Școala
informală
de IT

Generic Collections

Școala informală de IT

Agenda

- Linear Data Structures
  - List
  - Stack
  - Queue
- Dictionaries
  - Dictionary
  - SortedDictionary
- Sets
  - HashSet
  - SortedSet

Școala informală de IT

Basic Data Structures

- Linear structures
  - Lists: fixed size and variable size
  - Stacks: LIFO (Last In First Out) structure
  - Queues: FIFO (First In First Out) structure
- Dictionaries (maps)
  - Contain pairs (key, value)
  - Hash tables: use hash functions to search/insert

Școala
informală
de IT

Linear Data Structures

Lists, Stacks, Queues

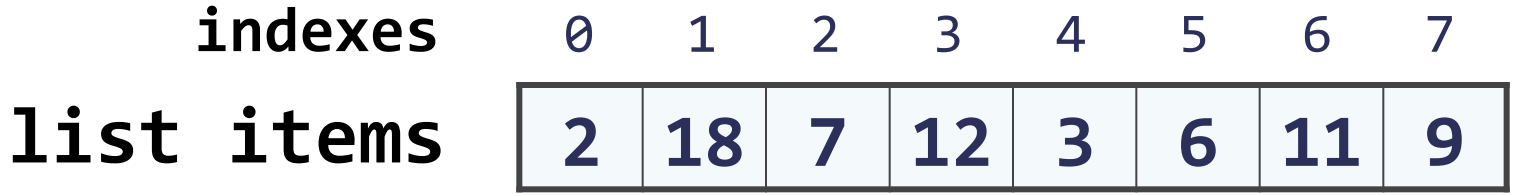
Școala
informală
de IT

Lists

Static and Dynamic Implementations

Static List

- Implemented by an array
  - Provides direct access by index
  - Has fixed capacity
  - Insertion, deletion and resizing are slow operations

