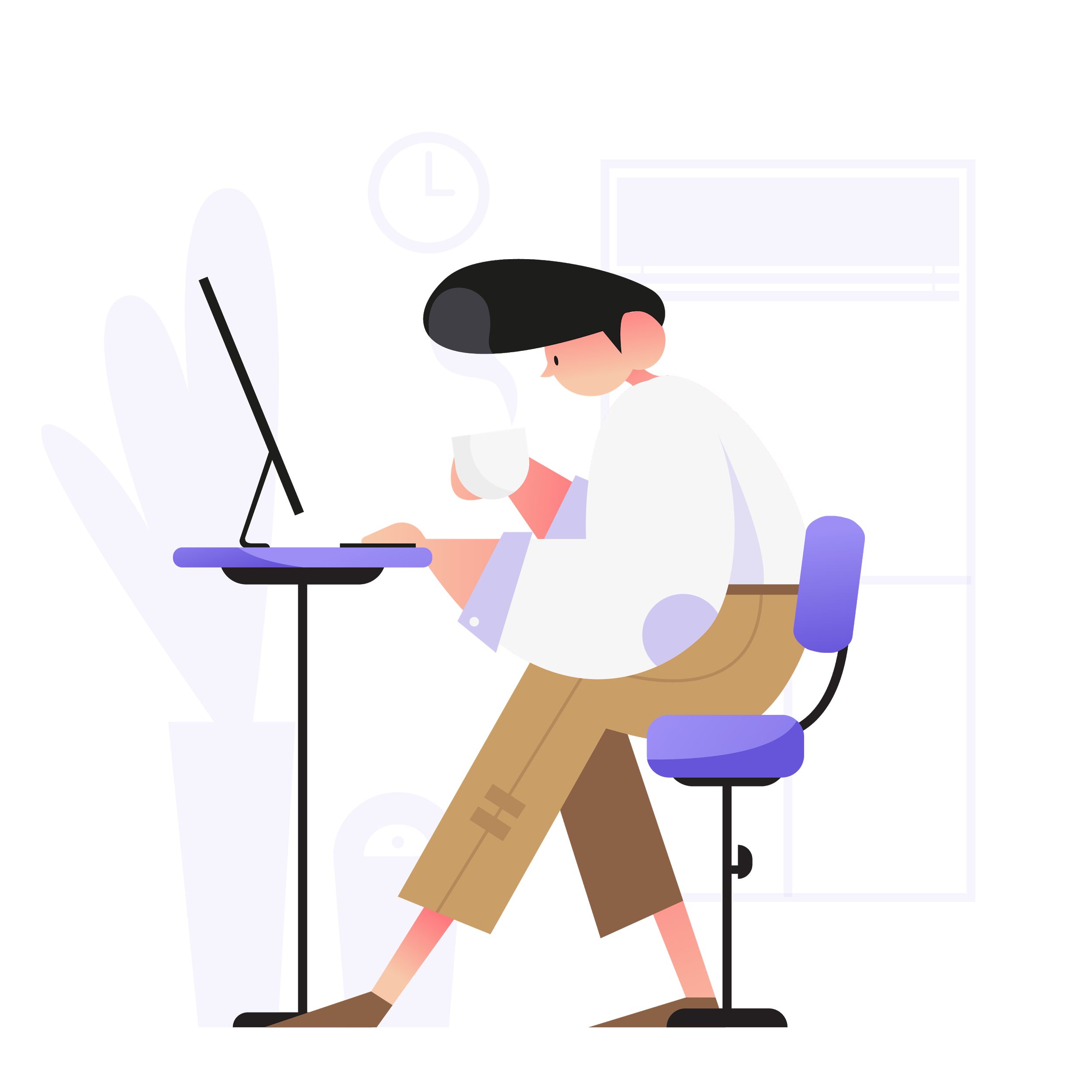
Алгоритмы и структуры данных на С#

Стек, очередь, словарь и коллекции в C#

Netcore 3.1





# 

**На этом уроке**

1. Узнаем, что такое очередь, стек и дек.
2. Выявим возможные пути реализации стека и очереди.
3. Узнаем, как осуществляется поиск в глубину и ширину.
4. Разберём к[оллекции с C#](#_2d2vjrhq2z2o).
5. Изучим threadsafe-коллекции: concurrent и immutable.
6. Рассмотрим интерфейсы коллекций

