Оглавление

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ О СТРУКТУРАХ ДАННЫХ И ОПТИМИЗАЦИИ КОДА

Данное пособие всецело посвящено двум важнейшим аспектам профессионального уровня создания программного обеспечения – грамотному, обоснованному выбору структур данных и программной оптимизации уровня исходных текстов (source-level code optimization). К большому сожалению, именно этим аспектам уделяется мало внимания в традиционных курсах программирования, хотя выигрыш в качестве программного обеспечения здесь довольно велик.Современный уровнь развития языковых средств реализации программного обеспечения позволяет в значительной степени уйти от традиционного трудоёмкого и обычно неэффективного создания собственных структур данных, опираясь на предельно эффективные встроенные в языки программирования возможности, в первую очередь, так называемые контейнеры. Знакомству с их устройством и функционалом посвящена первая часть пособия.

Вторая часть пособия нацелена на практическое освоение базовых и наиболее эффективных методов оптимизации кода применительно к самым вычислительно трудоёмким моментам в программировании – циклам и массивам. Формирование данных навыков является неотъемлемым элементом профессиональной разработки приложений.

Контейнеры в библиотеке STL

Классы, которые могут хранить информацию (контейнеры) можно условно поделить на два основных типа: последовательные (порядок, в котором добавлены элементы, сохраняется) и ассоциативные (элементы перестраиваются в контейнере специальным образом – чаще всего, чтобы соблюдалась та или иная упорядоченность).

К последовательным контейнерам относятся классы, которые используют модели массива, связного списка или производные от них. Контейнеры STL, основанные на этих структурах данных: array, vector, list, deque, queue, stack.

Ассоциативные контейнеры основаны на сбалансированных деревьях или на применении хеширования. Они позволяют выполнить поиск элемента с помощью операции [ ]. Обычно он работает быстрее, чем поиск в последовательных контейнерах. Чаще всего, применяются контейнеры set и map.

Операции на последовательных контейнерах

Массив – непрерывный участок памяти, выделенный под элементы одинакового размера. Благодаря этому, операция индексации [i] работает быстро: известен адрес начала массива в памяти, размер одного элемента, поэтому расположение i-го элемента получить легко. Например, для массива type *arr:

$$\&arr[i] = \&arr[0] + sizeof(type) * i,$$

где операция sizeof(type) показывает, сколько байт отведено под один элемент массива. Эта операция выполняется за O(1).

Такая организация элементов в памяти позволяет быстро добавлять элементы в конец массива (при условии, что выделено достаточно памяти и есть свободное место в конце массива). Если *len* – длина участка в памяти (в количестве элементов), выделенного под массив, а count – количество заполненных элементов, запись нового элемента *elem* в конец массива *arr* выглядит так (см. рисунок 1):

```
arr[count] = elem;
count++;
```

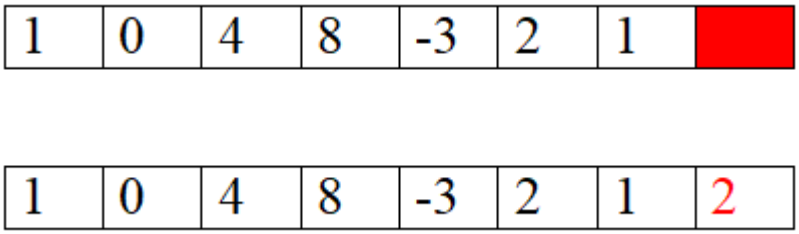


Рисунок 1. Добавление элемента 2 в конец массива.

Эта операция выполняется за константное время O(1).

Добавление элемента в середину выполняется медленнее, т.к. все элементы, которые должны оказаться после вставленного элемента нужно сдвинуть вправо. Выполнять это приходится, работая индивидуально с каждым элементом. Существуют структуры данных, которые позволяют выполнять эту операцию быстрее, чем за O(n) (например, см. операции над массивом на основе декартова дерева, treap). При использовании концепции массива операция вставки / удаления элемента из середины массива оценивается за O(n). Здесь и далее n – это оценка количества элементов в массиве. В целом, та же ситуация с добавлением элемента в начало массива. С точки зрения производительности, лучше всего, если удаётся переместить начало массива вперёд к новому элементу с индексом 0, но это возможно не всегда. В общем

случае надо сдвинуть все оставшиеся элементы массива на одну позицию (аналогично вставке / удалению из середины массива, см. рис.2).

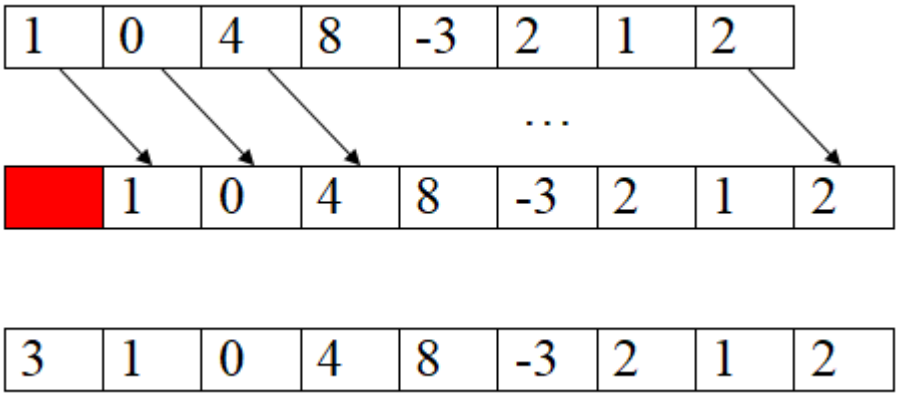


Рисунок 2. Добавление элемента 3 в начало массива.

При этом, за константное время можно выполнять операции, которые не требуют перемещения или сравнения большого количества элементов. Например, обмен местами двух элементов (swap(i, j), где i, j – индексы переставляемых элементов) не задействует никаких других элементов в массиве, поэтому выполняется за O(1).

Если массив не упорядочен, то во время поиска элемента *elem_to_find* по значению необходимо просматривать все элементы от начала массива к концу или в противоположном направлении. Такая операция оценивается в O(n):

```
for(i=0;i<len;i++)
    if(arr[i]==elem_to_find)
        ...
```

Если массив упорядочен, то поиск по нему можно выполнить за O(log n). Чаще всего применяется алгоритм бинарного поиска, на каждой итерации которого рассматриваемая часть массива делится на две части. При просмотре элемента, который расположен в середине, возможны четыре ситуации: 1) искомый элемент найден, 2) искомый элемент слева, 3) искомый элемент справа, 4) размер рассматриваемой части массива стал равен 0, элемент не найден. На рисунке 3 представлен пример бинарного поиска элемента 12, рассматриваемая часть массива выделена красным, середина массива выделена жирным шрифтом.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

1	2	3	4	5	6	7	8	9	10	**11**	12	13	14	15	16	17	18	19	20

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**16**	17	18	19	20

1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15	16	17	18	19	20

1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	15	16	17	18	19	20

Рисунок 3. Бинарный поиск элемента 12 в упорядоченном массиве

Итак, массив – это простая структура данных. Из-за простоты она, по умолчанию, используется как основная структура данных практически везде.

Операции, которые выполняются эффективно (за O(1)): индексация, добавление элемента в конец, обмен элементов местами, поиск в отсортированном массиве.

В STL на массиве основаны классы array (размер выделенного участка памяти зафиксирован) и vector (память можно выделять заново / увеличивать размер выделенного участка). Класс vector обладает более широкой функциональностью.

Массив используется и в других контейнерах. Например, deque представляет двунаправленную очередь (добавлять/удалять элементы можно и с начала, и с конца).

Пример работы класса array представлен ниже.

```
#include <array>
#include <iostream>

int main()
{
    array<int, 4> c0 = { 0, 1, 2, 3 };

    // просмотр содержимого массива даст: "0 1 2 3"
    for (int i = 0; i < 4; i++)
    {
        std::cout << c1[i] << " ";
    }
```

```
    std::cout << std::endl;

    return 0;
}
```

Связный список отличается от массива тем, что под него не обязательно выделяется непрерывный участок памяти. При отсутствии этого требования невозможно просчитать, где находится i-й элемент в памяти. Для перемещения между элементами вместе с данными хранят указатель на следующий элемент *Element* next* (рисунок 4). Здесь *Element* – это тип звеньев в списке.

Рисунок 4. Схема односвязного списка

В такой конфигурации речь идёт об односвязном списке. Указатель на первый элемент списка здесь обозначен как *Element* begin*. Если в каждом элементе списка, вдобавок, есть указатель на предыдущий элемент *Element* prev*, то речь идёт о двусвязном списке (рисунок 5).

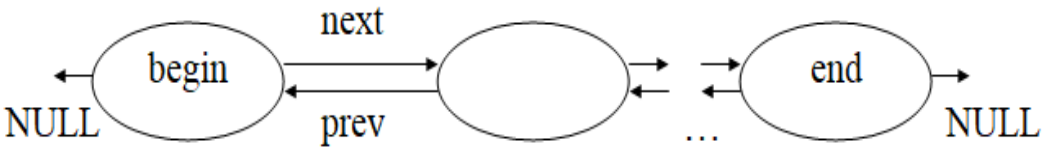


Рисунок 5. Схема двусвязного списка

Указатель на последний элемент списка здесь обозначен как *Element* end*.

Чтобы получить доступ к i-му элементу, необходимо установить указатель на начало списка и перечислить i элементов в цикле. Эта операция оценивается в O(n), где n – число элементов в списке. Из-за этого у контейнера list нет операции [i], а перебирать элементы надо при помощи итераторов. Более подробно о перечислении элементов контейнеров можно прочесть в разделе об итераторах.

В отличие от массива быстрее выполняется вставка и удаление элементов. Предполагается, что задан указатель на элемент, за или перед которым нужно выполнить вставку, или указатель на удаляемый элемент. При этом, перемещать множество элементов в памяти не требуется. Достаточно

настроить указатели элемента перед вставляемым / удаляемым элементом и после него. Вставка элемента *Element* newElem* в односвязный список вслед на элементом *Element* current* схематично представлена на рисунке 6. Красным цветом выделены редактируемые указатели. Кроме этого, может потребоваться изменить указатели на начало и конец списка. Количество этих действий ограничено и не зависит от длины списка. Эта операция оценивается в O(1).

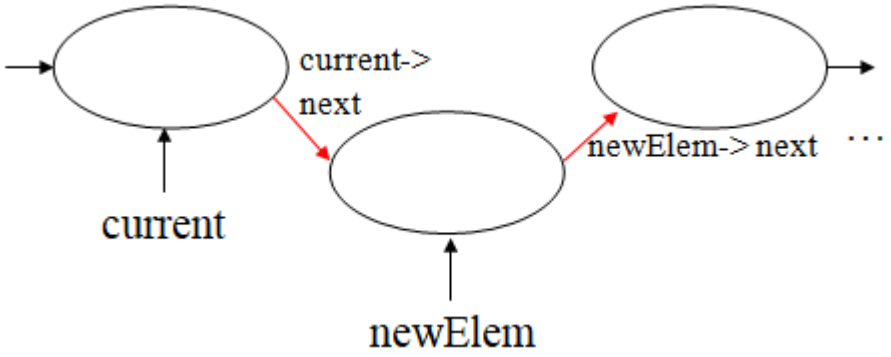


Рисунок 6. Вставка элемента в односвязный список.

Вставка элемента в двусвязный список, в целом, аналогична. Нужно отредактировать больше указателей: кроме, указателей на следующий элемент, указатели на предыдущий (см. рисунок 7).

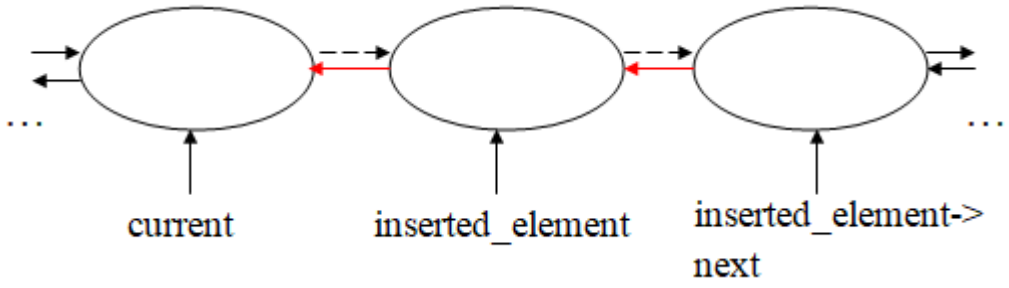


Рисунок 7. Вставка элемента в двусвязный список.

Эффективными операциями (выполняются за O(1)) также остаются добавление и удаление элемента с начала или конца списка. Применять бинарный поиск в списке неэффективно из-за неэффективной операции индексации (выполняется за O(n)), поэтому, даже при наличии упорядоченности элементов в списке, следует пользоваться схемой линейного поиска.

На основе массива или связного списка могут вводиться структуры данных стек и очередь. В стек элементы добавляются в конец и извлекаются с

конца (LIFO – Last In First Out). В очередь элементы также добавляются в конец, но извлекаются с начала (FIFO – First In First Out).

```
list<char> lst;
for (i = 0; i < 10; i++)
    lst.push_back('A' + i);

cout << "\nList:\n";
//итератор пробегает по списку
list<char>::iterator it_l = lst.begin();
while (it_l != lst.end())
{
//перемещение по списку с помощью итератора,
//нет операции [i]
    cout << *it_l << " ";
    it_l++;
}
```

Операции на ассоциативных контейнерах

Следующая важная группа структур данных – бинарные деревья.

У каждого элемента в дереве есть информация, которую нужно сохранить (обычно данные, data); также есть поле или набор полей, по которому информация упорядочена (для элементов введена операция <, обычно поле, по которому введено сравнение, - ключ, key).

В наличии связи в виде указателей: к родителю (parent) и к детям (children). В случае бинарных деревьев потомка два, которые чаще называются левым и правым дочерним элементом (left, right).

Узлы (nodes), у которых нет детей, - листья (leaf, leaves). Узел, у которого нет родителя, всего один – корневой (root).

Есть много примеров бинарных деревьев: деревья поиска, красно-чёрные деревья, АВЛ-деревья, бинарные кучи и т.д.

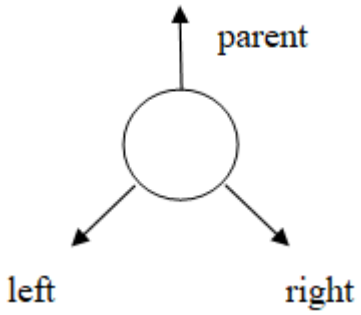


Рисунок 8. Узел в бинарном дереве.

Деревья поиска выделяет свойство упорядоченности: `left->key < parent->key <right->key`. Например, у куч (heap) `parent->key > child->key` или `parent->key < child->key`, где child – любой дочерний элемент (частичная упорядоченность).

Высота дерева – максимальное расстояние от корня к узлу (учтенное в узлах).

У сбалансированного дерева высоты всех ветвей отличаются не сильно. Для каждого узла current, гарантируется, что высоты поддеревьев, которые начинаются с его дочерних элементов отличаются по высоте не более, чем на 1.

На рисунке 9 высота узла 1 равна 2, высота 4 равна 3.

Но обычное дерево поиска не обязательно будет сбалансированным. Для его балансировки требуются дополнительные действия.

Предполагается, что, обрабатывая массив элементов, можно потратить ресурсы на построение дерева, к которому затем приходят запросы на поиск элемента по ключу. Дерево должно быть в состоянии обработать эти запросы быстрее, чем это произошло бы в массиве.

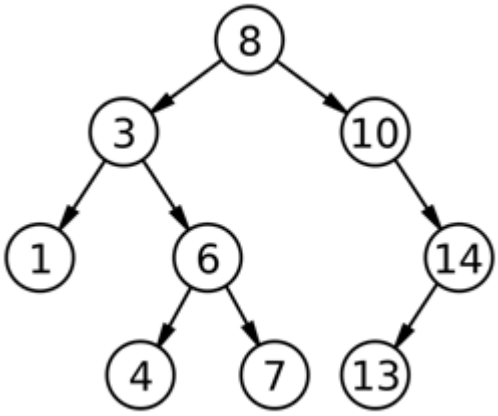


Рисунок 9. Сбалансированное дерево поиска.

Операция поиска заключается в спуске по дереву от корня. Причём, необходимо сравнить искомый ключ с ключом в текущем узле. Если в искомом узле значение больше, он должен быть в дереве справа – идём к правому потомку. Иначе, - идём влево (рисунок 10). В конце концов, либо мы окажемся в элементе с искомым ключом, либо попробуем спуститься по указателю со значением NULL. Это и есть проверка на то, присутствует ли искомый элемент в дереве.

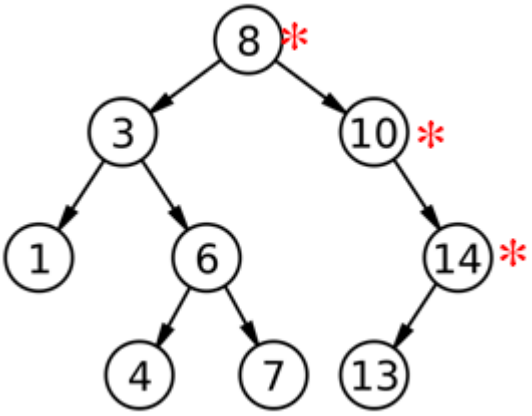


Рисунок 10. Поиск элемента 14 по дереву поиска (звездочкой обозначены просмотренные при спуске элементы).

Операция добавления элемента в дерево заключается в поиске места для элемента. Т.е. логику предыдущей операции нужно использовать практически полностью. Если элемент с тем же ключом, что добавляемый в дереве уже есть, такую ситуацию нужно дополнительно обсуждать. Если договориться, что её не будет, когда-то спуск приведёт к узлу, от которого нужно спускаться по указателю NULL. В это место и нужно добавить новый элемент.

Обе операции оцениваются по производительности в высоту дерева: O(h). В худшем случае имеем связный список h = n, O(n).

В лучшем случае дерево сбалансировано. Его высота оценивается в O(log n), а вместе с ней и операции добавления и поиска.

Для балансировки деревьев используются специальные преобразования узлов дерева: например, для АВЛ-дерева – повороты. Благодаря им, после добавления элементов дерево остаётся сбалансированным.

Поиск минимального и максимального элементов возводит принцип упорядоченности в абсолют. Дочерний элемент, меньший родителя, - слева, а больший – справа, поэтому для поиска минимума нужно организовать спуск по дереву по левой ветви (всегда выбираем левый дочерний элемент), для поиска максимума – по правой ветви.

При удалении элемента есть три возможные ситуации: 1) у узла не было дочерних элементов (например, удаляем 4 из дерева на рисунке 10), 2) у удаляемого узла один потомок – он и занимает его место (удаляем 14 – его место занимает 13), 3) у удаляемого узла есть два потомка.

Последний случай интереснее всего. Рекомендуется постулировать, что узел, который должен занять место удаляемого, находится среди его дочерних элементов.

Предположим, что он находится в левом поддереве. Он автоматически меньше всех элементов правого поддерева – правое поддерево вообще не надо перестраивать. Элемент, который займёт место удалённого, должен быть больше всех элементов, остающихся слева. Таким образом, это максимум в левом поддереве. Гарантируется, что у него не более одного потомка, поэтому операция его перемещения – это один из простых случаев. Аналогично можно использовать минимум из правого поддерева.

В целом, все операции, которые указаны выше, требуют прохода по ветвям от корня к узлу, поэтому оцениваются за O(h). Высота может быть любой – и «хорошей», и «плохой». Есть подходы к перестройке дерева, чтобы оно оставалось сбалансированным: 1) «повороты» дерева или операции балансировки используются в АВЛ-дереве и красно-чёрном дереве, 2) постоянные слияния и разрезания в декартовом дереве (treap) оставляют его сбалансированным в среднем, без необходимости дополнительно проводить балансировки.

Для балансировки дерева вводятся либо специальные правила (какие потомки/родители могут быть у красного/черного узла), либо повороты (АВЛ-дерево). Но результат получается примерно одинаковым по структуре дереве и одинаковым по оценкам операций на нем. Здесь предлагается использовать АВЛ-дерево.

При добавлении узла требуется выполнить те же действия, что и в дереве поиска, но потом нужно запустить процедуру балансировки, если после него дерево разбалансировалось, поэтому удобно использовать наследование от дерева поиска.

Естественно, процедуры поиска минимума, максимума и просто поиска элемента не меняются по сути. Только гарантируется, что они получают «хорошие» оценки по производительности (O(log n), т.к. ту же оценку имеет высота сбалансированного дерева).

Перечислить все элементы в дереве можно несколькими способами. Они разделяются по порядку обхода дочерних элементов.

Находясь в узле current, мы можем опросить: 1) сначала левого ребенка, потом current, потом его правого ребенка (InOrder-обход – получим все элементы в дереве в порядке возрастания), 2) сначала левого и правого ребенка, затем родителя – обратный обход, PostOrder, 3) сначала родитель, потом левый ребенок, затем – правый, PreOrder. Обход в порядке InOrder даёт отсортированную последовательность (рисунок 11).

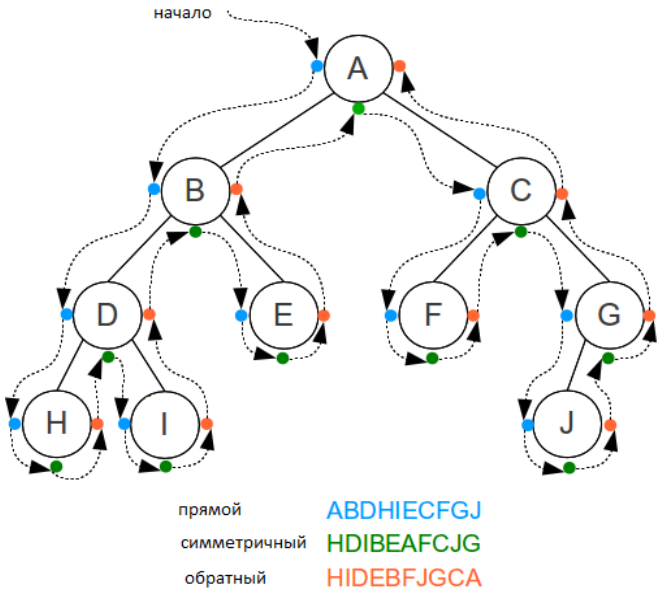


Рисунок 11. Обходы дерева.

