MskDotNet #HNY2018-2019: Algorithms and Data Structures in C#

Elias Fofanov

ENGINEER SPOCK

http://engineerspock.com

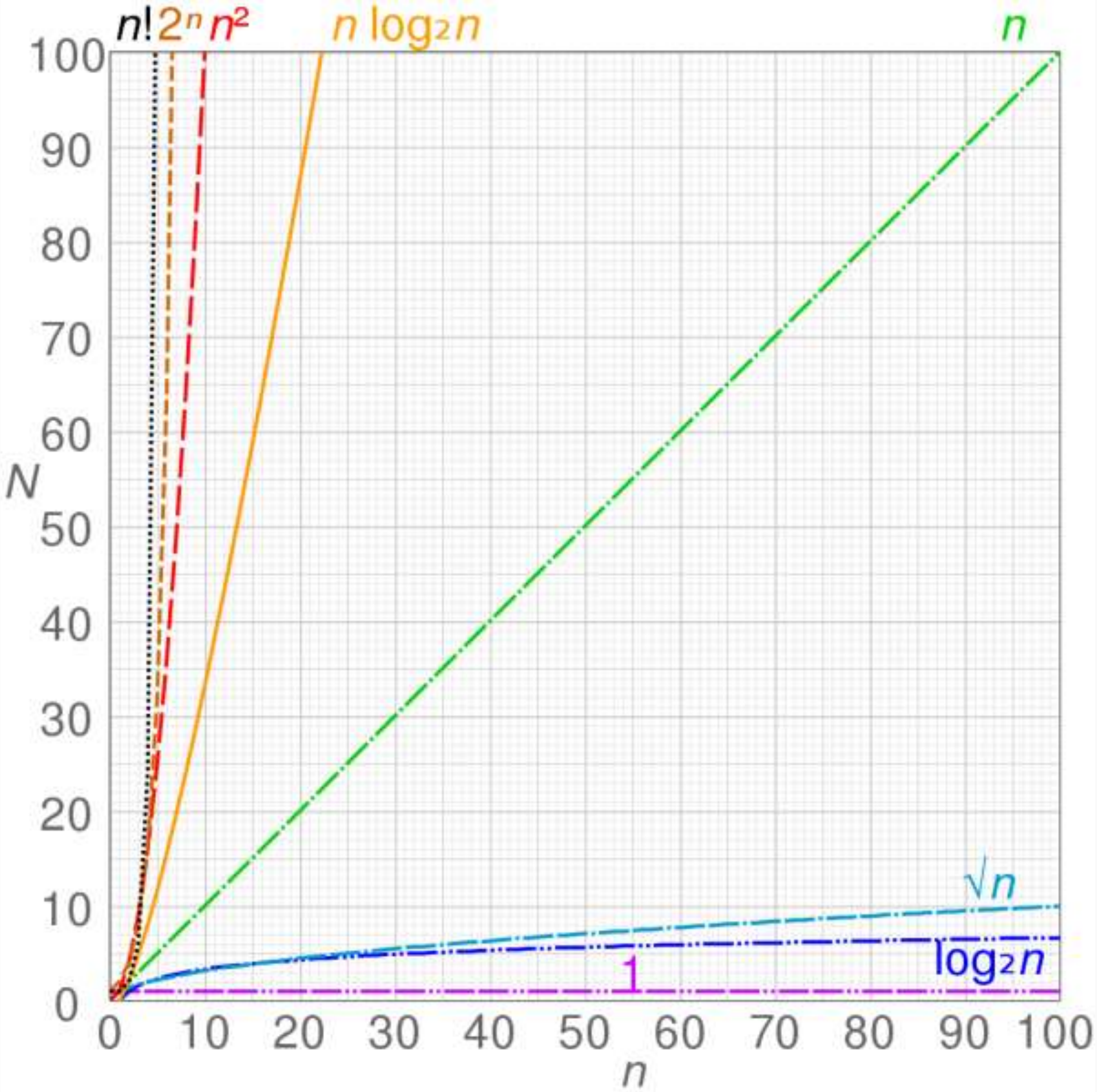
Why learn algorithms and data structures?

- If you're not good at algorithms and data structures,
  you'll never pass a coding interview in a decent company
- Better hardware is not a solution
- Understand what's going on under the hood

Algorithm Analysis

- how much time will our algorithm take for solving a problem?
- how much memory will our algorithm consume for solving a problem?

Running Time Complexities



Topics to Discuss

- Array.Sort
- Lists
- Stack and Queue
- Hashing
- Collisions
- Dictionaries: Dictionary, SortedList, SortedDictionary
- Sets: HashSet, SortedSet

Be Careful Even with Classic Algorithms

- Sure you can implement “trivial” Binary Search without bugs?

First binary search paper was published in 1946;
first binary search that works correctly for all values
of n appeared only in 1962

Be Careful Even with Classic Algorithms

- Bug in Java’s Array.binarySearch() discovered in 2006!
  (an integer overflow bug when calculating the midpoint
  of the range that you’re dividing the search over)
- QuickSort took N^2 in too many cases in the C-implementation (1990).
  In 1990 it has already been passed about 31 years since the invention of QSort!
- Reimplementing MergeSort you can make it unstable, simply by using “<=" (">=”)
  instead of “<" (">“) when comparing items

Array.Sort<T>

- if T is primitive -> TrySZSort() – native implementation
- if T is ref type ->

```
if(platform == .NET Core || platform >= .NET Framework 4.5)
{
  //combination of insertion sort, heap sort, QSort
  IntroSort();
}
else
{
   //actually IntroSort as well
   //QSort with 32-max recursion depth, if exceeded switches to HeapSort
   DepthLimitedQuickSort();
}
```

Array.Sort<T>

- Array.Sort demonstrates the following time complexity:
  $\theta(nlogn)$ **linearithmic on average**
  $O(n^2)$ quadratic – worst case

Какой алгоритм использует Array.Sort()?

- Quick Sort
- Intro Sort
- Heap Sort
- Зависит

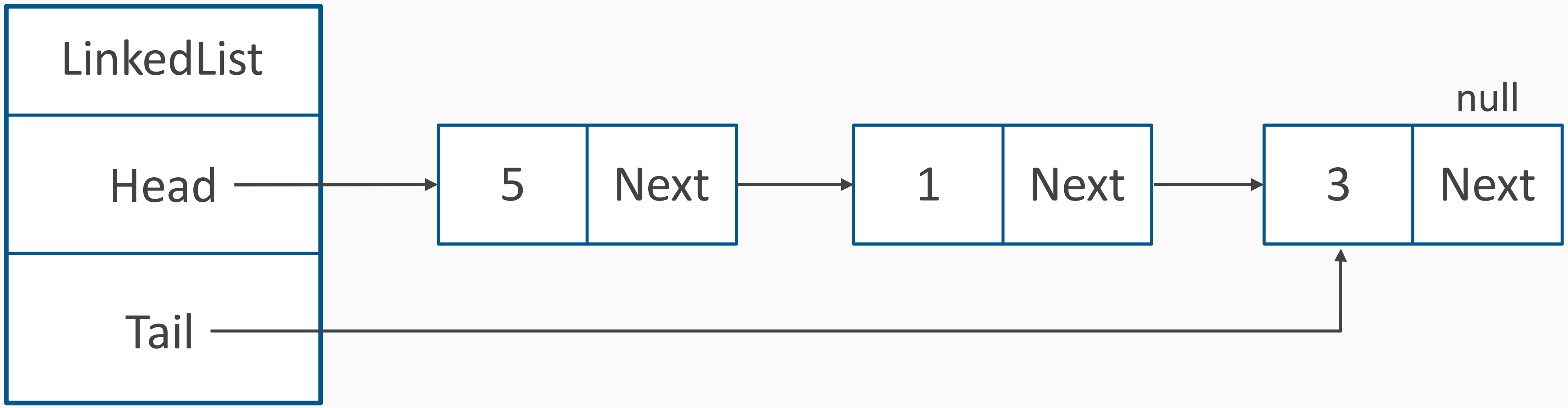
Shell Sort

- Based on Insertion Sort
- Insertion Sort is fast on pre-sorted arrays
- **Basic Idea:** pre-sort the input and switch to Insertion Sort
- Gap is used for pre-sorting => swap distant elements
- Shell Sort starts with a “large” gap and gradually reduces it
- When gap = 1, Insertion Sort finishes the sorting process

Shell Sort

- **In-place algorithm:**
  uses a small amount of extra memory (doesn't depend on *n*)
- **Unstable**
- $O(n^{3/2})$ time complexity (if sequence is $({}^{1}/_{2}\,(3^k - 1))$
  Can be even $O(n^{6/5})$

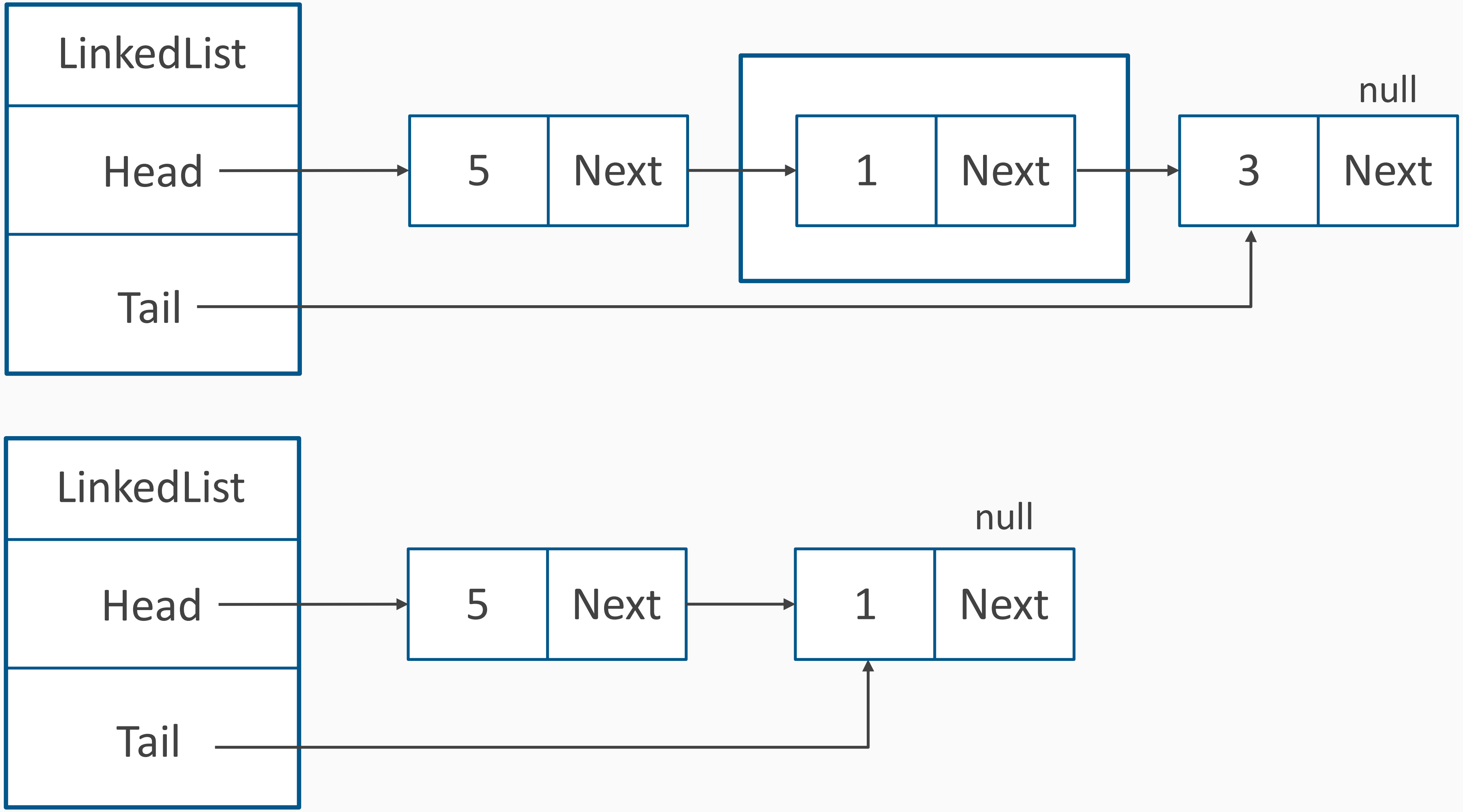
Singly-Linked List



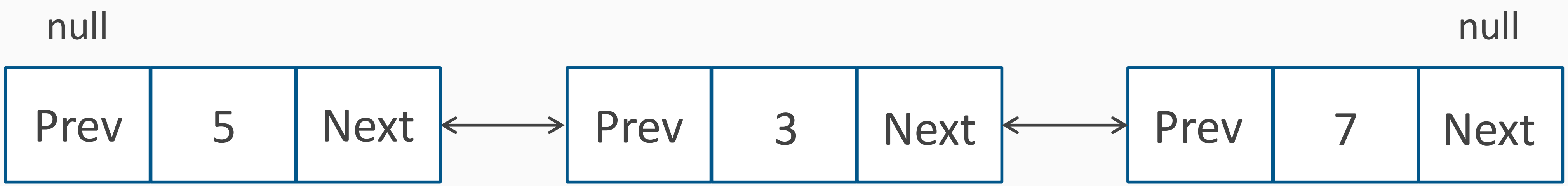
Какая операция в двусвязном списке работает существенно быстрее, чем в односвязном?

