Колледж Научно-Технологического Университета Сириус

ДОКЛАД

по дисциплине “Введение в специальность”

на тему “Структуры данных”

Выполнил:
Студент группы

К0709-23/3
Тутик Николай Алексеевич

Принял:

Старший преподаватель
Тенигин Альберт Андреевич

_____

IT-Колледж “Сириус”
2023

Содержание

Введение

Цели доклада:

- Узнаем, что такое структуры данных, их характеристики.
- Ответим на вопрос «Зачем нужны структуры данных в программировании?»
- Обозначим отличия линейных и нелинейных структур данных.
- Узнаем, какие структуры данных существуют, их классификации, для чего нужна каждая из них, какие операции можно проводить над структурами данных.
- Рассмотрим применение структур данных в реальных задачах.

Что такое структура данных?

Структура данных — так называемый контейнер, в который мы помещаем данные для дальнейшей удобной работы с ними.

Зачем нужны структуры данных в программировании?

Структуры данных, в первую очередь, нужны для того, чтоб упростить работу с большими объемами информации. Используя их, не нужно выделять отдельную переменную для каждой единицы данных. Можно объединить все в одну структуру и удобно проводить над информацией определенные операции. Для разных целей существует множество разновидностей структур данных, о них мы поговорим позже.

Характеристики структур данных.

Структура данных должна соответствовать нескольким критериям:

- Корректность – структура данных должна быть корректно технически описана.
- Пространственная сложность – реализация структуры данных должна занимать как можно меньше места
- Временная сложность – выполнение операция должно происходить максимально быстро

Классификации структур данных.

Структуры данных можно разделить на линейные и нелинейные. Линейные структуры данных – структуры, в которых элементы выстраиваются в последовательность. В таких структурах у каждого

элемента есть предыдущий и последующий (за исключением первого и последнего). Нелинейными структурами данных называют структуры, в которых элементы не расположены последовательно.

Еще структуры данных могут быть статическими и динамическими. Статические имеют постоянную размерность, то есть количество элементов в ней. Динамические же структуры могут изменять свою размерность в зависимости от того, сколько элементов мы в эту структуру «положили».

Массив (Array)

Массивы – наверное, самая широко используемая структура данных. Массив представляет собой список упорядоченный набор элементов. У каждого элемента есть индекс – номер этого элемента в массиве, причем индексы могут быть только со значениями от 0 до n-1, где n – размерность массива (количество элементов в нем). Массив подразумевает произвольный доступ, так как можно выбрать любой элемент, и доступ к ним будет осуществляться одинаково быстро. Отдельный элемент массива выбирается с помощью индекса (Обозначим массив переменной A. Чтоб получить доступ к элементу с индексом i, нужно написать A[i] или A_i (второе обозначение называется индексированной переменной)).

Обычно элементами массива могут быть только данные одного типа, поэтому об этой структуре данных говорят как об однородной. Но также бывают гетерогенные массивы, содержащие элементы разных типов данных.

Массивы делятся на одномерные и многомерные. Одномерные представляют собой список пронумерованных элементов. Многомерные массивы – структура, в которой каждый элемент сам является массивом. Доступ к элементам осуществляется с помощью двух или более индексов.

Классические массивы – статическая структура данных, то есть ее длинна всегда остается неизменной, но еще бывают динамические массивы: их длинна зависит от количества элементов, которых поместили в массив

Работать с массивом можно с помощью некоторых операций:

- Обход массива.
- Добавление нового элемента в массив.
- Изменение элемента.
- Определение размера массива.
- Срез (Создание нового массива из старого).
- Удаления элемента из массива (с помощью функции pop).

Массивы – базовая структура данных, которая применяется для решения огромного количества задач. Они используются для реализации более сложных структур данных. Массивы позволяют хранить большое количество данных в упорядоченном виде и осуществлять различные операции с ними, такие как добавление, удаление, изменение элементов и т.д.