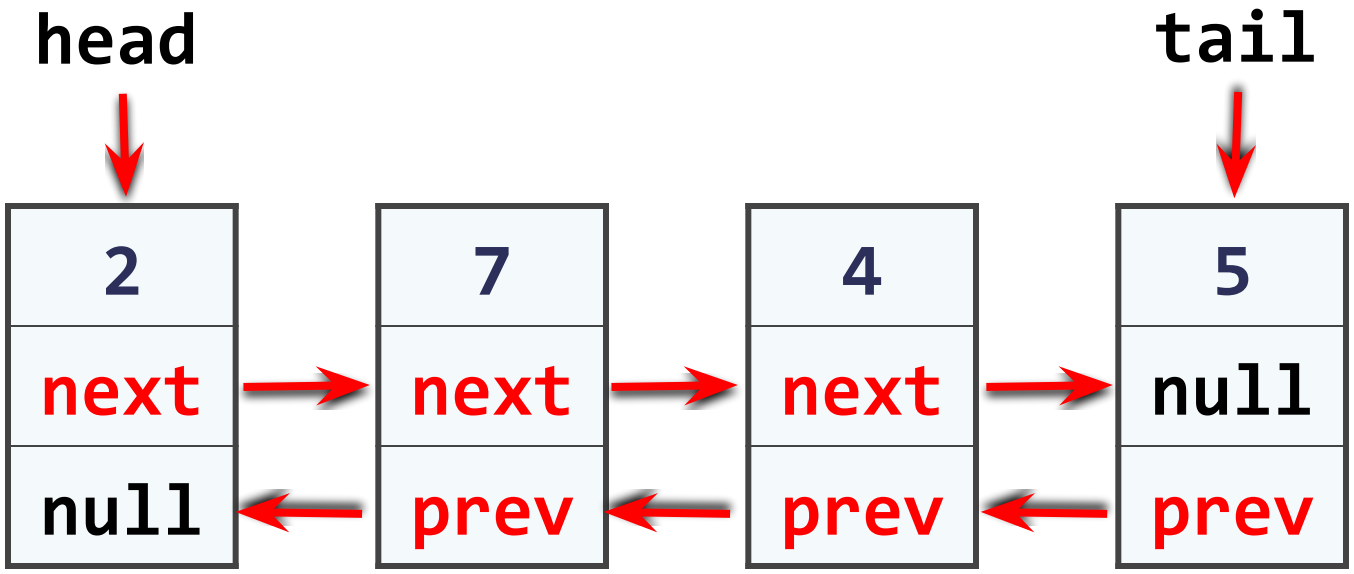


Linked List

- Dynamic (pointer-based) implementation
- Different forms
  - Single-linked and double-linked
  - Sorted and unsorted
- Single-lined list
  - Each item has 2 fields: value and next

Linked List

- Double-linked List
  - Each item has 3 fields: value, next and prev



Școala
informală
de IT

The `List<T>` Class

Auto-Resizable Indexed Lists

Școala informală de IT

The List<T> Class

- Implements the abstract data structure list using an array
  - All elements are of the same type T
  - T can be any type, e.g. `List<int>`, `List<string>`, `List<DateTime>`
  - Size is dynamically increased as needed
- Basic functionality:
  - `Count` – returns the number of elements
  - `Add(T)` – appends given element at the end

Școala informală de IT

List<T> - Functionality

- list[index] - access element by index
- Insert(index, T) - insert given element to the list at a specified position
- Remove(T) - removes the first occurrence of given element
- RemoveAt(index) - removes the element at the specified position
- Clear() - removes all elements
- Contains(T) - determines whether an element is part of the list

Școala informală de IT

List<T> - Functionality

- IndexOf() - returns the index of the first occurrence of a value in the list (zero-based)
- Reverse() - reverses the order of the elements in the list or a portion of it
- Sort() - sorts the element in the list or a portion of it
- ToArray() - converts the elements of the list to an array
- TrimExcess() - sets the capacity to the actual number of elements

Școala informală de IT

List<T> – Simple Example

```
static void Main()
{
    List<string> list = new List<string>() { "C#", "Java" };

    list.Add("SQL");
    list.Add("Python");

    foreach (string item in list)
    {
        Console.WriteLine(item);
    }

    // Result:
    //    C#
    //    Java
    //    SQL
    //    Python
}
```

Inline initialization: the compiler adds specified elements to the list.