- ▲ AddFirst
- ◆ AddLast
- ● RemoveFirst
- ■ RemoveLast

Singly-Linked List - RemoveLast



Doubly-Linked List



LinkedList

- Doubly-Linked Circular List
- AddFirst/AddLast – O(1)
  AddBefore/AddAfter – O(1)
  (if you know the node, otherwise you'll have to search at first for O(N))
- Remove – O(N) - searching
- RemoveFirst/RemoveLast – O(1)
- Contains, Find/FindLast – O(N) – have to traverse N nodes

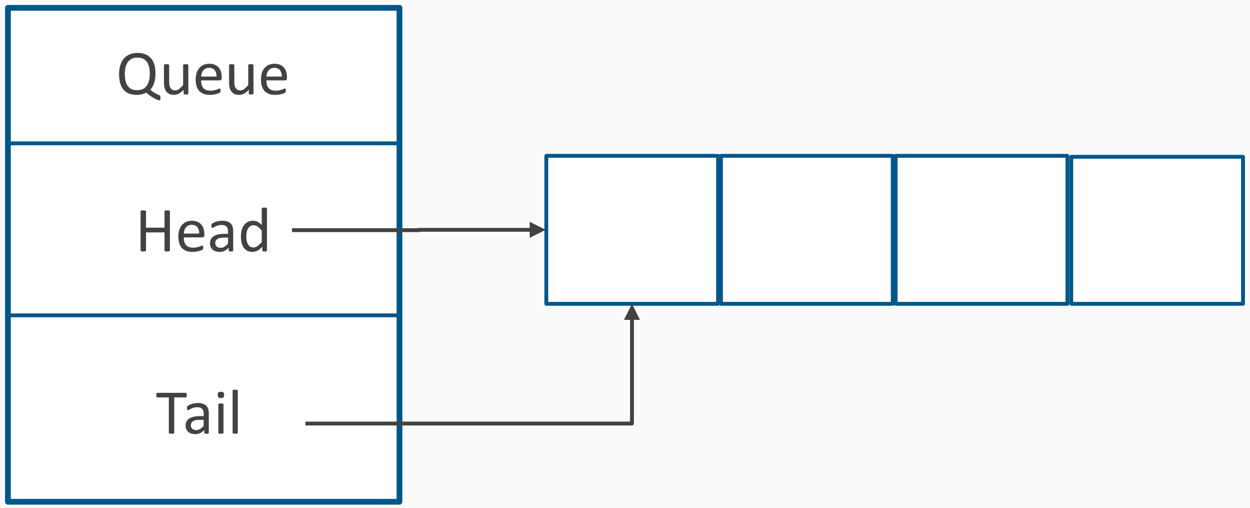
Stack

- Peek works for O(1) in any cases
- If backed up by a LinkedList:
  Push/Pop work for O(1)
- If backed up by an array, then Push/Pop:
  - if enough space Push – O(1)
  - If not enough space, Push - O(N) – resizing array
  - Pop works for O(1) if we never shrink array; O(N) when shrinking
- if there’s enough memory on a device, or the max number of items is not known
  -> linked list is preferable as a backing data structure
- if not enough memory or the max number of items is known
  -> array is preferable as a backing data structure

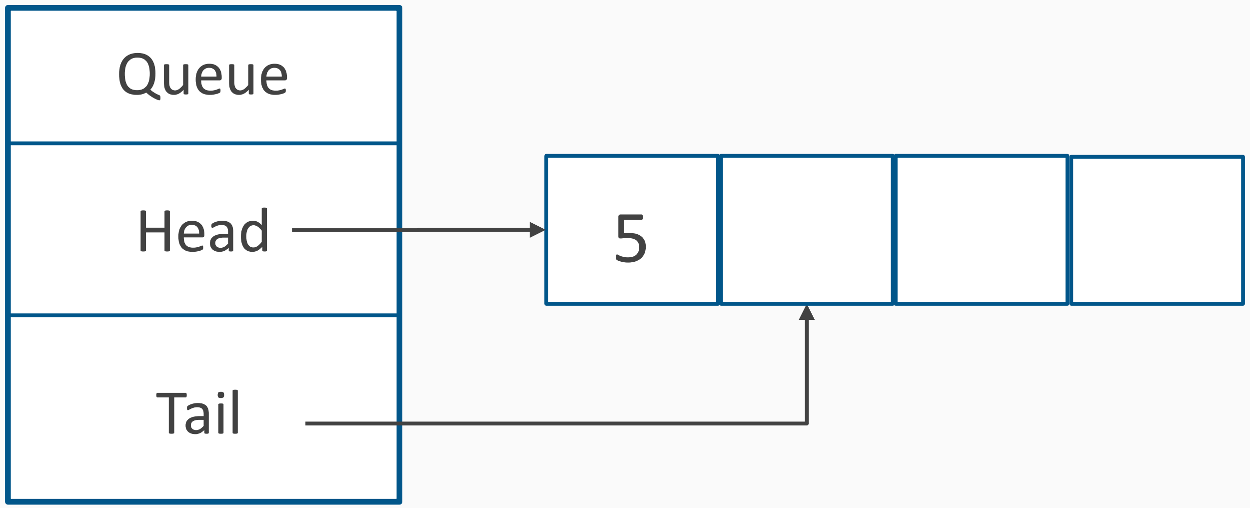
Queue

- Peek works for O(1) in any cases
- If backed up by a LinkedList:
  Enqueue/Dequeue work for O(1)
- If backed up by an array, then Enqueue/Dequeue:
  - if enough space Enqueue – O(1)
  - If not enough space, Enqueue - O(N) – resizing array
  - Dequeue works for O(1) if we never shrink array; O(N) when shrinking
- if there's enough memory on a device, or the max number of items is not known
  -> linked list is preferable as a backing data structure
- if not enough memory or the max number of items is known
  -> array is preferable as a backing data structure