Контейнеры map и set библиотеки STL используют красно-черное дерево, обладающее схожими характеристиками и похожими операциями.

Ключи, которые хранятся в контейнерах, должны иметь операцию operator<, чтобы их можно было упорядочить.

Если перечислять элементы в этих контейнерах с помощью итераторов, начиная с итератора begin() и заканчивая итератором end(), будет получена последовательность элементов с ключами в порядке возрастания. Этот процесс аналогичен InOrder-обходу дерева.

Таким образом, поиск по ключу, добавление и удаление элемента в map и set оценивается в O(log n). Стоит обратить внимание на то, что поиск элемента по массиву оценивается в O(log n) только в случае бинарного поиска в отсортированном массиве.

```
//красно-черное (сбалансированное) дерево map, есть
интерфейс доступа к значению по ключу
map<string, int> marks;
```

```
marks["Petrov"] = 5;
marks["Ivanov"] = 4;
marks["Sidorov"] = 5;
marks["Nikolaev"] = 3;
marks["Abramov"] = 4;
marks["Fedorov"] = 5;
marks["Kuznetsov"] = 4;

cout << "\nMap:\n";
//итератор пробегает по map
map<string, int>::iterator it_m = marks.begin();
while (it_m != marks.end())
{
    //перемещение по списку с помощью итератора, нет
операции [i]
    cout << "Key: " << it_m->first;
    cout << ", value: " << it_m->second << "\n";
    it_m++;
}
//вывод в порядке возрастания ключей
```

Класс set (множество) чаще используется для того, чтобы определить все уникальные ключи в некотором массиве. Добавив все элементы массива в него, а затем перечислив содержимое set, можно перечислить все уникальные элементы массива. Так же можно получить и количество различных значений в массиве (или в другой структуре данных / контейнере).

```
set<string> various_values;
various_values.insert("test1");
various_values.insert("test2");
various_values.insert("test1");
various_values.insert("test2");
various_values.insert("test3");
various_values.insert("test4");
various_values.insert("test3");

cout << "\nSet:\n";
//итератор пробегает по set
```

```
//(сбалансированному дереву / множеству)
set<string>::iterator it_s;
it_s = various_values.begin();
while (it_s != various_values.end())
{
//перемещение по списку с помощью
//итератора, нет операции [i]
    cout << *it_s << " ";
    it_s++;
}
//вывод: test1 test2 test3 test4
```

Контейнеры unordered_map / unordered_set реализуют тот же интерфейс доступа по ключу с помощью операции [ ], но в основе поиска лежит хеширование, поэтому получение элемента из контейнера по ключу занимает O(1). При этом, перечислить элементы в контейнере с помощью итераторов уже нет возможности.

Бинарная куча (heap) – это бинарное дерево, удовлетворяющее условию: родитель больше (или меньше) потомков. Дети определённого узла могут располагаться в произвольном порядке: левый ребенок не обязан быть меньше правого. Здесь вместо понятия ключа пользуются приоритетом.

Договоримся, что родитель больше потомков, тогда первое полезное свойство видно сразу: в корне находится максимум. На рисунке 1: 20>11 и 20>15.

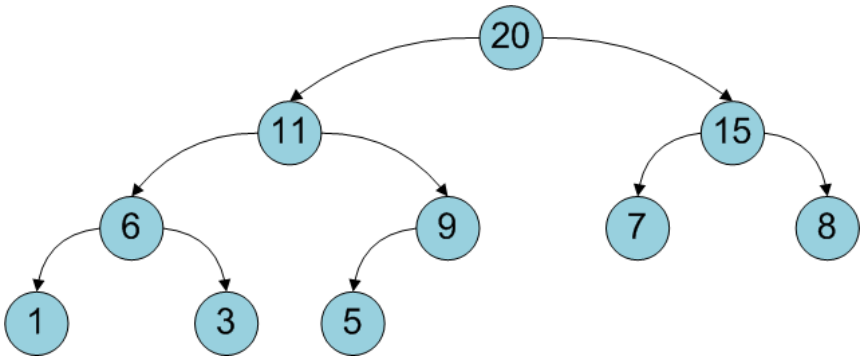


Рисунок 12. Пример бинарной кучи.

Подходов к работу с кучей два: на основе бинарного дерева и массива. Опишем «древесные» операции с кучей на основе массива.

Договоримся заполнять дерево сверху вниз по уровням и слева направо внутри уровня: нулевой элемент массива – корень, первый и второй – его дети на 2-м уровне (11 и 15), элементы с индексами 3 – 6 находятся на 3-м уровне (6, 9, 7, 8). Куче на рисунке 12 соответствует массив:

20, 11, 15, 6, 9, 7, 8, 1, 3, 5

Такой подход является нестандартным. В литературе чаще работают с кучей на основе дерева (см. Вирт, пункт 2.2.5). Он возможен благодаря следующему полезному свойству. Т.к. дерево является сбалансированным, а операция добавления использует самый нижний незаполненный полностью уровень (на рисунке 12 элементы 1, 3, 5), индексы родителя и потомков связаны простым соотношением: child_left_index = 2*parent_index + 1 и child_right_index = 2*parent_index + 2 (индексация с 0).

Единственное место, куда в массиве можно добавить элемент «быстро», - это конец. В нашем дереве ему соответствует первое свободное место слева на последнем уровне дерева (см. рисунок 13).

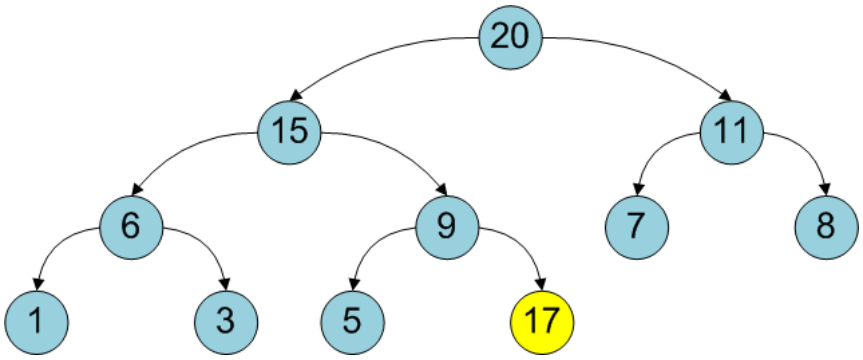


Рисунок 13. Добавление элемента 17 в конец массива (свойства кучи нарушились).

Теперь необходимо проверить, сохраняется ли свойство кучи: родитель > потомки. Здесь 9 и 17 расположены неверно, меняем их местами («просеиваем» 17 вверх, процедура sift up).

На следующем шаге получим ту же ситуацию для бывшего родителя элемента 9 и элемента 17, занявшего его место (рисунок 14). Процедура может быть реализована рекурсивно. Продолжаем её, пока родитель не будет больше потомков (рисунок15). В худшем случае 17 займёт место корня. Т.к. дерево сбалансированное, высота дерева оценивается по порядку в log n, и этот случай «стоит» O(log n).

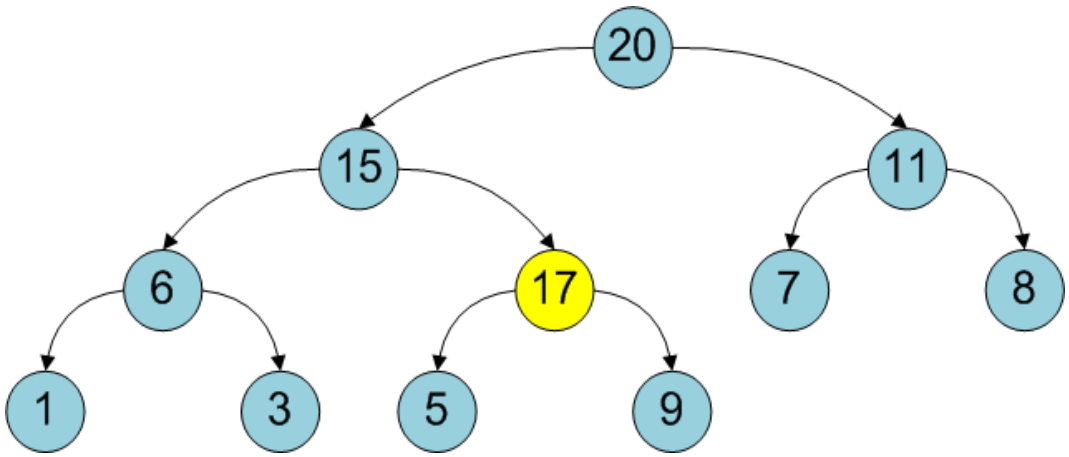


Рисунок 14. Добавление элемента 17 в конец массива (свойства кучи нарушились).

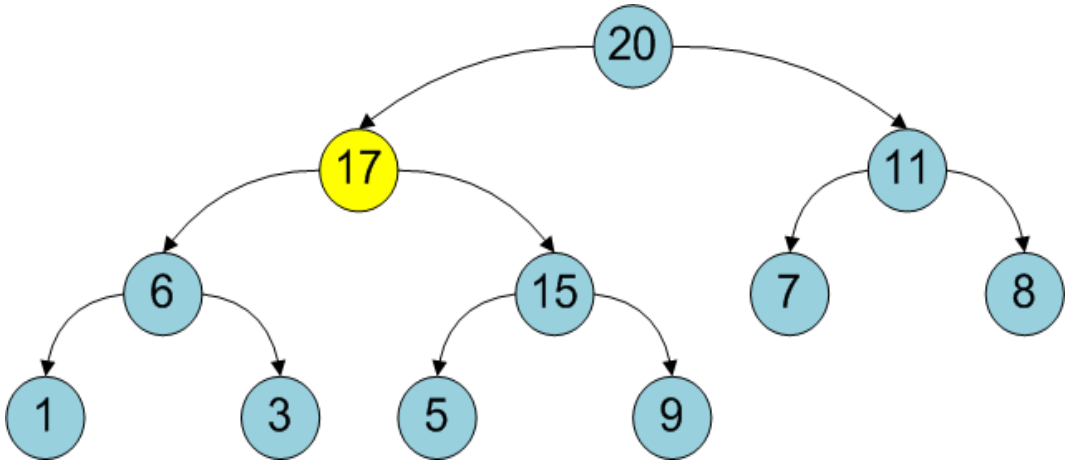


Рисунок 15. Просеивание элемента 17 вверх, sift up, третий шаг.

Т.к. 20 > 17, ситуация на рисунке 15 – конечная. Больше операций не нужно. С точки зрения массива, все эти действия – обмен элементов местами. Они оцениваются за O(1), поэтому такое «просеивание вверх», и вправду, стоит O(log n). Для выполнения этого действия можно реализовать рекурсивную процедуру, т.к. количество вызовов функции не будет большим – порядка log n, здесь n – число элементов в куче.

Естественно, операция удаления элемента должна быть связана с «просеиванием вниз» (sift down). Важная операция для кучи – удаление корня (в корне находится либо максимум, либо минимум).

Если мы уберём элемент в корне, его место должен занять другой элемент. Работая с массивом, нельзя просто так удалить элемент (тем более, 0-й), - это уже стоит O(n), но можно произвести обмен элементов местами. Единственный обмен, в результате которого не надо двигать массово другие

элементы: обмен местами корня и самого последнего элемента (см. рисунок 16, место корня занимает последний элемент в массиве, они оба отмечены желтым). «Забываем», что корень попал на последнее место: считаем, что количество элементов уменьшилось на 1.

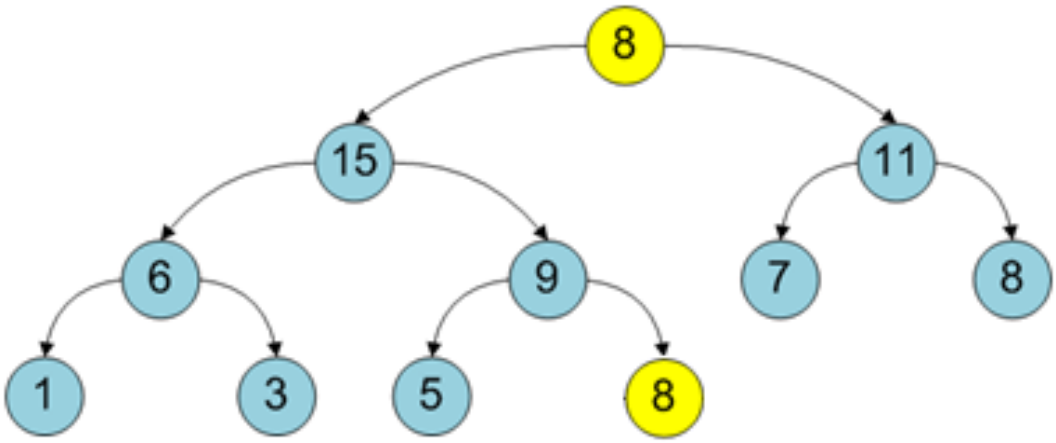


Рисунок 16. Удалили корень и поставили на его место последний элемент 8 (свойства кучи нарушились).

Теперь в корне дерева имеем потенциальное нарушение свойств кучи. Если корень меньше одного из детей: максимум из 3 элементов обменивается местами с корнем. Процедура повторяется рекурсивно для поддерева, корнем которого становится перемещаемый элемент (см. рисунки 17, 18).

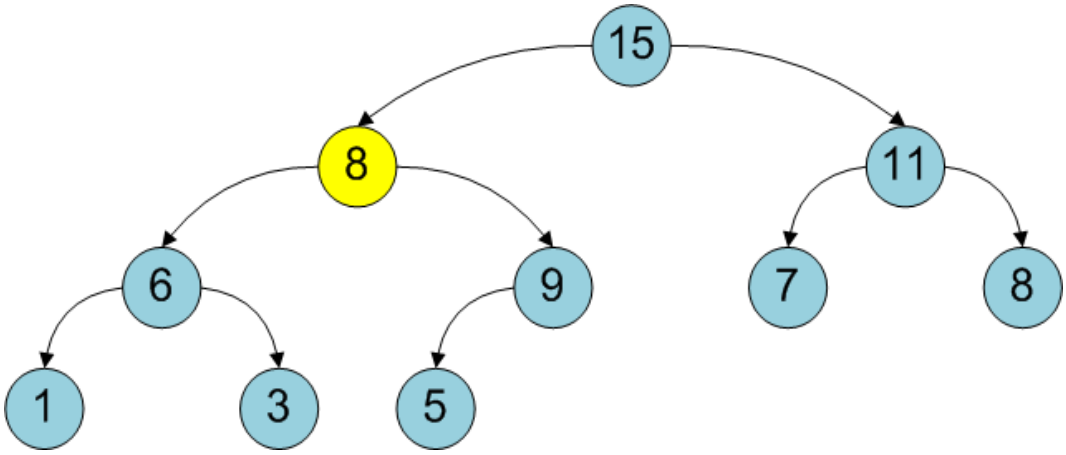


Рисунок 17. Просеивание элемента 8 вниз, sift down, первый шаг.

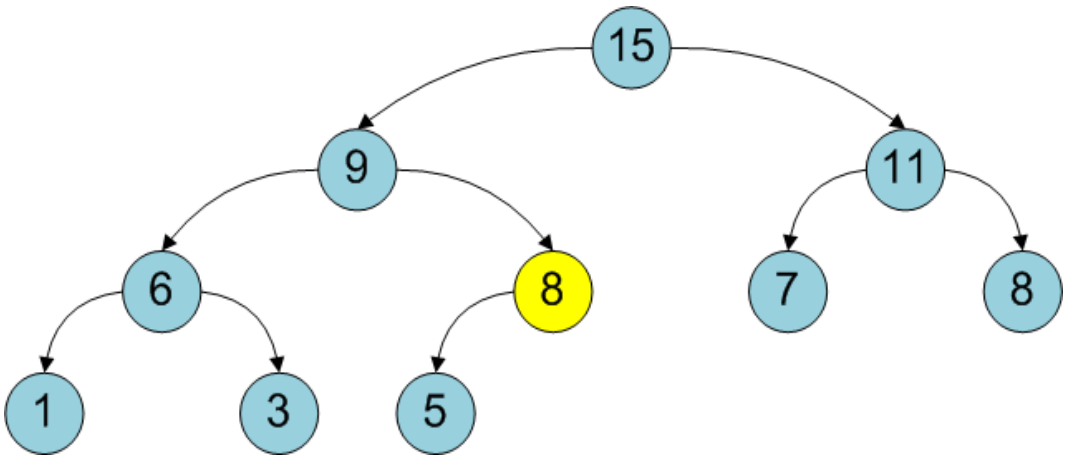


Рисунок 18. Просеивание элемента 8 вниз, sift down, второй шаг.

Естественно, из-за того, что высота дерева оценивается в O(log n), процедура «просеивания вниз» (sift down) «стоит» в худшем случае прохода по самой длинной ветке (в сбалансированном дереве отличается от самой короткой не более, чем на 1 узел) – O(log n).

Конечно, «хороший» поиск из АВЛ-дерева потерян. Но получаем простую процедуру сортировки HeapSort(): удаляем из кучи корень (максимум), сохраняя его в массиве-результате, пока все элементы кучи не закончатся. Всего n элементов, просеивание можно оценить за O(log n), поэтому сортировка является логарифмической: O(n log n).

Она также обладает интересным свойством. При создании кучи из отсортированного массива в порядке возрастания или убывания тратится в среднем одинаковое число действий. По этой причине у сортировки кучей нет ни хороших, ни плохих случаев. Всё стоит O(n log n).

Контейнер prority_queue, очередь с приоритетами, основан на куче и позволяет получать все элементы в порядке убывания приоритета.

Сравнение контейнеров STL по производительности

В заключение этого описания контейнеров приведем таблицу сложности их важных операций.

Таблица 1. Сравнение операций над контейнерами по сложности

тип контейнера	вставка нового элемента	удаление элемента	непосредственный доступ к произвольному элементу по индексу (позиции) или ключу	непосредственный доступ к произвольному элементу по его значению	естественная упорядоченность по значению или по хэшу	доступ к следующему и предыдущему элементу от данного для итераторов	доступ к первому и последнему элементам для итераторов
forward_list (односвязный список)	только в начало или после позиции итератора: O(range)	только начального элемента или после позиции итератора: O(range)	нет	нет	нет	только к **следующему**, O(1)	только к **первому**, O(1)
list (двусвязный список)	только в начало или в конец или после позиции итератора: O(1)	только начального или конечного элемента или после позиции итератора: O(1)				O(1)	O(1)
stack (стек, LIFO)	только в **конец**: O(1)	только из **конца**: O(1)	нет	нет	нет	нет	только к **последнему** элементу, O(1)
queue (очередь, FIFO)		только из **начала** (первого вставленного): O(1)					O(1)
priority_queue (очередь с приоритетами)		только **максимального** элемента			частичная по значению: всегда доступен только **максимальный** элемент		нет
deque (двунаправленная очередь, дек)	только в начало или в конец или после позиции ите-	только из начала или из конца или после позиции итератора:	по индексу	нет	нет	O(1)	O(1)

	ратора: O(1)	O(1)					
set (упорядоченное множество)	в место, зависящее только от значения вставляемого элемента или итератора или диапазона итераторов, в худшем случае $O(\log_2(n))$	удаляется элемент с заданным значением или с заданным итератором или диапазоном итераторов, в худшем случае $O(\log_2(n))$	нет	в худшем случае $O(\log_2(n))$	сортировка по значениям, каждый элемент **уникален**	в худшем случае $O(\log_2(n))$	O(1)
multiset (упорядоченное множество с повторениями)					сортировка по значениям, может быть **несколько** элементов с одинаковыми значениями		
unordered_map	с уникальным ключом O(1)	с уникальным ключом O(1)	даёт итератор или значение по ключу, O(1), в худшем случае O(n)	нет	сортировка по хэшам	только к **следующему**, O(1), в худшем случае O(n)	только к **первому**, O(1)
map	по ключу, который может быть не уникальным или диапазону итераторов	по ключу $O(\log_2(n))$ или итератору O(1) или диапазону итераторов O(range)	даёт итератор или значение по ключу		сортировка по ключам	O(1), в худшем случае O(n)	O(1)
multimap					сортировка по ключам с возможными повторениями		
unordered_multimap	по значению или диапазону итераторов				сортировка по хэшам	только к **следующему**, O(1), в худшем случае O(n)	только к **первому**, O(1)
vector	в конец, в худшем случае O(n)	последнего элемента, O(1)	по индексу, O(1)	нет	нет	O(1)	O(1)
array	нет, размер задаётся при компиляции						

Итераторы

Итераторы позволяют перебирать элементы в контейнере. Это особенно важно, если в соответствующем контейнере нет операции индексации, например: связный список list или бинарное дерево map / set.

При этом, также итераторы могут применяться, чтобы скопировать элементы контейнера в поток или обратно. Перебор элементов в векторе представлен ниже:

```
vector<int> v;
…
cout << "\nVector:\n";
//итератор пробегает по вектору
vector<int>::iterator it_v = v.begin();
while (it_v != v.end())
{
    cout << *it_v << " ";
    it_v++;
}
```

Перебор элементов в списке представлен ниже. Он не отличается по структуре от перечисления элементов в векторе:

```
list<char> lst;
    …
//итератор пробегает по списку
list<char>::iterator it_l = lst.begin();
while (it_l != lst.end())
{
//перемещение по списку с помощью
//итератора, нет операции [i]
    cout << *it_l << " ";
   it_l++;
}
```

При переборе элементов в map / set элементы будут перечисляться в порядке возрастания ключей (InOrder-обход дерева):

```
map<string, int> marks;

//итератор пробегает по map
map<string, int>::iterator it_m = marks.begin();
while (it_m != marks.end())
{
//перемещение по списку с помощью
//итератора, нет операции [i]
cout << "Key: " << it_m->first;
cout << ", value: " << it_m->second << "\n";
 it_m++;
}
```

Элементы контейнера можно перечислить по-другому. Например, перебор элементов очереди с приоритетами:

```
#include <iostream>
#include <queue>


template<typename T>
void print_queue(T& q)
{
    while (!q.empty())
    {
        cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}


int main() {
    priority_queue<int> q;
```

```
        for (int n : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
            q.push(n);

        print_queue(q);

    }
```

Алгоритмы

Алгоритмы содержат реализацию часто встречающихся операций: сортировку, поиск, применение ко всем элементам контейнера функции, перебор элементов в контейнере, подсчёт числа элементов, удовлетворяющих условию и т.д. Для работы с ними нужно подключить библиотеку algorithm.