Școala
informală
de IT

Stacks

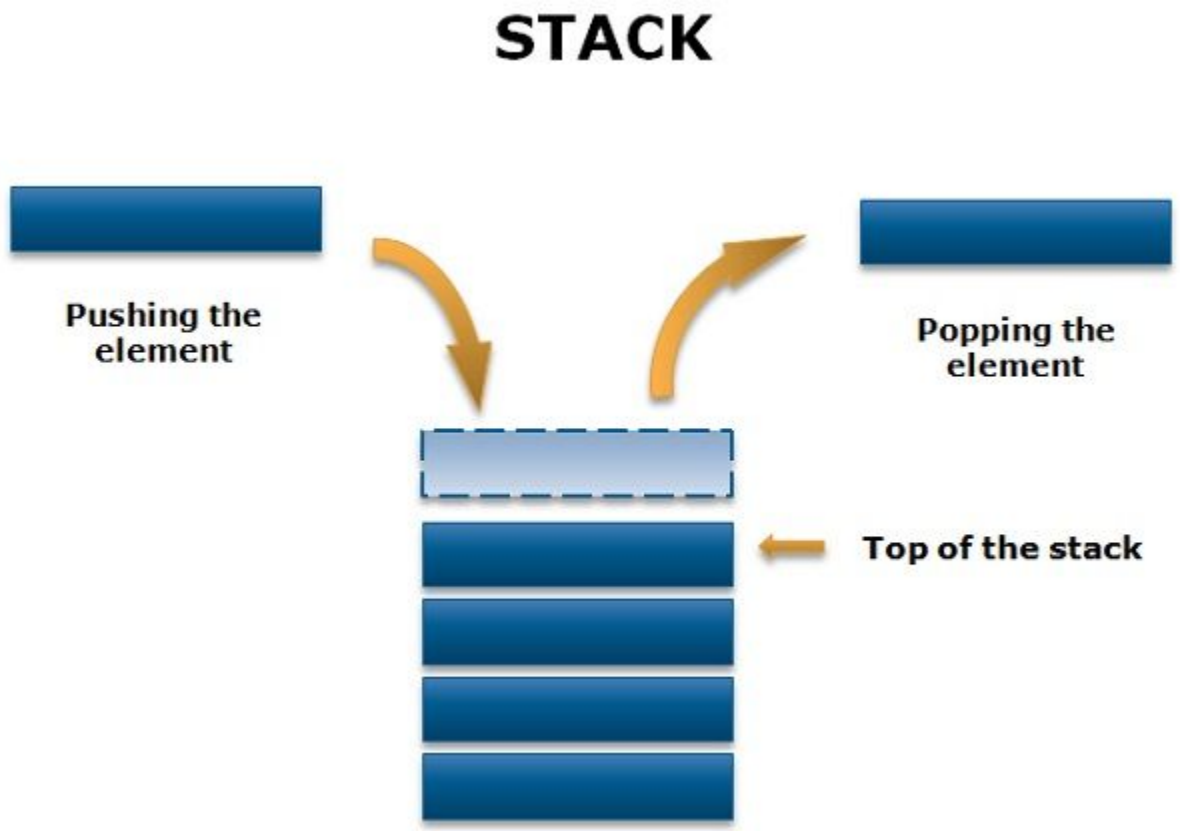
Static Dynamic Implementation

Școala
informală
de IT

The Stack

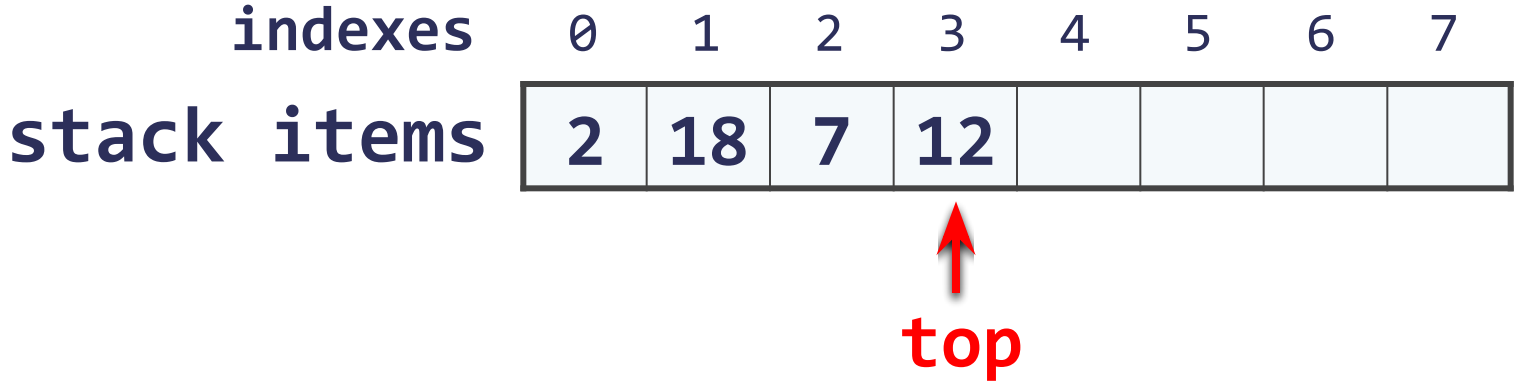
- LIFO (Last In - First Out) structure
- Elements inserted (push) at “top”
- Elements removed (pop) from “top”
- Useful in many situations
  - E.g. the execution stack of the program
- Can be implemented in several ways
  - Statically (using array)
  - Dynamically (linked implementation)
  - Using the `Stack<T>` class