**Оглавление**

[Введение](#_5033axa43fbw)

[Очередь и стек](#_kloy5qmwzu5)

[Стек](#_b83bzp6zurbq)

[Очередь](#_2uh7uku1csta)

[Дек](#_7aqm438zu2v1)

[Возможные пути реализации](#_7b7idtx4pfg0)

[Поиск в дереве](#_1svva7swgbam)

[BFS](#_k8999mvkup79)

[DFS](#_d29e8sox42rz)

[Dictionary](#_9tyuxr2m163t)

[Коллекции с C#](#_gbwp8dt58otj)

[HashSet<TKey,TValue>](#_btxesqtyblfg)

[SortedSet<TKey,TValue>](#_kt7guat1w4fp)

[Dictionary<TKey,TValue>](#_ls4fa5vqch6)

[List<T>](#_rw0z7pm1nerp)

[LinkedList<T>](#_ucgjepyuzb2r)

[Queue<T>](#_783iqu2qfxgo)

[Stack<T>](#_6f63zp7xa78t)

[SortedList<TKey,TValue>](#_lfy90n1zr7k6)

[Threadsafe-коллекции](#_gh6shpaf34gd)

[Concurrent](#_i9tho1xkvodq)

[Immutable](#_ffy5rt6y9dtq)

[Интерфейсы](#_isp4s88mjjbk)

[Заключение](#_tt9c2dv9g6k4)

[Практическое задание](#_h4fhoqi0j41m)

[Дополнительные материалы](#_2jxsxqh)

[Используемая литература](#_7c9jrnh1eqkx)

# 

# Введение

Такие структуры данных, как стек и очередь, — очень частые гости в современном программировании. Например, это очередь в брокерах сообщений, таких как RabbitMQ, или стек вызова в любой современной программе. Мы рассмотрим эти структуры на уроке.

В .NET существует реализация различных структур данных и интерфейсов доступа к ним, чтобы программист не писал всё с нуля. На уроке мы рассмотрим, какие это структуры, и узнаем, у каких из них есть потокобезопасные версии.

# 

# Очередь и стек

## Стек

Стек — это структура данных, которая организована по принципу «последним пришёл — первым ушёл» (Last In — First Out). Стек удобно использовать в некоторых алгоритмах, поэтому полезно уметь создавать эту структуру. Если представить стек как стопку блинов, то будет понятно, что первый испечённый блин (если, конечно, вы его не съедите сразу) будет лежать в самом низу стопки и будет взят самым последним. А последний испечённый блин будет лежать наверху стопки, и ваши гости (или вы) съедят его первым.

На практике стек используется в программах для контроля вызова функций. Туда кладутся аргументы функции, возвращаемое значение и адрес возврата в вызывающую функцию. Также стек может использоваться для вычисления математических выражений, где при разборе выражения более приоритетные операции будут находиться наверху стека, а операции с самым низким приоритетом — внизу стека.

Типовые операции:

1. Push — добавление в стек.
2. Pop — изъятие из стека.
3. Проверка наличия элементов или их количества.
4. Очистка.

## 

## Очередь

Очередь — структура данных, организованная по принципу «первым пришёл – первым ушёл» (First In — First Out). Это линейный список, для которого введены две операции:

1. Добавление нового элемента в конец очереди.
2. Удаление первого элемента из очереди.