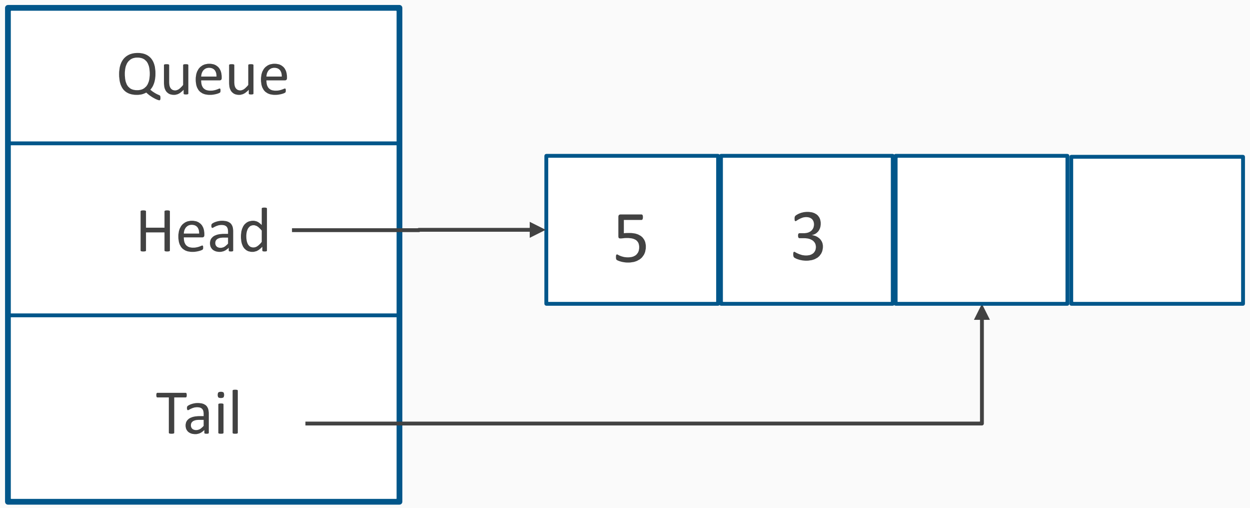
Queue



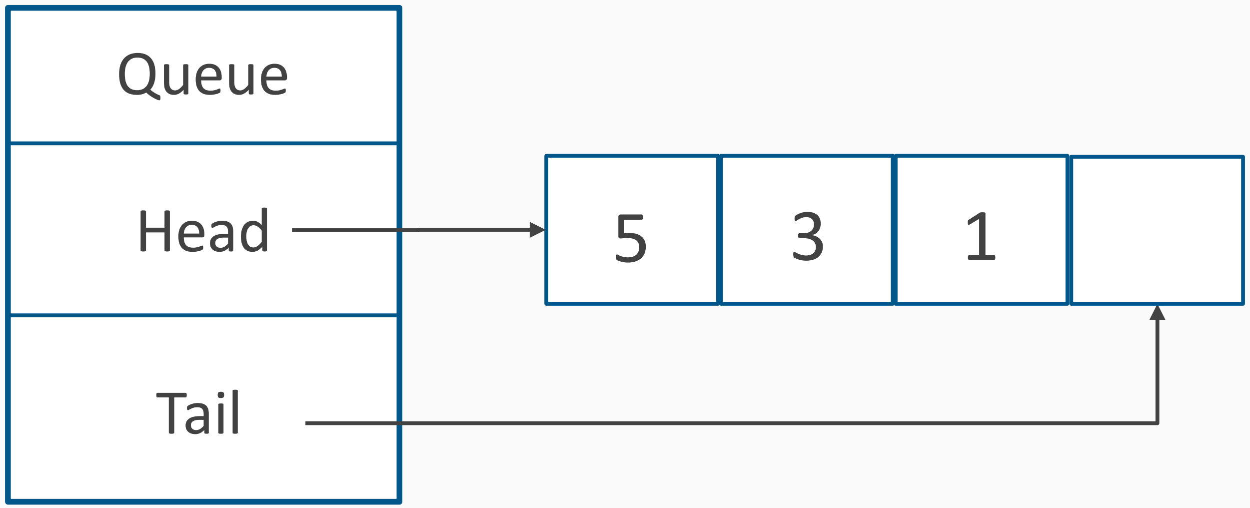
Queue



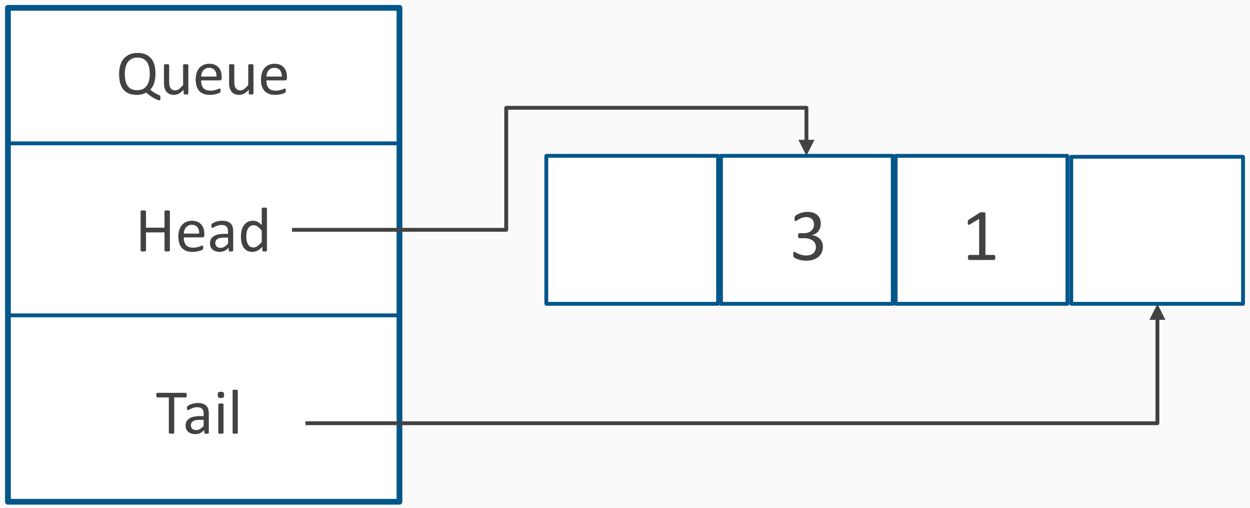
Queue



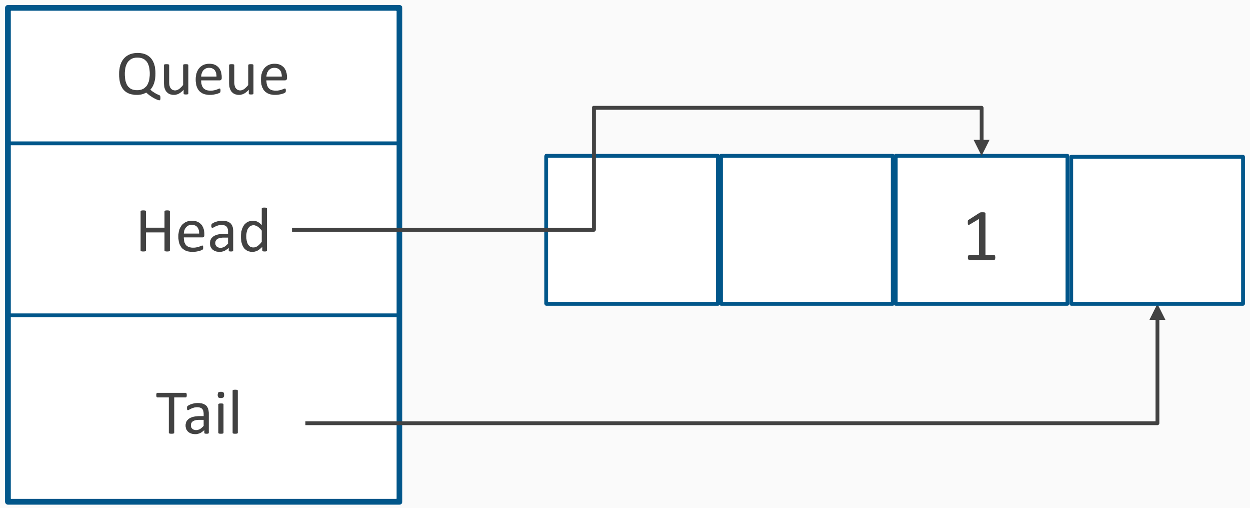
Queue



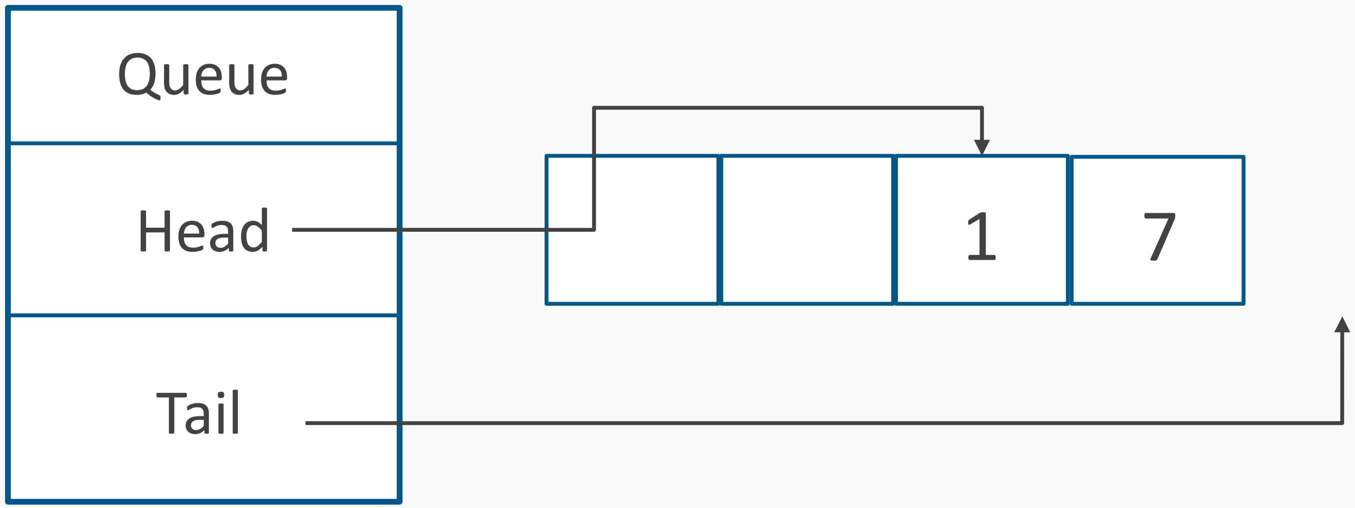
Queue



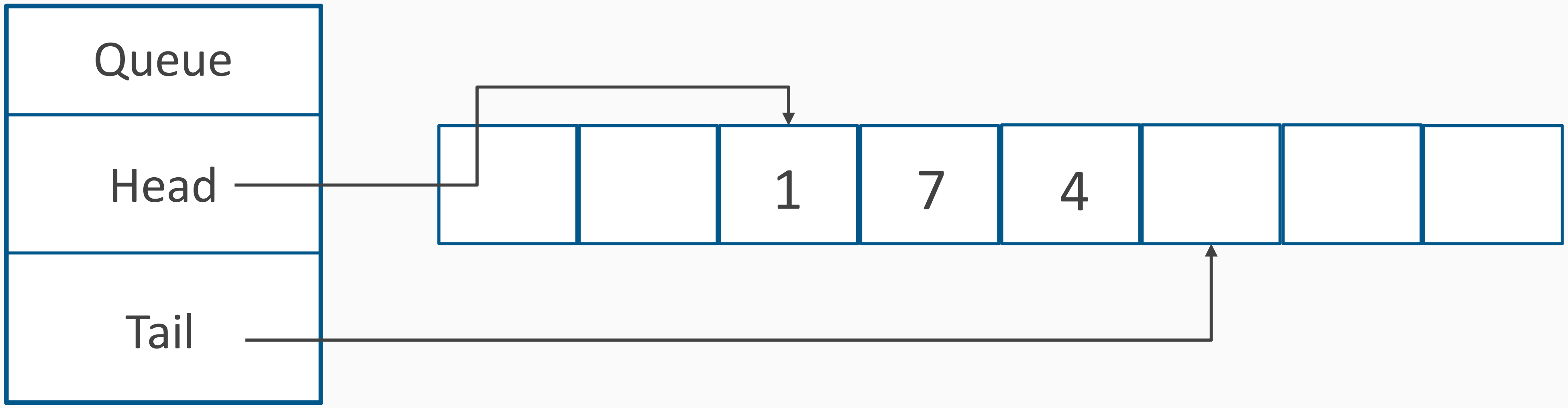
Queue



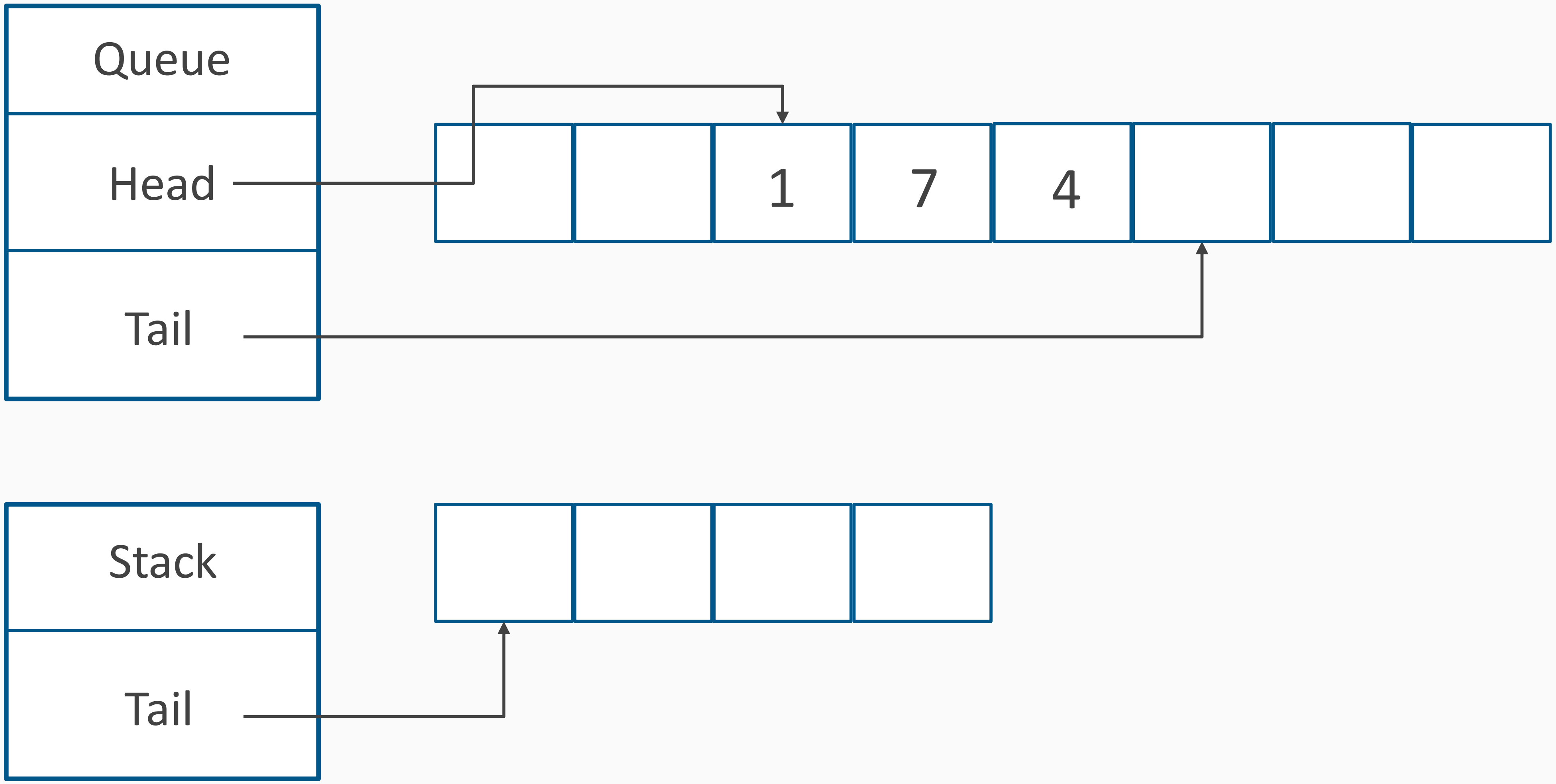
Queue



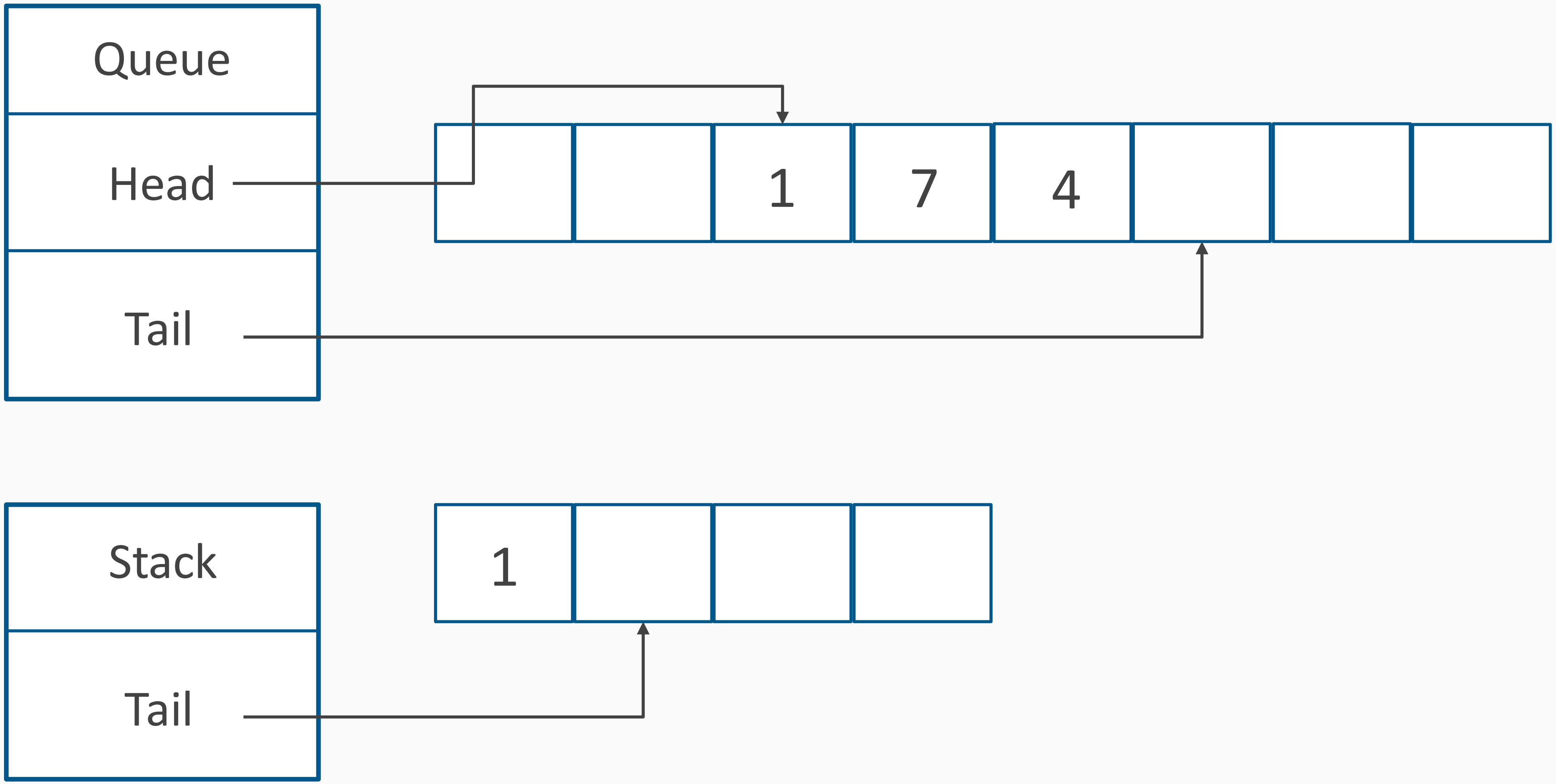
Queue



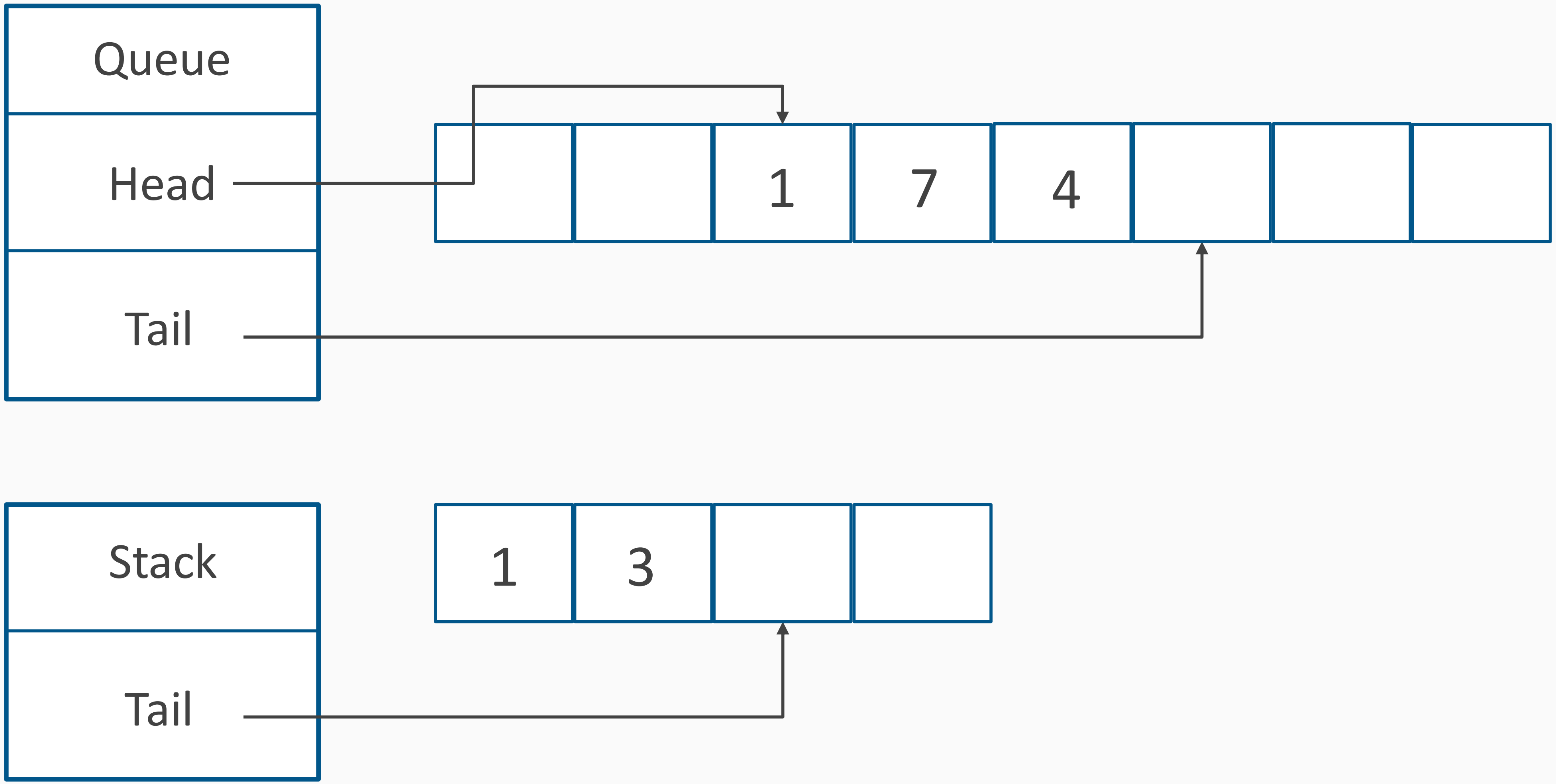
Queue



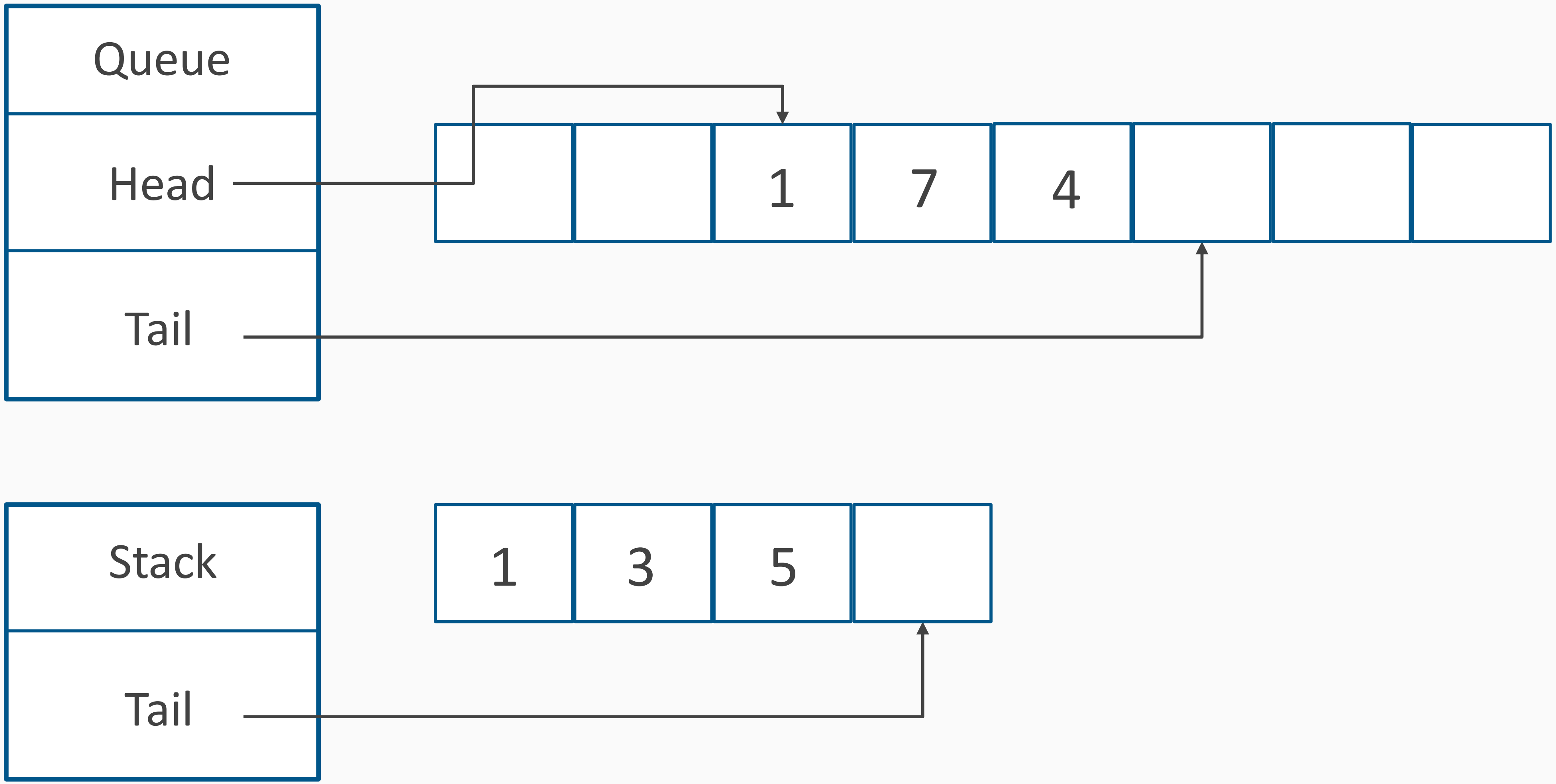
Queue



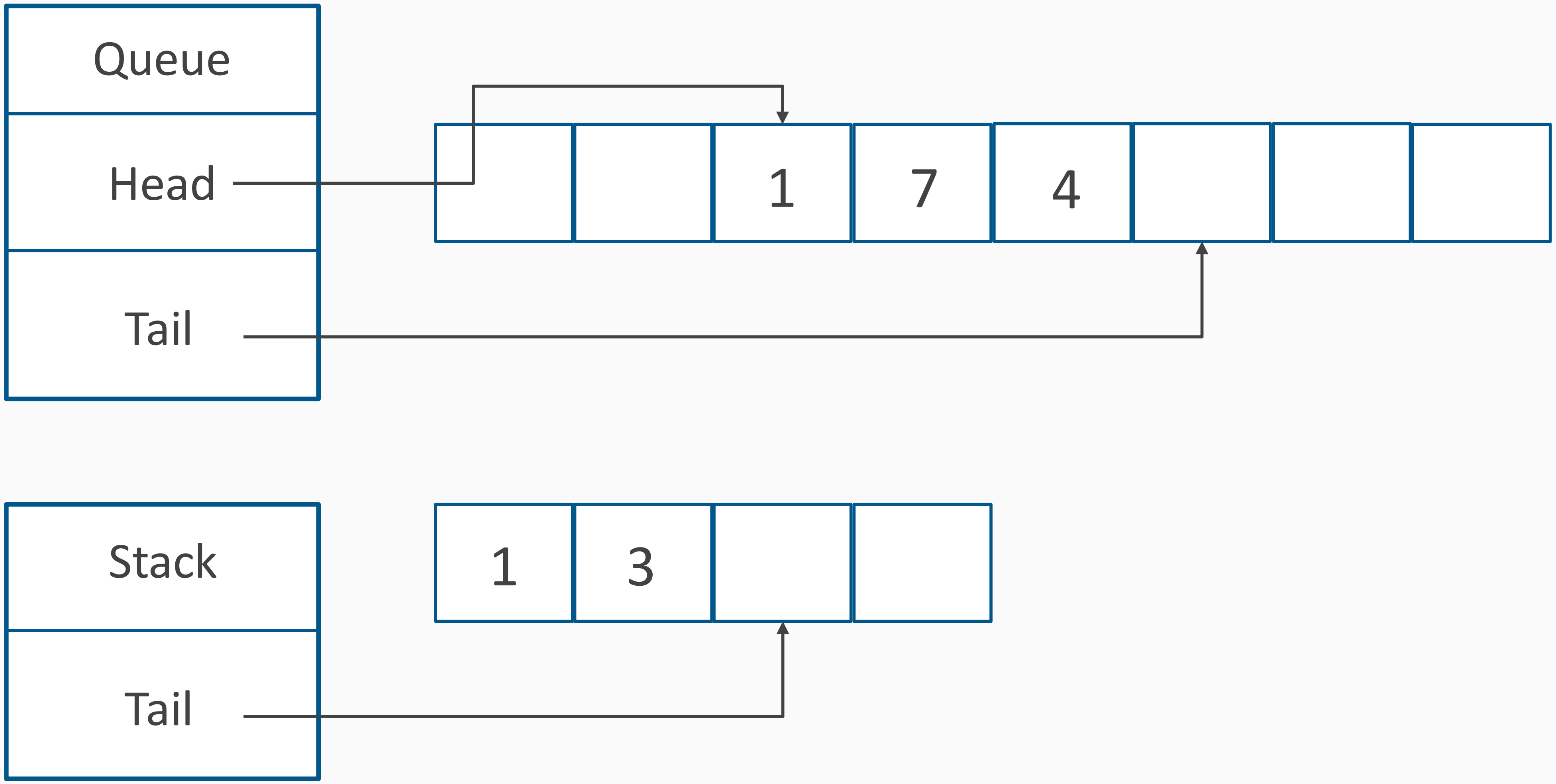
Queue



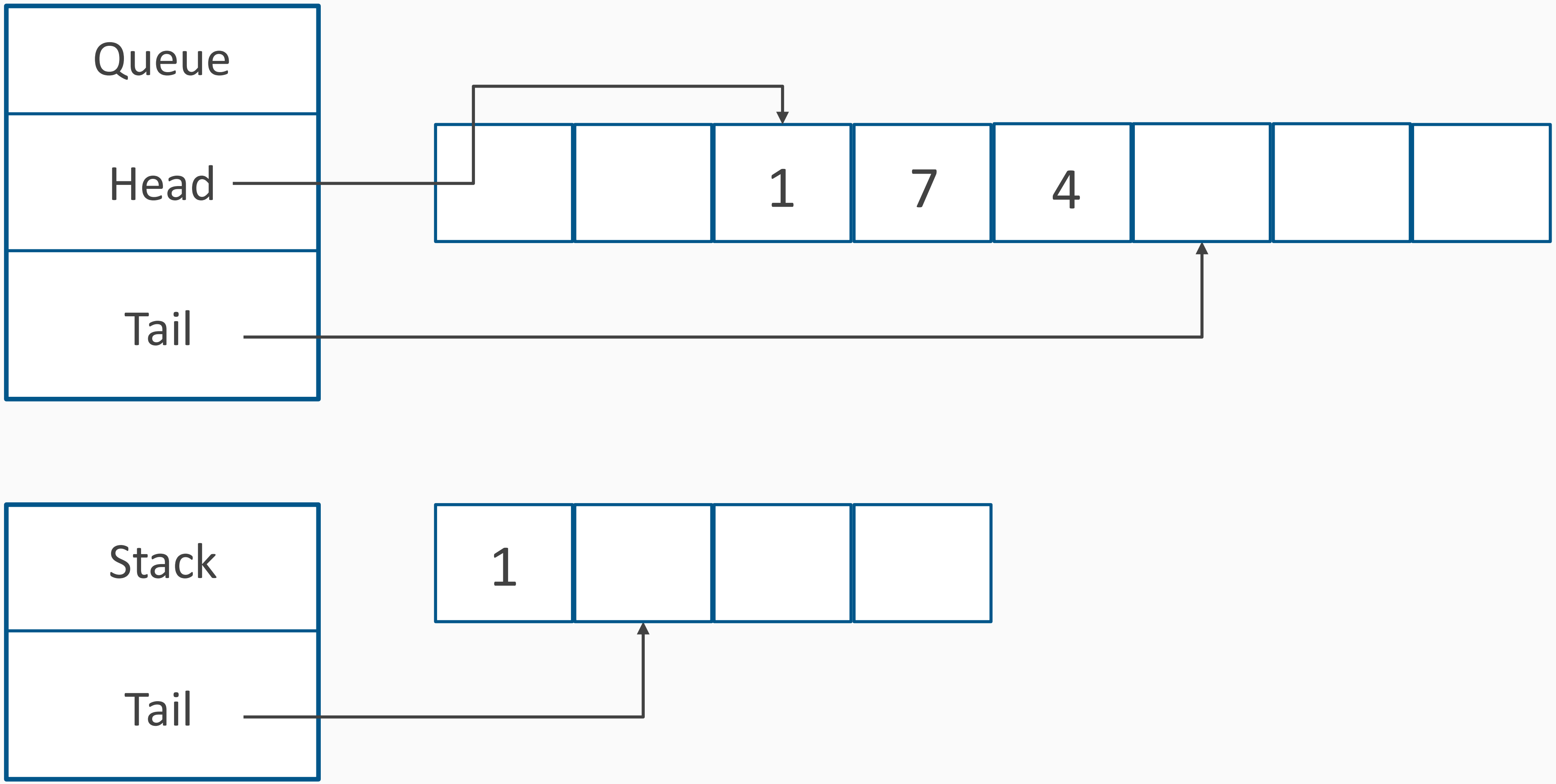
Queue



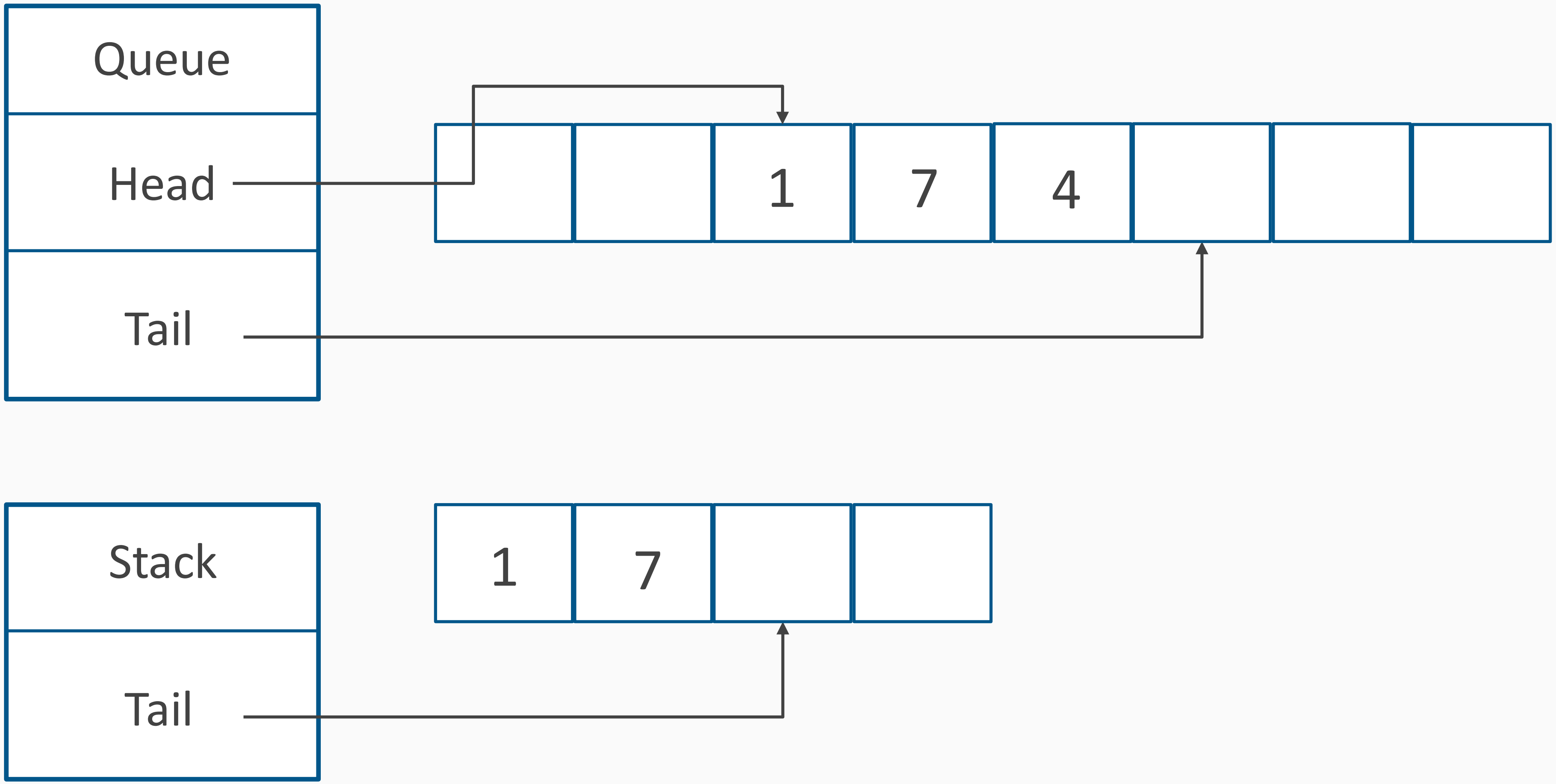
Queue



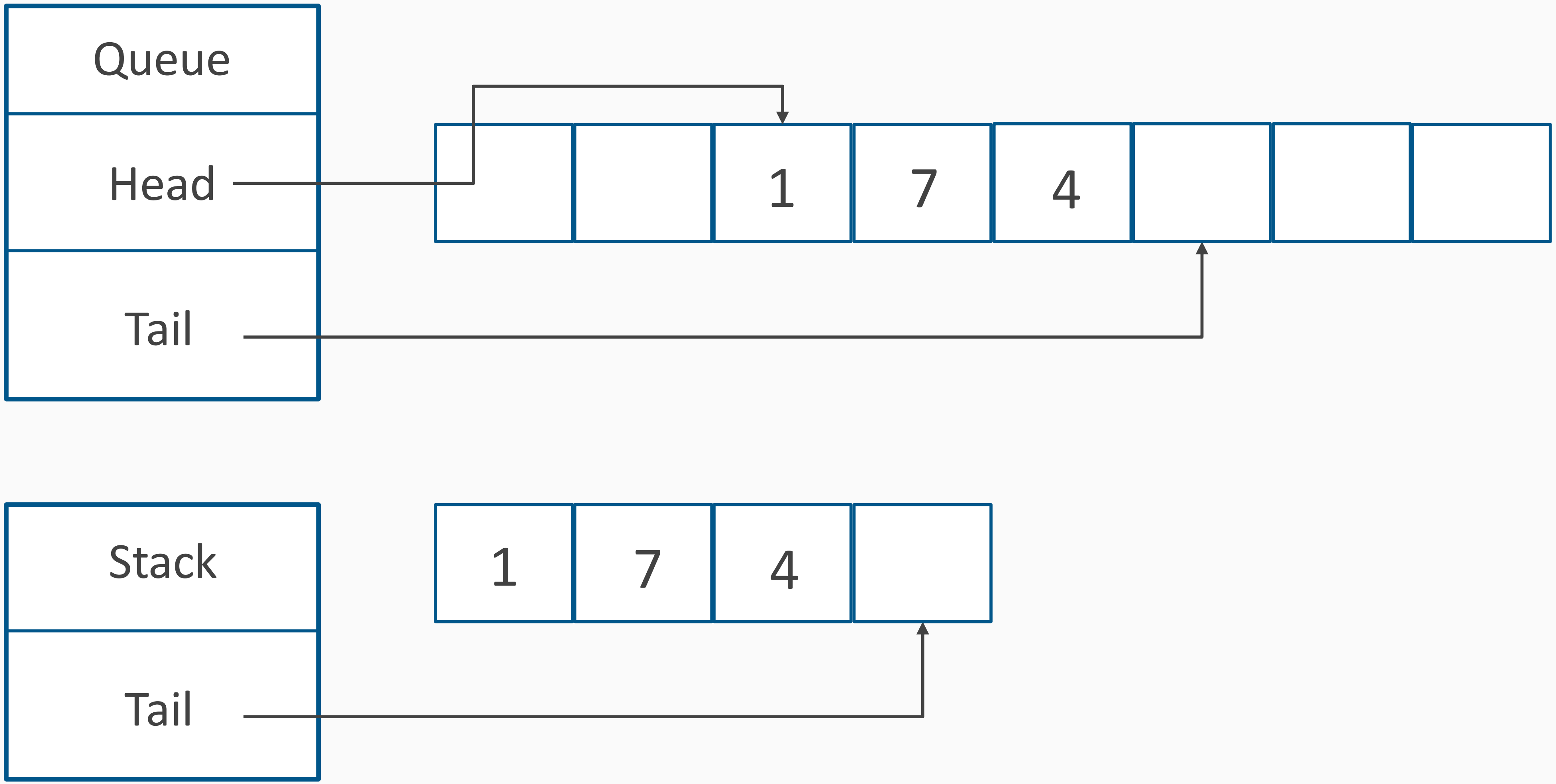
Queue



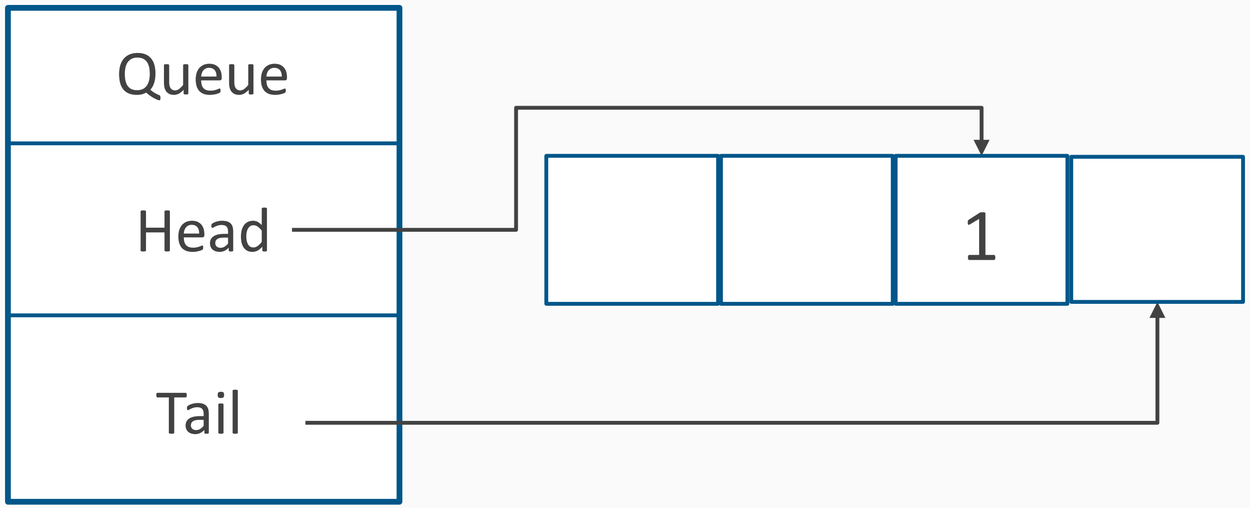
Queue



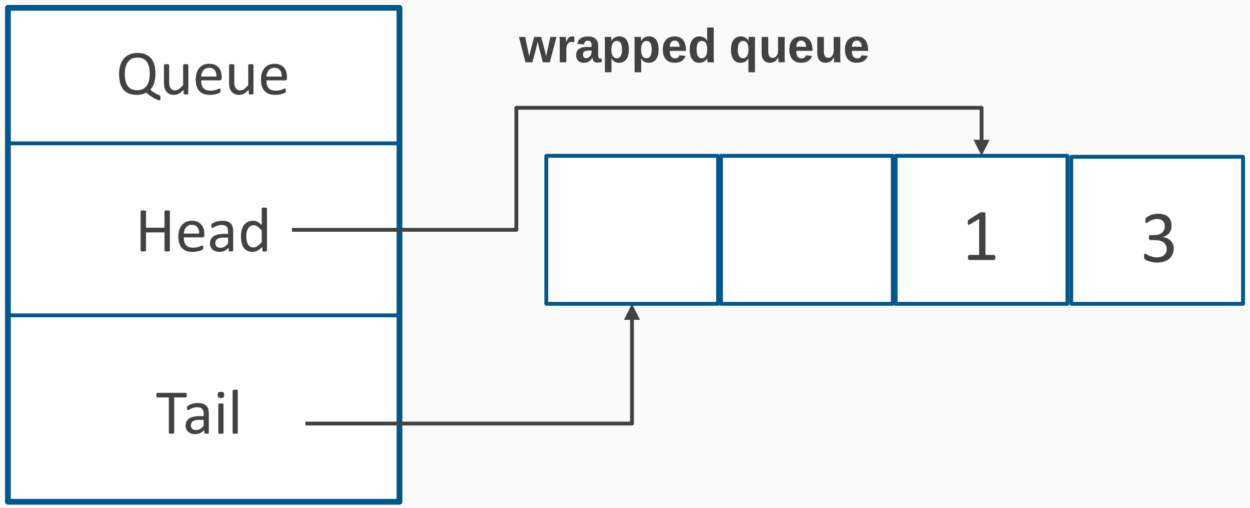
Queue



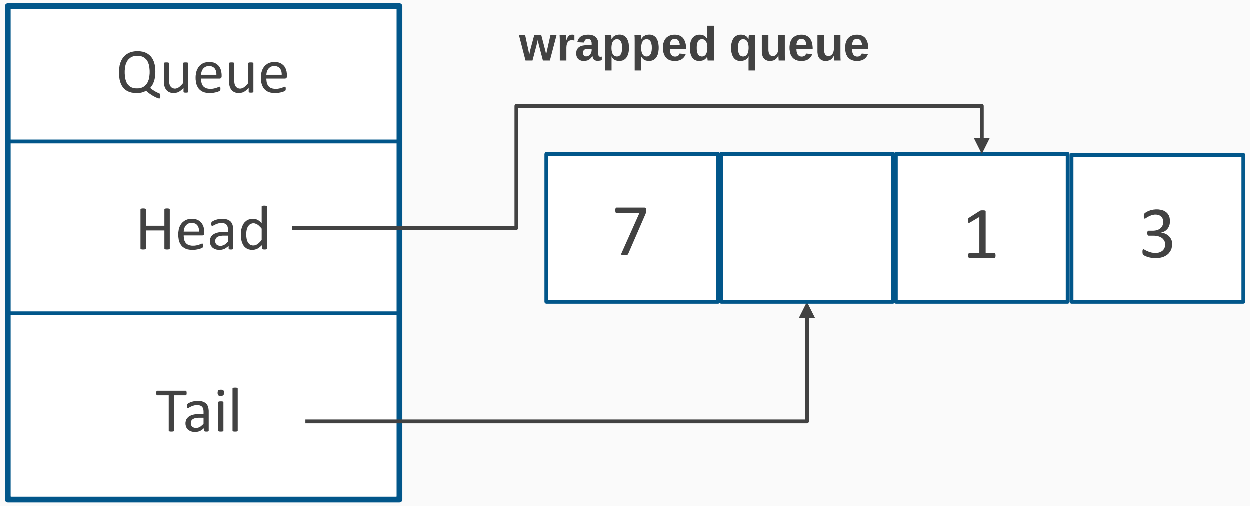
Circular Queue



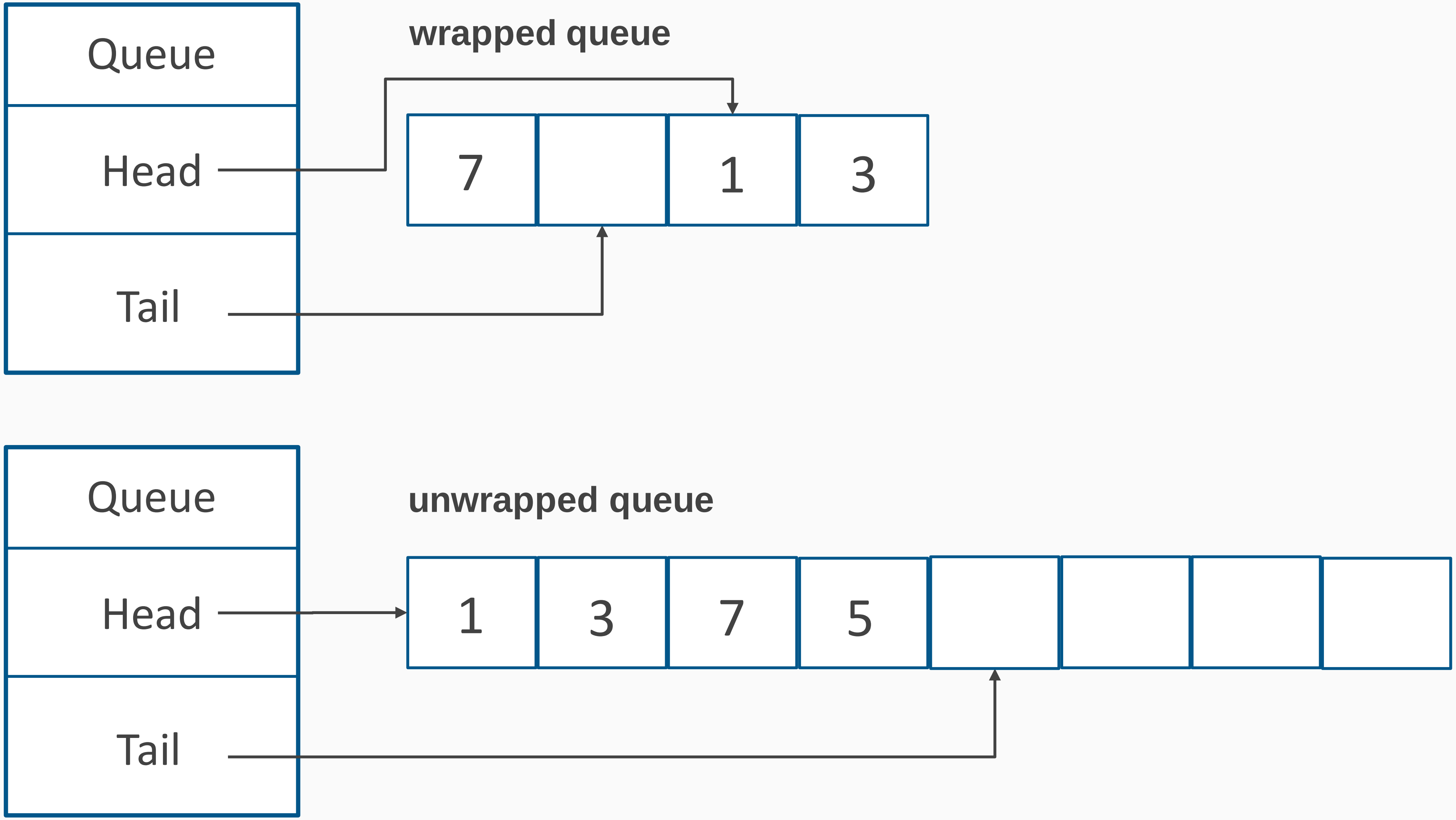
Circular Queue



Circular Queue



Circular Queue



Priority Queue

- Priority Queue is a Queue where items are weighted
- No built-in implementation in BCL
  check out here:
  https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp
  (stable priority queue implementation)

List<T>

```
[Serializable]
public class List<T> : IList<T>, System.Collections.IList, IReadOnlyList<T>
{
    private const int _defaultCapacity = 4;

    private T[] _items;