The Stack


STACK
Pushing the element
Popping the element
Top of the stack

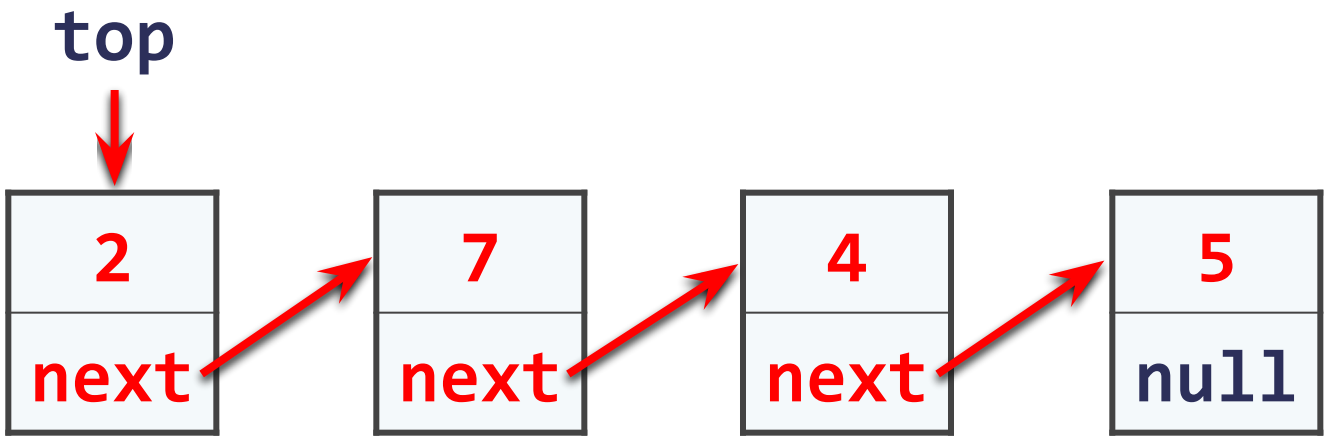
Static Stack

- Static (array-based) implementation
  - Has limited (fixed) capacity
  - The current index (`top`) moves left / right with each pop / push



Linked Stack

- Dynamic (pointer-based) implementation
  - Each item has 2 fields: `value` and `next`
  - Special pointer keeps the top element



Școala
informală
de IT

The Stack<T> Class

Școala informală de IT

The Stack<T> Class

- Implements the stack data structure using an array
  - Elements are from the same generic type T
  - T can be any type, e.g. `Stack<int>`
  - Size is dynamically increased as needed
- Basic functionality:
  - `Push(T)` – inserts elements to the stack
  - `Pop()` – removes and returns the top element from the stack

Școala informală de IT

The Stack<T> Class

- Basic functionality:
  - `Peek()` – returns the top element of the stack without removing it
  - `Count` – returns the number of elements
  - `Clear()` – removes all elements
  - `Contains(T)` – determines whether given element is in the stack
  - `ToArray()` – converts the stack to an array
  - `TrimExcess()` – sets the capacity to the actual number of elements

Școala informală de IT

Stack(T) - Example

- Using Push(), Pop() and Peek() methods

```
static void Main()
{
    Stack<string> stack = new Stack<string>();

    stack.Push("Cluj");
    stack.Push("Timisoara");
    stack.Push("Iasi");
    stack.Push("Oradea");

    Console.WriteLine("Top = {0}", stack.Peek());

    while (stack.Count > 0)
    {
        string personName = stack.Pop();
        Console.WriteLine(personName);
    }
}
```