Примеры некоторых полезных алгоритмов:

1. Поиск элемента по значению или условию: find, find_if.
2. Копирование элементов контейнера: copy.
3. Сортировка элементов контейнера: sort.
4. Перебор элементов контейнера: for_each.
5. Слияние элементов двух контейнеров: merge.
6. Подсчет числа элементов, совпадающих с заданным значением или удовлетворяющих условию: count, count_if.
7. Обмен местами элементов контейнера: swap.
8. Применение к элементам контейнера функции: transform.

```
#include <algorithm>
…

//алгоритмы count, count_if  - подсчёт числа
//элементов в контейнере, совпадающих с переданным,
//или тех, для которых функция glas вернёт true
int n1 = count(lst.begin(), lst.end(), 'a');
int n2 = count_if(lst.begin(), lst.end(), glas);
cout << "\nNumber 'a' = " << n1;
cout << "\tGlas = " << n2 << "\n";

//алгоритм transform - применить ко всем элементам
//списка lst функцию CaesarForward
```

```
p = transform(lst.begin(), lst.end(),
lst.begin(), CaesarForward);

//поиск элемента 4 в векторе v. Если ничего не найдено,
//возвращается итератор end()
vector<int>::iterator p1 = find(v.begin(), v.end(), 4);
cout << "\nFind algorithm (4): " << *p1;
//поиск в векторе v элемента, для которого
//функция MoreThan5 вернет true
p1 = find_if(v.begin(), v.end(), MoreThan5);
cout << "\nFind_if algorithm > 5: " << *p1;

//применить ко всем элементам вектора v функцию print()
for_each(v.begin(), v.end(), print);
```

Оптимизация кода

Оптимизация кода предполагает построение кода, который выполняется быстрее или требует меньше памяти для работы. На оптимизацию можно смотреть как на обобщённые эквивалентные преобразования, гарантированно приводящие к тому же результату, но с меньшими затратами ресурсов.

Выделяют алгоритмическую оптимизацию (применение более эффективных алгоритмов и структур данных для ускорения работы с памятью) и программную оптимизацию (построение кода, который работает быстрее или потребляет меньше памяти). Перед оптимизацией кода всегда следует сперва понять, какой выигрыш способен дать метод оптимизации и какими усилиями. Очень часто оказывается (например, в результате профилирования программы), что малый кусок кода программы является основным потребителем процессорного времени или памяти. Поскольку оптимизация кода – достаточно трудоёмкий процесс, логично сосредоточиться именно на оптимизации данного фрагмента кода программы. Опыт создания крупных приложений говорит в пользу первичности алгоритмической оптимизации. Она трудоёмка, но обычно даёт отменные результаты. Если же достаточен умеренный выигрыш или возможности алгоритмической оптимизации исчерпаны, единственный

выход – программная оптимизация, рассмотрению которой и посвящён этот раздел.

Оптимизация циклов

Оптимизация циклов (loop optimization) позволяет получить ускорение в работе с циклами. По этой причине она является важнейшим видом оптимизации. Если удаётся ускорить исполнение на процессоре тела цикла, этот выигрыш умножается на число итераций цикла, и таким образом даёт очень эффективную отдачу от усилий программиста.

Необходимо принимать во внимание, что условные операторы в теле цикла – это первый кандидат на оптимизацию, поскольку выполняются потенциально множество раз. Изъятие условных операторов из циклов столь важно, что ради этого можно решиться на радикальную переработку всего цикла, включая практически произвольное усложнение условия продолжения цикла, разрезку цикла на части и так далее.

Следующие методы оптимизации цикла являются стандартными и заслуживают детального описания.

1. Развертка (loop unrolling) используется при малом теле цикла (с известной длиной). Вместо цикла с n-итерациями, компилятор будет производить код, который просто повторяется n раз.

Метод может повысить производительность, так как устраняет любые инструкции перехода, что является целью многих оптимизаций. Однако, следует учесть, что, если увеличенный цикл не помещается в кэш, то данный способ не целесообразен.

Пример:

Количество итераций/размер массива: 10 000

До (111 мкс):

```
for (int i = 0; i < iN; i++)
{
    res *= a[i];
}
```

После (48 мкс):

```
for (int i = 0; i < iN; i+=3)
{
```

```
    res1 *= a[i];
    res2 *= a[i+1];
    res3 *= a[i+2];
}
res = res1 * res2 * res3;
//вынесли из цикла, так как операция однократная
```

2. Объединение циклов (loop fusion) используется, когда:
Количество итераций циклов одинаково либо кратно
Циклы не зависимы по данным, то есть, каждая последующая операция второго цикла не зависит от данных первого цикла
Это позволяет уменьшить дополнительные расходы на циклы. Такая оптимизация не всегда выигрывает в производительности, так как тело цикла может «вывалиться» из кэша, и в этом случае лучше применить разрезание циклов.
Инициализировать два массива в отдельных циклах, иногда лучше, чем оба массива одновременно в одном.
Пример:
Количество итераций/размер массива: 10 000
До (267 мкс):

```
for (int i = 0; i < iN; i++)
{
    a[i] = b[i] - 5;
}
for(int i = 0; i < iN-1; i++)
{
    d[i] = e[i] * 3;
}
```

После (125 мкс):

```
for (int i = 0; i < iN-1; i++)
{
    a[i] = b[i] - 5;
    d[i] = e[i] * 3;
}
a[iN-1] = b[iN-1] - 5;
```

3. Разрезание циклов (loop distribution) используется, когда большое тело программы не помещается в кэш, начало вытесняет конец и наоборот, в худшей ситуации данные вытесняются в оперативную память (ОП), что существенно замедляет программу.

Важно понять, какая часть попадет в кэш, и вывести эту часть в отдельный цикл. Это возможно тогда, когда вычисления внутри тела цикла не зависят друг от друга.

Метод помогает удалить последовательные операторы в отдельный цикл и собрать параллелизуемые операторы в другой цикл.

Применение может мешать зависимости по данным, в таком случае лучше применить объединение циклов.

Пример:

До:

```
for (i=0; i < n; i++)
{
    x[i] = y[i] + z[i]*w[i];                  /* S1 */
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;     /* S2 */
    y[i] = z[i] - x[i];                       /* S3 */
}
```

После:

```
/* L1: parallel loop */
for (i=0; i < n; i++)
{
    x[i] = y[i] + z[i]*w[i];                  /* S1 */
    y[i] = z[i] - x[i];                       /* S3 */
}
/* L2: sequential loop */
for (i=0; i < n; i++)
{
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

4. Перестановка вложенных в друг друга циклов (loop interchange)

Выделяют 2 основные цели:

1. Тело самого внутреннего цикла должно находиться в кэш памяти
2. Цикл, который находиться внутри, должен быть максимальной длины

Тут мы встречаем противоречие. Важно понять, что даст больший выигрыш.

Больший выигрыш даст «погружение» в кэш память, значит, возможно, вам придется применить разрезание циклов.

Рассмотрим несколько случаев:

1. «Идеальный». Всё помещается в кэш память. В этом случае самое выгодное сделать самый длинный цикл – внутренним, а самый короткий – самым внешним.
2. «Лучший». Вся или большая часть данных находится в памяти непрерывно и проходится самым внутренним циклом, помещаясь в кэш память. Такая ситуация может не соответствовать идее о том, что самый большой цикл должен находиться внутри. Главная задача - «сманить» данные в кэш последовательным чтением, чем больше обращений к этим данным, тем больше выигрыш.
3. «Огромные циклы». В любом случае данные не попадут в кэш. Внутрь вложенных циклов разумно вставить самый маленький цикл, потому что он работает с одной областью данных, которая уже находится в кэше. Но если данные устроены так, что из кэша ничего не удалось извлечь, тогда есть смысл сделать внутренним самым длинный, а внешним самый короткий, чтобы уменьшить накладные расходы на организацию циклов.

Важно: соизмеряйте выигрыш от разных оптимизаций, так как универсального приема не существует.

Метод нужен для того, чтобы помочь компилятору распараллеливать код.

Пример:

Количество итераций/размер массива: 200*200

До (50516 мкс):

```
for(int i = 0; i < iN; i++)
for(int j = 0; j < jN; j++)
for(int k = 0; k < kN; k++)
    c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

После (47038 мкс):

```
//меняем местами, чтобы данные шли последовательно
for(int i = 0; i < iN; i++)
for(int k = 0; k < kN; k++)
```

```
for(int j = 0; j < jN; j++)
    c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

К конечному циклу можно применить объединение.

5. Разбивка на блоки (loop tiling)

Если тело цикла сложное, то можно применить эту оптимизацию, и разбить цикл на более маленькие части. Разбиение пространства итерирования цикла приводит к разбиению массива на меньшие блоки, которые помещаются в кэш, что приводит к улучшению использования кэша, снижению количества промахов. Можно применять вместе с перестановкой циклов, но не во всех случаях.

Этот метод может увеличить накладные расходы на циклы.

Пример:

Количество итераций/размер массива: 100*100

До: (383 мкс)

```
for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
    c[i] = c[i] + a[i, j] * b[j];
```

После: (127 мкс)

```
for (i = 0; i < N; i += 2)
for (j = 0; j < N; j += 2)
for (ii = i; ii < min(i + 2, N); ii++)
for (jj = j; jj < min(j + 2,N); jj++)
    c[ii] = c[ii] + a[ii, jj] * b[ii];
```

6. Циклическое преобразование кода (loop-invariant code motion)

Применяем, если можно вынести одноразовые вычисления за цикл, чтобы не вычислять их несколько раз. Этот метод значительно повышает производительность.

Пример:

Количество итераций/размер массива: 10 000

До: (125 мкс)

```
for (int i = 0; i < n; ++i)
```

```
{
    x = y + z;
    a[i] = 6 * i + x * x;
}
```

После: (113 мкс)

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; ++i)
{
    a[i] = 6 * i + t1;
}
```

7. Loop unswitching

Используется в случаях, когда условие внутри тела цикла мешает его распараллеливанию, т.к. современные процессоры быстро работают на векторах.

Пример:

Количество итераций/размер массива: 100 000

До: (760 мкс)

```
int i, w, x[1000], y[1000];
  for (i = 0; i < 1000; i++)
 {
    x[i] += y[i];
    if (w)
      y[i] = 0;
  }
```

После: (527 мкс)

```
int i, w, x[1000], y[1000];
 if (w)
 {
   for (i = 0; i < 1000; i++)
   {
      x[i] += y[i];
```

```
      y[i] = 0;
   }
 }
 else
 {
    for (i = 0; i < 1000; i++)
    {
      x[i] += y[i];
    }
 }
```

Оптимизация массивов

Массив – это структура данных, представленная в виде ячеек определенного типа, объединенных одним именем. Массивы используются, когда необходимо обработать большое количество данных. Его имя является постоянным указателем, который указывает на первый элемент массива. Вследствие чего можно использовать указатели и их арифметику.

Вследствие значительных размеров массива, работа с ним осуществляется обычно посредством циклов и занимает относительно большое время. Отсюда вытекает важность оптимизации массивов, которая, вследствие определённого стандартного способа укладки данных в памяти и специфики работы компьютерной подсистемы обращения к памяти, выходит далеко за рамки оптимизации циклов.

Обращение к элементам массива выглядеть, как показано в примере 1.

Пример 1

```
for (int i=0; i<n; i++)
     array [i] = n_some_value;
```

Вместо приведенного выше кода эффективнее сделать, как показано в примере 2.

Пример 2

```
int* arrn=array +n;
for (int * ptr_int=array; ptr_int<arrn; ptr_int ++)
    * ptr_int= n_some_value;
```

Причина этого в операциях с указателями. В примере 2 есть указатель на тип int, который берет адрес из имени массива array и увеличивает его для каждого элемента.

В примере 3 рассмотрена очевидная обработка двумерного массива. (В данном случает его обнуление).

Пример 3

```
for  (int i=0; i<n_rows; i++)
     for (int j=0; j<n_cols; j++)
          array[i][j]=0;
```

Использование указателей сделает код из примера 3 более эффективным.

Пример 4

```
int *p = &array[0][0];
int *end = &array[0][0] + n_rows *n_cols;
for (p = *array; p < end; p++)   *p = 0;
```

Используя оптимизацию из примера 4, можно ускорить обработку массива приблизительно в 10 раз. Вместо того, чтобы вычислять позицию каждого элемента с помощью индексов, указатель перемещался по массиву, увеличивая на одну целую позицию после каждой итерации до конца – адреса, который был вычислен один раз, заранее. Конечно, хороший оптимизирующий компилятор упростил бы код примера 4.

В примере 5 требуется посчитать сумму элементов двумерного массива.

Пример 5

```
int a[n][m];
int sum =0;
for (int i=0; i<n;i++)
     for (int j=0; j<m;j++)
          sum+=a[i][j];
```

Время выполнения фрагмента кода из примера 5 может сильно увеличить при достаточно больших n и m, потому что на каждой итерации цикла при определении адреса [i][j]-го элемента компилятор использует много вычислений.

Зная о том, что элементы массива расположены в памяти последовательно, можно оптимизировать пример 5, введя вспомогательный указатель.

Пример 6

```
int *p= &a[0][0];
float size=sizeof(a)/sizeof(int);
for (int i=0; i<size;i++)
{
    sum+=*p;
    p++;
}
```

Так же можно использовать эквивалентное объявление указателя *p= a[0].

Пример 7

```
int *p; //указатель на массив элементов
x=*(p++);
y=*(++p);
```

В примере 7 первое присваивание значительно эффективней, чем второе. В первом случае будет осуществляться разыменование указателя и его инкремента параллельно, а во втором – последовательно.

Еще один совет по оптимизации работы с двумерными массивами.

Количество столбцов двумерного массива желательно должно быть равно степени двойки при небольших типах данных. Это ускорит работу с массивом, поскольку это выравнивает указатели на первые элементы каждой строки, что ускоряет доступ к элементам.

Пример 8

```
double array [7][8]; // 8 =2^3
```

Оптимизация ветвлений

В состав современных процессоров входит модуль предсказания ветвлений – branch predictor. Всякий раз, когда код имеет ветвь (например, if-структуру) или если мы находимся внутри цикла, микропроцессор не знает

заранее, какая из двух ветвей нужна для подачи в pipeline. Branch predictor нужен для того, чтобы предсказать по какой ветке пойдет if: выполнение останется внутри тела цикла или выйдет из него.

Сам процессор работает быстрее, чем оперативная память, из которой он берет данные. Поэтому желательно обращаться к памяти таким образом, чтобы данные заведомо находились либо в регистрах процессора, либо в кэше памяти. В таком случае, процессор может эти данные подтянуть, если будет знать, куда пойдут вычисления.

Вычисления развиваются линейно, поэтому предсказать, что понадобится через некоторое количество тактов процессора достаточно легко, если в этом коде нет ветвлений.

Первый способ оптимизировать ветвление – использовать в операторе if более вероятное условие. Код, который выполняется при верном условие, расположен там же, где и сама операция сравнения, а для перемещения к операторам в else необходимо сделать прыжок (безусловный переход выполняется командой jmp, эквивалентной оператору goto языков программирования высокого уровня, или выполняется переход по условию) в то место, где начинаются операторы внутри блока else.Роротские ветвления в большинстве случаев имеют преимущества над длинными. Их разумно выполнять в виде конструкции – с оператором “?:” (x = a>b ? y : z;). В приведенном ниже примере исключается предсказатель ветвления – просто выполняется логическое выражение.

Пример:

```
if (a > 0)
{x = 1, y = 2, z++ }
else {q = 3 z = 4 y = 5}
```

Этот пример можно переписать с помощью одной тернарной операции. Для начала нужно найти общие присваемые части в обоих случаях – это z и y. В z выполняются разные операции: z++ - инкрементируется; z = 4 записывается. А y в обоих случаях просто записывается, что довольно удобно.

`y = a > 0 ? { x = 1, z++, 2}` – непрерывная тернарная операция

В итоге получаем:

```
y = a > 0 ? ( x = 1, z++, 2) : ( q = 3, z = 4, 5);
```

Оператор switch работает быстрее многоуровневых if/else, но все же это условие. Для повышения производительности всегда можно воспользоваться константным массивом.

Пример:

```
int a, b = 0;
/*switch (a) {
case 0: b* = 10;
case 2: b* = 20;
case 3: b* = 30;
…
}*/
const int array[3] = {10,0,20,30};
b *= array[a];
```

Основной (неоптимизированный) способ составления оператора switch – трактовать его как цепочку `if … else if …` операторов. Обычный способ, которым компиляторы оптимизируют переключатель, - преобразование в таблицу переходов, которая может выглядеть примерно так:

```
if (condition1) goto label1;
if (condition2) goto label2;
if (condition3) goto label3;
else            goto default;
label1:
  <<<code from first `case statement`>>>
  goto end;
label2:
  <<<code from first `case statement`>>>
  goto end;
label3:
  <<<code from first `case statement`>>>
  goto end;
default:
  <<<code from `default` case>>>
  goto end;
end:
```

Одна из причин, по которой этот метод работает быстрее, заключается в том, что код внутри условных выражений меньше (поэтому при неправильном прогнозировании условного кэша будет меньше штраф). Кроме того, «провальный» случай становится более тривиальным для реализации (компилятор пропускает goto end оператор).

Компиляторы могут дополнительно оптимизировать таблицу переходов путем создания массива указателей (на места, отмеченные метками) и использовать значение, которое вы включаете, в качестве индекса в этом массиве. Это исключило бы почти все условные выражения из кода (за исключением того, что было необходимо для проверки того, соответствует ли значение, которое вы включаете, одному из ваших случаев или нет).

Важно отметить, что вложенные таблицы переходов трудно генерировать, и некоторые компиляторы отказываются даже пытаться их создать. По этой причине стоит избегать вложения switch внутрь другого, switch если важен максимально оптимизированный код.

Задача: имеется много разных параметров (**param**), от которых зависит решение задачи. Все параметры независимы друг от друга. Каждый из них принимает конечное количество вариантов.

Поставленную задачу можно решить несколькими способами.Первая ситуация складывается при необходимости закодировать все возможные варианты.

1. Каждый имеющийся параметр необходимо закодировать достаточным (но не более того) количеством битов. На рисунке 19 представлен пример подобной ситуации.

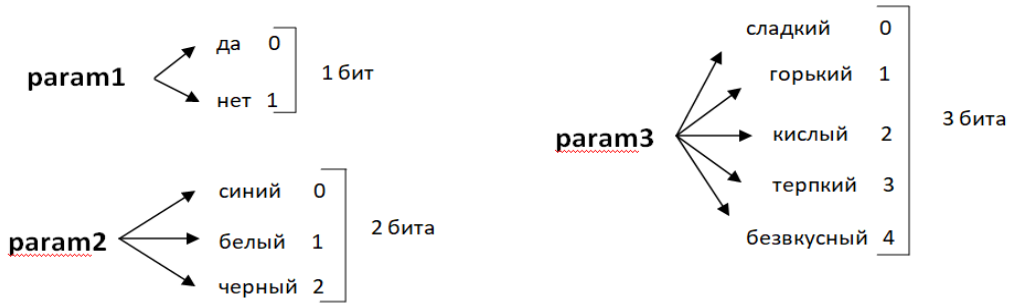
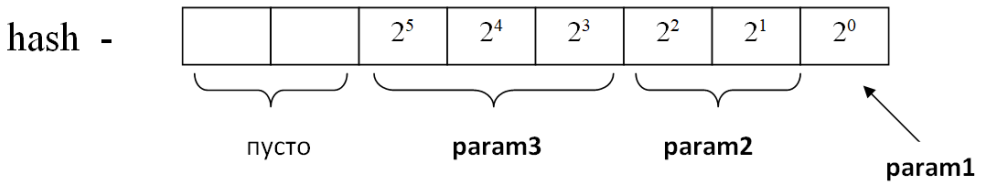


Рисунок 19. Кодирование факторных переменных

2. Из этих коротких целых чисел, каждое из которых содержит маленькое количество битов, при помощи сдвигов собирается одно число (аналог хеш-функции), которое в итоге попадает на выражение switch.



hash = **param1** + 2^1* **param2** + 2^3* **param3**

Рисунок 20. Формирование выражения для оператора switch на основе нескольких факторных переменных (аналог хеширования)

Вторая возможная ситуация – при к*одировании не всех сочетаний, а только возможных.* В этом случае нет возможности или необходимости закодировать все варианты (некоторые из них не возможны или не нужны).

Например, не все варианты param1 можно сопоставить со всеми вариантами параметра param2(рисунок 20). В этом случае хеш-функция образуется следующим образом:

а. логическое выражение

б. a = a0 + (сложное логическое выражение 1)* a1
b = b0 + (сложное логическое выражение 2)* b1
c = c0 + (сложное логическое выражение 3)* c1

Например, если положить a0 = b0 = c0 = 0, а a1 = b1 = c1 = 1, то
hash = a+2*b+4*c

в. закодировать возможные сочетания при помощи индексного массива (index arrays/ Look Up Table).

Пример создания индексного массива представлен ниже. Создадим вспомогательный массив. Исходные ситуации описываются числами, которые идут на вход switch. На рисунке 21 представлена визуализация индексного массива.

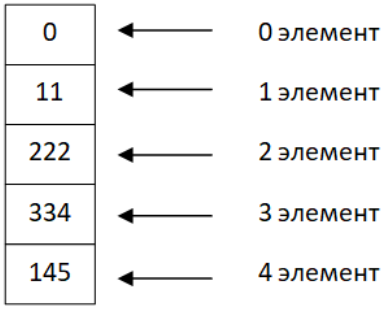


Рисунок.21. Индексный массив

Это необходимо для того, чтобы непрерывно идущий числовой ряд (элемент 0, 1, …,4) преобразовать в необходимые для switch значения, которые могут идти в различном порядке.