    [ContractPublicPropertyName("Count")]
    private int _size;
    private int _version;
    [NonSerialized]
    private Object _syncRoot;

    static readonly T[]  _emptyArray = new T[0];

    // Constructs a List. The list is initially empty and has a capacity
    // of zero. Upon adding the first element to the list the capacity is
    // increased to 16, and then increased in multiples of two as required.
    public List() {
        _items = _emptyArray;
    }
```

List<T>

```
public int Capacity {
    get {
        Contract.Ensures(Contract.Result<int>() >= 0);
        return _items.Length;
    }
    set {
        if (value < _size) {
            ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArgument.value,
        }
        Contract.EndContractBlock();

        if (value != _items.Length) {
            if (value > 0) {
                T[] newItems = new T[value];
                if (_size > 0) {
                    Array.Copy(_items, 0, newItems, 0, _size);
                }
                _items = newItems;
            }
            else {
                _items = _emptyArray;
            }
        }
    }
}
```

List<T>

- Backed up by an array internally
- Add – O(1) if enough space, O(N) if not enough
- Remove – O(N) – search + RemoveAt
- RemoveAt – O(N) - shifting
- Contains, IndexOf etc. – O(N) – have to traverse N elements
- Sort drills down to Array.Sort<T>
- TrimExcess for O(N)
- DO NOT USE ArrayList.
  Use List<object> instead.

На какой структуре данных базируется тип List<T> из BCL?

- Односвязный список
- Двусвязный список
- Массив
- Развёрнутый связный список

Сужается ли массив под List<T> если удалено более 50% элементов от Capacity?

- Да
- Нет
- Зависит
- Не знаю (зато честно)

Symbol Tables

- Fast access to information is almost the required condition for our existence nowadays. We need data structures which allow both extremely fast insertion and retrieval
- Symbol Table allows to add a value using a key and then retrieve that data by the key
- We often refer to symbol tables as to dictionaries
- Four ways of implementing a symbol table, 3 of which are competitive while one is basic and trivial

Hashing

Key	Hash	Value
a	1	quick
b	**3**	brown
c	0	fox
d	**2**	jumps

Key	Hash	Value
e	**3**	**lazy**

Collision

0	1	2	3
[c]->fox	[a]->quick	[d]->jumps	[b]->brown [e]->lazy

Two Problems

Building a data structure based on hashes, we need to solve two major problems:

- find a hashing algorithm which generates different indexes for different keys in such a way that collisions occur rarely
- find an algorithm of resolving collisions which will anyway occur

Hash function significantly depends on the type of the key.