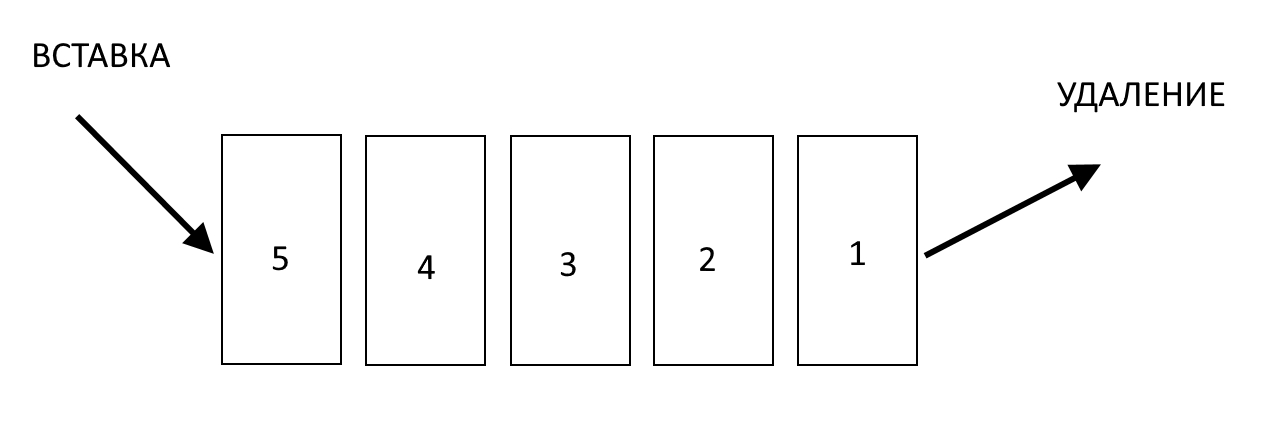
Операционные системы используют очереди для организации сообщений между программами: у каждой программы своя очередь сообщений. Контроллеры жёстких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создаётся очередь из пакетов данных, ожидающих отправки. Также при разработке распределённых приложений часто используются брокеры сообщений, содержащие очереди сообщений, которые требуется обработать.

В коммуникации между потоками очереди часто могут использоваться для чтения и обработки данных. Например, первый поток читает данные с устройства и кладёт в очередь, а второй поток её обрабатывает и кладёт результат на диск. Такое решение полезно, когда устройство теряет данные, если их незамедлительно не прочитать, а обработка занимает достаточное время.

Типовые операции:

1. Enqueue — добавление в очередь.
2. Dequeue — изъятие из очереди.
3. Проверка наличия элементов или их количества.
4. Очистка.

Пример очереди:



## Дек

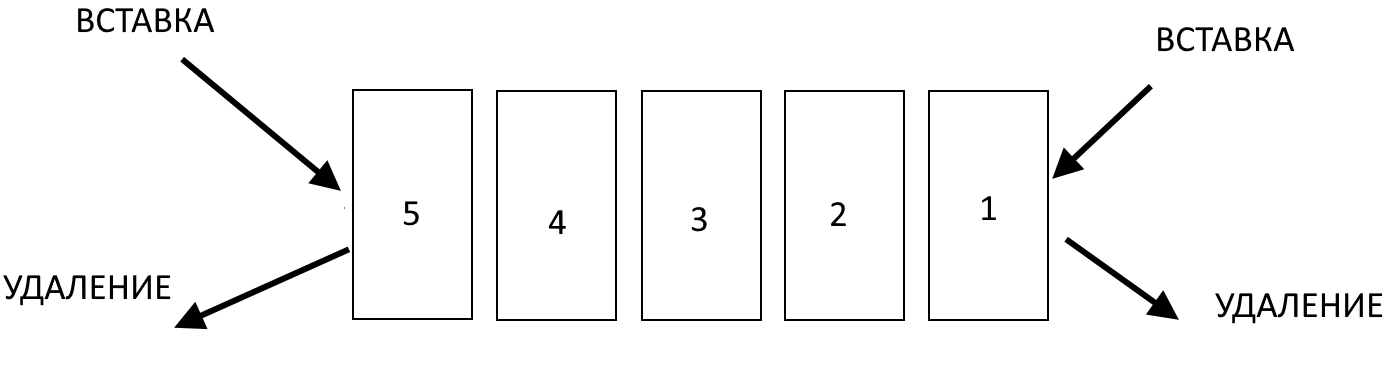
Дек (двусторонняя очередь) — линейный список, в котором можно добавлять и удалять элементы с одного и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт.