Применим эту рекомендацию на практике. Есть определенное правило, по которому осуществляется транслитерация, т.е. на вход поступает русский текст, известно правило, по которому всякой русской букве или буквосочетанию можно поставить в соответствие латинскую букву или буквосочетание. Необходимо преобразовать входной тест в соответствии с заданным правилом. (Например, ‘с’ ~ ‘s’, ‘ш’ ~ ‘sh’).

Нужно максимально оптимизировать транслитерацию.

1. Формируем вспомогательный массив таким образом, чтобы индексы массива получались посредством вычитания из кода буквы кода буквы ‘а’.

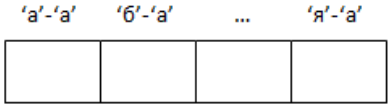


Рисунок 22. Индексы массива для символов алфавита

2. В каждую ячейку массива помещаем указатель на массив, где хранится транслитерация отдельно взятой буквы.
3. Массивы с транслитерацией буквы могут быть разной длины, потому что некоторые буквы транслитерируются несколькими символами. Например, ‘щ’ ~ ‘sch’ (рисунок 23).

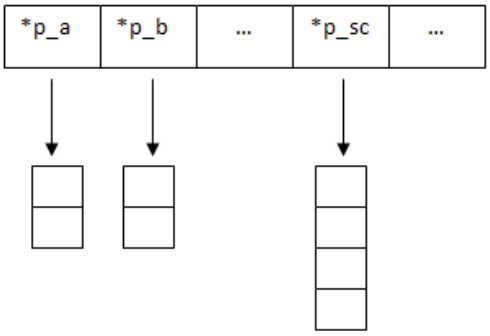


Рисунок 23. Указатели на строки с данными транслитерации по каждому символу

4. Результат изображен на рисунке 24, а в примере ниже представлен фрагмент кода.

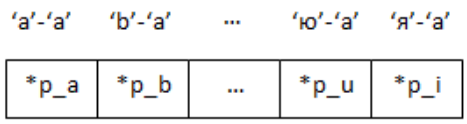


Рисунок 24. Использование индексного массива для транслитерации

Пример применения оператора switch для решения задачи транслитерации:

```
switch (character){
      case 'а'-'а':
      *p_a;
       break;
       ...
       case 'ж'-'а':
       *p_zh;
        break;
        …
}
```

Оптимизация вызова функций

Вызов функций сопряжён со следующими действиями: 1) необходимо передать управление из точки вызова в функцию, 2) после выполнения всех действий необходимо вернуть управление в точку вызова, 3) необходимо передать параметры в функцию. Для передачи параметров чаще всего используют общую область памяти – системный стек. При передаче

параметров их значения копируются в стек, а потом извлекаются оттуда в вызываемой функции. Возврат результата с помощью return осуществляется так же через стек.

Наилучшее решение, оптимизирующее этот процесс, - исключение процесса вызова как такового и работа тела функции прямо в точке вызова. Для этого можно применить два подхода: на основе макросов и подставляемых (inline) функций.

Макрос – это способ обработки и замены программного кода. Эта замена происходит на этапе препроцессирования, то есть на этапе обработки исходных текстов, до компиляции. В макросе нет информации о типе операндов, что может быть как удобным (одна версия для всех типов), так и потенциальным источников ошибок. Вместо вызова функции при использовании макроса происходит подстановка текста макроса прямо в точку вызова. Пример макроса для определения минимума двух чисел представлен ниже:

```
#define min(a, b) (a < b ? a: b)
```

Аналогичным образом работает и подставляемая функция (inline). Отличие состоит в том, что это функция (типы параметров остаются, но можно сделать шаблон функции), и это просьба компилятору сделать подстановку кода (тело функции подставляется в место вызова функции). Компилятор может и не сделать подстановку, если тело функции слишком сложное.

Пример циклического обмена значений переменных местами представлен ниже.

```
inline static void cyc_comp (Q  & a, Q  & b, Q  &c )
{
    Q temp = a; a = b; b = c; c = temp;
};
```

ТРЕБОВАНИЯ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Работы выполняются в среде разработки по выбору студента, например: Visual Studio, CodeBlocks, GCC + Eclipse, Clang.

Каждая практическая работа выполняется в виде отдельного проекта.

Решение каждой задачи или подзадачи оформляется в виде отдельной функции (кроме функции main()). Входные параметры задаются непосредственно в коде или запрашиваются у пользователя с консоли, а затем передаются в эту функцию. Функция возвращает результат с помощью оператора return, или он может быть доступен по указателю. Вывод результата в консоль осуществляется вне этой функции.

Необходимо тщательно протестировать работу решения на разнообразных входных данных.

Код курсовой работы и отчет по курсовой работе сдаются первично в электронном виде. Подробности описаны в разделе «Требования к оформлению курсовой работы».

Желательно задолго до сдачи отчетности по курсовой работе согласовать с преподавателем: 1) оглавление курсовой работы, 2) теоретический обзор литературы по теме работы, 3) структуру кода курсовой работы, 4) выводы об оценке построенного решения по производительности и потреблению памяти.

ПРАКТИЧЕСКИЕ РАБОТЫ

Практическая работа №1. Работа с последовательными контейнерами

Задание 1.1

Постройте связный список (используйте класс list библиотеки STL), который содержит объекты указанного в таблице 1.1 типа T. Постройте функции добавления push() и удаления pop() элементов таким образом, чтобы список оставался отсортированным при выполнении этих операций (допустимо удаление из начала контейнера, с конца и из произвольного места). Постройте функцию filter(), которая принимает предикат P (см. таблицу 1.1) и возвращает новый список с объектами, для которых предикат принимает истинное значение. Постройте функцию вывода содержимого списка с помощью итераторов.
Примечание: *В этом задании не требуется создавать класс списка, нужно использовать класс list из библиотеки STL и написать отдельно требуемые функции (не методы класса).*

Код 1.1. Пример функции добавления в список элемента с сохранением упорядоченности

```
#include <list>
#include <iostream>

using namespace std;


template<class T>
void insert(list<T>& lst, T element)
{
	list<T>::iterator p = lst.begin();
	while (p != lst.end())
	{
		if (*p > element)
			break;
		p++;
	}
	lst.insert(p, element);
}
```

```
int main()
{
	list<char> lst;
	int i=0;
	for(i=0;i<10;i+=2)
		lst.push_back('A' + i);

	insert(lst, 'X');
	list<char>::iterator p = lst.begin();
	while(p!=lst.end())
	{
		//перемещение по контейнеру с помощью указателя, нет
операции [i]
		cout<<*p<<"";
		p++;
	}
	return 0;
}
```

Таблица 1.1. Варианты типов хранимых в контейнере значений и условие предиката в функции фильтрации

Вариант	Тип T	Условие предиката P
1.	char	Только буквы верхнего регистра. Сортировка по коду символа.
2.	double	Только положительные числа
3.	int	Только простые числа
4.	Complex	Только комплексные числа с отрицательной действительной частью. Сортировка по модулю комплексного числа.
5.	Point2D	Только точки, лежащие во втором октанте. Сортировка по расстоянию до центра координат.
6.	Fraction	Только правильные дроби. Сортировка по величине дроби.
7.	char	Только гласные
8.	double	Числа, модули которых больше некоторого значения *a*
9.	int	Только числа, являющиеся факториалами
10.	int	Только квадраты некоторых целых чисел (1, 4, 9, 16 и

		т.д.)
11.	Complex	Только чисто мнимые числа. Сортировка по модулю комплексного числа.
12.	Fraction	Только дроби с числителями, представляющими простые числа.
13.	Point2D	Только точки, лежащие за пределами единичного круга.
14.	int	Только элементы последовательности Фибоначчи
15.	Fraction	Дроби, по модулю превосходящие некоторое значения *a*
16.	char	Только согласные.
17.	double	Числа, модули которых меньше некоторого значения *a*
18.	int	Только числа, кратные 3
19.	Complex	Только комплексные числа с четной действительной частью. Сортировка по модулю комплексного числа.
20.	Point2D	Только точки, лежащие внутри единичного квадрата с центров в начале координат. Сортировка по расстоянию до центра координат.
21.	Fraction	Только дроби, у которых числитель квадрат некоторого числа. Сортировка по величине дроби.
22.	char	Только буквы нижнего регистра.
23.	double	Числа, дробная часть которых не превосходит *a*
24.	int	Только числа, являющиеся факториалами четных значений
25.	int	Только кубы некоторых целых чисел (1, 8, 27, 64 и т.д.)
26.	Complex	Только комплексные числа с нечетной действительной и мнимой частью. Сортировка по модулю комплексного числа.
27.	Fraction	Только дроби с числителями, представляющими простые числа.
28.	Point2D	Только точки, лежащие внутри единичного круга.
29.	int	Только числа, кратные 7, отрицательные
30.	Fraction	Дроби, по модулю, не превосходящие некоторое значения *a*

Задание 1.2

Заполните список из пункта 1 объектами класса C (таблица 1.2), сохраняя убывание по приоритету: полю или группе полей, указанных в варианте. Функция pop() должна удалять объект из контейнера и возвращать

как результат объект с наибольшим приоритетом (определяется по полям, указанным в третьем столбце таблицы 1.2: больший приоритет имеет объект с большим значением первого поля; если значение первого поля совпадает, то сравниваются значения второго поля и так далее). Если больший приоритет имеют объекты с меньшим значением поля (упорядоченность по возрастанию), это указано в скобках.

Пример из варианта 1: *объекты недвижимости сортируются по убыванию цены. Если цена совпадает, то сравниваем по адресу, но для адреса уже используется упорядочение по возрастанию (“меньший” адрес - больший приоритет, строки сравниваются в лексикографическом порядке, “как в словаре”).*

Таблица 1.2. Варианты типов хранимых в контейнере значений и порядок сравнения полей для упорядочения объектов

Вариант	Класс С	Приоритет
1.	«Объект жилой недвижимости». Минимальный набор полей: адрес, тип (перечислимый тип: городской дом, загородный дом, квартира, дача), общая площадь, жилая площадь, цена.	Цена; адрес (по возрастанию)
2.	«Сериал». Минимальный набор полей: название, продюсер, количество сезонов, популярность, рейтинг, дата запуска, страна.	Рейтинг; название (по возрастанию)
3.	«Смартфон». Минимальный набор полей: название, размер экрана, количество камер, объем аккумулятора, максимальное количество часов без подзарядки, цена.	Цена, количество камер, размер экрана; название марки (по возрастанию)
4.	«Спортсмен». Минимальный набор полей: фамилия, имя, возраст, гражданство, вид спорта, количество медалей.	Количество медалей; возраст (по возрастанию); фамилия и имя (по возрастанию)
5.	«Врач». Минимальный набор полей: фамилия, имя,	Рейтинг, стаж; фамилия и имя (по возрас-

	специальность, должность, стаж, рейтинг (вещественное число от 0 до 100).	танию)
6.	«Авиакомпания». Минимальный набор полей: название, международный код, количество обслуживаемых линий, страна, интернет-адрес сайта, рейтинг надёжности (целое число от -10 до 10).	Надёжность, количество обслуживаемых линий; название (по возрастанию)
7.	«Книга». Минимальный набор полей: фамилия (первого) автора, имя (первого) автора, название, год издания, название издательства, число страниц, вид издания (перечислимый тип: электронное, бумажное или аудио), тираж.	Тираж; год издания (по возрастанию); название (по возрастанию)
8.	«Небесное тело». Минимальный набор полей: тип (перечислимый тип: астероид, естественный спутник, планета, звезда, квазар), имя (может отсутствовать), номер в небесном каталоге, удаление от Земли, расчётная масса в миллиардах тонн (для сверхбольших объектов допускается значение Inf, которое должно корректно обрабатываться).	Масса; номер в каталоге (по возрастанию)
9.	«Населённый пункт». Минимальный набор полей: название, тип (перечислимый тип: город, посёлок, село, деревня), числовой код региона, численность населения, площадь.	Площадь, численность населения; числовой код региона (по возрастанию)
10.	«Музыкальный альбом». Минимальный набор полей: имя или псевдоним исполнителя, название альбома, количество композиций, год выпуска, количество проданных экземпляров.	Количество проданных экземпляров; количество композиций; год выпуска (по возрастанию); имя или псевдоним исполнителя (по возрастанию)
11.	«Фильм».	Доход, стоимость; год

	Минимальный набор полей: фамилия, имя режиссёра, название, страна, год выпуска, стоимость, доход.	выпуска (по возрастанию); фамилия и имя режиссера (по возрастанию); название фильма (по возрастанию)
12.	«Автомобиль». Минимальный набор полей: имя модели, цвет, серийный номер, количество дверей, год выпуска, цена.	Цена; год выпуска; марка (по возрастанию); серийный номер (по возрастанию)
13.	«Автовладелец». Минимальный набор полей: фамилия, имя, регистрационный номер автомобиля, дата рождения, номер техпаспорта.	Регистрационный номер автомобиля; номер техпаспорта; фамилия и имя автовладельца (по возрастанию)
14.	«Стадион». Минимальный набор полей: название, виды спорта, год постройки, вместимость, количество арен.	Вместимость, количество арен, год постройки; название (по возрастанию)
15.	«Спортивная Команда». Минимальный набор полей: название, город, число побед, поражений, ничьих, количество очков.	Число побед, число ничьих; число поражений (по возрастанию); название (по возрастанию)
16.	«Пациент». Минимальный набор полей: фамилия, имя, дата рождения, телефон, адрес, номер карты, группа крови.	Номер карты; группа крови; фамилия и имя (по возрастанию)
17.	«Покупатель». Минимальный набор полей: фамилия, имя, город, улица, номера дома и квартиры, номер счёта, средняя сумма чека.	Средняя сумма чека; номер счёта; фамилия и имя (по возрастанию)
18.	«Школьник». Минимальный набор полей: фамилия, имя,	Класс; дата рождения (по возрастанию); фа-

	пол, класс, дата рождения, адрес.	милия и имя (по возрастанию)
19.	«Человек». Минимальный набор полей: фамилия, имя, пол, рост, возраст, вес, дата рождения, телефон, адрес.	Возраст, рост; вес (по возрастанию); фамилия и имя (по возрастанию)
20.	«Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь.	Численность населения; площадь; название (по возрастанию)
21.	«Сайт». Минимальный набор полей: название, адрес, дата запуска, язык, тип (блог, интернет-магазин и т.п.), cms, дата последнего обновления, количество посетителей в сутки.	Количество посетителей в сутки, дата последнего обновления; адрес (по возрастанию)
22.	«Программа». Минимальный набор полей: название, версия, лицензия, есть ли версия для android, iOS, платная ли, стоимость, разработчик, открытость кода, язык кода.	Стоимость, версия; название (по возрастанию)
23.	«Ноутбук». Минимальный набор полей: производитель, модель, размер экрана, процессор, количество ядер, объем оперативной памяти, объем диска, тип диска, цена.	Цена, количество ядер, объем оперативной памяти, размер экрана; модель (по возрастанию)
24.	«Велосипед». Минимальный набор полей: марка, тип, тип тормозов, количество колес, диаметр колеса, наличие амортизаторов, детский или взрослый.	Диаметр колеса, количество колес; марка (по возрастанию)
25.	«Программист». Минимальный набор полей: фамилия, имя, email, skype, telegram, основной язык программирования, текущее место работы, уровень (число от 1 до 10).	Уровень; основной язык программирования (по возрастанию); фамилия и имя (по возрастанию)
26.	«Профиль в соц.сети».	Количество друзей;

	Минимальный набор полей: псевдоним, адрес страницы, возраст, количество друзей, интересы, любимая цитата.	псевдоним (по возрастанию)
27.	«Супергерой». Минимальный набор полей: псевдоним, настоящее имя, дата рождения, пол, суперсила, слабости, количество побед, рейтинг силы.	Рейтинг силы, количество побед; псевдоним (по возрастанию)
28.	«Фотоаппарат». Минимальный набор полей: производитель, модель, тип, размер матрицы, количество мегапикселей, вес, тип карты памяти, цена.	Цена, вес, размер матрицы; модель (по возрастанию)
29.	«Файл». Минимальный набор полей: полный адрес, краткое имя, дата последнего изменения, дата последнего чтения, дата создания.	Дата последнего изменения, дата последнего чтения; полный адрес (по возрастанию)
30.	«Самолет». Минимальный набор полей: название, производитель, вместимость, дальность полета, максимальная скорость.	Вместимость, дальность полета; производитель (по возрастанию), название (по возрастанию)

Задание 1.3

Постройте шаблон класса двусвязного списка путём наследования от класса IteratedLinkedList. Реализуйте функции добавления элемента push() и удаления элемента pop() в классе-наследнике D (для четных вариантов D – Стек, для нечетных – Очередь) согласно схеме: для класса Стек элементы добавляются в конец, извлекаются с конца; для класса Очередь элементы добавляются в конец, извлекаются с начала. Постройте наследник класса D. Переопределите функцию добавления нового элемента таким образом, чтобы контейнер оставался упорядоченным. Реализуйте функцию filter() из пункта 1.

Код 1.3. Абстрактный класс для связного списка LinkedListParent (функции push() и pop() чисто виртуальные) и IteratedLinkedList (введен механизм работы итераторов) и другие вспомогательные классы

```
#include <iostream>
#include <fstream>

using namespace std;

template <class T>
class Element
{
	//элемент связного списка
private:
	//указатель на предыдущий и следующий элемент
	Element* next;
	Element* prev;

	//информация, хранимая в поле
	T field;
public:
	Element(T value = 0, Element<T> * next_ptr = NULL, Ele-
ment<T> * prev_ptr = NULL)
	{
		field = value;
		next = next_ptr;
		prev - prev_ptr;
	}
	//доступ к полю *next
	virtual Element* getNext() { return next; }
	virtual void setNext(Element* value) { next = value; }

	//доступ к полю *prev
	virtual Element* getPrevious() { return prev; }
	virtual void setPrevious(Element* value) { prev = value; }

	//доступ к полю с хранимой информацией field
	virtual T getValue() { return field; }
	virtual void setValue(T value) { field = value; }

	template<class T> friend ostream& operator<< (ostream&
ustream, Element<T>& obj);
};

template<class T>
```

```
ostream& operator << (ostream& ustream, Element<T>& obj)
{
	ustream << obj.field;
	return ustream;
}

template <class T>
class LinkedListParent
{
protected:
	//достаточно хранить начало и конец
	Element<T>* head;
	Element<T>* tail;
	//для удобства храним количество элементов
	int num;
public:
	virtual int Number() { return num; }

	virtual Element<T>* getBegin() { return head; }

	virtual Element<T>* getEnd() { return tail; }

	LinkedListParent()
	{
		//конструктор без параметров
		cout << "\nParent constructor";
		head = NULL;
		num = 0;
	}

	//чисто виртуальная функция: пока не определимся с типом
списка, не сможем реализовать добавление
	virtual Element<T>* push(T value) = 0;

	//чисто виртуальная функция: пока не определимся с типом
списка, не сможем реализовать удаление
	virtual Element<T>* pop() = 0;

	virtual ~LinkedListParent()
	{
		//деструктор - освобождение памяти
		cout << "\nParent destructor";
	}
```

```
	//получение элемента по индексу - какова асимптотическая
оценка этого действия?
	virtual Element<T>* operator[](int i)
	{
		//индексация
		if (i<0 || i>num) return NULL;
		int k = 0;

		//ищем i-й элемент - вставем в начало и отсчитываем i
шагов вперед
		Element<T>* cur = head;
		for (k = 0; k < i; k++)
		{
			cur = cur->getNext();
		}
		return cur;
	}

	template<class T> friend ostream& operator<< (ostream&
ustream, LinkedListParent<T>& obj);
	template<class T> friend istream& operator>> (istream&
ustream, LinkedListParent<T>& obj);
};

template<class T>
ostream& operator << (ostream& ustream, LinkedListParent<T>&
obj)
{
	if (typeid(ustream).name() == typeid(ofstream).name())
	{
		ustream << obj.num << "\n";
		for (Element<T>* current = obj.getBegin(); current !=
NULL; current = current->getNext())
			ustream << current->getValue() << " ";
		return ustream;
	}

	ustream << "\nLength: " << obj.num << "\n";
	int i = 0;
	for (Element<T>* current = obj.getBegin(); current != NULL;
current = current->getNext(), i++)
		ustream << "arr[" << i << "] = " << current->getValue()
<< "\n";
```

```
	return ustream;
}

template<class T>
istream& operator >> (istream& ustream, LinkedListParent<T>&
obj)
{
	//чтение из файла и консоли совпадают
	int len;
	ustream >> len;
	//здесь надо очистить память под obj, установить obj.num = 0
	double v = 0;
	for (int i = 0; i < len; i++)
	{
		ustream >> v;
		obj.push(v);
	}
	return ustream;
}

template<typename ValueType>
class ListIterator : public
std::iterator<std::input_iterator_tag, ValueType>
{
private:

public:
	ListIterator() { ptr = NULL; }
	//ListIterator(ValueType* p) { ptr = p; }
	ListIterator(Element<ValueType>* p) { ptr = p; }
	ListIterator(const ListIterator& it) { ptr = it.ptr; }

	bool operator!=(ListIterator const& other) const { return
ptr != other.ptr; }
	bool operator==(ListIterator const& other) const { return
ptr == other.ptr; }//need for BOOST_FOREACH
	Element<ValueType>& operator*()
	{
		return *ptr;
	}
	ListIterator& operator++() { ptr = ptr->getNext(); return
*this; }
	ListIterator& operator++(int v) { ptr = ptr->getNext(); re-
turn *this; }
```

```
    ListIterator& operator=(const ListIterator& it) { ptr =
it.ptr; return *this; }
    ListIterator& operator=(Element<ValueType>* p) { ptr = p;
return *this; }
private:
    Element<ValueType>* ptr;
};

template <class T>
class IteratedLinkedList : public LinkedListParent<T>
{
public:
    IteratedLinkedList() : LinkedListParent<T>() { cout <<
"\nIteratedLinkedList constructor"; }
    virtual ~IteratedLinkedList() { cout <<
"\nIteratedLinkedList destructor"; }

    ListIterator<T> iterator;

    ListIterator<T> begin() { ListIterator<T> it =
LinkedListParent<T>::head; return it; }
    ListIterator<T> end() { ListIterator<T> it =
LinkedListParent<T>::tail; return it; }
};
```

Задание 1.4

Постройте итераторы для перемещения по списку. Переопределите функцию вывода содержимого списка с помощью итераторов. Итераторы двунаправленные.