Școala
informală
de IT

Queues

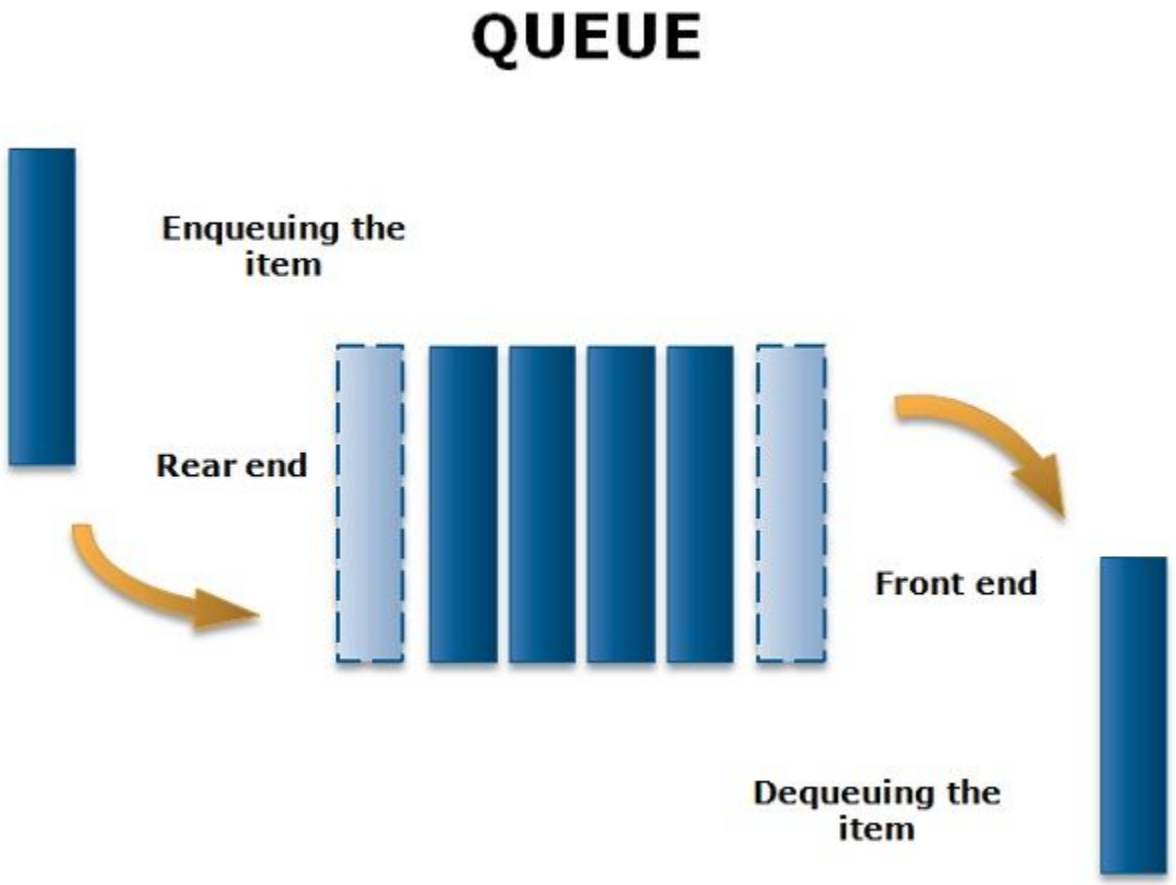
Static and Dynamic Implementation

Școala
informală
de IT

The Queue

- FIFO (First In - First Out) structure
- Elements inserted at the tail (Enqueue)
- Elements removed from the head (Dequeue)
- Useful in many situations
  - Print queues, message queues, etc.
- Can be implemented in several ways
  - Statically (using array)
  - Dynamically (using pointers)
  - Using the `Queue<T>` class