Типовые операции:

1. pushBack — добавление в конец очереди.
2. pushFront — добавление в начало очереди.
3. popBack — выборка с конца очереди.
4. popFront — выборка с начала очереди.
5. Проверка наличия элементов или их количества.
6. Очистка.

Пример дека:



## 

## Возможные пути реализации

Эти структуры легко реализовать с помощью обычного массива или связного списка, выбор зависит от задачи. Связный список подходит для всех описанных выше структур. Это универсальное решение, когда неизвестно, сколько будет вставок или удалений. Время операций вставки и удаления для стека, очереди и дека на связном списке будет O(1).

Если же известно максимальное количество элементов, которое будет содержаться в стеке или очереди, то можно использовать массив. Это позволит уменьшить трафик памяти и промахи кеша, так как все элементы структуры данных будут находиться в памяти рядом, в заранее выделенном участке. В случае неправильного использования время работы со стеком или очередью на вставку и удаление может стать O(N).

# 

# Поиск в дереве

Рассмотрим пример, где могут использоваться стек и очередь: поиск в ширину и в глубину.

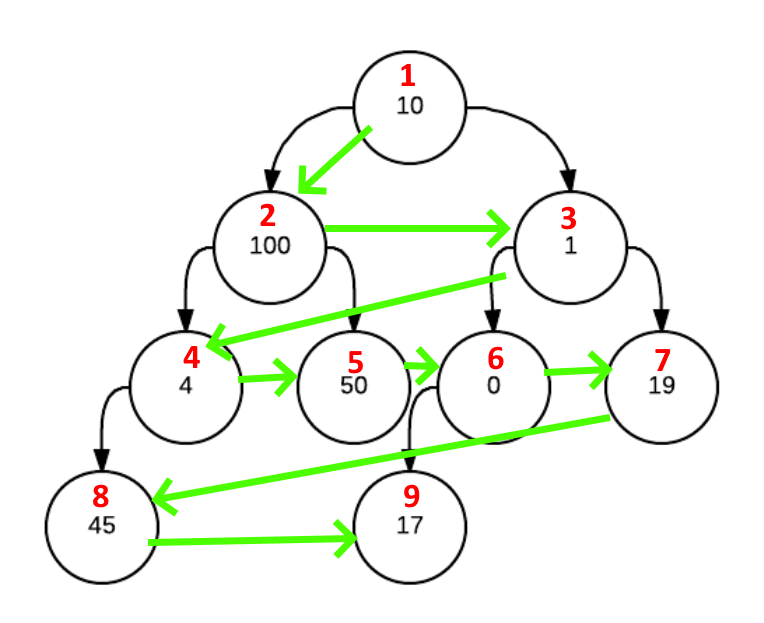
## BFS

BFS (breadth-first search) — поиск в ширину. При поиске в дереве это значит, что мы ищем нужный нам элемент последовательно на каждом уровне дерева: сначала первый уровень, потом второй, потом третий и т. д. Для реализации этого алгоритма часто используют очередь. BFS также неплохо подходит для поиска кратчайшего пути между двумя точками в невзвешенном графе.

Алгоритм работы:

1. Положить корень дерева в очередь.
2. Если очередь пуста, завершить работу алгоритма.
3. Вынуть из очереди элемент.
4. Если элемент искомый, вернуть его и завершить работу алгоритма.
5. Положить все дочерние узлы элемента в очередь.
6. Вернуться к пункту 2.

