Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

**РЕФЕРАТ**

По предмету “Основы алгоритмизации и программирования”

На тему: “Структуры и другие типы данных”

Выполнил:

Студент 1 курса 9 группы

Павлович Ян Андреевич

Преподаватель: Белодед Н.И.

2023, Минск

Содержание

1. Введение
2. Определение и роль структур данных в программировании
   1. Что представляют собой структуры данных в программировании?
   2. Почему структуры данных имеют значение в разработке программного обеспечения
3. Структуры и типы данных
   1. Классификации
   2. Простейшие структуры и основные типы
4. Примеры основных структур данных
   1. Массив (Array)
   2. Динамический массив (Dynamic array)
   3. Связный список (Linked list)
   4. Стек (Stack)
   5. Очередь (Queue)
   6. Множество (Set)
   7. Карта (Map)
   8. Двоичное дерево поиска (Binary search tree)
   9. Префиксное дерево (Trie)
5. Заключение
6. Источники

**Введение**

Структуры данных в программировании играют важную роль в организации и управлении данными. Они представляют собой способ структурирования информации, позволяющий эффективно добавлять, извлекать, изменять, анализировать и сортировать данные. Каждая структура данных имеет свои алгоритмы и преимущества, которые позволяют нам выбирать наиболее подходящую структуру в зависимости от задачи. Благодаря структурам данных мы можем создавать пользовательские типы данных, делая код более понятным и удобным для работы с информацией. Структуры данных представляют собой основу для эффективного программирования. Они позволяют нам решать сложные задачи, основанные на обработке и анализе данных.

В этом реферате мы рассмотрим различные типы структур данных, их особенности и применение в программировании. Мы изучим такие структуры данных, как массивы, списки, стеки, очереди, деревья, хеш-таблицы. Каждая из этих структур имеет свои уникальные свойства и алгоритмы, которые позволяют эффективно работать с данными.

Все это позволит нам получить некоторое понимание структур данных и их роли в программировании, а также развить навыки проектирования и разработки программных решений.

**Определение и роль структур данных в программировании**

**Что представляют собой структуры данных в программировании?**

В программировании, структуры данных - это способ организации и хранения данных, которые могут быть взаимосвязаны друг с другом. Например, массив - это структура. Со структурой можно работать: добавлять данные, извлекать их и обрабатывать, например изменять, анализировать, сортировать. Для каждой структуры данных - свои алгоритмы.

Одним из преимуществ использования структур данных является возможность создания пользовательских типов данных. Это позволяет разработчикам создавать собственные структуры, сочетающие в себе различные переменные и данные. Например, в C++ можно создать структуру, объединяющую информацию о человеке, такую как имя, возраст и адрес. Это делает код более понятным и удобным для работы с данными.

Главное свойство структур данных в том, что у любой единицы данных должно быть чёткое место, по которому её можно найти. Как определяется это место и как происходит поиск, зависит от конкретной структуры. То есть, структуры нужны, чтобы упорядочивать, искать, анализировать и использовать данные с применением алгоритмов программирования.

Фактически использование структур данных в программировании начинается ещё с задания переменной. Формат переменной - определённая структура данных, так как в память переменная записывается конкретным способом. Но на практике программисты работают с другими структурами, которые объединяют в себе разные переменные и типы данных. Далее приведем описание и классификацию основных структур данных, чтобы было удобнее в них разобраться.

**Почему структуры данных имеют значение в разработке программного обеспечения**

Структуры данных играют решающую роль в разработке программного обеспечения по нескольким причинам:

1. **Эффективность.** Правильно выбранные структуры данных оптимизируют доступ и обработку данных, что приводит к созданию более быстрого и эффективного программного обеспечения. Правильная структура данных может значительно сократить временную сложность стандартных операций, таких как поиск, вставка и удаление.
2. **Масштабируемость.** По мере роста приложений и обработки больших объемов данных эффективные структуры данных становятся еще более важными. Хорошо спроектированная структура данных может поддерживать расширение приложений без серьезного снижения производительности, гарантируя, что программное обеспечение остается отзывчивым и стабильным.
3. **Сопровождение кода.** Приложение с организованными структурами данных легче поддерживать, изменять и расширять. Выбор подходящих структур данных упрощает код и способствует улучшению практики кодирования, тем самым улучшая качество программного обеспечения.
4. **Разработка алгоритмов.** Поскольку большинство алгоритмов построены на основе одной или нескольких структур данных, их эффективность во многом зависит от базовых структур. Правильная структура данных позволяет лучше реализовать алгоритм и может существенно повлиять на производительность программного обеспечения.