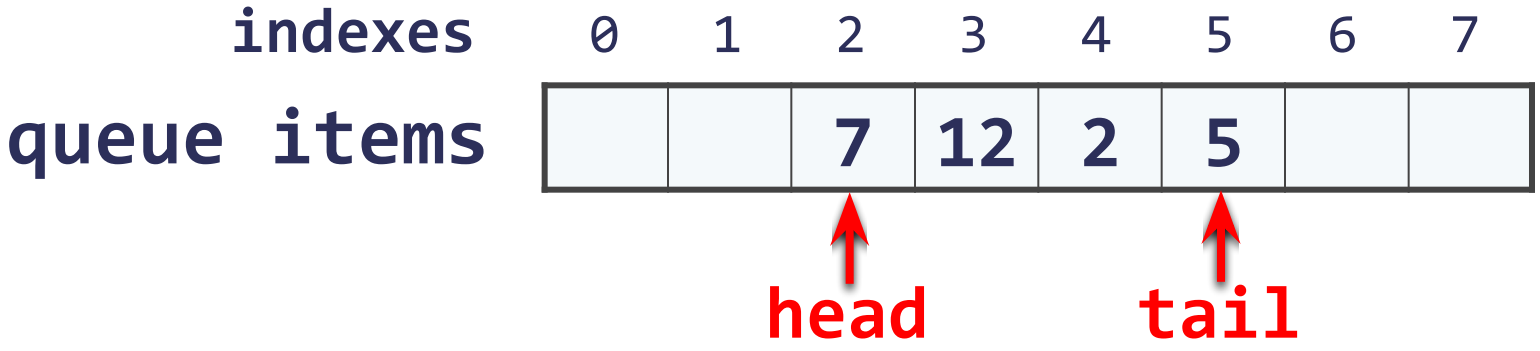
The Queue


QUEUE
Enqueuing the item
Rear end
Front end
Dequeuing the item

Școala informală de IT

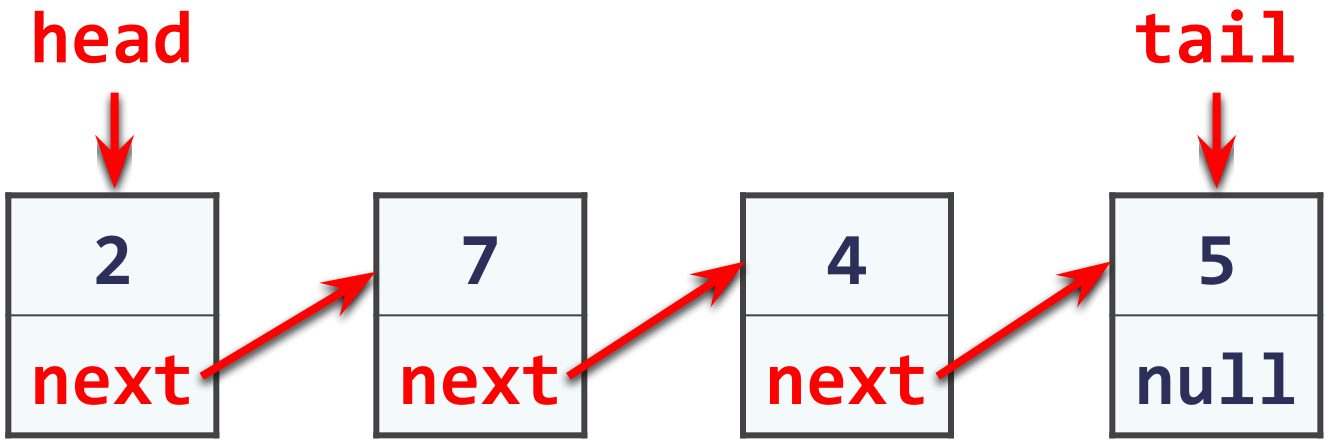
Static Queue

- Static (array-based) implementation
  - Has limited (fixed) capacity
  - Implement as a “circular array”
  - Has `head` and `tail` indices, pointing to the head and the tail of the cyclic queue



Linked Queue

- Dynamic (pointer-based) implementation
  - Each item has 2 fields: `value` and `next`
  - Dynamically create and delete objects



Școala
informală
de IT

The Queue<T> Class

Școala informală de IT

The Queue<T> Class

- Implements the queue data structure using a circular resizable array
  - Elements are from the same generic type `T`
  - T can be any type, e.g. `Stack<int>`
  - Size is dynamically increased as needed
- Basic functionality:
  - `Enqueue(T)` – adds an element to the end of the queue
  - `Dequeue()` – removes and returns the element at the beginning of the queue