Связанные списки (Linked lists)

Связанный список – структура данных, являющаяся последовательностью узлов, содержащих значение и ссылку на следующий и/или предыдущий элемент. Первый элемент структуры называется Head (голова), последний – Tail (хвост). Последний элемент списка имеет ссылку на Null. Одной из черт, отличающих список от массива, является то, что в списке могут находиться данные разных типов.

Списки бывают односвязными, двусвязными и кольцевыми: односвязные списки состоят из узлов, содержащих только значение и ссылку на следующий элемент. В двусвязных списках, помимо значения и ссылки на следующий элемент, он содержит и ссылку на предыдущий элемент списка. В кольцевых списках первый и последний узлы связаны между собой (в tail есть ссылка на head). Это позволяет обходить список полностью независимо от узла, с которого начали обход.

Основные операции над связанными списками:

- Обход списка
- Переход к следующему или предыдущему элементу
- Добавление нового элемента
- Удаление элемента из списка

В целом, связанные списки очень схожи с массивами. Их применение рационально в случаях, когда часто нужно добавлять или удалять элементы, ведь тогда не нужно будет перераспределять все элементы структуры. Связанные списки используют для реализации более сложных структур: очередей, стеков, хэш-таблиц, бинарных деревьев.

Очереди (Queues)

Очереди – линейная динамическая структура данных, похожая на связанный список. Эта структура работает по принципу FIFO (first in – first out). Принцип заключается в том, что первый вошедший в очередь элемент выйдет из нее первым. Начало очереди называют head, а конец – tail. Сравнить эту структуру можно с обыкновенной очередью в магазине.

Очереди можно реализовать разными способами, например, на связанном списке. Очередь на однонаправленном списке – это список, в котором все новые элементы включаются в tail, а исключаются из head.

Основные операции, которые можно проводить над очередями:

- Добавление элемента в очередь (enqueue),
- Извлечение элемента из очереди (dequeue),
- Проверка элемента в начале очереди (peek),
- Проверка очереди на наличие элементов в ней (isEmpty),
- Проверка размера очереди (size).

Вариантов применения очередей безграничное количество, однако я приведу лишь некоторые из них:

- Очереди используются в системе обработки запросов сервера: запросы поступают в очередь и обрабатываются в порядке поступления.
- С помощью очередей реализуют буфер: очередь используют для временного хранения данных, пока те не будут обработаны или переданы.

Стеки (Stacks)

Стеки – структура данных, очень схожая с очередью. Различаются они в том, что стеки работают по принципу LIFO (last in – first out). Этот принцип заключается в следующем: последний вошедший в стек элемент, выйдет из него первым. Последний добавленный элемент называется верхушкой стека или «top» Эту структуру можно сравнить со стопкой тарелок. Не важно, сколько тарелок в стопке, мы сможем взять только самую верхнюю, которую положили последней.

Основные операции, которые можно проводить над стеком:

- Добавление элемента в стек (push),
- Извлечение элемента из стека (pop),
- Просмотр верхушки стека (peek),
- Проверка стека на наличие элементов в нем (isEmpty),
- Проверка размера стека (size).

У стеков также есть огромное количество применений, но приведу я лишь пару:

- Реализация отмены действия. Все произведенные действия сохраняются в стеке, а при отмене действия, оно извлекается из стека.
- Стеки применяются при создании калькуляторов, использующих обратную польскую запись. Они нужны для преобразования выражений из одной формы в другую. Операнды помещаются в стек, а операторы выполняются на верхних элементах стека.

Множество (Set)

Понятие множества как структуры данных схоже с математическим понятием множеств. Это набор элементов, обязательно уникальных по значению, которые хранятся в произвольном порядке.

Работают с множеством с помощью операций, которые так же применяют к математическим множествам:

- Добавление и удаление элементов,
- Проверка множества на наличие в нём элементов,
- Объединение нескольких множеств (По итогу получается одно множество, которое содержит в себе все элементы первого и второго множеств,
- Разность множеств (По итогу получается множество, которое содержит элементы первого множества за исключением тех, которые есть во втором множестве),
- Пересечение множеств (По итогу получаем множество, содержащее только общие элементы двух изначальных множеств).