Глубокое понимание структур данных необходимо для эффективной и масштабируемой разработки программного обеспечения. Они позволяют разработчикам более эффективно решать проблемы, что приводит к созданию более качественных программных приложений.

**Структуры и типы данных**

**Классификации**

Вряд ли кто-нибудь решится спорить с тем, что в памяти компьютера данные (data) представлены в виде **последовательности битов**. Эти последовательности структурированы недостаточно, что затрудняет их применение на практике. Именно поэтому широко используются специальные структуры данных.

**Структурой данных** можно назвать некое количество элементов, которые имеют между собой внутренние связи. Существуют как **простые**, так и **интегрированные**структуры данных. Простые организуются из битов, вот их примеры: числовые; символьные; битовые; логические; указатели.

**Интегрированные** организуются с помощью простых и других интегрированных структур. Также структуры бывают **физические и логические**.

Немаловажно знать и такой термин, как **изменчивость структуры** - речь идет об изменении количества элементов и связей между ними. Учитывая понятие изменчивости, можно разделить структуры на **статические и динамические**. **Статические** структуры данных мы все хорошо знаем - из основных можно вспомнить массив, множество, вектор, запись, таблицу. Программистам хорошо известны и **динамические** структуры данных (три наиболее популярные - очередь, стек, списки).

Как уже было сказано выше, структура состоит из элементов данных. Эти элементы бывают как **упорядоченными, так и неупорядоченными**. С учетом этого признака, структуры данных можно разделить на следующие группы:

- **нелинейные** (к примеру, многосвязные списки, графы, деревья с их корневыми узлами, потомками и т. д.);

- **линейные с последовательным распределением** (это вектор, массив, строка, очередь, стек);

- **линейные**, но уже с **произвольным связным распределением** (это односвязные и двусвязные списки).

**Простейшие структуры и основные типы**

Такие конструкции называют примитивами либо базовыми структурами данных. К примеру, в языках программирования они представлены простыми типами данных. В зависимости от языка набор типов может отличаться, но эти различия не очень существенны, поэтому можно говорить о наличии неких общих принципов.

Первый тип - **целочисленный** (целый тип), который применяется для обозначения целых чисел (int, integer). Из школьного курса математики мы знаем, что целые числа бывают отрицательными либо беззнаковыми. Во внутреннем машинном представлении целое число может занимать 1, 2 либо 4 байта.

Второй тип- **вещественный.** Он уже имеют вид числа с плавающей точкой. Такое число представляется посредством двух целых чисел – матиссы и порядка, плюс знака.

Третье - **десятичный** **тип** (decimal). Его поддерживает не каждый язык программирования. К примеру, такой тип есть в C# - он имеет разрядность 128 бит и может представлять числовые значения в пределах от 1Е-28 до 7,9Е+28. Применяется в финансовых расчетах.

Если предполагается работа с отдельными двоичными числовыми разрядами, существует **битовый тип**. Здесь данные - это набор битов, которые объединены в байты либо слова. При выполнении операций предполагается обращение к каждому биту отдельно.

Идем дальше. Переменная, имеющая **логический тип**, способна принимать одно из 2-х значений: либо истину, либо ложь. Для хранения такой переменной требуется 1 байт памяти. False кодируется нулевым значением байта, True - любым значением, отличным от нуля.

**Символьный тип** дает возможность представлять данные в виде последовательности символов какого-нибудь определенного заранее множества. Каждый символ хранится в памяти в качестве последовательности битов. Соответствие символов и последовательностей называют кодировкой. Разные кодировки представляют символы в форме битовых последовательностей разной длины.

**Указатель** - это переменная, ее значение - адрес ячейки памяти. В результате указатель ссылается на какой-нибудь блок данных и указывает на его первую ячейку.

**Примеры основных структур данных**

1. Массив (Array)

Одна из самых простых структур данных, которая встречается чаще всего. Именно на массивах основаны многие другие структуры данных: списки, стеки, очереди.

Для простоты восприятия можно считать, что массив - это таблица. Каждый его элемент имеет индекс - «адрес», по которому этот элемент можно извлечь. В большинстве языков программирования индексы начинаются с нуля. То есть первый элемент массива имеет индекс не [1], а [0]. Данные в массиве можно просматривать, сортировать и изменять с помощью специальных операций.

Массивы бывают двух видов:

1. Одномерные

У каждого элемента только один индекс. Можно представить это как строку с данными, где одного номера достаточно, чтобы чётко определить положение каждой переменной.

// Объявление и инициализация одномерного массива целых чисел

**int** numbers[5] = {1, 2, 3, 4, 5};

1. Многомерные

У каждого элемента два или больше индексов. По сути, это комбинация из нескольких одномерных массивов, то есть вложенная структура.