Școala informală de IT

The Queue<T> Class

- Basic functionality:
  - `Peek()` – returns the element at the beginning of the queue without removing it
  - `Count` – returns the number of elements
  - `Clear()` – removes all elements
  - `Contains(T)` – determines whether given element is in the queue
  - `ToArray()` – converts the queue to an array
  - `TrimExcess()` – sets the capacity to the actual number of elements in the queue

Școala informală de IT

Queue(T) - Example

- Using Enqueue( ) and Dequeue( ) methods

```
static void Main()
{
    Queue<string> queue = new Queue<string>();
    queue.Enqueue("Message One");
    queue.Enqueue("Message Two");
    queue.Enqueue("Message Three");
    queue.Enqueue("Message Four");
    while (queue.Count > 0)
    {
        string message = queue.Dequeue();
        Console.WriteLine(message);
    }
}
```

Școala
informală
de IT

Dictionaries and Sets

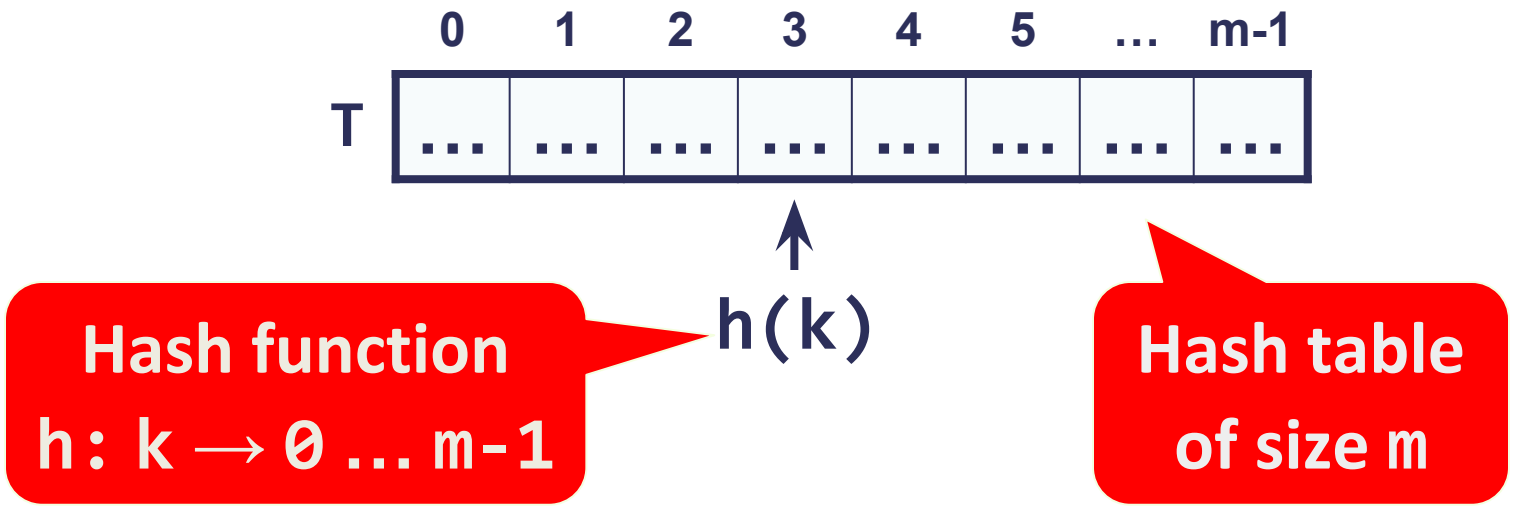
Dictionary, SortedDictionary, HashSet, SortedSet

Dictionary

- The abstract data type "dictionary" maps key to values
  - Also known as "map" or "associative array"
  - Contains a set of (key, value) pairs
- Dictionary abstract operations:
  - `Add(key, value)`
  - `FindByKey(key)` → `value`
  - `Delete(key)`
- Can be implemented in several ways
  - List, array, hash table, balanced tree, ...

