When to Use a Particular Data Structure?

Array (T[])

The arrays are collections of fixed numbers of elements from a given type (for example numbers) where the elements preserved their order. Each element can be accessed through its index. The arrays are memory areas, which have a predefined size.

Adding a new element in an array is a slow operation. To do this we have to allocate a memory with the same size plus one and copy all the data from the original array to the new one.

Searching in an array takes time because we have to compare every element to the searched value. It takes N/2 comparisons in the average case.

Removing an element from an array is a slow operation. We have to allocate a memory with the same size minus one and copy all the old elements except the removed one.

Accessing by index is direct, and thus, a fast operation.

The arrays should be used only when we have to process a fixed number of elements to which we need a quick access by index. For example, if we have to sort some numbers, we can keep them in an array and then apply some of the well-known sorting algorithms. If we have to change the elements' count, the array is not the correct data structure we should use.

~Use arrays when you have to process a fixed number of elements to which you need an access through index.

Singly / Doubly Linked List (LinkedList<T>)

Singly and doubly linked lists hold collections of elements, which preserve their order. Their representation in the memory is dynamic, pointer-based. They are linked sequences of elements.

Adding is a fast operation but it is a bit slower than adding to a List<T> because every time when we add an element to a linked list we allocate a new memory area. The memory allocation works at speed, which cannot be easily predicted.

Searching in a linked list is a slow operation because we have to traverse through all of its elements.

Accessing an element by index is a slow operation because there is no indexing in singly and doubly linked lists. You have to go through all the elements from the start one by one instead.

Removing an element at a specified index is a slow operation because reaching the element through its index is a slow operation. Removing an element with a specified value is a slow operation too, because it involves searching.

Linked list can quickly add and remove elements (with a constant complexity) at its two ends (head and tail). Hence, it is very handy for an implementation of stacks, queues and similar data structures.

Linked lists are rarely used in practice because the dynamic arrays (List<T>) can do almost exactly the same operations LinkedList does, plus for most of them it works faster and more comfortably.

Dynamic Array (List<T>)

Dynamic array (List<T>) is one of the most popular data structures used in programming. It does not have fixed size like arrays, and allows direct access through index, unlike linked lists (LinkedList<T>). The dynamic array is also known as "array list", "resizable array" and "dynamic array".

List<T> holds its elements in an array, which has a bigger size than the count of the stored elements. Usually when we add an element, there is an empty cell in the list's inner array. Therefore this operation takes a constant time. Occasionally the array has been filled and it has to expand. This takes linear time, but it rarely happens. If we have a large amount of additions, the average-case complexity of adding an element to List<T> will be constant – O(1). If we sum the steps needed for adding 100,000 elements (for both cases – "fast add" and "add with expand") and divide by 100,000, we will obtain a constant which will be nearly the same as for adding 1,000,000 elements.

This statistically-averaged complexity calculated for large enough operations is called amortized complexity. Amortized linear complexity means that if we add 10,000 elements consecutively, the overall count of steps will be of the order of 10,000. In most cases, adding it will execute in a constant time, while very rarely adding will execute in linear time.

Searching in List<T> is a slow operation because you have to traverse through all the elements.

Removing by index or value executes in a linear time. It is a slow operation because we have to move all the elements after the deleted one with one position to the left.

The indexed access in List<T> is instant, in a constant time, since the elements are internally stored in an array.

Practically List<T> combines the best of arrays and lists, for which it is a preferred data structure in many situations. For example if we have to process a text file and to extract from it all words (with duplicates), which match a regular expression, the most suitable data structure in which we can accumulate them is List<T>, because we need a list, the length of which is unknown in advance and can grow dynamically.

The dynamic array (List<T>) is appropriate, when we have to add elements frequently as well as keeping their order of addition and access them through index. If we often we have to search or delete elements, List<T> is not the right data structure.

Stack

Stack is a linear data structure in which there are 3 operations defined: adding an element at the top of the stack (push), removing an element from the top of the stack (pop) and inspect the element from the top without removing it (peek). All these operations are very fast – it takes a constant time to execute them. The stack does not support the operations search and access through index.

The stack is a data structure, which has a LIFO behavior (last in, first out). It is used when we have to model such a behavior – for example, if we have to keep the path to the current position in a recursive search.

Queue

Queue is a linear data structure in which there are two operations defined: adding an element to the tail (enqueue) and extract the front-positioned element from the head (dequeue). These two operations take a constant time to execute, because the queue is usually implemented with a linked list. We remind that the linked list can quickly add and remove elements from its both ends.

The queue's behavior is FIFO (first in, first out). The operations searching and accessing through index are not supported. Queue can naturally model a list of waiting people, tasks or other objects, which have to be processed in the same order as they were added (enqueued).

As an example of using a queue we can point out the implementation of the BFS (breadth-first search) algorithm, in which we start from an initial element and all its neighbors are added to a queue. After that they are processed in the order they were added and their neighbors are added to the queue too. This operation is repeated until we reach the element we are looking for or we process all elements.

Dictionary, Implemented with a Hash-Table (Dictionary<K, T>)

The data structure "dictionary" suggests storing key-value pairs and provides a quick search by key. The implementation with a hash table has a very fast add, search and remove of elements – constant complexity at the average case. The operation access through index is not available, because the elements in the hash-table have no order, i.e. an almost random order.

Dictionary<K,T> keeps internally the elements in an array and puts every element at the position calculated by the hash-function. Thus the array is partially filled – in some cells there is a value, others are empty. If more than one element should be placed in a single cell, elements are stored in a linked list. It is called chaining. This is one of the few ways to resolve the collision problem. When the load factor exceeds 75%, the size is doubled and all the elements occupy new positions. This operation has a linear complexity, but it is executed so rarely, that the amortized complexity remains a constant.

Hash-table has one peculiarity: if we choose a bad hash-function causing many collisions, the basic operations can become very inefficient and reach linear complexity. In practice, however, this hardly happens. Hash-table is considered to be the fastest data structure, which provides adding and searching by key.

From time to time one key will have to keep multiple values. This is not standardly supported but we can store the values matching this key in a List<T> as a sequence of elements. For example if we need a hash-table Dictionary<int, string>, in which to accumulate pairs {integer, string} with duplicates, we can use Dictionary<int, List<string>>. Some external libraries have ready to use data structures called MultiDictionary<K,V>.

Hash-table is recommended to be used every time we need fast addition and fast search by key. For example if we have to count how many times each word is encountered in a set of words in a text file, we can use Dictionary<string, int> – the key will be a particular word, the value – how many times we have seen it.

Use a hash-table, when you want to add and search by key very fast.

The main advantage of the hash-table over the other data structures is a very quick searching and addition. The comfort for the developers is a secondary factor.

Use HashSet<T>, when you have to quickly add elements to a set and check whether a given element belongs to a set.

Set, Implemented with a Balanced Tree (SortedSet<T>)

The data structure set, implemented with a red-black tree, is a special case of SortedDictionary<K,T> in which keys and values coincide.

Similar to SortedDictionary<K,T>, the basic operations in SortedSet<T> are executed with logarithmic complexity O(log(N)), which is the same in the average and worst case.

As an example of using a SortedSet<T> we can point out the task of finding all the different words in a given text file and printing them alphabetically ordered.

Use SortedSet<T>, when you have to quickly add an element to a set and check whether given element belongs to the set as well as need all the elements sorted in ascending order.