// Объявление и инициализация двумерного массива целых чисел (3x3)

**int** matrix[3][3] = {

    {1, 2, 3},

    {4, 5, 6},

    {7, 8, 9}

};

Как применяют массивы:

1. В качестве блоков для более сложных структур данных. Массивы предусмотрены в синтаксисе большинства языков программирования, и на их основе удобно строить другие структуры.
2. Для хранения несложных данных небольших объёмов.
3. Для сортировки данных.

###### 2. Динамический массив (Dynamic array)

В классическом массиве размер задан заранее - мы точно знаем, сколько в нём индексов. А динамический массив - это тот, у которого размер может изменяться. При его создании задаётся максимальная величина и количество заполненных элементов. При добавлении новых элементов они сначала заполняются до максимальной величины, а при превышении сразу создаётся новый массив, с большей максимальной величиной.

Элементы в динамический массив можно добавлять без ограничений и куда угодно. Однако, если добавлять их в середину, остальные придётся сдвигать, что занимает много времени. Поэтому лучше всего [динамический массив работает](https://practicum.yandex.ru/blog/chto-takoe-dinamicheskii-massiv-dannyh/) при добавлении элементов в конце.

**const** **int** N = 100;  // Максимальный размер массива

**int**\* arr = **new** **int**[N];  // Выделение памяти под динамический массив

Также нужно учитывать, что необходимо очищать память при завершении работы с динамическим массивом.

**delete**[] arr;

Как применяют динамические массивы:

1. В качестве блоков для структур данных.
2. Для хранения неопределённого количества элементов.

###### 3. Связный список (Linked list)

Ещё одна базовая структура данных, которую, как и массивы, используют для реализации других структур. Связный список - это группа из узлов. В каждом узле содержатся:

1. Данные.
2. Указатель или ссылка на следующий узел.
3. В некоторых списках - ещё и ссылка на предыдущий узел.

В итоге получается список, у которого есть чёткая последовательность элементов. При этом сами элементы более разрозненны, чем в массиве, поскольку хранятся отдельно. Быстро перемещаться между элементами списка помогают указатели.

// Определение структуры узла списка

**struct** Node {

**int** data;

    Node\* next;

};

**int** main() {

    // Создание узлов списка

    Node\* head = **new** Node{1, **nullptr**};

    Node\* second = **new** Node{2, **nullptr**};

    Node\* third = **new** Node{3, **nullptr**};

    // Связывание узлов

    head->next = second;

second->next = third;

...

}

Как применяют связные списки:

1. Для построения более сложных структур данных.
2. Для реализации файловых систем.
3. Для формирования хэш-таблиц.
4. Для выделения памяти в динамических структурах данных.

4. Стек (Stack)

Эта структура данных позволяет добавлять и удалять элементы только из начала. Она работает по принципу LIFO - Last In, First Out (англ. «последним пришёл - первым ушёл»). Последний добавленный в стек элемент должен будет покинуть его раньше остальных.

// Создание пустого стека

std::stack<**int**> myStack;

// Добавление элементов в стек

myStack.push(1);

myStack.push(2);

myStack.push(3);

// Вывод верхнего элемента без удаления

std::cout << "Верхний элемент: " << myStack.top() << std::endl;

// Удаление верхнего элемента

myStack.pop();

Как применяют стеки:

1. Для реализации рекурсии.
2. Для вычислений постфиксных значений.
3. Для временного хранения данных, например истории запросов или изменений.

5. Очередь (Queue)

Этот вид структуры представляет собой ряд данных, как и стек. Но в отличие от него она работает по принципу FIFO - First In, First Out (англ. «первым пришёл - первым ушёл»). Данные добавляют в конец, а извлекают из начала.

Бывают неклассические, двусторонние очереди. В них можно добавлять элементы и извлекать их из начала и конца структуры. Элементы посередине недоступны.

// Создание пустой очереди

std::queue<**int**> myQueue;

// Добавление элементов в очередь

myQueue.push(1);

myQueue.push(2);

myQueue.push(3);

// Вывод первого элемента без удаления

std::cout << "Элемент: " << myQueue.front() << std::endl;

// Удаление первого элемента

myQueue.pop();

Как применяют очереди:

1. Для реализации очередей, например на доступ к определённому ресурсу.
2. Для управления потоками в многопоточных средах.
3. Для генерации значений.
4. Для создания буферов.