- integer numbers
- floating-point numbers
- strings
- custom value types or structures
- custom reference types or classes

Hashing Strings

```
public int GetHashCode() {
#if FEATURE_RANDOMIZED_STRING_HASHING
    if(HashHelpers.s_UseRandomizedStringHashing) {
        return InternalMarvin32HashString(this, this.Length, 0);
    }
#endif
    unsafe {
        fixed (char* src = this) {
          int hash1 = (5381<<16) + 5381;
          int hash2 = hash1;

          // 32 bit machines.
          int* pint = (int *)src;
          int len = this.Length;
          while (len > 2) {
             hash1 = ((hash1 << 5) + hash1 + (hash1 >> 27)) ^ pint[0];
             hash2 = ((hash2 << 5) + hash2 + (hash2 >> 27)) ^ pint[1];
             pint += 2;
             len  -= 4;
          }
          if (len > 0) {
             hash1 = ((hash1 << 5) + hash1 + (hash1 >> 27)) ^ pint[0];
          }
          return hash1 + (hash2 * 1566083941);
        }
    }
}
```

Hashing Strings

Guidelines for using the built-in hash algorithm for strings:

- hash codes should never be used outside of the application domain in which they were created
- string hashes should never be used as key fields in a collection
- they should never be persisted

Guidelines

Guidelines are caused by two major facts:

- If two string objects are equal, the GetHashCode method returns identical values.
  There is not a unique hash code value for each unique string value.
  Different strings can return the same hash code.
- The hash code itself is not guaranteed to be stable.

Какие значения хэш-кода будут выведены, если запустить этот код дважды?

```
static void Main()
{
    string str = "Hello, world!";
    WriteLine(str.GetHashCode());
}
```

- Эквивалентные
- Различные
- Зависит
- Лучше застрелите

Hashing Guidelines

- GetHashCode is useful for only one thing: putting an object in a hash table
- Equal Items should have equal hashes
- The integer returned by GetHashCode must never change while the object is contained in a data structure that depends on the hash code remaining stable
- GetHashCode must never throw an exception and must return

Hashing

A good hash code implementation should be:

- Fast
- Well distributed across the space of 32-bit integers
  for the given distribution of inputs.

Do not use hash codes:

- as a unique key for an object; probability of collision is extremely high
- as part of a digital signature or as a password equivalent

GetHashCode – ValueType

```
/*=================================GetHashCode==================================
**Action: Our algorithm for returning the hashcode is a little bit complex.  We look
**        for the first non-static field and get it's hashcode.  If the type has no
**        non-static fields, we return the hashcode of the type.  We can't take the
**        hashcode of a static member because if that member is of the same type as
**        the original type, we'll end up in an infinite loop.
**Returns: The hashcode for the type.
**Arguments: None.
**Exceptions: None.
==============================================================================*/
[System.Security.SecuritySafeCritical]  // auto-generated
[ResourceExposure(ResourceScope.None)]
[MethodImplAttribute(MethodImplOptions.InternalCall)]
public extern override int GetHashCode();

[MethodImplAttribute(MethodImplOptions.InternalCall)]
internal static extern int GetHashCodeOfPtr(IntPtr ptr);

public override String ToString()
{
    return this.GetType().ToString();
}
```

GetHashCode – ValueType

```
//source is in coreclr\src\vm\comutilnative.cpp

if(CanCompareBitsOrUseFastGetHashCode()) {
    FastGetValueTypeHashCodeHelper(mt, pObjRef);
}
else {
    RegularGetValueTypeHashCode(mt, pObjRef);
}

static INT32 FastGetValueTypeHashCodeHelper(MethodTable *mt, void *pObjRef)
{
    INT32 hashCode = 0;
    INT32 *pObj = (INT32*)pObjRef;

    //this is a struct with no refs and no "strange" offsets,
    //just go through the obj and xor the bits
    INT32 size = mt->GetNumInstanceFieldBytes();
    for (INT32 i = 0; i < (INT32)(size / sizeof(INT32)); i++)
        hashCode ^= *pObj++;

    return hashCode;
}
```

```csharp
static void Main() {
    var c1 = new Customer {
        Age = 18,
        Ssn = 1000
    };
    var c2 = new Customer {
        Age = 18,
        Ssn = 2000
    };

    WriteLine(c1.GetHashCode() ==
              c2.GetHashCode());
}
```

```csharp
public struct Customer
{
    public string Name { get; set; }
    public int Age { get; set; }
    public int Ssn { get; set; }
}
```

- true
- false
- Зависит
- Лучше застрелите

```csharp
static void Main() {
    var c1 = new Customer {
        Age = 18,
        Ssn = 1000
    };
    var c2 = new Customer {
        Age = 18,
        Ssn = 2000
    };
    var hs = new HashSet<Customer>();
    hs.Add(c1);
    hs.Add(c2);

    WriteLine(hs.Count);
}
```

```csharp
public struct Customer
{
    public string Name { get; set; }
    public int Age { get; set; }
    public int Ssn { get; set; }
}
```

- 1
- 2
- Зависит
- Лучше застрелите

GetHashCode – Native Implementation

```
public virtual int GetHashCode() { return RuntimeHelpers.GetHashCode(this); }
```

GetHashCode – Native Implementation

```
DWORD Object::ComputeHashCode()
{
    DWORD hashCode;

    // note that this algorithm now uses at most HASHCODE_BITS so that it will
    // fit into the objheader if the hashcode has to be moved back into the objheader
    // such as for an object that is being frozen
    do
    {
        // we use the high order bits in this case because they're more random
        hashCode = GetThread()->GetNewHashCode() >> (32-HASHCODE_BITS);
    }
    while (hashCode == 0);   // need to enforce hashCode != 0

    // verify that it really fits into HASHCODE_BITS
     _ASSERTE((hashCode & ((1<<HASHCODE_BITS)-1)) == hashCode);

    return hashCode;
}
```

GetHashCode – Native Implementation

```
inline DWORD GetNewHashCode()
{
    LIMITED_METHOD_CONTRACT;
    // Every thread has its own generator for hash codes so that we won't get into a situation
    // where two threads consistently give out the same hash codes.
    // Choice of multiplier guarantees period of 2**32 - see Knuth Vol 2 p16 (3.2.1.2 Theorem A).
    DWORD multiplier = GetThreadId()*4 + 5;
    m_dwHashCodeSeed = m_dwHashCodeSeed*multiplier + 1;
    return m_dwHashCodeSeed;
}
```

$$X_{n+1} = (aX_n + c)\ \%\ m, n \geq 0$$

```
// Initialize this variable to a very different start value for each thread
// Using linear congruential generator from Knuth Vol. 2, p. 102, line 24
dwHashCodeSeed = dwHashCodeSeed * 1566083941 + 1;
m_dwHashCodeSeed = dwHashCodeSeed;
```

GetHashCode – Native Implementation

Таблица 1

ВЫБОРОЧНЫЕ РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ СПЕКТРАЛЬНОГО КРИТЕРИЯ

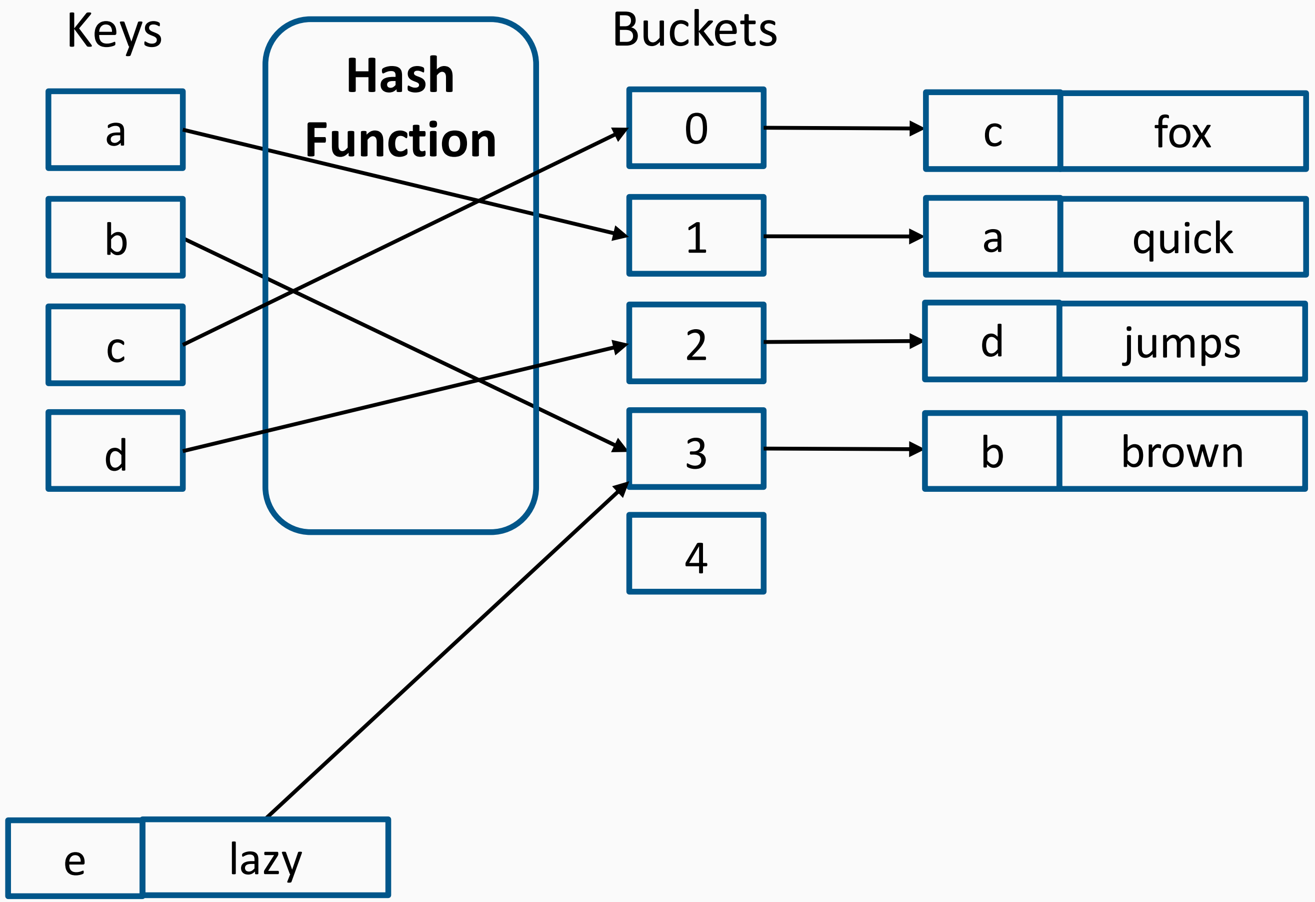
Строка	a	m	ν_2^2	ν_3^2	ν_4^2	ν_5^2	ν_6^2
1	23	10^8+1	530	530	530	530	447
2	2^7+1	2^{35}	16642	16642	16642	15602	252
3	$2^{18}+1$	2^{35}	34359738368	6	4	4	4
4	3141592653	2^{35}	2997222016	1026050	27822	1118	1118
5	137	256	274	30	14	6	4
6	3141592621	10^{10}	4577114792	1034718	62454	1776	542
7	3141592221	10^{10}	4293881050	276266	97450	3366	2382
8	4219755981	10^{10}	10721093248	2595578	49362	5868	820
9	4160984121	10^{10}	9183801602	4615650	16686	6840	1344
10	$2^{24}+2^{13}+5$	2^{35}	8364058	8364058	21476	16712	1496
11	5^{13}	2^{35}	33161885770	2925242	113374	13070	2256
12	$2^{16}+3$	2^{29}	536936458	118	116	116	116
13	1812433253	2^{32}	4326934538	1462856	15082	4866	906
14	1566083941	2^{32}	4659748970	2079590	44902	4652	662
15	69069	2^{32}	4243209856	2072544	52804	6990	242
16	1664525	2^{32}	4938916874	2322494	63712	4092	1038
17	314159269	$2^{31}-1$	1432232969	899290	36985	3427	1144
18	62089911	$2^{31}-1$	1977289717	1662317	48191	6101	1462
19	16807	$2^{31}-1$	282475250	408197	21682	4439	895
20	48271	$2^{31}-1$	1990735345	1433881	47418	4404	1402
21	40692	$2^{31}-249$	1655838865	1403422	42475	6507	1438
22	44485709377909	2^{46}	5.6×10^{13}	1180915002	1882426	279928	26230

```
static void Main() {
    var c1 = new Customer {
        Age = 18,
        Ssn = 1000
    };
    var c2 = new Customer {
        Age = 18,
        Ssn = 1000
    };

    WriteLine(c1.GetHashCode() ==
              c2.GetHashCode());
}
```

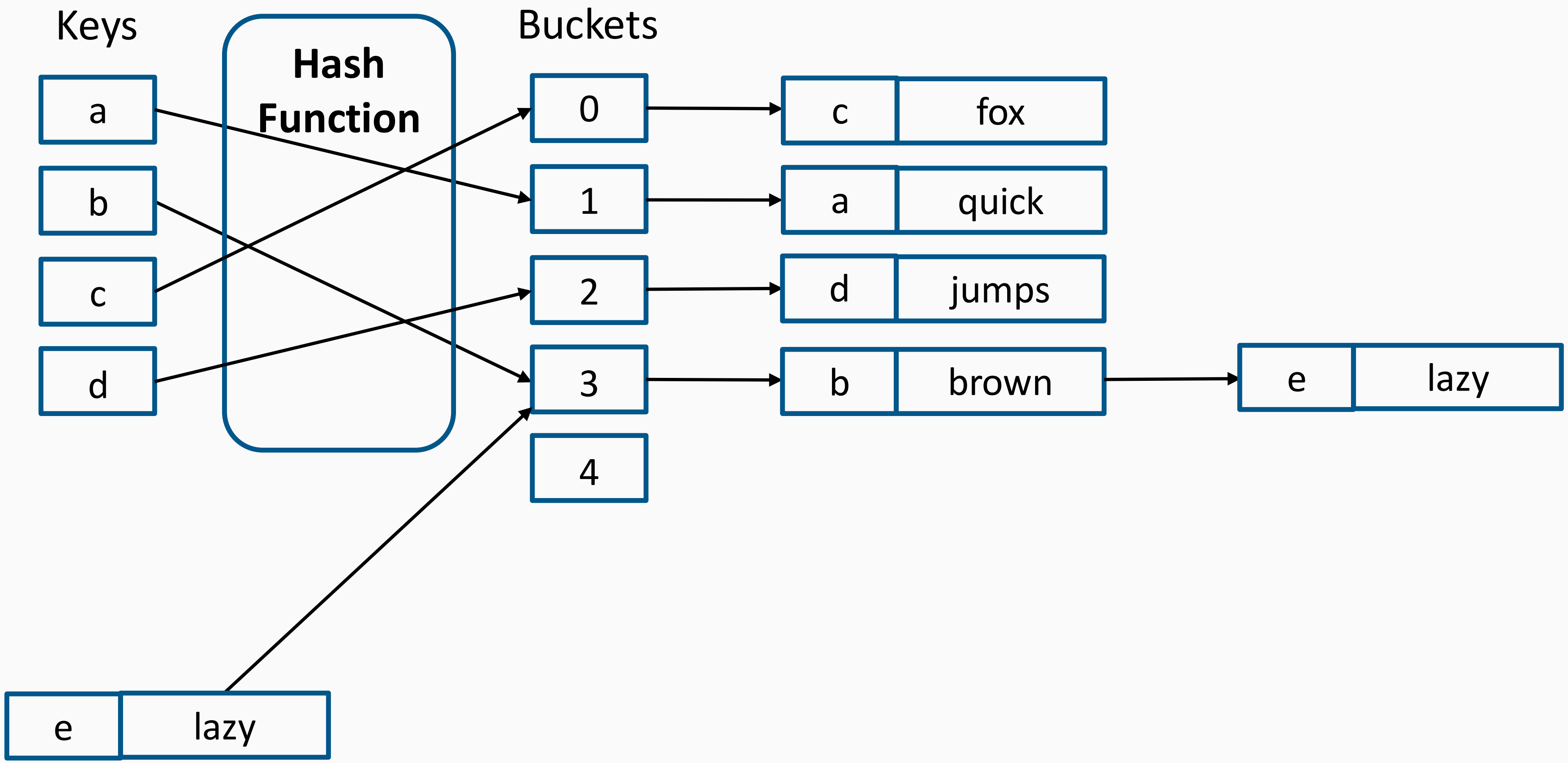
```
public class Customer
{
    public string Name { get; set; }
    public int Age { get; set; }
    public int Ssn { get; set; }
}
```