Пример прохода по дереву с помощью BFS:



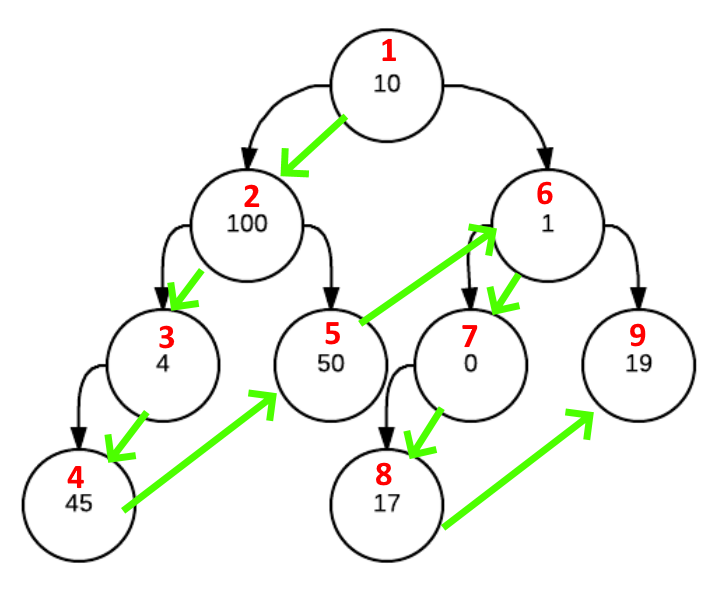
## DFS

DFS (deep-first search) — поиск в глубину. В отличие от поиска в ширину тут мы последовательно проходим по каждой ветви дерева до конца. Для реализации этого алгоритма часто используют стек.

Алгоритм работы:

1. Положить корень дерева в стек.
2. Если стек пуст, завершить работу алгоритма.
3. Вынуть из стека элемент.
4. Если элемент искомый, вернуть его и завершить работу алгоритма.
5. Положить все дочерние узлы элемента в стек.
6. Вернуться к пункту 2.

Пример прохода по дереву с помощью DFS:



# Dictionary

Класс Dictionary представляет собой хеш-таблицу, которая хранит пары ключ-значение. Ключ в рамках одного Dictionary должен быть уникален, иначе при попытке добавить дублирующее значение будет выдан exception. Принцип работы Dictionary схож с HashSet, только в случае с Dictionary данные хранятся парой ключ-значение. Ключом может быть любой тип, но если это пользовательский тип, то требуется, как в случае с HashSet, переопределить HashSet и GetHashcode.

В случае с Dictionary бакет — это связный список, который лежит в общем для всех бакетов массиве, и адрес каждого элемента списка — это адрес в массиве. Каждый узел списка — структура значимого типа, что позволяет хранить весь узел в массиве целиком.

Также есть ConcurrentDictionary — потокобезопасная версия Dictionary. В отличие от Dictionary узел связанного списка в бакете уже ссылочного типа и не хранится в массиве. Это увеличивает промахи кеша, но позволяет более независимо производить операции вставки и удаления. Также в ConcurrentDictionary есть список объектов синхронизации, ограничивающий работу с одним бакетом и общее количество потоков, которые могут одновременно работать с ConcurrentDictionary.

# Коллекции с C#

Актуальные сейчас коллекции из System.Collections.Generic:

* HashSet<T>;
* SortedSet<T>;
* Dictionary<TKey,TValue>;
* List<T>;
* LinkedList<T>;
* Queue<T>;
* Stack<T>;
* SortedList<TKey,TValue>.

## 

## HashSet<TKey,TValue>

HashSet — хеш-таблица, которую мы рассматривали на одном из предыдущих уроков.

Асимптотическая сложность на вставку, удаление, проверку наличия элемента — O(1).

Пример работы с этой структурой:

|  |
| --- |
| var hashSet = new HashSet<string>(); hashSet.Add("Santa"); *// добавление элемента* Console.WriteLine(hashSet.Contains("Santa")); |

## SortedSet<TKey,TValue>

Хранит уникальные значения в отсортированном виде. Удобен, когда данные нужно обрабатывать диапазонами в отсортированном порядке. Асимптотическая сложность на вставку, удаление и получение значения по ключу — O(Log(N)).