Задание 1.5

Постройте шаблон класса списка D (из задания в пункте 3), который хранит объекты класса C (из задания в пункте 2), сохраняя упорядоченность по приоритету: полю или группе полей, указанных в варианте.

Практическая работа №2. Работа с ассоциативными контейнерами

Задание 2.1

Постройте сбалансированное дерево (используйте класс map библиотеки STL), которое содержит значения V по ключам K (таблица 2.1). Постройте функции поиска элемента по значению и по ключу. Постройте функцию вывода содержимого дерева с помощью итераторов. Постройте функцию filter(), которая принимает предикат P и возвращает новое дерево с объектами, для которых предикат принимает истинное значение (для всех вариантов условие предиката: значение поля V выше некоторого порога *threshold*, в случае хранения нескольких полей достаточно проверить одно из них).

Примечание: *В этом задании не требуется создавать класс дерева, нужно использовать класс map из библиотеки STL и написать отдельно требуемые функции (не методы класса).*

Код 2.1. Пример работы с контейнером map

```
//красно-черное (сблансированное) дерево map, есть интерфейс доступа к
значению по ключу

using namespace std;

#include <map>
#include <iostream>

int main()
{

   map<string, int> marks;
   marks["Petrov"] = 5;
   marks["Ivanov"] = 4;
   marks["Sidorov"] = 5;
   marks["Nikolaev"] = 3;
   marks["Abramov"] = 4;
   marks["Fedorov"] = 5;
   marks["Kuznetsov"] = 4;

   cout << "\nMap:\n";
   //итератор пробегает по map
   map<string, int>::iterator it_m = marks.begin();
   while (it_m != marks.end())
   {
```

```
            //перемещение по списку с помощью итератора, нет операции [i]
            cout << "Key: " << it_m->first << ", value: " << it_m->second <<
"\n";
            it_m++;
    }
}
```

Таблица 2.1. Ключи и хранимая в ассоциативном контейнере map информация

Вариант	Ключ K	Хранимая информация
1.	Адрес	«Объект жилой недвижимости». V: цена квартиры
2.	Название	«Сериал». V: рейтинг
3.	Название	«Смартфон». V: цена
4.	Фамилия и имя	«Спортсмен». V: количество медалей.
5.	Фамилия и имя	«Врач». V: рейтинг (вещественное число от 0 до 100) количество медалей
6.	Международный код	«Авиакомпания». V: количество обслуживаемых линий
7.	Название	«Книга». V: тираж
8.	Номер в небесном каталоге	«Небесное тело». V: расчётная масса в миллиардах тонн
9.	Название	«Населённый пункт». V: численность населения
10.	Имя или псевдоним исполнителя, название альбома	«Музыкальный альбом». V: количество проданных экземпляров
11.	Название фильма	«Фильм». V: доход

12.	Название производителя, имя модели	«Автомобиль». V: цена
13.	Регистрационный номер автомобиля	«Автовладелец». V: фамилия, имя
14.	Название, год постройки	«Стадион». V: вместимость
15.	Название, город	«Спортивная Команда». V: число побед, поражений, ничьих, количество очков
16.	Номер карты	«Пациент». V: группа крови
17.	Фамилия и имя	«Покупатель». V: средняя сумма чека
18.	Фамилия и имя	«Школьник». V: дата рождения
19.	Фамилия и имя	«Человек». V: адрес
20.	Название	«Государство». V: численность населения
21.	Адрес	«Сайт». V: количество посетителей в сутки.
22.	Название	«Программа». V: разработчик
23.	Производитель, модель	«Ноутбук». V: размер экрана, количество ядер, объем оперативной памяти
24.	Марка, диаметр колеса	«Велосипед». V: тип, наличие амортизаторов
25.	Фамилия и имя	«Программист». V: уровень (число от 1 до 10)

26.	Псевдоним	«Профиль в соц.сети». V: количество друзей
27.	Псевдоним	«Супергерой». V: суперсила
28.	Производитель, модель	«Фотоаппарат». V: размер матрицы, количество мегапикселей
29.	Полный адрес	«Файл». V: дата последнего изменения
30.	Производитель, название	«Самолет». V: дальность полета, максимальная скорость

Задание 2.2

Постройте очередь с приоритетами на основе адаптера priority_queue. Типы ключей и значений соответствуют пункту 2 задания №1. Выведите элементы очереди в порядке убывания приоритета.

Код 2.2. Пример работы с адаптером “очередь с приоритетом”

```
using namespace std;

#include <iostream>
#include <queue>

template<typename T>
void print_queue(T& q) {
	while (!q.empty()) {
		cout << q.top() << " ";
		q.pop();
	}
	std::cout << '\n';
}


int main() {
	priority_queue<int> q;

	for (int n : {1, 8, 5, 6, 3, 4, 0, 9, 7, 2})
		q.push(n);

	print_queue(q);

   }
```

Задание 2.3

Постройте шаблон сбалансированного дерева. Используйте его для хранения объектов класса C по ключам K в соответствии с таблицей (2.3). Переопределите функцию вывода содержимого дерева с помощью итераторов (в порядке возрастания / убывания ключей). Добавьте функции поиска элемента по ключу, значению.

Код 2.3. Класс бинарного дерева поиска

```
#include <iostream>

using namespace std;
//узел
template<class T>
class Node
{
protected:
    //закрытые переменные Node N; N.data = 10 вызовет ошибку
    T data;

    //не можем хранить Node, но имеем право хранить указатель
    Node* left;
    Node* right;
    Node* parent;

    //переменная, необходимая для поддержания баланса дерева
    int height;
public:
    //доступные извне переменные и функции
    virtual void setData(T d) { data = d; }
    virtual T getData() { return data; }
    int getHeight() { return height; }

    virtual Node* getLeft() { return left; }
    virtual Node* getRight() { return right; }
    virtual Node* getParent() { return parent; }

    virtual void setLeft(Node* N) { left = N; }
    virtual void setRight(Node* N) { right = N; }
    virtual void setParent(Node* N) { parent = N; }

    //Конструктор. Устанавливаем стартовые значения для указателей
    Node<T>(T n)
    {
           data = n;
           left = right = parent = NULL;
           height = 1;
    }

    Node<T>()
```

```
    {
        left = NULL;
        right = NULL;
        parent = NULL;
        data = 0;
        height = 1;
    }


    virtual void print()
    {
        cout << "\n" << data;
    }

    virtual void setHeight(int h)
    {
        height = h;
    }

    template<class T> friend ostream& operator<< (ostream& stream, Node<T>& N);
};

template<class T>
ostream& operator<< (ostream& stream, Node<T>& N)
{
    stream << "\nNode data: " << N.data << ", height: " << N.height;
    return stream;
}
template<class T>
void print(Node<T>* N) { cout << "\n" << N->getData(); }

template<class T>
class Tree
{
protected:
    //корень - его достаточно для хранения всего дерева
    Node<T>* root;
public:
    //доступ к корневому элементу
    virtual Node<T>* getRoot() { return root; }

    //конструктор дерева: в момент создания дерева ни одного узла нет, корень смотрит
в никуда
    Tree<T>() { root = NULL; }

    //рекуррентная функция добавления узла. Устроена аналогично, но вызывает сама себя
- добавление в левое или правое поддерево
    virtual Node<T>* Add_R(Node<T>* N)
    {
        return Add_R(N, root);
    }

    virtual Node<T>* Add_R(Node<T>* N, Node<T>* Current)
    {
```

```
            if (N == NULL) return NULL;
            if (root == NULL)
            {
                  root = N;
                  return N;
            }

            if (Current->getData() > N->getData())
            {
                  //идем влево
                  if (Current->getLeft() != NULL)
                        Current->setLeft(Add_R(N, Current->getLeft()));
                  else
                        Current->setLeft(N);
                  Current->getLeft()->setParent(Current);
            }
            if (Current->getData() < N->getData())
            {
                  //идем вправо
                  if (Current->getRight() != NULL)
                        Current->setRight(Add_R(N, Current->getRight()));
                  else
                        Current->setRight(N);
                  Current->getRight()->setParent(Current);
            }
            if (Current->getData() == N->getData())
                  //нашли совпадение
                  ;
            //для несбалансированного дерева поиска
            return Current;
      }

      //функция для добавления числа. Делаем новый узел с этими данными и вызываем нуж-
ную функцию добавления в дерево
      virtual void Add(int n)
      {
            Node<T>* N = new Node<T>;
            N->setData(n);
            Add_R(N);
      }

      virtual Node<T>* Min(Node<T>* Current=NULL)
      {
            //минимум - это самый "левый" узел. Идём по дереву всегда влево
            if (root == NULL) return NULL;

            if(Current==NULL)
                  Current = root;
            while (Current->getLeft() != NULL)
                  Current = Current->getLeft();

            return Current;
      }
```

```
	virtual Node<T>* Max(Node<T>* Current = NULL)
	{
		//минимум - это самый "правый" узел. Идём по дереву всегда вправо
		if (root == NULL) return NULL;

		if (Current == NULL)
			Current = root;
		while (Current->getRight() != NULL)
			Current = Current->getRight();

		return Current;
	}

	//поиск узла в дереве. Второй параметр - в каком поддереве искать, первый - что
искать
	virtual Node<T>* Find(int data, Node<T>* Current)
	{
		//база рекурсии
		if (Current == NULL) return NULL;

		if (Current->getData() == data) return Current;

		//рекурсивный вызов
		if (Current->getData() > data) return Find(data, Current->getLeft());

		if (Current->getData() < data) return Find(data, Current->getRight());


	}

	//три обхода дерева
	virtual void PreOrder(Node<T>* N, void (*f)(Node<T>*))
	{
		if (N != NULL)
			f(N);
		if (N != NULL && N->getLeft() != NULL)
			PreOrder(N->getLeft(), f);
		if (N != NULL && N->getRight() != NULL)
			PreOrder(N->getRight(), f);
	}

	//InOrder-обход даст отсортированную последовательность
	virtual void InOrder(Node<T>* N, void (*f)(Node<T>*))
	{
		if (N != NULL && N->getLeft() != NULL)
			InOrder(N->getLeft(), f);
		if (N != NULL)
			f(N);
		if (N != NULL && N->getRight() != NULL)
			InOrder(N->getRight(), f);
	}

	virtual void PostOrder(Node<T>* N, void (*f)(Node<T>*))
	{
```

```
            if (N != NULL && N->getLeft() != NULL)
                PostOrder(N->getLeft(), f);
            if (N != NULL && N->getRight() != NULL)
                PostOrder(N->getRight(), f);
            if (N != NULL)
                f(N);
        }
    };

int main()
{
        Tree<double> T;
        int arr[15];
        int i = 0;
        for (i = 0; i < 15; i++) arr[i] = (int)(100 * cos(15 * double(i+1)));
        for (i = 0; i < 15; i++)
            T.Add(arr[i]);

        Node<double>* M = T.Min();
        cout << "\nMin = " << M->getData() << "\tFind " << arr[3] << ": " <<
T.Find(arr[3], T.getRoot());

        void (*f_ptr)(Node<double>*); f_ptr = print;
        cout << "\n-----\nInorder:";
        T.InOrder(T.getRoot(), f_ptr);
        char c; cin >> c;
        return 0;
}
```

Таблица 2.3. Ключ и тип объекта, хранимого в контейнере АВЛ-дерево

Вари-ант	Ключ	Класс С
1.	Адрес	«Объект жилой недвижимости». Минимальный набор полей: адрес, тип (перечислимый тип: городской дом, загородный дом, квартира, дача), общая площадь, жилая площадь, цена.
2.	Название	«Сериал». Минимальный набор полей: название, продюсер, количество сезонов, популярность, рейтинг, дата запуска, страна.
3.	Название	«Смартфон». Минимальный набор полей: название, размер экрана, количество камер, объем аккумулятора, максимальное количество часов без подзарядки, цена.

4.	Фамилия и имя	«Спортсмен». Минимальный набор полей: фамилия, имя, возраст, гражданство, вид спорта, количество медалей.
5.	Фамилия и имя	«Врач». Минимальный набор полей: фамилия, имя, специальность, должность, стаж, рейтинг (вещественное число от 0 до 100).
6.	Международный код	«Авиакомпания». Минимальный набор полей: название, международный код, количество обслуживаемых линий, страна, интернет-адрес сайта, рейтинг надёжности (целое число от -10 до 10).
7.	Название	«Книга». Минимальный набор полей: фамилия (первого) автора, имя (первого) автора, название, год издания, название издательства, число страниц, вид издания (перечислимый тип: электронное, бумажное или аудио), тираж.
8.	Номер в каталоге	«Небесное тело». Минимальный набор полей: тип (перечислимый тип: астероид, естественный спутник, планета, звезда, квазар), имя (может отсутствовать), номер в небесном каталоге, удаление от Земли, расчётная масса в миллиардах тонн (для сверхбольших объектов допускается значение Inf, которое должно корректно обрабатываться).
9.	Название	«Населённый пункт». Минимальный набор полей: название, тип (перечислимый тип: город, посёлок, село, деревня), числовой код региона, численность населения, площадь.
10.	Имя или псевдоним исполните-	«Музыкальный альбом». Минимальный набор полей: имя или псевдоним исполнителя, название альбома, количество композиций, год выпуска, количество проданных экземпляров.

	ля, название альбома	
11.	Название фильма	«Фильм». Минимальный набор полей: фамилия, имя режиссёра, название, страна, год выпуска, стоимость, доход.
12.	Серийный номер	«Автомобиль». Минимальный набор полей: имя модели, название производителя, цвет, серийный номер, количество дверей, год выпуска, цена.
13.	Регистрационный номер автомобиля	«Автовладелец». Минимальный набор полей: фамилия, имя, регистрационный номер автомобиля, дата рождения, номер техпаспорта.
14.	Название, год постройки	«Стадион». Минимальный набор полей: название, виды спорта, год постройки, вместимость, количество арен.
15.	Название, город	«Спортивная Команда». Минимальный набор полей: название, город, число побед, поражений, ничьих, количество очков.
16.	Фамилия и имя	«Пациент». Минимальный набор полей: фамилия, имя, дата рождения, телефон, адрес, номер карты, группа крови.
17.	Фамилия и имя	«Покупатель». Минимальный набор полей: фамилия, имя, город, улица, номера дома и квартиры, номер счёта, средняя сумма чека.
18.	Фамилия и имя	«Школьник». Минимальный набор полей: фамилия, имя, пол, класс,

		дата рождения, адрес.
19.	Фамилия и имя	«Человек». Минимальный набор полей: фамилия, имя, пол, рост, возраст, вес, дата рождения, телефон, адрес.
20.	Название	«Государство». Минимальный набор полей: название, столица, язык, численность населения, площадь.
21.	Адрес	«Сайт». Минимальный набор полей: название, адрес, дата запуска, язык, тип (блог, интернет-магазин и т.п.), cms, дата последнего обновления, количество посетителей в сутки.
22.	Название	«Программа». Минимальный набор полей: название, версия, лицензия, есть ли версия для android, iOS, платная ли, стоимость, разработчик, открытость кода, язык кода.
23.	Производитель, модель	«Ноутбук». Минимальный набор полей: производитель, модель, размер экрана, процессор, количество ядер, объем оперативной памяти, объем диска, тип диска, цена.
24.	Марка, диаметр колеса	«Велосипед». Минимальный набор полей: марка, тип, тип тормозов, количество колес, диаметр колеса, наличие амортизаторов, детский или взрослый.
25.	Фамилия и имя	«Программист». Минимальный набор полей: фамилия, имя, email, skype, telegram, основной язык программирования, текущее место работы, уровень (число от 1 до 10).
26.	Псевдоним	«Профиль в соц.сети». Минимальный набор полей: псевдоним, адрес страницы, возраст, количество друзей, интересы, любимая цитата.
27.	Псевдоним	«Супергерой». Минимальный набор полей: псевдоним, настоящее имя,

		дата рождения, пол, суперсила, слабости, количество побед, рейтинг силы.
28.	Производитель, модель	«Фотоаппарат». Минимальный набор полей: производитель, модель, тип, размер матрицы, количество мегапикселей, вес, тип карты памяти, цена.
29.	Полный адрес	«Файл». Минимальный набор полей: полный адрес, краткое имя, дата последнего изменения, дата последнего чтения, дата создания.
30.	Производитель, название	«Самолет». Минимальный набор полей: название, производитель, вместимость, дальность полета, максимальная скорость.

Задание 2.4

Используйте шаблон класса Heap (куча, пирамида) для хранения объектов в соответствии с пунктом 2 задания №1 (используется упорядоченность по приоритету, в корне дерева – максимум). Реализуйте функцию удаления корня дерева ExtractMax(). Выведите элементы Heap в порядке убывания приоритета с её помощью.

Код 2.4. Класс Heap (куча, пирамида)

```
#include <iostream>

using namespace std;

//узел дерева
template <class T>
class Node
{
private:
	T value;
public:
	//установить данные в узле
	T getValue() { return value; }
	void setValue(T v) { value = v; }

	//сравнение узлов
	int operator<(Node N)
	{
		return (value < N.getValue());
	}
```

```
	int operator>(Node N)
	{
		return (value > N.getValue());
	}

	//вывод содержимого одного узла
	void print()
	{
		cout << value;
	}
};

template <class T>
void print(Node<T>* N)
{
	cout << N->getValue() << "\n";
}

//куча (heap)
template <class T>
class Heap
{
private:
	//массив
	Node<T>* arr;
	//сколько элементов добавлено
	int len;
	//сколько памяти выделено
	int size;
public:

	//доступ к вспомогательным полям кучи и оператор индекса
	int getCapacity() { return size; }
	int getCount() { return len; }

	Node<T>& operator[](int index)
	{
		if (index < 0 || index >= len)
			;//?

		return arr[index];
	}

	//конструктор
	Heap<T> (int MemorySize = 100)
	{
		arr = new Node<T>[MemorySize];
		len = 0;
		size = MemorySize;
	}

	//поменять местами элементы arr[index1], arr[index2]
	void Swap(int index1, int index2)
	{
```

```
		if (index1 <= 0 || index1 >= len)
			;
		if (index2 <= 0 || index2 >= len)
			;
		//здесь нужна защита от дурака

		Node<T> temp;
		temp = arr[index1];
		arr[index1] = arr[index2];
		arr[index2] = temp;
	}

	//скопировать данные между двумя узлами
	void Copy(Node<T>* dest, Node<T>* source)
	{
		dest->setValue(source->getValue());
	}

	//функции получения левого, правого дочернего элемента, родителя или их индексов в
массиве
	Node<T>* GetLeftChild(int index)
	{
		if (index < 0 || index * 2 >= len)
			;
		//здесь нужна защита от дурака
		return &arr[index * 2 + 1];
	}

	Node<T>* GetRightChild(int index)
	{
		if (index < 0 || index * 2 >= len)
			;
		//здесь нужна защита от дурака

		return &arr[index * 2 + 2];
	}

	Node<T>* GetParent(int index)
	{
		if (index <= 0 || index >= len)
			;
		//здесь нужна защита от дурака

		if (index % 2 == 0)
			return &arr[index / 2 - 1];
		return &arr[index / 2];
	}

	int GetLeftChildIndex(int index)
	{
		if (index < 0 || index * 2 >= len)
			;
		//здесь нужна защита от дурака
		return index * 2 + 1;
```

```
	}

	int GetRightChildIndex(int index)
	{
		if (index < 0 || index * 2 >= len)
			;
		//здесь нужна защита от дурака

		return index * 2 + 2;
	}

	int GetParentIndex(int index)
	{
		if (index <= 0 || index >= len)
			;
		//здесь нужна защита от дурака

		if (index % 2 == 0)
			return index / 2 - 1;
		return index / 2;
	}

	//просеить элемент вверх
	void SiftUp(int index = -1)
	{
		if (index == -1) index = len - 1;
		int parent = GetParentIndex(index);
		int index2 = GetLeftChildIndex(parent);
		if (index2 == index) index2 = GetRightChildIndex(parent);
		int max_index = index;

		if (index < len && index2 < len && parent >= 0)
		{
			if (arr[index] > arr[index2])
				max_index = index;
			if (arr[index] < arr[index2])
				max_index = index2;
		}
		if (parent < len && parent >= 0 && arr[max_index]>arr[parent])
		{
			//нужно просеивание вверх
			Swap(parent, max_index);
			SiftUp(parent);
		}
	}
	//добавление элемента - вставляем его в конец массива и просеиваем вверх
	template <class T>
	void Add(T v)
	{
		Node<T>* N = new Node<T>;
		N->setValue(v);
		Add(N);
	}
```

```
	template <class T>
	void Add(Node<T>* N)
	{
		if (len < size)
		{
			Copy(&arr[len], N);
			len++;
			SiftUp();
		}
	}

	//перечислить элементы кучи и применить к ним функцию
	void Straight(void(*f)(Node<T>*))
	{
		int i;
		for (i = 0; i < len; i++)
		{
			f(&arr[i]);
		}
	}

	//перебор элементов, аналогичный проходам бинарного дерева
	void InOrder(void(*f)(Node<T>*), int index = 0)
	{
		if (GetLeftChildIndex(index) < len)
			PreOrder(f, GetLeftChildIndex(index));
		if (index >= 0 && index < len)
			f(&arr[index]);
		if (GetRightChildIndex(index) < len)
			PreOrder(f, GetRightChildIndex(index));
	}
};

int main()
{
	Heap<int> Tree;

	Tree.Add(1);
	Tree.Add(-1);
	Tree.Add(-2);
	Tree.Add(2);
	Tree.Add(5);
	Tree.Add(6);
	Tree.Add(-3);
	Tree.Add(-4);
	Tree.Add(4);
	Tree.Add(3);

	cout << "\n-----\nStraight:";
	void(*f_ptr)(Node<int>*); f_ptr = print;
	Tree.Straight(f_ptr);

	char c; cin >> c;
	return 0;}
```

Практическая работа №3. Алгоритмы на графах

Дана матрица смежности неориентированного взвешенного графа (таблица 3.1, 0 означает отсутствие ребра).

1. Необходимо построить минимальное остовное дерево.
2. Запустите функцию, реализующую алгоритм поиска в глубину, для перечисления всех вершин в минимальном остовном дереве. Результат должен быть представлен с помощью одного из контейнеров STL.
3. Напишите функцию для поиска минимального пути (в смысле суммарного веса пройденных рёбер) между i-м и всеми остальными пунктами, куда можно построить маршрут. Результат должен быть представлен с помощью одного из контейнеров STL.
4. Реализовать функцию подсчета степени (количества инцидентных ребер) вершин в полученном дереве (обход дерева сделать на основе поиска в ширину). Реализовать функцию подсчета средней степени по всему дереву.