- true
- false
- Зависит
- Лучше застрелите

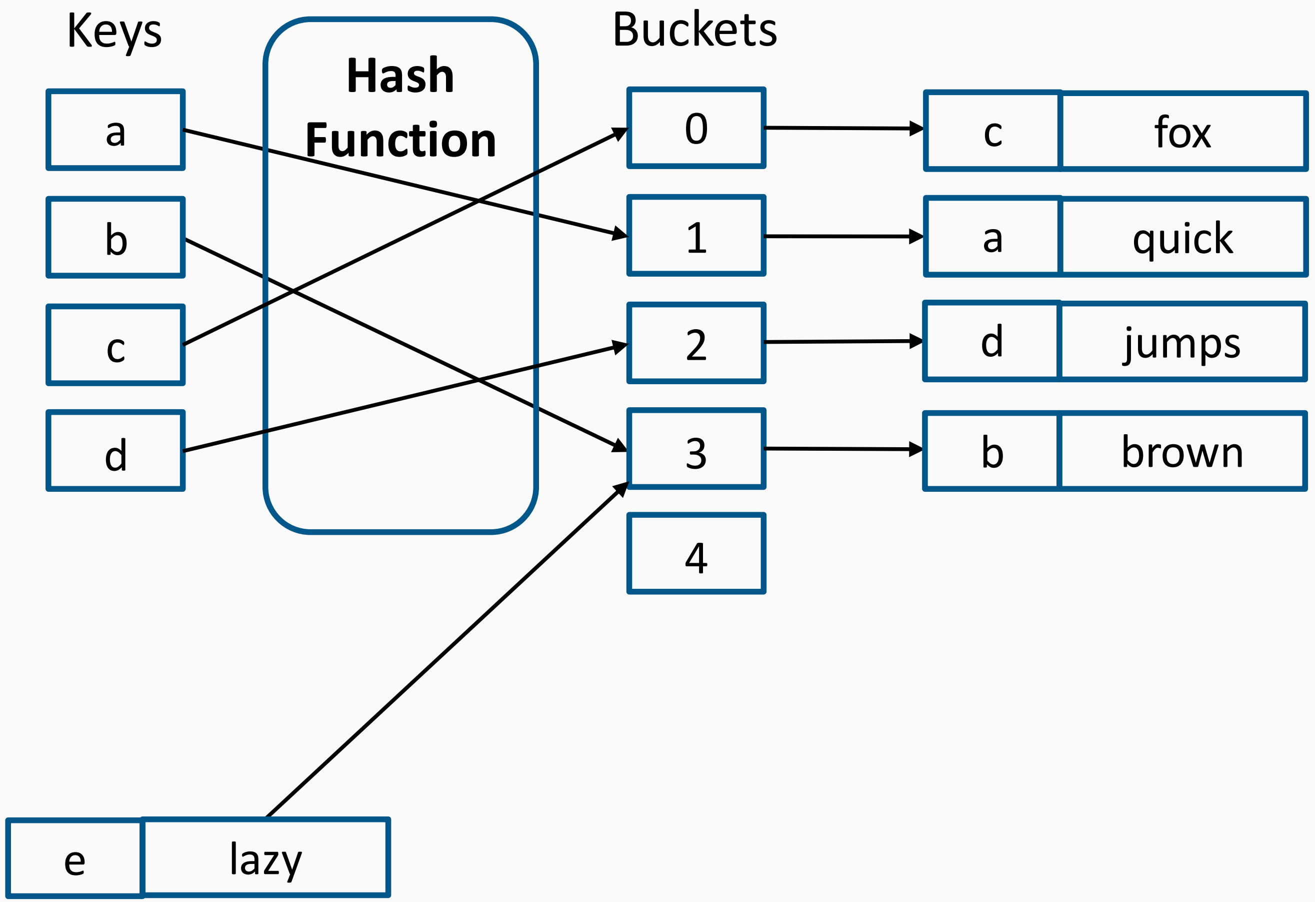
Resolving Collisions



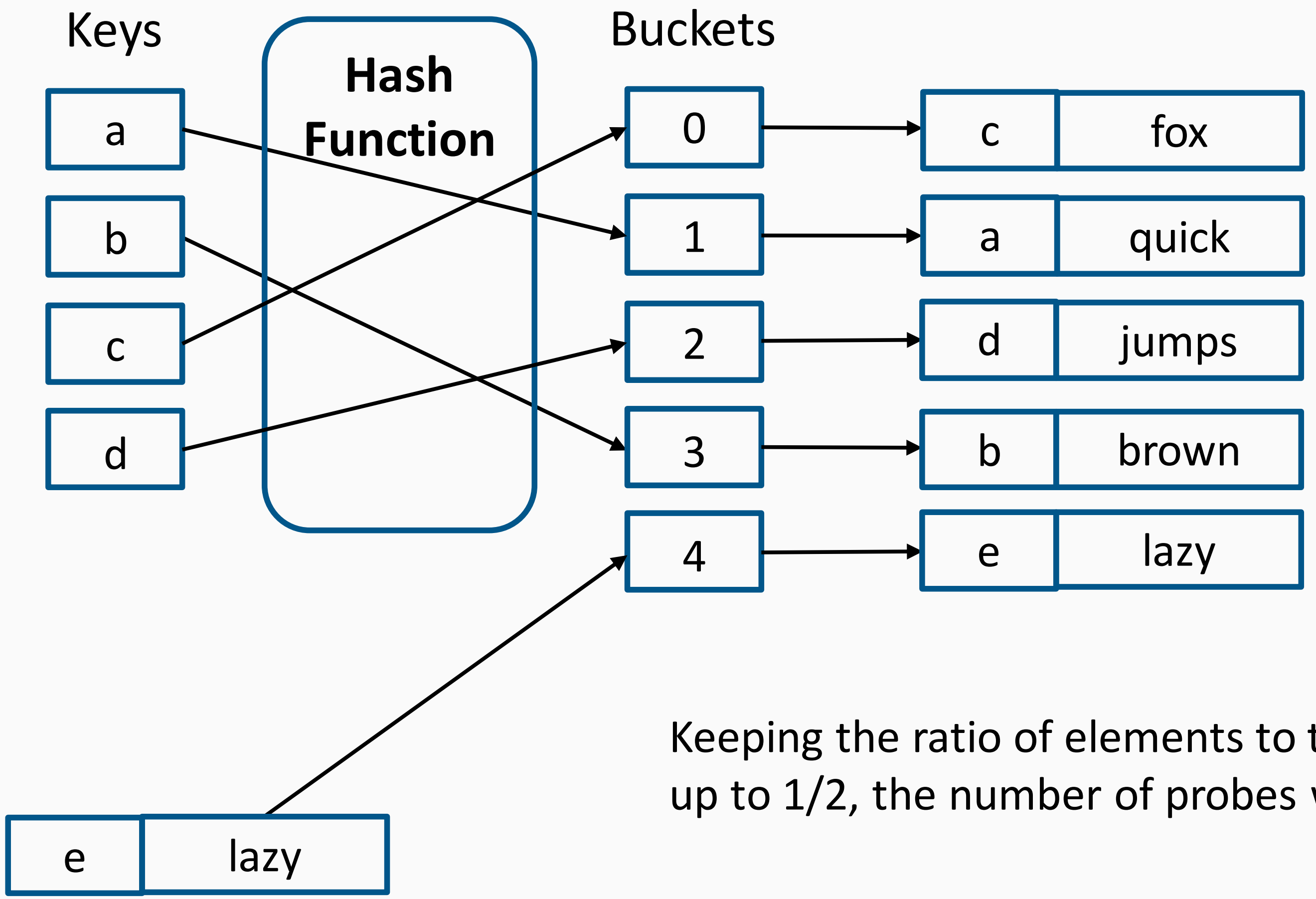
Separate Chaining



Resolving Collisions



Linear Probing



Keeping the ratio of elements to the buckets size between 1/8 up to 1/2, the number of probes will vary between 1.5 and 2.5!

Dictionary

```
[System.Runtime.InteropServices.ComVisible(false)]
public class Dictionary<TKey,TValue>: IDictionary<TKey,TValue>, IDictionary

    private struct Entry {
        public int hashCode;    // Lower 31 bits of hash code, -1 if unused
        public int next;        // Index of next entry, -1 if last
        public TKey key;           // Key of entry
        public TValue value;         // Value of entry
    }

    private int[] buckets;
    private Entry[] entries;
    private int count;
    private int version;
    private int freeList;
    private int freeCount;
    private IEqualityComparer<TKey> comparer;
    private KeyCollection keys;
    private ValueCollection values;
    private Object _syncRoot;
```

Dictionary

```
private void Insert(TKey key, TValue value, bool add)
{
    // Calc hash code of the key eliminating negative values.
    int hashCode = comparer.GetHashCode(key) & 0x7FFFFFFF;

    // Usual way of narrowing the value set
    // of the hash code to the set of possible bucket indices.
    int targetBucket = hashCode % buckets.Length;

    for (int i = buckets[targetBucket]; i >= 0; i = entries[i].next)
    {
        if (entries[i].hashCode == hashCode && comparer.Equals(entries[i].key, key))
        {
            entries[i].value = value;
            version++;
            return;
        }
    }
}
```

Dictionary

```
internal static class HashHelpers
{
    public static readonly int[] primes =
    {
        3, 7, 11, 17, 23, 29, 37, 47, 59, 71, 89, 107, 131, 163, 197,
        239, 293, 353, 431, 521, 631, 761, 919,
        1103, 1327, 1597, 1931, 2333, 2801, 3371, 4049, 4861,
        5839, 7013, 8419, 10103, 12143, 14591,
        17519, 21023, 25229, 30293, 36353, 43627, 52361, 62851,
        75431, 90523, 108631, 130363, 156437,
        187751, 225307, 270371, 324449, 389357, 467237,
        560689, 672827, 807403, 968897, 1162687, 1395263,
        1674319, 2009191, 2411033, 2893249, 3471899,
        4166287, 4999559, 5999471, 7199369
    };
}
```

Dictionary

```
private void Insert(TKey key, TValue value, bool add)
{
    // Calc hash code of the key eliminating negative values.
    int hashCode = comparer.GetHashCode(key) & 0x7FFFFFFF;

    // Usual way of narrowing the value set
    // of the hash code to the set of possible bucket indices.
    int targetBucket = hashCode % buckets.Length;

    for (int i = buckets[targetBucket]; i >= 0; i = entries[i].next)
    {
        if (entries[i].hashCode == hashCode && comparer.Equals(entries[i].key, key))
        {
            entries[i].value = value;
            version++;
            return;
        }
    }
}
```

Dictionaries

	SortedList	Dictionary	SortedDictionary	SortedSet
based on	2 arrays- keys (sorted)/values	Hash Table	Red-Black Tree	Red-Black Tree
Add	O(n)**	O(1)*	log(n)	log(n)
Remove (by key)	O(n)	O(1)	log(n)	log(n)
RemoveAt	O(n)	-	-	-
TryGetValue	log(n) – binary search	O(1)	log(n)	log(n)
ContainsKey	log(n)	O(1)	log(n)	log(n) - **Contains**
ContainsValue	O(n)	O(n)	O(n)	-
Clear	O(n)	O(n)	O(1)	O(n) – O(1)?
IndexOfKey	log(n)	-	-	-
IndexOfValue	O(n)	-	-	-
Indexed access [key]	log(n)	-	log(n)	-

* - O(n) в случае resize;

** - O(log n) operation if the new element is added at the end of the list. If insertion causes a resize, the operation is O(n)

SortedList

```
    [System.Runtime.InteropServices.ComVisible(false)]
    public class SortedList<TKey, TValue> :
        IDictionary<TKey, TValue>, System.Collections.IDictionary, IReadOnlyDictionary<TKey, TValue>
    {
        private TKey[] keys;
        private TValue[] values;
        private int _size;
        private int version;
        private IComparer<TKey> comparer;
        private KeyList keyList;
        private ValueList valueList;
#if !FEATURE_NETCORE
        [NonSerialized]
#endif
        private Object _syncRoot;

        static TKey[] emptyKeys = new TKey[0];
        static TValue[] emptyValues = new TValue[0];

        private const int _defaultCapacity = 4;
```

Dictionaries

	SortedList	Dictionary	SortedDictionary	SortedSet
based on	2 arrays- keys (sorted)/values	Hash Table	Red-Black Tree	Red-Black Tree
Add	O(n)**	O(1)*	log(n)	log(n)
Remove (by key)	O(n)	O(1)	log(n)	log(n)
RemoveAt	O(n)	-	-	-
TryGetValue	log(n) – binary search	O(1)	log(n)	log(n)
ContainsKey	log(n)	O(1)	log(n)	log(n) - **Contains**
ContainsValue	O(n)	O(n)	O(n)	-
Clear	O(n)	O(n)	O(1)	O(n) – O(1)?
IndexOfKey	log(n)	-	-	-
IndexOfValue	O(n)	-	-	-
Indexed access [key]	log(n)	-	log(n)	-

* - O(n) в случае resize;