6. Множество (Set)

В отличие от предыдущих базовых структур, во множестве данные не упорядочены. Они хранятся группой, их нельзя структурировать и в некоторых случаях нельзя сортировать. Зато с ними можно работать как с классическими математическими множествами: объединять, искать пересечения, вычислять разность и смотреть, является ли одно множество подмножеством другого.

// Создание пустого множества

std::set<**int**> mySet;

// Добавление элементов в множество

mySet.insert(3);

mySet.insert(1);

mySet.insert(2);

mySet.insert(3); // Дубликаты игнорируются

// Удаление элемента из множества

mySet.erase(2);

Как применяют множества:

1. Для поддержания множества уникальных элементов.
2. Для хранения данных, которые не нужно сортировать.
3. Для сравнения, объединения наборов данных и других операций с ними.

7. Карта (Map)

Её ещё называют ассоциативным массивом или словарём. Данные здесь хранятся в паре «ключ/значение», причем каждый ключ уникален, а вот значения могут повторяться. То есть определённому уникальному ключу всегда соответствует конкретное значение.

// Создание пустой карты

std::map<std::string, **int**> myMap;

// Добавление пар "ключ-значение" в карту

myMap["Alice"] = 25;

myMap["Bob"] = 30;

myMap["Charlie"] = 22;

Как применяют Карту:

1. Для быстрого поиска данных.
2. Для создания баз, в которых нужно хранить уникальное соответствие между двумя множествами значений. Их помещают в ключ, и структура проверяет, чтобы ключ не повторялся.

Частный случай map - это **hash-map, или хэш-таблица.** В ней есть ключи и значения, а для их реализации добавляются индексы. Специальная хэш-функция позволяет по ключу вычислить индекс, чтобы найти нужные данные.

Когда в хэш-таблицу что-то вносят, то вписывают ключ и данные. Далее функция хэширует ключ, переводит его в число и записывает данные в ячейку, соответствующую этому числу. Когда нужно запросить данные, снова вводят ключ - и по нему хэш-функция находит нужные данные.

**const** **int** SIZE = 10; // Здесь SIZE предполагается как константа для размера хэш-таблицы

// Функция хэширования

**int** hash\_function(**char**\* key) {

    // Получаем ASCII-код первого символа ключа, преобразуем его в заглавную букву и вычитаем код 'A'

**int** hash = toupper(key[0]) - 'A';

    // Возвращаем остаток от деления хэша на SIZE, чтобы получить индекс в пределах размера таблицы

**return** hash % SIZE;

}

8. Двоичное дерево поиска (Binary search tree)

Дерево - это структура, данные в которой лежат в узлах. У каждого узла могут быть один или несколько дочерних и только один родитель, то есть они расходятся, как ветви дерева:

Деревья бывают разных типов, но наиболее распространены двоичные деревья поиска. У них есть следующие особенности:

1. У каждого узла не больше двух дочерних.
2. Если новое значение меньше, оно становится левым дочерним либо дочерним левого дочернего.
3. Если значение больше, оно становится правым дочерним или дочерним правого дочернего.

Дерево всегда имеет корневой узел и дополнительные дочерние узлы, как слева, так и справа. Алгоритм выполняет все операции путем сравнения значений с корнем и его дальнейшими дочерними узлами в левом или правом поддереве соответственно.

В зависимости от элемента, который необходимо вставить, найти или удалить, после сравнения алгоритм может легко удалить левое или правое поддерево корневого узла.

// Функция для вставки ключа в дерево

Node\* insert(Node\* root, **int** key) {

    // Если дерево пусто, создаем новый узел

**if** (root == **nullptr**) {

**return** **new** Node(key);

    }

    // Рекурсивно ищем место для вставки ключа

**if** (key < root->key) {

        root->left = insert(root->left, key);

    } **else** **if** (key > root->key) {

        root->right = insert(root->right, key);

    }

**return** root;

}

Как применяют двоичные деревья:

1. Для быстрого поиска данных.
2. Для хранения данных в отсортированном виде с возможностью быстро их добавлять и удалять.

9. Префиксное дерево (Trie)

Другие названия этой структуры данных - Бор и нагруженное дерево. Данные в нём хранятся последовательно: каждый узел - это префикс, по которому находятся следующие узлы.

Префиксные деревья помогают прогнозировать пользовательский ввод -например, предлагая слово, наиболее близкое по составу введенных букв. Так работают системы проверки орфографии или дополнитель кода в среде разработки.

В качестве примера префиксного дерева рассмотрим хранение слов или некоторых ассоциативных массивов.

**Префиксным деревом** называется дерево, в котором каждая вершина помечена некоторой буквой и имеет дополнительный признак терминальности. Если мы читаем вершины по порядку, мы получаем последовательность символов, соответствующую некоторому слову.