Таблица 3.1

Вариант	Матрица смежности

1.	{
	{ 0, 9, 5, 4, 8, 5, 1, 6, 1, 9, 5, 0 },
	{ 9, 0, 1, 7, 7, 8, 2, 8, 2, 0, 7, 3 },
	{ 5, 1, 0, 4, 0, 9, 4, 4, 5, 2, 3, 1 },
	{ 4, 7, 4, 0, 5, 3, 8, 2, 3, 4, 5, 9 },
	{ 8, 7, 0, 5, 0, 5, 4, 7, 7, 5, 4, 4 },
	{ 5, 8, 9, 3, 5, 0, 5, 9, 4, 8, 1, 1 },
	{ 1, 2, 4, 8, 4, 5, 0, 1, 6, 4, 2, 0 },
	{ 6, 8, 4, 2, 7, 9, 1, 0, 8, 9, 4, 4 },
	{ 1, 2, 5, 3, 7, 4, 6, 8, 0, 2, 0, 7 },
	{ 9, 0, 2, 4, 5, 8, 4, 9, 2, 0, 3, 4 },
	{ 5, 7, 3, 5, 4, 1, 2, 4, 0, 3, 0, 2 },
	{ 0, 3, 1, 9, 4, 1, 0, 4, 7, 4, 2, 0 },
	}

2.	{ { 0, 5, 3, 6, 8, 9, 7, 8, 1, 7, 0, 0, 4, 8 },

	{ 5, 0, 0, 3, 6, 9, 6, 5, 0, 8, 0, 0, 5, 6 }, { 3, 0, 0, 2, 8, 1, 3, 0, 8, 8, 5, 5, 8, 4 }, { 6, 3, 2, 0, 4, 6, 6, 4, 6, 8, 8, 6, 9, 4 }, { 8, 6, 8, 4, 0, 2, 8, 0, 9, 0, 8, 2, 0, 5 }, { 9, 9, 1, 6, 2, 0, 8, 5, 5, 9, 8, 8, 9, 8 }, { 7, 6, 3, 6, 8, 8, 0, 3, 6, 6, 8, 1, 5, 6 }, { 8, 5, 0, 4, 0, 5, 3, 0, 7, 1, 4, 7, 8, 5 }, { 1, 0, 8, 6, 9, 5, 6, 7, 0, 1, 2, 5, 2, 2 }, { 7, 8, 8, 8, 0, 9, 6, 1, 1, 0, 6, 2, 4, 8 }, { 0, 0, 5, 8, 8, 8, 8, 4, 2, 6, 0, 8, 4, 3 }, { 0, 0, 5, 6, 2, 8, 1, 7, 5, 2, 8, 0, 5, 5 }, { 4, 5, 8, 9, 0, 9, 5, 8, 2, 4, 4, 5, 0, 3 }, { 8, 6, 4, 4, 5, 8, 6, 5, 2, 8, 3, 5, 3, 0 }, }

3.	{
	{ 0, 1, 7, 0, 9, 2, 4, 9, 3, 1, 4, 7, 3 },
	{ 1, 0, 8, 6, 0, 0, 4, 8, 5, 7, 6, 7, 4 },
	{ 7, 8, 0, 9, 6, 0, 6, 1, 3, 0, 4, 4, 9 },
	{ 0, 6, 9, 0, 4, 5, 1, 1, 5, 6, 4, 9, 3 },
	{ 9, 0, 6, 4, 0, 7, 0, 0, 9, 0, 4, 7, 6 },
	{ 2, 0, 0, 5, 7, 0, 4, 5, 3, 8, 5, 1, 8 },
	{ 4, 4, 6, 1, 0, 4, 0, 3, 4, 3, 4, 8, 0 },
	{ 9, 8, 1, 1, 0, 5, 3, 0, 3, 5, 7, 5, 6 },
	{ 3, 5, 3, 5, 9, 3, 4, 3, 0, 2, 3, 0, 4 },
	{ 1, 7, 0, 6, 0, 8, 3, 5, 2, 0, 7, 9, 4 },
	{ 4, 6, 4, 4, 4, 5, 4, 7, 3, 7, 0, 9, 8 },
	{ 7, 7, 4, 9, 7, 1, 8, 5, 0, 9, 9, 0, 6 },
	{ 3, 4, 9, 3, 6, 8, 0, 6, 4, 4, 8, 6, 0 },
	}

4.	{ { 0, 8, 2, 0, 5, 1, 7, 3, 5, 9, 3, 7 }, { 8, 0, 7, 5, 7, 1, 9, 1, 1, 6, 6, 9 }, { 2, 7, 0, 9, 3, 5, 1, 9, 1, 0, 8, 0 }, { 0, 5, 9, 0, 8, 8, 4, 0, 3, 5, 7, 8 }, { 5, 7, 3, 8, 0, 1, 7, 3, 0, 6, 8, 9 }, { 1, 1, 5, 8, 1, 0, 7, 0, 0, 8, 6, 9 }, { 7, 9, 1, 4, 7, 7, 0, 0, 7, 2, 5, 8 },

	{ 3, 1, 9, 0, 3, 0, 0, 0, 1, 8, 8, 1 }, { 5, 1, 1, 3, 0, 0, 7, 1, 0, 8, 6, 9 }, { 9, 6, 0, 5, 6, 8, 2, 8, 8, 0, 2, 7 }, { 3, 6, 8, 7, 8, 6, 5, 8, 6, 2, 0, 4 }, { 7, 9, 0, 8, 9, 9, 8, 1, 9, 7, 4, 0 }, }

5.	{
	{ 0, 1, 3, 0, 1, 3, 6, 6, 7, 1 },
	{ 1, 0, 1, 8, 6, 0, 0, 0, 8, 8 },
	{ 3, 1, 0, 6, 6, 4, 5, 7, 6, 2 },
	{ 0, 8, 6, 0, 1, 3, 0, 3, 4, 3 },
	{ 1, 6, 6, 1, 0, 4, 6, 8, 5, 7 },
	{ 3, 0, 4, 3, 4, 0, 1, 3, 6, 1 },
	{ 6, 0, 5, 0, 6, 1, 0, 4, 7, 1 },
	{ 6, 0, 7, 3, 8, 3, 4, 0, 1, 8 },
	{ 7, 8, 6, 4, 5, 6, 7, 1, 0, 1 },
	{ 1, 8, 2, 3, 7, 1, 1, 8, 1, 0 },
	}

6.	{
	{ 0, 5, 6, 6, 6, 4, 5, 0, 0, 8, 8, 4, 4 },
	{ 5, 0, 8, 0, 3, 8, 4, 8, 6, 6, 6, 3, 4 },
	{ 6, 8, 0, 2, 0, 0, 0, 9, 3, 5, 3, 8, 1 },
	{ 6, 0, 2, 0, 2, 4, 7, 7, 7, 9, 5, 5, 5 },
	{ 6, 3, 0, 2, 0, 1, 5, 5, 4, 4, 1, 4, 2 },
	{ 4, 8, 0, 4, 1, 0, 8, 1, 5, 4, 5, 8, 6 },
	{ 5, 4, 0, 7, 5, 8, 0, 6, 9, 0, 1, 2, 0 },
	{ 0, 8, 9, 7, 5, 1, 6, 0, 4, 6, 7, 3, 3 },
	{ 0, 6, 3, 7, 4, 5, 9, 4, 0, 4, 4, 1, 1 },
	{ 8, 6, 5, 9, 4, 4, 0, 6, 4, 0, 9, 2, 7 },
	{ 8, 6, 3, 5, 1, 5, 1, 7, 4, 9, 0, 6, 9 },
	{ 4, 3, 8, 5, 4, 8, 2, 3, 1, 2, 6, 0, 3 },
	{ 4, 4, 1, 5, 2, 6, 0, 3, 1, 7, 9, 3, 0 },
	}

7.	{ { 0, 6, 5, 6, 7, 5, 8, 8, 2 }, { 6, 0, 2, 5, 1, 4, 4, 3, 2 }, { 5, 2, 0, 0, 6, 7, 5, 4, 2 },

	{ 6, 5, 0, 0, 2, 4, 1, 7, 4 }, { 7, 1, 6, 2, 0, 8, 0, 9, 5 }, { 5, 4, 7, 4, 8, 0, 9, 8, 0 }, { 8, 4, 5, 1, 0, 9, 0, 7, 5 }, { 8, 3, 4, 7, 9, 8, 7, 0, 7 }, { 2, 2, 2, 4, 5, 0, 5, 7, 0 }, }

8.	{
	{ 0, 7, 8, 9, 1, 6, 3, 2, 0, 8, 2, 3, 0 },
	{ 7, 0, 7, 6, 0, 6, 7, 1, 4, 1, 1, 1, 1 },
	{ 8, 7, 0, 4, 0, 8, 0, 1, 0, 7, 7, 7, 6 },
	{ 9, 6, 4, 0, 8, 1, 2, 4, 5, 2, 2, 9, 8 },
	{ 1, 0, 0, 8, 0, 5, 7, 6, 7, 3, 4, 9, 0 },
	{ 6, 6, 8, 1, 5, 0, 7, 2, 1, 8, 9, 2, 9 },
	{ 3, 7, 0, 2, 7, 7, 0, 9, 5, 9, 6, 4, 9 },
	{ 2, 1, 1, 4, 6, 2, 9, 0, 4, 3, 2, 6, 9 },
	{ 0, 4, 0, 5, 7, 1, 5, 4, 0, 7, 1, 3, 6 },
	{ 8, 1, 7, 2, 3, 8, 9, 3, 7, 0, 9, 8, 3 },
	{ 2, 1, 7, 2, 4, 9, 6, 2, 1, 9, 0, 5, 6 },
	{ 3, 1, 7, 9, 9, 2, 4, 6, 3, 8, 5, 0, 9 },
	{ 0, 1, 6, 8, 0, 9, 9, 9, 6, 3, 6, 9, 0 },
	}

9.	{ { 0, 9, 7, 1, 5, 4, 5, 3, 8, 1, 0, 7, 4, 0, 8 }, { 9, 0, 7, 3, 2, 7, 0, 9, 8, 5, 0, 6, 4, 1, 3 }, { 7, 7, 0, 2, 2, 2, 2, 3, 9, 5, 1, 5, 0, 4, 4 }, { 1, 3, 2, 0, 4, 4, 1, 0, 6, 9, 7, 2, 3, 6, 2 }, { 5, 2, 2, 4, 0, 4, 4, 8, 4, 2, 4, 5, 7, 6, 9 }, { 4, 7, 2, 4, 4, 0, 9, 0, 3, 1, 6, 4, 8, 8, 8 }, { 5, 0, 2, 1, 4, 9, 0, 2, 2, 4, 5, 4, 2, 6, 1 }, { 3, 9, 3, 0, 8, 0, 2, 0, 4, 1, 9, 9, 5, 5, 7 }, { 8, 8, 9, 6, 4, 3, 2, 4, 0, 0, 7, 3, 7, 4, 1 }, { 1, 5, 5, 9, 2, 1, 4, 1, 0, 0, 7, 6, 1, 2, 9 }, { 0, 0, 1, 7, 4, 6, 5, 9, 7, 7, 0, 9, 6, 7, 8 }, { 7, 6, 5, 2, 5, 4, 4, 9, 3, 6, 9, 0, 6, 2, 2 }, { 4, 4, 0, 3, 7, 8, 2, 5, 7, 1, 6, 6, 0, 2, 7 }, { 0, 1, 4, 6, 6, 8, 6, 5, 4, 2, 7, 2, 2, 0, 7 },

	{ 8, 3, 4, 2, 9, 8, 1, 7, 1, 9, 8, 2, 7, 7, 0 }, }

10.	{
	{ 0, 9, 0, 2, 7, 4, 7, 8, 4 },
	{ 9, 0, 5, 5, 0, 7, 8, 1, 5 },
	{ 0, 5, 0, 6, 3, 9, 3, 3, 0 },
	{ 2, 5, 6, 0, 3, 4, 3, 9, 5 },
	{ 7, 0, 3, 3, 0, 0, 9, 0, 4 },
	{ 4, 7, 9, 4, 0, 0, 5, 9, 6 },
	{ 7, 8, 3, 3, 9, 5, 0, 9, 8 },
	{ 8, 1, 3, 9, 0, 9, 9, 0, 1 },
	{ 4, 5, 0, 5, 4, 6, 8, 1, 0 },
	}

11.	{
	{ 0, 8, 0, 1, 4, 1, 4, 9, 3, 9 },
	{ 8, 0, 5, 1, 4, 7, 9, 0, 2, 0 },
	{ 0, 5, 0, 9, 6, 0, 5, 6, 0, 6 },
	{ 1, 1, 9, 0, 0, 8, 5, 1, 1, 0 },
	{ 4, 4, 6, 0, 0, 3, 9, 0, 6, 5 },
	{ 1, 7, 0, 8, 3, 0, 1, 1, 4, 7 },
	{ 4, 9, 5, 5, 9, 1, 0, 2, 1, 8 },
	{ 9, 0, 6, 1, 0, 1, 2, 0, 7, 8 },
	{ 3, 2, 0, 1, 6, 4, 1, 7, 0, 7 },
	{ 9, 0, 6, 0, 5, 7, 8, 8, 7, 0 },
	}

12.	{ { 0, 1, 4, 2, 5, 6, 4, 0, 2, 1, 1 }, { 1, 0, 7, 6, 5, 2, 9, 5, 9, 8, 3 }, { 4, 7, 0, 7, 0, 9, 0, 8, 2, 0, 8 }, { 2, 6, 7, 0, 9, 7, 2, 8, 0, 8, 1 }, { 5, 5, 0, 9, 0, 7, 7, 5, 0, 1, 8 }, { 6, 2, 9, 7, 7, 0, 1, 8, 5, 9, 0 }, { 4, 9, 0, 2, 7, 1, 0, 6, 4, 9, 4 }, { 0, 5, 8, 8, 5, 8, 6, 0, 9, 6, 6 }, { 2, 9, 2, 0, 0, 5, 4, 9, 0, 8, 8 }, { 1, 8, 0, 8, 1, 9, 9, 6, 8, 0, 7 }, { 1, 3, 8, 1, 8, 0, 4, 6, 8, 7, 0 },

	}

13.	{
	{ 0, 9, 9, 7, 6, 9, 9, 5, 3 },
	{ 9, 0, 3, 9, 0, 7, 8, 9, 5 },
	{ 9, 3, 0, 8, 1, 7, 1, 2, 4 },
	{ 7, 9, 8, 0, 9, 0, 4, 2, 2 },
	{ 6, 0, 1, 9, 0, 3, 1, 9, 1 },
	{ 9, 7, 7, 0, 3, 0, 8, 0, 3 },
	{ 9, 8, 1, 4, 1, 8, 0, 7, 7 },
	{ 5, 9, 2, 2, 9, 0, 7, 0, 4 },
	{ 3, 5, 4, 2, 1, 3, 7, 4, 0 },
	}

14.	{
	{ 0, 6, 2, 1, 9, 1, 8, 1, 4, 8, 6, 1, 3 },
	{ 6, 0, 2, 5, 1, 9, 9, 8, 1, 7, 9, 1, 1 },
	{ 2, 2, 0, 4, 2, 2, 5, 3, 4, 6, 0, 3, 0 },
	{ 1, 5, 4, 0, 1, 2, 4, 9, 4, 8, 8, 0, 9 },
	{ 9, 1, 2, 1, 0, 3, 5, 4, 4, 4, 5, 4, 8 },
	{ 1, 9, 2, 2, 3, 0, 2, 5, 1, 6, 9, 5, 8 },
	{ 8, 9, 5, 4, 5, 2, 0, 7, 9, 3, 5, 9, 6 },
	{ 1, 8, 3, 9, 4, 5, 7, 0, 5, 2, 0, 9, 2 },
	{ 4, 1, 4, 4, 4, 1, 9, 5, 0, 6, 9, 2, 9 },
	{ 8, 7, 6, 8, 4, 6, 3, 2, 6, 0, 9, 5, 4 },
	{ 6, 9, 0, 8, 5, 9, 5, 0, 9, 9, 0, 5, 1 },
	{ 1, 1, 3, 0, 4, 5, 9, 9, 2, 5, 5, 0, 0 },
	{ 3, 1, 0, 9, 8, 8, 6, 2, 9, 4, 1, 0, 0 },
	}

15.	{ { 0, 6, 1, 9, 4, 4, 2, 3, 5 }, { 6, 0, 2, 2, 4, 0, 5, 5, 0 }, { 1, 2, 0, 1, 6, 9, 4, 6, 3 }, { 9, 2, 1, 0, 1, 9, 9, 4, 3 }, { 4, 4, 6, 1, 0, 2, 8, 3, 1 }, { 4, 0, 9, 9, 2, 0, 9, 1, 2 }, { 2, 5, 4, 9, 8, 9, 0, 8, 8 }, { 3, 5, 6, 4, 3, 1, 8, 0, 9 }, { 5, 0, 3, 3, 1, 2, 8, 9, 0 },

	}

16.	{
	{ 0, 5, 3, 1, 0, 3, 4, 3, 6, 0, 4 },
	{ 5, 0, 6, 7, 9, 4, 3, 3, 6, 9, 6 },
	{ 3, 6, 0, 4, 5, 1, 9, 5, 3, 1, 8 },
	{ 1, 7, 4, 0, 5, 5, 2, 4, 2, 5, 8 },
	{ 0, 9, 5, 5, 0, 3, 8, 2, 6, 4, 3 },
	{ 3, 4, 1, 5, 3, 0, 3, 2, 2, 2, 8 },
	{ 4, 3, 9, 2, 8, 3, 0, 7, 6, 7, 6 },
	{ 3, 3, 5, 4, 2, 2, 7, 0, 4, 3, 4 },
	{ 6, 6, 3, 2, 6, 2, 6, 4, 0, 7, 3 },
	{ 0, 9, 1, 5, 4, 2, 7, 3, 7, 0, 9 },
	{ 4, 6, 8, 8, 3, 8, 6, 4, 3, 9, 0 },
	}

17.	{
	{ 0, 9, 7, 9, 6, 9, 5, 5, 6, 3, 6 },
	{ 9, 0, 2, 3, 7, 6, 5, 6, 7, 7, 0 },
	{ 7, 2, 0, 5, 0, 0, 6, 8, 0, 5, 6 },
	{ 9, 3, 5, 0, 6, 2, 5, 1, 1, 2, 2 },
	{ 6, 7, 0, 6, 0, 0, 1, 0, 5, 8, 3 },
	{ 9, 6, 0, 2, 0, 0, 4, 2, 8, 3, 0 },
	{ 5, 5, 6, 5, 1, 4, 0, 5, 9, 7, 4 },
	{ 5, 6, 8, 1, 0, 2, 5, 0, 9, 2, 6 },
	{ 6, 7, 0, 1, 5, 8, 9, 9, 0, 1, 0 },
	{ 3, 7, 5, 2, 8, 3, 7, 2, 1, 0, 4 },
	{ 6, 0, 6, 2, 3, 0, 4, 6, 0, 4, 0 },
	}

18.	{ { 0, 0, 0, 7, 6, 0, 1, 4, 4, 5, 6, 6, 7 }, { 0, 0, 6, 4, 4, 1, 2, 3, 2, 0, 1, 4, 2 }, { 0, 6, 0, 8, 8, 4, 3, 6, 5, 3, 6, 6, 5 }, { 7, 4, 8, 0, 5, 4, 5, 8, 0, 9, 3, 6, 8 }, { 6, 4, 8, 5, 0, 1, 4, 2, 7, 7, 7, 2, 0 }, { 0, 1, 4, 4, 1, 0, 7, 4, 4, 2, 4, 2, 6 }, { 1, 2, 3, 5, 4, 7, 0, 7, 1, 2, 2, 9, 8 },

	{ 4, 3, 6, 8, 2, 4, 7, 0, 8, 4, 2, 3, 2 }, { 4, 2, 5, 0, 7, 4, 1, 8, 0, 2, 5, 8, 1 }, { 5, 0, 3, 9, 7, 2, 2, 4, 2, 0, 5, 9, 7 }, { 6, 1, 6, 3, 7, 4, 2, 2, 5, 5, 0, 9, 6 }, { 6, 4, 6, 6, 2, 2, 9, 3, 8, 9, 9, 0, 5 }, { 7, 2, 5, 8, 0, 6, 8, 2, 1, 7, 6, 5, 0 }, }

19.	{
	{ 0, 4, 0, 7, 1, 8, 9, 7, 6, 8, 3 },
	{ 4, 0, 5, 3, 5, 3, 2, 3, 9, 6, 2 },
	{ 0, 5, 0, 4, 9, 9, 9, 6, 2, 7, 2 },
	{ 7, 3, 4, 0, 5, 7, 8, 4, 1, 8, 1 },
	{ 1, 5, 9, 5, 0, 4, 8, 2, 3, 4, 2 },
	{ 8, 3, 9, 7, 4, 0, 6, 1, 5, 4, 6 },
	{ 9, 2, 9, 8, 8, 6, 0, 0, 7, 7, 6 },
	{ 7, 3, 6, 4, 2, 1, 0, 0, 9, 2, 7 },
	{ 6, 9, 2, 1, 3, 5, 7, 9, 0, 1, 1 },
	{ 8, 6, 7, 8, 4, 4, 7, 2, 1, 0, 5 },
	{ 3, 2, 2, 1, 2, 6, 6, 7, 1, 5, 0 },
	}