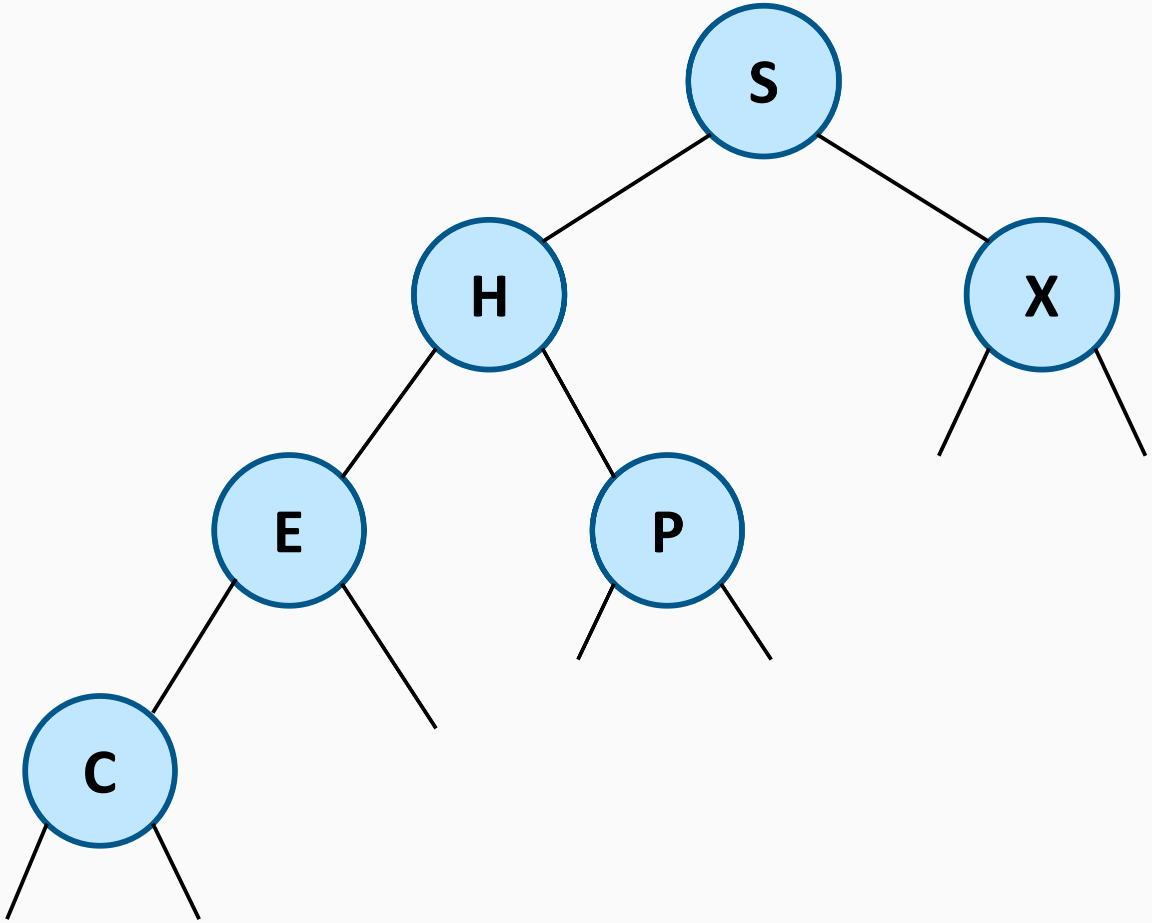
** - O(log n) operation if the new element is added at the end of the list. If insertion causes a resize, the operation is O(n)

SortedDictionary

```
    public class SortedDictionary<TKey, TValue> : IDictionary<TKey, TValue>,
#if !FEATURE_NETCORE
        [NonSerialized]
#endif
        private KeyCollection keys;

#if !FEATURE_NETCORE
        [NonSerialized]
#endif
        private ValueCollection values;

        private TreeSet<KeyValuePair<TKey, TValue>> _set;
```

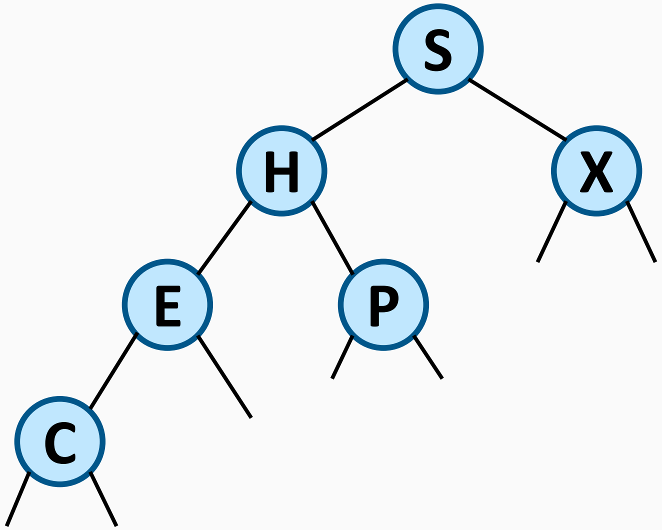
```
  internal class TreeSet<T> : SortedSet<T> {


      internal override bool AddIfNotPresent(T item) {
          bool ret = base.AddIfNotPresent(item);
          if (!ret) {
              ThrowHelper.ThrowArgumentException(ExceptionResource.Argument_AddingDuplicate);
          }
          return ret;
      }

  }
```

Binary Tree

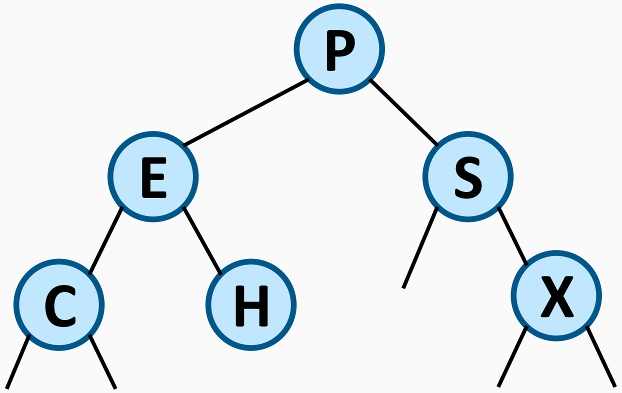


Tree

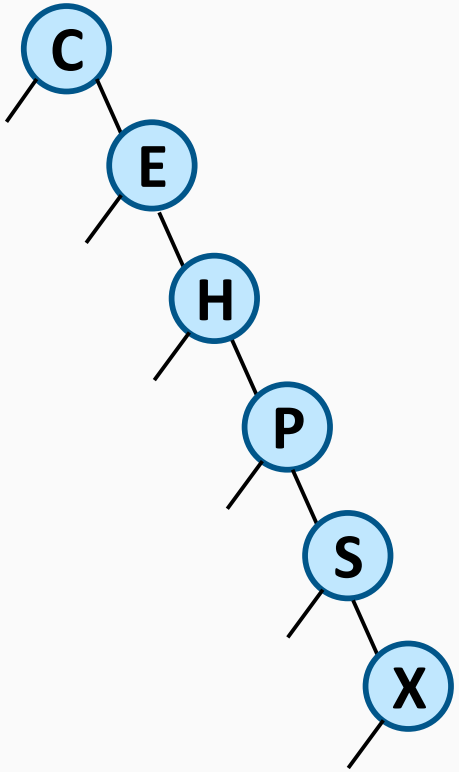
Typical Case



Best Case

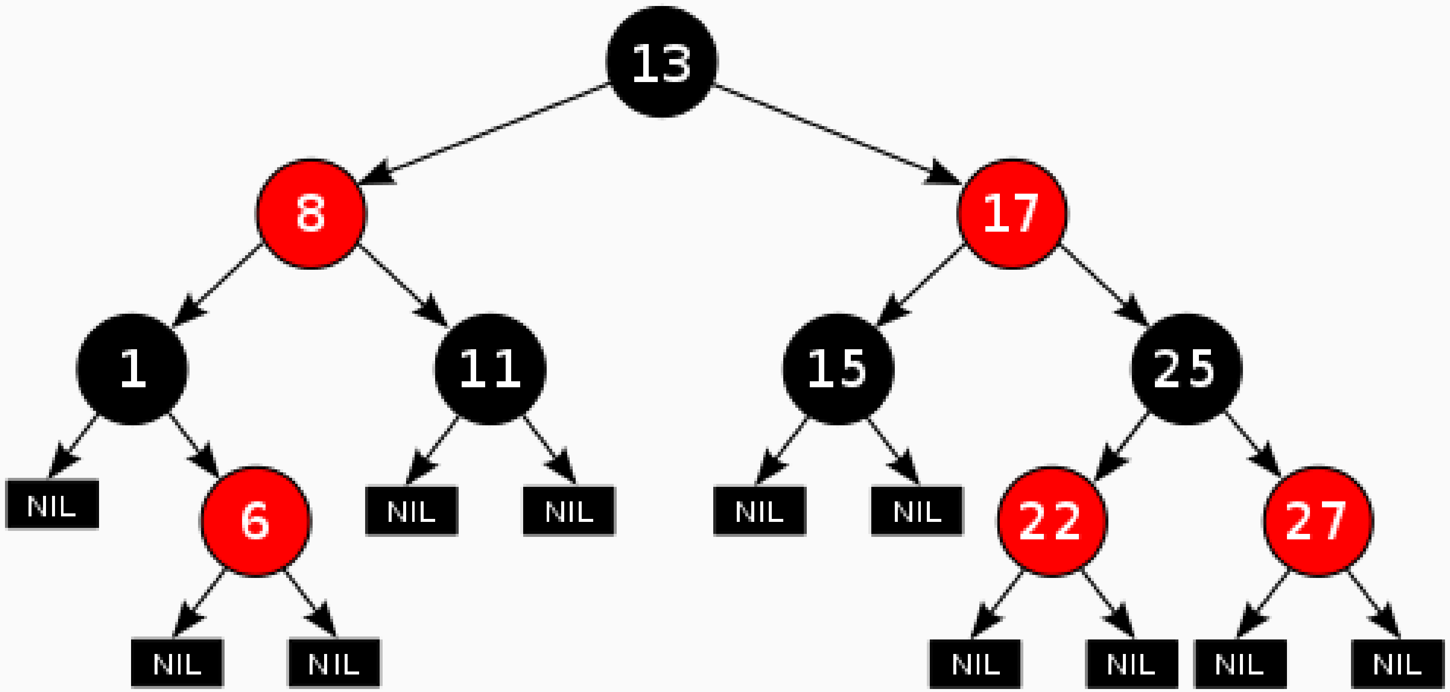


Worst Case



Red-Back Tree

Close to Ideally Balanced



Operations on Sets

- **Intersections**:
  - *Example: The intersection of {1,2,5} and {2,4,9} is the set {2}.*
- **Unions**:
  - *Example: The union of {1,2,5} and {2,4,9} is {1,2,4,5,9}.*
- **Differences**:
  - *Example: The difference of {1,2,5} and {2,4,9} is {1,5}.*
- **Supersets**:
  - *Example: The set {1,2,5} is a superset of {1,5}.*
- **Subsets:**
  - *Example: The set {1,5} is a subset of {1,2,5}.*

ISet<T>

Method	Description
ExceptWith	Removes all elements in the specified collection from the current set.
IntersectWith	Modifies the current set so that it contains only elements that are also in a specified collection.
IsProperSubsetOf	Determines whether the current set is a proper (strict) subset of a specified collection.
IsProperSupersetOf	Determines whether the current set is a proper (strict) superset of a specified collection.
IsSubsetOf	Determines whether a set is a subset of a specified collection.
IsSupersetOf	Determines whether the current set is a superset of a specified collection.
Overlaps	Determines whether the current set overlaps with the specified collection.
SetEquals	Determines whether the current set and the specified collection contain the same elements.
SymmetricExceptWith	Modifies the current set so that it contains only elements that are present either in the current set or in the specified collection, but not both.
UnionWith	Modifies the current set so that it contains all elements that are present in the current set, in the specified collection, or in both.

Sets

	HashSet	SortedSet	List
based on	HashTable	Red-Black Tree	Array
Add	O(1) / O(n)	log(n)	O(1) / O(n)
Remove (by key)	O(1)	log(n)	O(n)
RemoveAt	-	-	O(n)
TryGetValue	O(1)	log(n)	-
Contains	O(1)	log(n)	O(n)
Clear	O(n)	O(n) – **O(1)?**	O(n)
Indexed access [key]	-	-	O(1) – by index (not key)

ISet<T>

Method	HashSet	SortedSet
ExceptWith	**O(N)**	~
IntersectWith	O(N) / O(N+M) *	~
IsProperSubsetOf	O(N) / O(N+M) *	~
IsProperSupersetOf	O(N) / O(N+M) *	~
IsSubsetOf	O(N) / O(N+M) *	~
IsSupersetOf	O(N) / O(N+M) *	~
Overlaps	**O(N)**	~
SetEquals	O(N) / O(N+M) *	O(logN) / O(N+M)
SymmetricExceptWith	O(N) / O(N+M) *	~
UnionWith	**O(N)**	**~**

* - O(N) if other is a HashSet / SortedSet with the same comparer, otherwise O(N+M)

** https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1.setequals?view=netcore-2.1

На какой структуре данных базируется тип SortedDictionary<T> из BCL?

- Массив
- Два параллельных массива
- Красно-чёрное дерево
- Хэш таблица

На какой структуре данных базируется тип SortedList<T> из BCL?

- Массив
- Два параллельных массива
- Красно-чёрное дерево
- Хэш таблица

На какой структуре данных базируется тип SortedSet<T> из BCL?

- Массив
- Два параллельных массива
- Красно-чёрное дерево
- Хэш таблица

На какой структуре данных базируется тип Dictionary<T> из BCL?

- Массив
- Два параллельных массива
- Красно-чёрное дерево
- Хэш таблица

Dead Horses

- StringCollection
- StringDictionary
- OrderedDictionary
- NameValueCollection
- ListDictionary
- HybridDictionary
- HashTable
- ArrayList

Conclusion

- Be extremely careful implementing even standard algorithms
- Choose right data structures to improve performance significantly
- Hashing algorithm has to be fast and well-distributed
- It's easy to fail implementing a hashing algorithm
- Default hash for Value Types depends on the first non-static field
- Default hash for a Reference Type doesn't depend on its internal data at all
- No hashing algorithms without collisions
- There are two major approaches to resolve collisions:
  separate chains and open addressing
- There is almost always a room for applying slick optimizations

Data Structures in BCL

- Array.Sort<T> runs either a custom Intro Sort or native QSort
- List<T>, Stack<T>, Queue<T> are based on Array
- LinkedList<T> is a doubly-linked circular list
- No PriorityQueue in BCL
- Dictionary<T> is lightening fast but is not sorted.
  Almost all operations work for O(1).
  Resolves collisions combining separate chaining and open addressing.
- SortedList<T> is a dictionary based on 2-parallel arrays
- SortedDictionary<T> is based on SortedSet<T> which is based on a Red-Black Tree.
  Almost all operations work for log(n).

Resources

https://habr.com/post/188038/

https://blogs.msdn.microsoft.com/ericlippert/2010/03/22/socks-birthdays-and-hash-collisions/

https://en.wikipedia.org/wiki/Pigeonhole_principle

https://stackoverflow.com/questions/3841602/why-is-valuetype-gethashcode-implemented-like-it-is

https://blog.markvincze.com/back-to-basics-dictionary-part-2-net-implementation/

Get a Course

If you want to **get my "Algorithms & Data Structures Course in C#"** course **for $9.99:**

Visit this URL:
https://www.udemy.com/algorithms-data-structures-csharp/

And apply **your coupon: MSKDOTNET**

or: https://bit.ly/2BgaiVI (coupon is applied already)