Есть большое количество применений множества. Например, эту структуру данных удобно использовать для удаления дубликатов в коллекциях, ведь во множестве не может находиться два одинаковых элемента.

Ассоциативный массив/словарь (Map)

Словарь – структура данных, очень схожая с массивом. Главное их отличие – то, что в качестве индекса (в словаре это называется ключом) могут выступать данные любого типа: строки, вещественные и целые числа, коллекции.

Операции над словарями проводятся те же, что и над массивами (см. стр. 5). Но в словарях по-другому реализован обход структуры: в массивах

достаточно было прибавлять к индексу 1 единицу, чтоб перейти к следующему элементу, а в словарях для этого есть несколько других решений. Например, можно перебирать ключи через цикл for и обращаться к значениям словаря по ключам.

Словари используют для хранения информации, ведь доступ к любому элементу очень легко получить по ключу.

Хэш – таблицы (Hash-maps)

Хэш-таблицы – частный случай словарей. В этой структуре данные также хранятся по типу «ключ, значение». Отличие от словарей заключается в том, что поиск элемента таблицы происходит по индексу. Чтоб получить индекс, нужно провести определенные операции над ключом. Эти операции называются хэш-функцией. Поиск элемента в хэш-таблице происходит очень быстро, ведь в отличие от словаря, обход всей структуры не требуется: достаточно «прогнать» ключ через хеш-функцию и получить индекс элемента в таблице. Но при использовании хеш-функций может возникнуть ситуация, в которой применение хеш-функции к разным ключам будет давать один и тот же индекс. Такая ситуация называется коллизией. Не существует хеш-функции, при применении которой не смогут возникнуть коллизии, поэтому необходимо научиться решать их.

Некоторые методы решения коллизий:

- Метод открытой адресации. В случае возникновения коллизии, нужно преобразовывать индекс (возможно, повторно применять к нему хеш-функцию) до тех пор, пока не получим индекс пустой ячейки. Преобразования индексов называются пробированием, а полученные индексы – последовательностью проб. У этого метода, как и у любого другого, есть свои преимущества и недостатки: главным минусом является зависимость от размера внутреннего массива (ячейки рано или поздно закончатся), а плюсами можно считать быстрый обход и небольшой расход памяти.
- Метод цепочек. Он заключается в следующем: вместо значений элементов в самой таблице, помещаем туда ссылку на значение, которое будет располагаться в отдельной области памяти. Если возникает коллизия, то ссылку на элемент с повторяющимся индексом нужно поместить в последний элемент по этому индексу. Этот метод тяжело описать без визуального представления, поэтому прикрепляю рисунок, который немного прокомментирую (Рис. 1)

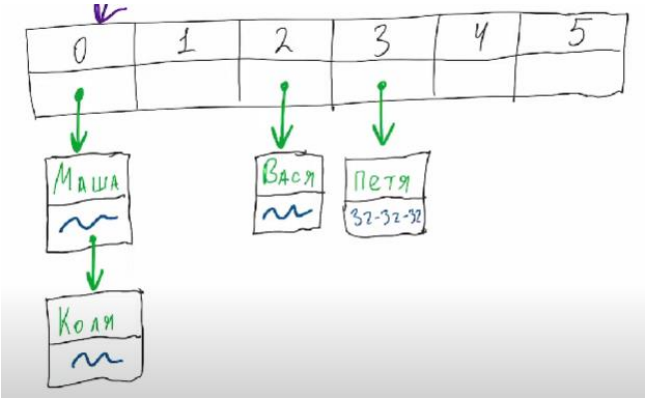

Рисунок 1 – визуальное представление метода цепочек.

В данном случае, применение функции к ключам «Маша» и «Коля» дает одинаковый индекс, поэтому мы сохраняем ссылку на элемент «Коля» в элементе «Маша». Плюсами метода цепочек можно ответить независимость от размера внутреннего массива, а также простоту реализации. Минусы: медленный обход, относительно большой расход памяти.