20.	{
	{ 0, 5, 2, 7, 4, 8, 8, 8, 0, 6, 8, 4 },
	{ 5, 0, 7, 2, 0, 8, 9, 6, 4, 2, 5, 4 },
	{ 2, 7, 0, 1, 3, 3, 8, 3, 2, 6, 3, 6 },
	{ 7, 2, 1, 0, 6, 8, 0, 6, 3, 9, 4, 3 },
	{ 4, 0, 3, 6, 0, 9, 2, 3, 5, 9, 6, 8 },
	{ 8, 8, 3, 8, 9, 0, 9, 5, 4, 6, 9, 1 },
	{ 8, 9, 8, 0, 2, 9, 0, 0, 0, 5, 5, 8 },
	{ 8, 6, 3, 6, 3, 5, 0, 0, 0, 7, 6, 3 },
	{ 0, 4, 2, 3, 5, 4, 0, 0, 0, 6, 5, 4 },
	{ 6, 2, 6, 9, 9, 6, 5, 7, 6, 0, 8, 1 },
	{ 8, 5, 3, 4, 6, 9, 5, 6, 5, 8, 0, 9 },
	{ 4, 4, 6, 3, 8, 1, 8, 3, 4, 1, 9, 0 },
	}

21.	{ { 0, 6, 7, 6, 2, 9, 4, 6, 4, 7, 1 }, { 6, 0, 1, 1, 7, 7, 4, 7, 4, 8, 3 },

	{ 7, 1, 0, 4, 5, 5, 7, 2, 3, 9, 0 }, { 6, 1, 4, 0, 4, 6, 6, 8, 5, 3, 6 }, { 2, 7, 5, 4, 0, 9, 5, 0, 6, 9, 7 }, { 9, 7, 5, 6, 9, 0, 9, 2, 0, 8, 1 }, { 4, 4, 7, 6, 5, 9, 0, 4, 5, 8, 5 }, { 6, 7, 2, 8, 0, 2, 4, 0, 0, 4, 0 }, { 4, 4, 3, 5, 6, 0, 5, 0, 0, 7, 1 }, { 7, 8, 9, 3, 9, 8, 8, 4, 7, 0, 4 }, { 1, 3, 0, 6, 7, 1, 5, 0, 1, 4, 0 }, }

22.	{
	{ 0, 7, 7, 9, 0, 4, 7, 6, 4, 4, 1, 4 },
	{ 7, 0, 7, 5, 5, 2, 1, 1, 8, 5, 9, 0 },
	{ 7, 7, 0, 8, 9, 0, 9, 8, 9, 7, 4, 1 },
	{ 9, 5, 8, 0, 7, 3, 9, 6, 5, 5, 5, 2 },
	{ 0, 5, 9, 7, 0, 9, 9, 3, 5, 9, 0, 2 },
	{ 4, 2, 0, 3, 9, 0, 3, 2, 3, 2, 9, 3 },
	{ 7, 1, 9, 9, 9, 3, 0, 2, 7, 2, 4, 7 },
	{ 6, 1, 8, 6, 3, 2, 2, 0, 3, 3, 6, 9 },
	{ 4, 8, 9, 5, 5, 3, 7, 3, 0, 5, 6, 2 },
	{ 4, 5, 7, 5, 9, 2, 2, 3, 5, 0, 6, 4 },
	{ 1, 9, 4, 5, 0, 9, 4, 6, 6, 6, 0, 7 },
	{ 4, 0, 1, 2, 2, 3, 7, 9, 2, 4, 7, 0 },
	}

23.	{
	{ 0, 8, 2, 8, 9, 3, 7, 1, 5, 6 },
	{ 8, 0, 2, 9, 4, 7, 7, 1, 3, 5 },
	{ 2, 2, 0, 9, 5, 2, 9, 5, 9, 9 },
	{ 8, 9, 9, 0, 0, 9, 7, 7, 2, 8 },
	{ 9, 4, 5, 0, 0, 3, 2, 0, 3, 1 },
	{ 3, 7, 2, 9, 3, 0, 3, 0, 5, 9 },
	{ 7, 7, 9, 7, 2, 3, 0, 8, 7, 0 },
	{ 1, 1, 5, 7, 0, 0, 8, 0, 2, 6 },
	{ 5, 3, 9, 2, 3, 5, 7, 2, 0, 7 },
	{ 6, 5, 9, 8, 1, 9, 0, 6, 7, 0 },
	}

24.	{

	{ 0, 7, 7, 7, 0, 2, 4, 4, 3, 2, 3 }, { 7, 0, 0, 7, 1, 3, 0, 5, 0, 4, 6 }, { 7, 0, 0, 4, 6, 7, 0, 3, 9, 4, 1 }, { 7, 7, 4, 0, 4, 8, 4, 8, 5, 3, 5 }, { 0, 1, 6, 4, 0, 5, 5, 2, 2, 9, 6 }, { 2, 3, 7, 8, 5, 0, 7, 1, 5, 9, 5 }, { 4, 0, 0, 4, 5, 7, 0, 8, 6, 5, 2 }, { 4, 5, 3, 8, 2, 1, 8, 0, 2, 2, 8 }, { 3, 0, 9, 5, 2, 5, 6, 2, 0, 9, 4 }, { 2, 4, 4, 3, 9, 9, 5, 2, 9, 0, 2 }, { 3, 6, 1, 5, 6, 5, 2, 8, 4, 2, 0 }, }

25.	{
	{ 0, 1, 4, 4, 8, 5, 6, 7, 5, 2, 4, 2 },
	{ 1, 0, 6, 9, 1, 2, 3, 1, 2, 8, 9, 5 },
	{ 4, 6, 0, 7, 4, 8, 9, 6, 2, 6, 7, 6 },
	{ 4, 9, 7, 0, 2, 0, 0, 8, 8, 8, 7, 8 },
	{ 8, 1, 4, 2, 0, 3, 9, 2, 7, 7, 3, 1 },
	{ 5, 2, 8, 0, 3, 0, 0, 6, 4, 4, 5, 3 },
	{ 6, 3, 9, 0, 9, 0, 0, 2, 2, 9, 2, 3 },
	{ 7, 1, 6, 8, 2, 6, 2, 0, 7, 4, 2, 6 },
	{ 5, 2, 2, 8, 7, 4, 2, 7, 0, 3, 4, 6 },
	{ 2, 8, 6, 8, 7, 4, 9, 4, 3, 0, 3, 4 },
	{ 4, 9, 7, 7, 3, 5, 2, 2, 4, 3, 0, 7 },
	{ 2, 5, 6, 8, 1, 3, 3, 6, 6, 4, 7, 0 },
	}

26.	{ { 0, 3, 8, 7, 6, 6, 0, 2, 0, 0 }, { 3, 0, 9, 6, 3, 9, 9, 5, 1, 4 }, { 8, 9, 0, 4, 2, 4, 9, 8, 8, 0 }, { 7, 6, 4, 0, 5, 8, 5, 0, 3, 7 }, { 6, 3, 2, 5, 0, 2, 8, 8, 9, 4 }, { 6, 9, 4, 8, 2, 0, 6, 9, 7, 6 }, { 0, 9, 9, 5, 8, 6, 0, 1, 8, 4 }, { 2, 5, 8, 0, 8, 9, 1, 0, 6, 7 }, { 0, 1, 8, 3, 9, 7, 8, 6, 0, 6 }, { 0, 4, 0, 7, 4, 6, 4, 7, 6, 0 },

	}

27.	{
	{ 0, 9, 4, 3, 8, 7, 8, 2, 1 },
	{ 9, 0, 8, 2, 1, 0, 7, 9, 5 },
	{ 4, 8, 0, 8, 4, 4, 7, 5, 5 },
	{ 3, 2, 8, 0, 3, 4, 6, 2, 6 },
	{ 8, 1, 4, 3, 0, 5, 3, 2, 2 },
	{ 7, 0, 4, 4, 5, 0, 5, 2, 6 },
	{ 8, 7, 7, 6, 3, 5, 0, 0, 0 },
	{ 2, 9, 5, 2, 2, 2, 0, 0, 6 },
	{ 1, 5, 5, 6, 2, 6, 0, 6, 0 },
	}

28.	{
	{ 0, 7, 6, 4, 7, 0, 9, 2, 7 },
	{ 7, 0, 7, 6, 8, 5, 7, 4, 6 },
	{ 6, 7, 0, 6, 2, 1, 8, 6, 0 },
	{ 4, 6, 6, 0, 5, 8, 4, 7, 1 },
	{ 7, 8, 2, 5, 0, 0, 0, 2, 5 },
	{ 0, 5, 1, 8, 0, 0, 2, 2, 3 },
	{ 9, 7, 8, 4, 0, 2, 0, 6, 1 },
	{ 2, 4, 6, 7, 2, 2, 6, 0, 3 },
	{ 7, 6, 0, 1, 5, 3, 1, 3, 0 },
	}

29.	{
	{ 0, 9, 9, 3, 9, 6, 2, 9, 1, 5, 7 },
	{ 9, 0, 4, 3, 1, 3, 3, 3, 2, 6, 0 },
	{ 9, 4, 0, 4, 6, 1, 7, 5, 6, 7, 6 },
	{ 3, 3, 4, 0, 7, 0, 6, 6, 9, 5, 9 },
	{ 9, 1, 6, 7, 0, 8, 2, 3, 7, 3, 8 },
	{ 6, 3, 1, 0, 8, 0, 6, 9, 3, 7, 2 },
	{ 2, 3, 7, 6, 2, 6, 0, 8, 3, 6, 6 },
	{ 9, 3, 5, 6, 3, 9, 8, 0, 3, 0, 3 },
	{ 1, 2, 6, 9, 7, 3, 3, 3, 0, 4, 0 },
	{ 5, 6, 7, 5, 3, 7, 6, 0, 4, 0, 8 },
	{ 7, 0, 6, 9, 8, 2, 6, 3, 0, 8, 0 },
	}

30.	{

	{ 0, 9, 9, 8, 5, 9, 5, 1, 0, 5, 3, 6 }, { 9, 0, 7, 4, 7, 5, 6, 6, 5, 8, 4, 3 }, { 9, 7, 0, 2, 7, 7, 6, 5, 8, 0, 8, 1 }, { 8, 4, 2, 0, 0, 9, 0, 9, 4, 0, 4, 8 }, { 5, 7, 7, 0, 0, 0, 1, 1, 8, 9, 7, 5 }, { 9, 5, 7, 9, 0, 0, 6, 5, 3, 2, 3, 7 }, { 5, 6, 6, 0, 1, 6, 0, 5, 7, 5, 4, 4 }, { 1, 6, 5, 9, 1, 5, 5, 0, 2, 2, 6, 2 }, { 0, 5, 8, 4, 8, 3, 7, 2, 0, 3, 8, 1 }, { 5, 8, 0, 0, 9, 2, 5, 2, 3, 0, 4, 7 }, { 3, 4, 8, 4, 7, 3, 4, 6, 8, 4, 0, 6 }, { 6, 3, 1, 8, 5, 7, 4, 2, 1, 7, 6, 0 }, }

Код 3.1. Обход в глубину и ширину графа, заданного с помощью матрицы смежности

```
#include <iostream>

using namespace std;
int main()
 {

	// матрица смежности

	vector<vector<int> > mat =

	{
		{0, 1, 2, 0, 0, 0, 0},
		{1, 0, 2, 0, 0, 0, 0},
		{2, 2, 0, 4, 0, 0, 1},
		{0, 0, 4, 0, 1, 2, 2},
		{0, 0, 0, 1, 0, 1, 0},
		{0, 0, 0, 2, 1, 0, 0},
		{0, 0, 1, 2, 0, 0, 0}
	};

	vector <int> used(7, 0);
	//0 – вершина не посещена при поиске, 1 – помещена в структуру данных для вершин,
	//но не обработана, 2 – обработана, смежные вершины помещены в структуру данных
	//DFS – поиск в глубину
	stack<int> Stack;
	int iter = 0;

	Stack.push(0); // помещаем в очередь первую вершину
	while (!Stack.empty())
```

```
	{ // пока стек не пуст
		int node = Stack.top(); // извлекаем вершину
		Stack.pop();

		std::cout << "\nDFS at vertex " << node<< endl;

		if (used[node] == 2) continue;
		used[node] = 2; // отмечаем ее как посещенную
		iter++;
		for (int j = 0; j < 7; j++)
		{ // проверяем для нее все смежные вершины
			if (mat[node][j] > 0 && used[j] != 2)
			{ // если вершина смежная и не обнаружена
				Stack.push(j); // добавляем ее в стек
				used[j] = 1; // отмечаем вершину как обнаруженную
			}
		}
		std::cout << node << endl; // выводим номер вершины
	}
	std::cout << "\nVisited vertices";
	for (int i = 0; i < 7; i++) std::cout << used[i] << " ";

	for (int i = 0; i < 7; i++)
		used[i] = 0;
	queue<int> Queue;

	//BFS – поиск в ширину
	Queue.push(0);              //в качестве начальной вершины используем 0.
	used[0] = 2;
	vector <int> dist(7, 10000); //расстояния до вершин от 0-й в числе ребер
	dist[0] = 0;
	 iter = 0;
	while (!Queue.empty())
	{
		int node = Queue.front();  //извлекаем из очереди текущую вершину
		Queue.pop();

		//Здесь должна быть обработка текущей вершины.
		std::cout << "\nBFS at vertex " << node<< endl;
		if (used[node] == 2) continue;
			used[node] = 2;
		iter++;
		for (int j = 0; j < 7; j++)
		{ // проверяем для нее все смежные вершины
			if (mat[node][j] > 0 && used[j] != 2)
			{ // если вершина смежная и не обнаружена
				Queue.push(j); // добавляем ее в очередь
				used[j] = 1; // отмечаем вершину как обнаруженную
				if (dist[j] > dist[node] + 1)
					dist[j] = dist[node] + 1;
			}
		}
	}
	std::cout << "\nVisited vertices";
```

```
        for (int i = 0; i < 7; i++) std::cout << used[i] << " ";
        std::cout << "\nDistances";
        for (int i = 0; i < 7; i++) std::cout << dist[i] << " ";


        char c; cin >> c;
        return 0;
}
```

Практическая работа №4. Операции над многомерными массивами

Многомерные массивы применяются при реализации свёрточных нейронных сетей (convolutional neural networks, CNN), т.к. в них, в основном, работа идет с цветными изображениями. Для кодирования изображения применяется трехцветная модель RGB (red, green, blue). Каждое изображение представляет набор трех матриц (каждому цветовому каналу соответствует своя матрица). При этом, часто на вход сети подается сразу набор из нескольких изображений. Таким образом, на входе системы – четырехмерный массив.

В свёрточном слое нейронной сети используются фильтры, параметры которых настраиваются в процессе обучения сети. В задаче 4.1 используем уже готовые фильтры, применяемые при обработке изображений.

Задание 4.1. Свертка

Считайте данные изображения из файла или сгенерируйте многомерный массив соответствующего размера и заполните его случайным образом. Примените к нему фильтр, указанный в задании.

В результате применения фильтра получается изображение меньшего размера. Каждый пиксель нового изображения – это результат умножения матрицы фильтра на область изображения и затем суммирования элементов полученной матрицы. При этом умножение матриц происходит поэлементно, перемножаются только соответствующие пиксели (не по стандартному правилу умножения матриц). Затем, смещаясь на 1 пиксель по исходному изображению и выполняя те же действия, получаем новый результат (см. рисунок 4.1).

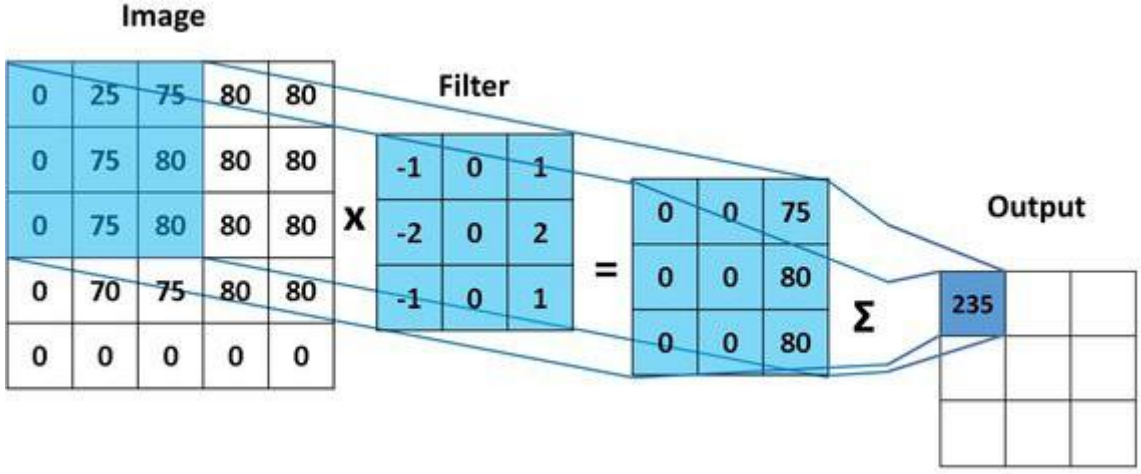


Рисунок 4.1. Схема работы фильтра изображений размером 3*3

В нечетных вариантах фильтр применяется к изображению в оттенках серого, предварительно полученному по формуле:

$$Y = 0.3R + 0.59G + 0.11B,$$

где R соответствует значению красной компоненты в пикселе, G – зеленой, B – синей (значения Y вычисляются для каждой точки исходного изображения).

В чётных вариантах преобразование надо применить к каждому каналу по отдельности, затем просуммировать результат от каждого канала.
После этого сохраните результирующее изображение.

*Необязательное задание: Для того, чтобы результирующее изображение было того же размера, что и исходное, необходимо добавить отступ по краям изображения и заполнить его, например, нулями («zero padding»). Размер отступа обычно зависит от размера фильтра

Вариант	Фильтр	Вари-ант	Фильтр
1,16	Гауссовое размытие 3x3 $\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	9,24	Выделение диагональных линий $\begin{bmatrix}-1&-1&2\\-1&2&-1\\2&-1&-1\end{bmatrix}$
2,17	Собеля по X $\begin{bmatrix}1&0&-1\\2&0&-2\\1&0&-1\end{bmatrix}$	10,25	Тиснение 1 $\begin{bmatrix}0&1&0\\0&0&0\\0&-1&0\end{bmatrix}$
3,18	Собеля по Y $\begin{bmatrix}1&2&1\\0&0&0\\-1&-2&-1\end{bmatrix}$	11,26	Тиснение 2 $\begin{bmatrix}0&0&0\\1&0&-1\\0&0&0\end{bmatrix}$
4,19	Прюитта по X $\begin{bmatrix}1&0&-1\\1&0&-1\\1&0&-1\end{bmatrix}$	12,27	Тиснение 3 $\begin{bmatrix}1&0&0\\0&0&0\\0&0&-1\end{bmatrix}$
5,20	Прюитта по Y $\begin{bmatrix}1&1&1\\0&0&0\\-1&-1&-1\end{bmatrix}$	13,28	Тиснение 4 $\begin{bmatrix}0&0&1\\0&0&0\\-1&0&0\end{bmatrix}$

6,21	Лапласа $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	14,29	Резкость (выделение границ) $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
7,22	Фильтр нижних частот $\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	15,30	Фильтр верхних частот $\frac{-1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
8,23	Гауссовое размытие 5x5 $\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$	31,32	Нерезкое маскирование $\frac{-1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & -476 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$

В качестве изображений могут выступать 3-х мерные векторы, заполненные случайным образом числами от 0 до 255. Для тех, кто хочет работать с реальными изображения, предлагается (но не обязательно) использовать библиотеку OpenCV (см. пример кода 4.1).

Код 4.1. Использование библиотеки OpenCV для загрузки, изменения и сохранения изображений

```
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;


int main() {
    // Открываем файл изображения
    Mat3b img = cv::imread("test.jpg");

    if (img.empty()) // Проверяем, удалось ли загрузить изобра-
жение
    {
        cout << "Cannot load image!" << endl; // выводим сообще-
ние в случае ошибки
```

```
        return -1;
    }
    // Создаем переменную для выходного изображения того же раз-
мера
    Mat3b img_out(img.size());

    imshow("img", img);  // Открываем окно с изображением

    //cout << "M = " << endl << " " << img << endl << endl;

    // Создаем объект точка - передаем координаты нужного пиксе-
ля (x,y)
    Point point(23, 42);
    // Создаем объект для пикселя из трех значений в таком по-
рядке: blue, green, red
    // Получаем значения яркостей по цветам в данном пикселе
    const cv::Vec3b& bgr = img(point); // или
img.at<Vec3b>(point);
    cout << bgr << endl;

    for(int y=0;y<img.rows;y++) {
        for (int x = 0; x < img.cols; x++) {
            // Получаем значение пикселя в точке
            Vec3b color = img.at<Vec3b>(Point(x, y));
            // что-то с ним делаем
            color[0] += 50;
            color[1] -= 33;
            color[2] += 33;

            // устанавливаем новое значение цвета в пикселе для
выходного изображения
            img_out.at<Vec3b>(Point(x, y)) = color;
            // если использовать ссылку Vec3b& color =
img.at<Vec3b>(Point(x, y));
            // то это обратное присваивание не нужно
        }
    }
    imshow("img_out", img_out);  // Открываем окно с измененным
изображением

    // записываем результат в файл
    imwrite("test_out.jpg", img);
```

```
    Mat channels[3]; // массив для 3 каналов изображения
    Mat merged; // переменная для объединенного изображения

    split(img, channels);  // разделяем изображение на 3 канала

    vector<Mat> channel_vec =
{channels[0],channels[1],channels[2]}; // вектор из 3 каналов -
нужно для объединения обратно
    //merge(channel_vec,merged);
    imshow("red", channels[2]);
    imshow("blue", channels[0]);
    imshow("green",channels[1]);

    // Здесь можно произвести изменения над каждым каналом
    // Занулим яркость в синем канале
    Mat& blue = channels[0]; // создаем переменную для удобства,
заметьте, что используем ссылку
    //cout << blue.size();
    for(int y=0;y<blue.rows;y++) {
        for (int x = 0; x < blue.cols; x++) {
            // Получаем значение пикселя в точке
            auto& color = blue.at<unsigned char>(Point(x, y));
            // что-то с ним делаем
            color = 0;
            // Т.к. мы выше использовали ссылку auto&, то от-
дельно
            // устанавливать новое значение цвета в пикселе для
выходного изображения
            // не нужно, оно изменится автоматически:
            // blue.at<unsigned char>(Point(x, y)) = color;
        }
    }

    /*
    // Аналогично, но теперь рассматриваем изображение как мат-
рицу, вытянутую в одномерный вектор
    for(int i=0; i < blue.rows*blue.cols; i++) {
            // Получаем значение пикселя в точке
            unsigned char color = blue.at<unsigned char>(i);
            // что-то с ним делаем
            color = 255;
            // устанавливаем новое значение цвета в пикселе для
выходного изображения
            blue.at<unsigned char>(i) = color;
```

```
            // или сразу blue.at<unsigned char>(i) = 255;
    }
     */

    // объединяем каналы обратно в одно изображение
    merge(channel_vec, merged);

    imshow("merged", merged);

    // записываем результат в файл
    imwrite("test_out_merged.jpg", merged);

    waitKey(0); // Для предотвращения закрытия окна программы,
ожидаем нажатия клавиши
    cout<<"End";
    return 0;
}
```

Задание 4.2. Активация

Применить функцию активации к изображению, полученному в пункте 4.1. Функция активации позволяет добавить нелинейности в процессе обучения нейронной сети.