Если полученная последовательность заканчивается признаком терминальности, то такое слово считается существующим в этой структуре данных.

При этом два узла не могут быть связаны с общим предком с помощью одной и той же буквы.

Для латинского алфавита (в котором 26 строчных букв) изначально пустой бор можно реализовать так:

**const** **int** k = 26;

**struct** Vertex {

    Vertex\* to[k] = {0}; // нулевой указатель означает, что перехода нет

**bool** terminal = 0;

};

Vertex \*root = **new** Vertex();

Чтобы добавить слово в бор, нужно пройтись от корня по символам слова. Если перехода по для очередного символа нет - создать его, иначе пройти по уже существующему. В конце текущее состояние нужно не забыть пометить терминальным.

// Функция для добавления строки в префиксное дерево

**void** add\_string(string& s) {

    Vertex\* v = root;  // Используем указатель для обхода дерева, начиная с корня

**for** (**char** c : s) {

        c -= 'a';  // Преобразование символа в число от 0 до 25

**if** (!v->to[c]) {

            // Если перехода по символу c ещё нет, создаем новую вершину и устанавливаем переход

            v->to[c] = **new** Vertex();

        }

        v = v->to[c];  // Переходим к следующей вершине

    }

    v->terminal = **true**;  // Помечаем последнюю вершину как конечную

}

Как применяют префиксные деревья:

1. Для хранения данных, которые нужно выдавать по цепочке. Например, слова для функции автозаполнения в телефоне: в зависимости от одной набранной буквы дерево выдаёт следующие.
2. Для хранения данных, у которых есть повторяющиеся участки. Например IP-адресов.

**Заключение**

Структуры данных в программировании представляют собой способ организации и хранения данных, которые могут быть взаимосвязаны друг с другом. Они позволяют эффективно добавлять, извлекать, изменять, анализировать и сортировать данные. Каждая структура данных имеет свои алгоритмы и преимущества, которые позволяют выбирать наиболее подходящую структуру для каждой задачи.

Одним из основных преимуществ использования структур данных является возможность создания пользовательских типов данных. Это позволяет разработчикам создавать собственные структуры, которые объединяют различные переменные и данные, делая код более понятным и удобным для работы с информацией.

Важно отметить, что структуры данных играют важную роль в программировании, предоставляя нам средства для эффективного управления и организации данных. Они являются неотъемлемой частью разработки программ и позволяют решать сложные задачи, основанные на обработке и анализе данных. Правильный выбор структуры данных может существенно повлиять на производительность и эффективность программы.

Изучение и практика работы с различными структурами данных являются неотъемлемой частью процесса обучения программированию. Это помогает развивать навыки анализа и улучшения программного кода, а также повышает общую эффективность разработки программ. Понимание роли структур данных в программировании позволяет нам создавать более эффективные и масштабируемые программы, сокращая затраты по времени и памяти.

В заключение, структуры данных являются фундаментальным инструментом в программировании, обеспечивая нам средства для эффективной работы с данными. Использование подходящих структур данных помогает нам решать сложные задачи и создавать программы, которые максимально оптимизированы и эффективны.

**Источники**

1. 10 основных структур, которые должен знать разработчик.

URL: https://practicum.yandex.ru/blog/10-osnovnyh-struktur-dannyh/#:~:text=В%20программировании%20структурой%20обычно%20называют,каждой%20структуры%20данных%20—%20свои%20алгоритмы.

1. Что такое структуры данных? Определение и типы.

URL: https://appmaster.io/ru/blog/chto-takoe-opredelenie-i-tipy-struktur-dannykh

1. Типы структур данных в программировании.

URL: https://gb.ru/blog/tipy-struktur-dannyh-v-programmirovanii/

1. Структуры и типы данных.

URL: https://otus.ru/journal/struktury-i-tipy-dannyh/

1. Хеш-таблицы, деревья и префиксные деревья.

URL: https://javarush.com/quests/lectures/questharvardcs50.level05.lecture06

1. Двоичное дерево поиска (Binary Search Tree (BST)).

URL: https://evileg.com/ru/post/491/

1. Двоичное дерево поиска (BST) с примером.

URL: https://www.guru99.com/ru/binary-search-tree-data-structure.html

1. Префиксные деревья.

URL: https://ru.hexlet.io/courses/algorithms-trees/lessons/prefix/theory\_unit

1. Префиксное дерево.

URL: https://ru.algorithmica.org/cs/string-structures/trie/

1. 10 типов структур данных, которые нужно знать + видео и упраженения. URL:https://habr.com/ru/companies/netologyru/articles/334914/