Чтоб считать хеш-функцию хорошей, она должна соответствовать четырем критериям:

- Детерминизм. Применение функции к одному ключу всегда должно давать один и тот же результат.
- Равномерность. Данные в таблице должны распределяться равномерно.
- Эффективность. Хеш-функция должна вычисляться быстро.
- Ограниченность. Индексы, которые выдает хеш-функция, должны быть в пределах таблицы.

Хеш-таблицы часто используются для реализации ассоциативных массивов, при построении кэша, индексированных баз данных и т.д.

Деревья (Trees)

Деревья – нелинейная структура данных, состоящая из корня, узлов и листьев. Корень – узел дерева, не имеющий предков, листья – узлы дерева, не имеющие потомков. У узла может быть несколько потомков, но предок может быть только один. За счет этого, дерево – иерархическая структура данных.

Типов деревьев огромное множество, поэтому остановимся на основных.

Бинарные деревья поиска (Binary search trees).

Отличительной чертой бинарного дерева поиска является то, что у каждого узла не может быть больше двух потомков. В узлах, помимо

значения, находятся ссылки на потомков. Потомок, меньший по значению, располагается слева, больший – справа, благодаря чему данные хранятся упорядоченно. Бинарные деревья поиска нужны, как раз таки, для быстрого поиска элементов. Чтоб найти какое-то значение, нужно сравнить его с корнем, если оно больше корня, то переходим в правый узел, если меньше – в левый, повторяем это, пока не найдем нужного значения. В бинарное дерево поиска можно вставлять элемент, удалять его из структуры, сохраняя порядок дерева. В случае, если добавлять и удалять элементы не нужно, рациональнее использовать упорядоченный массив, потому что в деревьях бинарного поиска может возникать дополнительная нагрузка из-за работы с ссылками и неравномерного распределения данных. Бинарные деревья поиска применяются в СУБД, в реализации поисковых движков и во многих других задачах.

Префиксные деревья (Tries).

Префиксные деревья еще называют нагруженными деревьями и борами. Боры – деревья, в котором элементы ищут с помощью строк. Они состоят из вершин, которые, в свою очередь, состоят из нескольких компонентов:

- Символ, причем только один.
- Признак терминальности. Этот параметр показывает, является ли этот узел конечным в каком-то ключе.
- Значение. Если проверка на флаг терминальности успешно пройдена, то можно получить значение, соответствующее данному ключу.
- Указатели на потомков.

Чтоб найти необходимое значение по ключу, нужно, начиная с первой пустой вершины, перемещаться по указателям на следующую букву в ключе. После того, как ключ будет полностью введен, необходимо выполнить проверку на флаг терминальности. При наличии флага, можно считать, что ключ найдет.

Одно из преимуществ нагруженных деревьев над словарями или хеш-таблицами – эффективная трата памяти, поскольку у многих ключей структуры могут быть одинаковые начала ключей – префиксы, для которых не потребуется дополнительное выделение памяти.

Префиксные деревья можно использовать для хранения данных. Также на них строятся системы, проверяющие правописание. С помощью префиксных деревьях реализованы подсказки и автоматическое дополнение при написании текста или кода.

Двоичные кучи (Heaps).

Двоичные кучи – полные бинарные деревья. Бывают max-heaps min-heaps и. В первом случае необходимо, чтоб соблюдалось свойство: потомок всегда должен быть меньше предка, а значение корневого узла должно быть максимальным. Во втором – наоборот, каждый потомок должен быть больше своего предка, а значение корневого узла – минимальным.

В кучу можно добавить элемент (push), а можно его удалить (pop). При добавлении элемента необходимо вставить его на место согласно правилам заполнения полных бинарных деревьев (слева – направо, снизу – вверх), а далее, если свойство кучи не соблюдается, выполнить «просеивание» кучи снизу вверх (если потомок больше предка в max-heap, то необходимо поменять их местами), пока свойство не будет соблюдаться. Удаляется элемент из кучи немного по-другому. Удалить мы можем только элемент, который находится в корневом узле, поэтому сначала необходимо поменять удаляемый элемент с элементом в корневом узле, достать его операцией pop, а потом выполнить «просеивание» кучи, но уже сверху вниз, пока куча не будет соответствовать свойству кучи.