Вариант	Функция	Вариант	Функция
1,16	Единичная ступенька	9,24	Гауссова
2,17	Сигмоида	10,25	Sinc
3,18	Гиперболический тангенс	11,26	Синусоида
4,19	Арктангенс	12,27	SoftPlus
5,20	Softsign	13,28	SELU
6,21	Обратный квадратный корень	14,29	SiLU
7,22	ReLU	15,30	ISRLU
8,23	PReLU	31,32	Выгнутая тождественная функция

Название функции	**Функция**
Единичная ступенька	$f(x)=\begin{cases}0, x<0\\1, x\geq 0\end{cases}$
Сигмоида	$f(x)=\sigma(x)=\frac{1}{1+e^{-x}}$
Гиперболический тангенс	$f(x)=th(x)=\frac{(e^{x}-e^{-x})}{(e^{x}+e^{-x})}$
Арктангенс	$f(x)=tg^{-1}(x)$
Softsign	$f(x)=\frac{x}{1+\lvert x\rvert}$
Обратный квадратный корень	$f(x)=\frac{x}{\sqrt{1+\alpha x^{2}}}$
ReLU	$f(x)=\begin{cases}0, x<0\\x, x\geq 0\end{cases}$
PReLU	$f(x)=\begin{cases}\alpha x, x<0\\x,\ x\geq 0\end{cases}$
Гауссова	$f(x)=e^{-x^{2}}$
Sinc	$f(x)=\begin{cases}1, & x=0\\\frac{\sin(x)}{x}, & x\neq 0\end{cases}$
Синусоида	$f(x)=\sin(x)$
SoftPlus	$f(x)=\ln(1+e^{x})$
SELU	$f(x)=\lambda\begin{cases}\alpha x, x<0\\x,\ x\geq 0\end{cases}$ $\lambda=1.0507, \alpha=1.67326$
SiLU	$f(x)=x\sigma(x)=\frac{x}{1+e^{-x}}$
ISRLU	$f(x)=\begin{cases}\frac{x}{\sqrt{1+\alpha x^{2}}}, & x<0\\x, & x\geq 0\end{cases}$
Выгнутая тождественная функция	$f(x)=\frac{\sqrt{x^{2}+1}-1}{2}+x$

Задание 4.3. Пулинг

Пулинг используется для уменьшения размеров изображения. Существует два типа пулинга: максимальный (max pooling) и средний (average pooling). Максимальный пулинг возвращает максимальное

значение из части изображения, покрываемой ядром. А средний пулинг возвращает среднее всех значений из части изображения, покрываемой ядром.

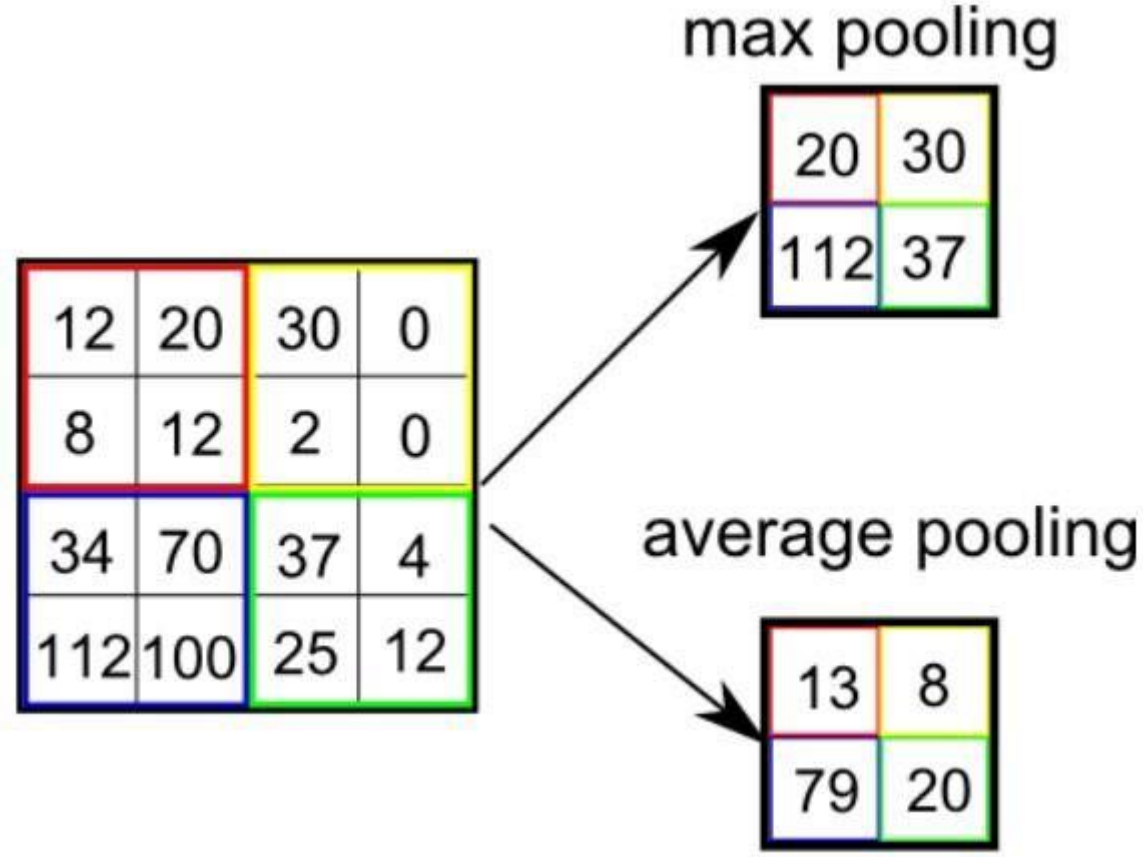


Рисунок 4.2. Схема применения среднего и максимального пулинга размером 2*2 к изображению 4*4

К изображению, полученному в пункте 4.2 применить пулинг размером 2x2. Для четных вариантов – максимальный пулинг, для нечетных – средний пулинг.

Задание 4.4. Оптимизация*

Произведите оптимизацию циклов в коде, построенном для решения задач 4.1-4.3. Произведите серию замеров времени, которое требуется на выполнение кода, до и после введения изменений согласно основным подходам оптимизации циклов.

Код 4.2. Пример замера времени исполнения кода

```
#include <iostream>
#include <chrono>
using namespace std;
```

```
int main() {
  int i, j, temp;
  int k=1;
  const int n = 10000000;
  int a[n];

  // стартуем замер времени
  auto begin = std::chrono::steady_clock::now();
  // здесь код, время работы которого нужно померить.
  // сортировка пузырьком
    for(i=0; i<n;i++){
        a[i] = rand();
    }

     for(i=0; i<n; i++){
        for(j=0; j< n - 1; j++){
            if(a[j]>a[j+1]){
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
  // замеряем время окончания
  auto end = std::chrono::steady_clock::now();

  // считаем время работы
  auto elapsed_ms = std::chrono::duration_cast<std::chrono::milliseconds>(end
- begin);
  std::cout << "The time: " << elapsed_ms.count() << " ms\n";
}
```

ВОПРОСЫ К ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ

1. Чем отличается класс от структуры в смысле Си? Как выбрать между классом и структурой в C++?
2. Что следует оставлять в основном теле определения класса и что разумно выносить из него? Приведите убедительный пример.
3. Могут ли разные части определения класса находиться в разных файлах?
4. Пространство имён класса. Как обращаться к членам класса: а) изнутри него, б) извне класса, в) из производных классов?
5. Можно ли внутри определения класса полноценно пользоваться объектами этого класса?
6. Доступ к членам класса: спецификаторы private, protected, public, дружественные функции.
7. Понятие интерфейса класса: что нежелательно туда выносить? Почему?
8. Устройство записи определения класса в самом общем виде: возможности.
9. Может ли один класс включен внутрь другого несколько раз?
10. Статические и нестатические атрибуты и методы класса: различия и возможности.
11. Чем отличается статический метод от статического атрибута класса?
12. Есть ли особенности наследования у классов со статическими членами?
13. Константные и неконстантные параметры метода. Константный метод: назначение, оформление.
14. Могут ли одноимённые методы с одинаковыми параметрами различаться лишь константностью по одному из параметров?
15. Предопределённый указатель this. Назначение, использование и предосторожности.
16. Можно ли указатель this передать не методу класса, а во внешнюю функцию?
17. Что делать, если часть начинки двух классов должна совпадать, а классы "не должны догадываться" друг о друге?
18. Способы наследования private, protected, public и спецификаторы в родительском классе private, protected, public. Что доступно в порождённом классе?

19. Способы наследования private, protected, public и спецификаторы в родительском классе private, protected, public.
20. Что доступно в классе, порождённом от порождённого от базового (2 ступень наследования) при первичном наследовании public?
21. Что доступно в классе, порождённом от порождённого от базового (2 ступень наследования) при первичном наследовании protected?
22. Наследование классов и пространства имён родительского и порождённого класса: как обращаться к члену родительского класса а) извне, б) из производных классов.
23. Конфликты имён в классах: причина и разрешение. Всегда ли оно возможно?
24. Сравните вставку (встраивание, погружение) класса в другой класс с наследованием: каковы возможности и ограничения в каждом случае?
25. В каком случае доступ к члену класса работает быстрее - при вставке (встраивании) класса или при наследовании? Почему?
26. Что такое множественное наследование, какие могут возникнуть проблемы? Как они решаются?
27. Виртуальные функции: способы задания, отличия от невиртуальных. Приведите сравнение на простом примере.
28. Что будет, если в родительском классе функция объявлена как виртуальная, в унаследованном от него - нет?
29. Имеется иерархия порождения классов A->B->C. Что будет в классе C, если в A функция была объявлена как обыкновенная, в B - как чисто виртуальная, а в C - снова как обыкновенная?
30. Виртуальные методы и абстрактные классы: назначение, отличия от обычных.
31. Зачем нужны абстрактные классы?
32. Статическое и динамическое связывание в контексте указателей на классы в иерархии порождения и преобразования типов C++.
33. Имеются ли хоть какие-то проверки допустимости преобразований при динамическом связывании?
34. Дружественные функции: назначение и отличия от методов класса. Понятие дружественного класса.
35. Может ли класс A быть другом класса B, а обратное не справедливо?
36. Дружественные базовому классу функции и их использование из порождённых классов.

37. Может ли одна функция быть дружественной нескольким классам одновременно?
38. Может ли метод одного класса быть другом другого класса?
39. Перегрузка функций: назначение и правила. Проиллюстрируйте простыми примерами допустимых и недопустимых вариантов перегрузки.
40. Что будет, если в классе перегружена функция, внешняя по отношению к классу?
41. Перегрузка операций: основные правила и ограничения. Выбор реализации между чисто внешними, дружественными функциями и методами класса.
42. Может ли одна и та же операция в одном и том же классе быть перегруженной разными способами?
43. Общий случай выбора функции при наличии обычных перегруженных функций, функций с переменным количеством параметров, базового шаблона функции и его уточнений (спецификаций, перегрузок).
44. Перегрузка операции () и понятие функтора.
45. Перегрузка операций -> и ->* и концепция "умного указателя".
46. Перегрузка операции индексирования [], её константный и неконстантный варианты. Обобщение индекса и многомерные массивы.
47. Перегрузка операций << и >> и введение в класс собственных инструментов ввода/вывода.
48. Что и как нужно сделать, чтобы мочь выводить содержимое объекта некоторого класса на экран?
49. Можно ли так перегрузить вывод содержимого объекта некоторого класса на экран несколькими разными способами и их сочетать?
50. Перегрузка одноместных операций -, * и &. Зачем она нужна?
51. Как автоматически обнаруживать потери динамической памяти?
52. Перегрузка new и delete. Различие скалярной и векторной форм.
53. В чём различия обработки поступившего указателя в delete и delete[]?
54. Перегрузка двуместных операций +, -, *, /, %, &, |, ^.
55. Перегрузка двуместных операций с присваиванием =, +=, -=, *=, /=, %=, &=, |=, ^=.
56. Перегрузка операций сравнения ==, !=, <, <=, >, >= и её особенности.
57. Перегрузка одноместных операций ++ и --. Различия префиксной и суффиксной форм.
58. В каких случаях при перегрузке операций создаётся временный объект?

59. Конструктор и деструктор: назначение, решаемые задачи, взаимодействие с средствами выделения памяти.
60. Какими функциями могут быть конструктор и деструктор (статическим и нестатическим методом класса, дружественной функцией, виртуальной функцией)? Почему? Аргументируйте ответ.
61. Можно ли перегружать конструктор? Почему?
62. Можно ли перегружать деструктор? Почему?
63. Передача параметров в конструктор и перегрузка конструктора. Что можно сказать о перегрузке деструктора? Конструктор по умолчанию.
64. Перегрузка конструкторов и наследование: как эффективно программировать конструктор порождённого класса?
65. Сложное наследование (несколько предков). Последовательность вызовов конструкторов и деструкторов для объектов производного класса в этом случае.
66. Потоковый ввод/вывод в C++. Перегрузка операций для этого и манипуляторы ввода/вывода.
67. Средства динамического выделения/освобождения памяти и их отличие от конструктора и деструктора.
68. Стандартные средства динамического выделения/освобождения памяти и наследование: что будет использовано, если имеется перегрузка new/delete в базовом/порождённом классе?
69. Пространства имён: определение, использование, наименование, вложенность.
70. Как сделать, чтобы в некотором блоке использовать несколько пространств имён сразу? Каким требованиям они должны отвечать?
71. Могут ли в одном классе быть несколько пространств имён?
72. Могут ли в одном пространстве имён быть несколько классов?
73. Могут ли два пространства имён пересекаться (но не одно не содержится в другом)?
74. Генерация и обработка исключительных ситуаций: является ли необходимой, чем обусловлено её введение, как осуществляется (правила перехватывания).
75. Что будет, если для некоторого класса исключительных ситуаций нет обработчика?
76. Что будет, если один класс исключительных ситуаций порождён от другого?

77. Бывают ли такие функции, в которых принципиально нельзя генерировать исключительные ситуации?
78. Классы памяти в C++: автоматический, динамический, временный, статический, внешний, регистровый: различия в поведении и использовании.
79. Когда вызываются конструкторы статических атрибутов класса?
80. Что дальше будет происходить с временным объектом после завершения его создавшей функции?
81. Шаблоны функций: назначение, определение, применение.
82. Что позволяют шаблоны и не позволяет препроцессор Си?
83. Что позволяет препроцессор Си и не позволяют шаблоны?
84. Правила перегрузки шаблонов: приведите короткие примеры.
85. Типизированные, нетипизированные и шаблонные параметры шаблонов - назначение и примеры использования.
86. Чем может быть нетипизированный параметр шаблона?
87. Всякое ли имя класса может быть передано в типизированный параметр шаблона?
88. Что может получиться из шаблона при подстановке в него всех типизированных и нетипизированных параметров?
89. Параметры по умолчанию в шаблонах и перегрузка шаблонов: правила задания и выбора используемого шаблона.
90. Шаблоны классов: назначение, определение, применение и правила порождения объектов.
91. Способы порождения функций и классов из шаблонов. Приведите короткие примеры.
92. Уточнение (спецификация, перегрузка) шаблонов: назначение, выяснение, какой шаблон будет в каком случае использован.
93. Как меняются правила перегрузки функции при появлении шаблона функции?
94. Неявные преобразования типа в C++: перечислите, приведите короткие примеры.
95. Явные преобразования типа в C++: встроенные формы const_cast, static_cast, dynamic_cast, reinterpret_cast. Правила выбора типа преобразования.
96. Функциональная форма преобразования типа в C++. Её реализация.
97. Что требуется сделать, чтобы можно было использовать функциональную запись преобразования из класса A в класс B?

98. Какие преобразования типа принимаются в расчёт только во время компиляции?
99. Преобразования указателей и ссылок в C++: особенности с учётом наследования.
100. Инициализация объектов классов, перечислите все её способы и проиллюстрируйте примерами.
101. Что нужно сделать, чтобы имелась возможность инициализировать значение объекта в месте его определения?
102. Можно ли проинициализировать значение объекта в месте его определения без вызова конструктора?
103. Инициализирующий конструктор: назначение, способ оформления, способы применения, отличие от перегрузки операции присваивания.
104. Понятие ссылки: назначение, способы применения, побочные эффекты.
105. Всегда ли передача по ссылке лучше передачи по значению? Аргументируйте ответ.
106. Всегда ли передача по ссылке лучше передачи по указателю? Аргументируйте ответ.
107. Всегда ли можно обойтись без ссылки?
108. Отличия в синтаксисе C++ от Си, связанные с перегрузкой функций, преобразованием типов и доступом к полям структур.
109. Контейнеры STL. Последовательные контейнеры и ассоциативные контейнеры
110. Операции над массивом. Эффективность операций
111. Операции над связным списком. Эффективность операций
112. Операции над бинарным деревом поиска. Эффективность операций
113. Очереди с приоритетами. Операции над кучей (heap). Эффективность операций
114. Бинарные деревья поиска. Балансировка. АВЛ-деревья. Эффективность операций
115. Ассоциативные контейнеры STL. Класс map. Эффективность операций.
116. Ассоциативные контейнеры STL. Класс set. Эффективность операций.
117. Ассоциативные контейнеры STL. Классы unordered_map, unordered_set. Эффективность операций.
118. Итераторы. Способы перечисления элементов в контейнерах STL.
119. Алгоритмы. Поиск в контейнере STL. Структуры данных для эффективного поиска.

120. Итераторы. Способы перечисления элементов в ассоциативных контейнерах STL.
121. Лямбда-функции. Применение в алгоритмах STL.
122. Алгоритмы на невзвешенных графах. Поиск в глубину. Поиск в ширину. Особенности реализации с использованием средств STL
123. Алгоритмы на взвешенных графах. Поиск минимального пути. Алгоритм Дейкстры. Особенности реализации с использованием средств STL
124. Алгоритмы на взвешенных графах. Поиск минимального пути. Алгоритм Беллмана-Форда. Поиск отрицательных циклов. Особенности реализации с использованием средств STL
125. Построение минимального остовного дерева. Алгоритм Прима. Особенности реализации с использованием средств STL
126. Определение максимального потока. Алгоритм Форда-Фалкерсона. Особенности реализации с использованием средств STL
127. Определение максимального паросочетания. Сведение к задаче определение максимального потока. Алгоритм Форда-Фалкерсона. Особенности реализации с использованием средств STL
128. Определение максимального паросочетания. Алгоритм Куна. Особенности реализации с использованием средств STL
129. Поиск клик в графе. Алгоритм Брона-Кербоша. Особенности реализации с использованием средств STL
130. Оптимизация кода. Методы оптимизации циклов
131. Оптимизация кода. Методы оптимизации ветвлений
132. Оптимизация кода. Методы оптимизации операций над массивами
133. Оптимизация кода. Методы оптимизации работы функций. Подставляемые функции. Макросы.

СПИСОК ЛИТЕРАТУРЫ

1. Кнут, Д. Искусство программирования, том 1. Основные алгоритмы / Д. Кнут – Москва: «Вильямс», 2015. – 720 с.
2. Кнут, Д. Искусство программирования, том 3. Сортировка и поиск / Д. Кнут – Москва: «Вильямс», 2013. – 824 с.
3. Топп, У. Структуры данных в C++ / У. Топп, У. Форд; пер. с англ. – Москва: ЗАО «Издательство Бином», 1999. – 816 с.
4. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт; пер. с англ. – Санкт-Петербург, «Невский диалект», 2001. – 352 с.
5. Кормен, Т.Х. Алгоритмы: построение и анализ, 3-е издание / Т. Х. Кормен, Ч. И. Лейзерстон, Р.Л. Ривест, К. Штайн; пер. с англ. – Москва: «Вильямс», 2013 – 1328 с.
6. Шилдт, Г. Самоучитель C++, 3-е издание / Г. Шилдт; пер. с англ. – Санкт-Петербург: БХВ-Петербург, 2003 – 688 с.
7. Шилдт, Г. Полный справочник по C++, 4-е издание / Г. Шилдт; пер. с англ. – Москва: Вильямс, 2006 – 800 с.
8. Шилдт, Г. Искусство программирования на C++ / Г. Шилдт; пер. с англ. – Санкт-Петербург: БХВ-Петербург, 2005 – 496 с.
9. Керниган, Б. Язык программирования Си, 3-е издание / Б. Керниган, Д. Ритчи; пер. с англ. – Москва: Вильямс, 2017 – 288 с.
10. Парфёнов, Д.В. Язык Си: кратко и ясно: учебное пособие / Д.В. Парфёнов – Москва: «Альфа-М», 2014 – 320 с.
11. Страуструп Б. Программирование. Принципы и практика с использованием C++ / Б.Страуструп; пер. с англ. – Москва: Вильямс, 2016 – 1328 с.
12. Страуструп Б. Язык программирования C++, 2-е издание / Б. Страуструп; пер. с англ. – Москва: Вильямс, 2019 – 320 с.
13. Страуструп Б. Язык программирования C++ для профессионалов, 2-е издание / Б.Страуструп; пер. с англ. – Москва: Вильямс, 2019 – 320 с.
14. Фог А. Оптимизация программ на C++. Руководство по оптимизации / А. Фог; пер. с англ – 145 с. URL: https://www.studmed.ru/agner-fog-tom-1-optimizaciya-programm-na-si-rukovodstvo-po-optimizacii-dlya-platform-windows-linux-i-mac_402b67bce5d.html

15. Гербер Р. Оптимизация ПО. Сборник рецептов / Р. Гербер, К. Тиан, К. Смит, А. Бик; пер. с англ. – Санкт-Петербург: Питер, 2010 – 352 с.