Пример работы с этой структурой:

|  |
| --- |
| var sortedSet = new SortedSet<int>(); sortedSet.Add(5); sortedSet.Add(99); sortedSet.Add(1); foreach (var item in sortedSet) {  Console.WriteLine(item); } |

## Dictionary<TKey,TValue>

Чуть выше мы уже упоминали Dictionary и его работу. Асимптотическая сложность на вставку, удаление и получение значения по ключу — O(1).

Пример работы с этой структурой:

|  |
| --- |
| var dictionary = new Dictionary<string, string>(); dictionary["Barbara"] = "Santa"; *// добавление или изменение элемента* dictionary.Add("Max", "Mad"); *// добавление нового элемента* |

## List<T>

По сути, это динамический массив, то есть его размер не строгий и может изменяться. Это достигается за счёт того, что внутри List<T> обычный массив ограниченного размера. В момент вставки, когда не хватает места во внутреннем массиве, List<T> создаёт новый массив и копирует туда все значения из старого. Обычный пользователь List<T> не видит это снаружи.

Также List<T> — амортизированный массив. Это значит, что при нехватке места во внутреннем массиве он не создаёт новый массив размера N + 1, а создаёт массив размера N \* 2, чтобы уменьшить количество пересозданий внутреннего массива при частых вставках.

Сложность вставки и удаления в конце массива — O(1) за счёт амортизации. Сложность вставки и удаления в середине, в начале и любом другом месте, если это не конец массива, — O(N). Сложность получения значения по индексу — O(1).

Пример работы с этой структурой:

|  |
| --- |
| var list = new List<int>(); list.Add(99); *// добавить новый элемент, добавление всегда происходит в конец* Console.WriteLine(list[0]); |

## LinkedList<T>

Это связный список, мы рассматривали его выше.

Пример работы с этой структурой:

|  |
| --- |
| var linkedList = new LinkedList<int>(); linkedList.AddLast(99); linkedList.AddFirst(10); var node = linkedList.Find(10); linkedList.AddAfter(node, 77); Console.WriteLine($"{node.Value}, {node.Next.Value}, {node.Next.Next.Value}"); |

## Queue<T>

Реализация очереди в C#, внутри используется массив. Время вставки и удаления — O(1).

Пример работы с этой структурой:

|  |
| --- |
| var queue = new Queue<int>(); queue.Enqueue(1); queue.Enqueue(2); var qItemOne = queue.Dequeue(); var qItemTwo = queue.Dequeue(); Console.WriteLine($"{qItemOne}, {qItemTwo} "); |

## Stack<T>

Реализация стека в C#, внутри используется массив. Время вставки и удаления — O(1).

Пример работы с этой структурой:

|  |
| --- |
| var stack = new Stack<int>(); stack.Push(1); stack.Push(2); var sItemOne = stack.Pop(); var sItemTwo = stack.Pop(); Console.WriteLine($"{sItemOne}, {sItemTwo} "); |

## SortedList<TKey,TValue>

Отсортированный массив. Сложность вставки — O(N), сложность поиска — O(Log(N)). Сложность поиска Log(N) достигается за счёт знания, что массив отсортирован, и использования бинарного поиска.

Пример работы с этой структурой:

|  |
| --- |
| var sortedList = new SortedList<int, string>(); sortedList.Add(5, "B"); sortedList.Add(99, "C"); sortedList.Add(1, "A");  Console.WriteLine(sortedList[99]);  foreach (var item in sortedList) {  Console.WriteLine(item); } |

# Threadsafe-коллекции

## Concurrent

Рассмотренные выше коллекции применимы, если одновременно к ним обращается только один поток. При переходе к многопоточным приложениям, где обращение на чтение и изменение происходит не из одного потока, стоит использовать соответствующие потокобезопасные коллекции.