Кучи можно хранить в виде деревьев с указателями на потомков, но есть вариант проще и быстрее – хранение в массиве. В таком случае, массив заполняется элементами кучи в порядке заполнения ими самой кучи. Для получения индекса нужного элемента есть некоторые формулы:

- Получить индекс левого потомка определенного узла можно с помощью формулы i (индекс узла) * 2 + 1.
- Индекс правого потомка можно получить благодаря формуле i * 2 + 2.
- Чтоб получить индекс предка, необходимо воспользоваться формулой (i – 1)/2.

Двоичные кучи зачастую используются для реализации приоритетных очередей, так как из них удобно извлекать элементы с самым высоким и самым низким приоритетом. Также эта структура данных используется в некоторых алгоритмах сортировки, например, в пирамидальной сортировке (HeapSort). Двоичные кучи используются в алгоритмах обхода графов, таких как алгоритм Дейкстры.

Графы (Graphs).

Графы – нелинейная динамическая структура данных, состоящая из вершин и ребер. Вершины – объекты, содержащие значение, ребра – пути, связывающие вершины между собой (Рис. 2).

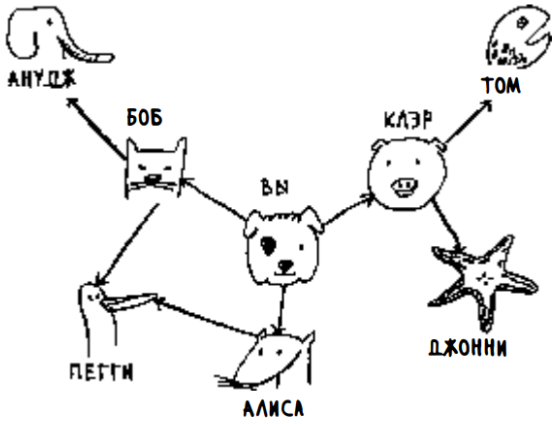

Рисунок 2 – направленный граф

Граф может быть ненаправленным и направленным. Ненаправленный граф – граф, ребра которого не имеют направлений и могут быть пройдены в обе стороны. Направленный – граф, в котором ребра имеют направление и перемещаться можно только вдоль указанного направления.

Также графы можно разделить на взвешенные и невзвешенные. В первом случае ребра имеют числовые значения, которые могут подразумевать под собой что угодно: длину пути, вес, стоимость и т.д. Во втором – у ребер нет значений.

Граф применяется в огромном количестве задач. Их применяют в алгоритмах для поиска кратчайшего пути, на них строятся графовые базы данных, они используются при поиске общих друзей в социальных сетях. Графы используются в машинном обучении. Карты используют графы для построения пути и нахождения расстояния между точками.

Заключение.

Структуры данных имеют огромную роль в программировании на данный момент. Они кардинально упрощают работу с большими и не очень объемами данных, на их основе строится множество алгоритмов.

В ходе данного доклада мы достаточно подробно рассмотрели основные структуры данных, рассмотрели их строение, выделили основные классификации структур данных, научились работать со структурами данных

(пока только в теории), рассмотрели примеры применения структур данных в программировании.

Все цели доклада считаю достигнутыми.

Список используемой литературы:

1. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. - СПб.: Питер, 2017, стр. 20-23, 41-43, 43-45, 45-46, 46-47, 65-69, 100-124, 128-135, 138, 152, 254-258.
2. Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010, стр. 20-22, 26-29, 29-31, 72-81, 170-175, 175-191, 191-210, 220-227, 246-250, 257-261.
3. Алгоритмы и структуры данных. Новая версия для Оберона + CD / Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010, стр. 78 – 79, 79-82, 82-84.
4. Структуры и алгоритмы обработки данных: оценка сложности алгоритмов, линейные и иерархические структуры данных. Гулаков В. К., Трубаков А. О., Трубаков Е. О. Научная библиотека library.ru