* BlockingCollection<T> — потокобезопасная коллекция producer-consumer, по функционалу аналогична очереди;
* ConcurrentBag<T> — потокобезопасный неупорядоченный набор элементов;
* ConcurrentDictionary<TKey,TValue> — потокобезопасный Dictionary;
* ConcurrentQueue<T> — потокобезопасная очередь;
* ConcurrentStack<T> — потокобезопасный стек.

## 

## Immutable

Стоит заметить, что если коллекция не меняет или не может менять своё внутреннее состояние в процессе работы, то при работе в многопоточном приложении все операции происходят на чтение. Соответственно, такая коллекция будет потокобезопасной. Основные проблемы возникают в тот момент, когда обычную коллекцию один поток пытается изменить, а другой — прочитать или тоже изменить.

Если коллеция статична и не изменяется, то для добавления туда чего-либо мы можем создать новую коллекцию с изменённым состоянием и просто поменять ссылку в переменной. В таком случае, если кто-то уже вошёл в коллекцию и ищет там что-то, он не получит блокировок и сломанных состояний, а закончит свою работу корректно. Тот, кто только собирается обратиться к коллекции, получит уже новую изменённую версию, потому что ссылка поменялась.

Чтобы не реализовывать такую логику изменения самостоятельно, в C# есть immutable-коллекции:

* ImmutableArray<T> — массив;
* ImmutableDictionary<TKey,TValue> — Dictionary;
* ImmutableList<T> — список с доступом по индексу;
* ImmutableQueue<T> — очередь;
* ImmutableSortedDictionary<TKey,TValue> — отсортированный Dictionary;
* ImmutableSortedSet<T> — отсортированная структура, которая содержит уникальные значения;
* ImmutableStack<T> — стек.

# Интерфейсы

Есть ряд интерфейсов, которые дают такой же контракт взаимодействия, как в одноимённой структуре. Также существуют интерфейсы только для чтения. ReadOnly-интерфейсы следует использовать, чтобы определить, что метод, который использует коллекцию, не будет её менять. Это удобно для ограничения вызываемых функций.

* ICollection<T> — общий интерфейс коллекций;
* IDictionary<TKey,TValue> — общий интерфейс для структур ключ-значение;
* IEqualityComparer<T> — интерфейс, который явно указывает на наличие реализации GetHashCode и Equals;
* IList<T> — интерфейс коллекции с функциями вставки, удаления и получения элемента по индексу;
* ISet<T> — общий интерфейс структур, которые содержат уникальные значения;
* IReadOnlyCollection<T> — вариант ICollection<T>, но только для чтения;
* IReadOnlyDictionary<TKey,TValue> — IDictionary<TKey,TValue>, но только для чтения;
* IReadOnlyList<T> — вариант IList<T>, но только для чтения;
* IReadOnlySet<T> — вариант ISet<T>, но только для чтения.

IEnumerator — универсальный интерфейс для прохода по коллекции. Имеет методы MoveNext(), Reset() и свойство Current. В C# этот интерфейс реализуют все коллекции, это позволяет их использовать в foreach- и LINQ-выражениях.

# 

# Заключение

Мы узнали, что стек — это LIFO-структура данных, а очередь — это FIFO-структура. DFS использует стек, а BFS — очередь. Также узнали о коллекциях:

* HashSet<T>;
* SortedSet<T>;
* Dictionary<TKey,TValue> ;
* List<T>;
* LinkedList<T>;
* Queue<T>;
* Stack<T>;
* SortedList<TKey,TValue>.

Выяснили, что есть альтернативные коллекции для многопоточных приложений: изменяемые и неизменяемые. Неизменяемые при попытке их изменить всегда возвращают ссылку на изменённую копию структуры.

# 

# Практическое задание

Реализуйте DFS и BFS для дерева с выводом каждого шага в консоль.

# Дополнительные материалы

[Коллекции в C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/collections).

# Используемая литература

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. Д. Рихтер CLR via C# М.: Питер, 2019.