Hash Table

- A hash table is an array that holds a set of (key, value) pairs
- The process of mapping a key to a position in a table is called hashing



Hash Table

h("Apple") = 4

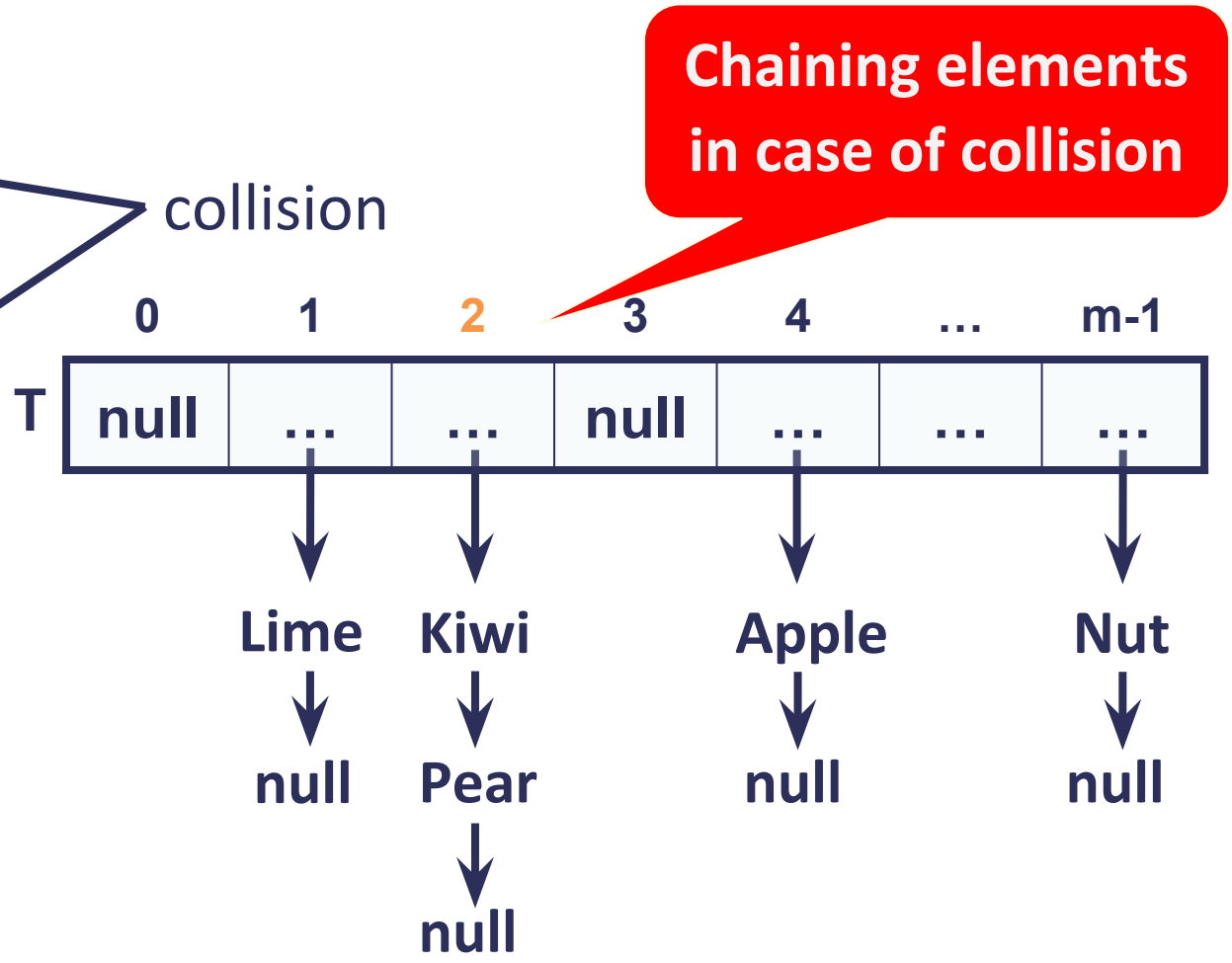
h("Kiwi") = 2

h("Lime") = 1

h("Pear") = 2

h("Nut") = m-1

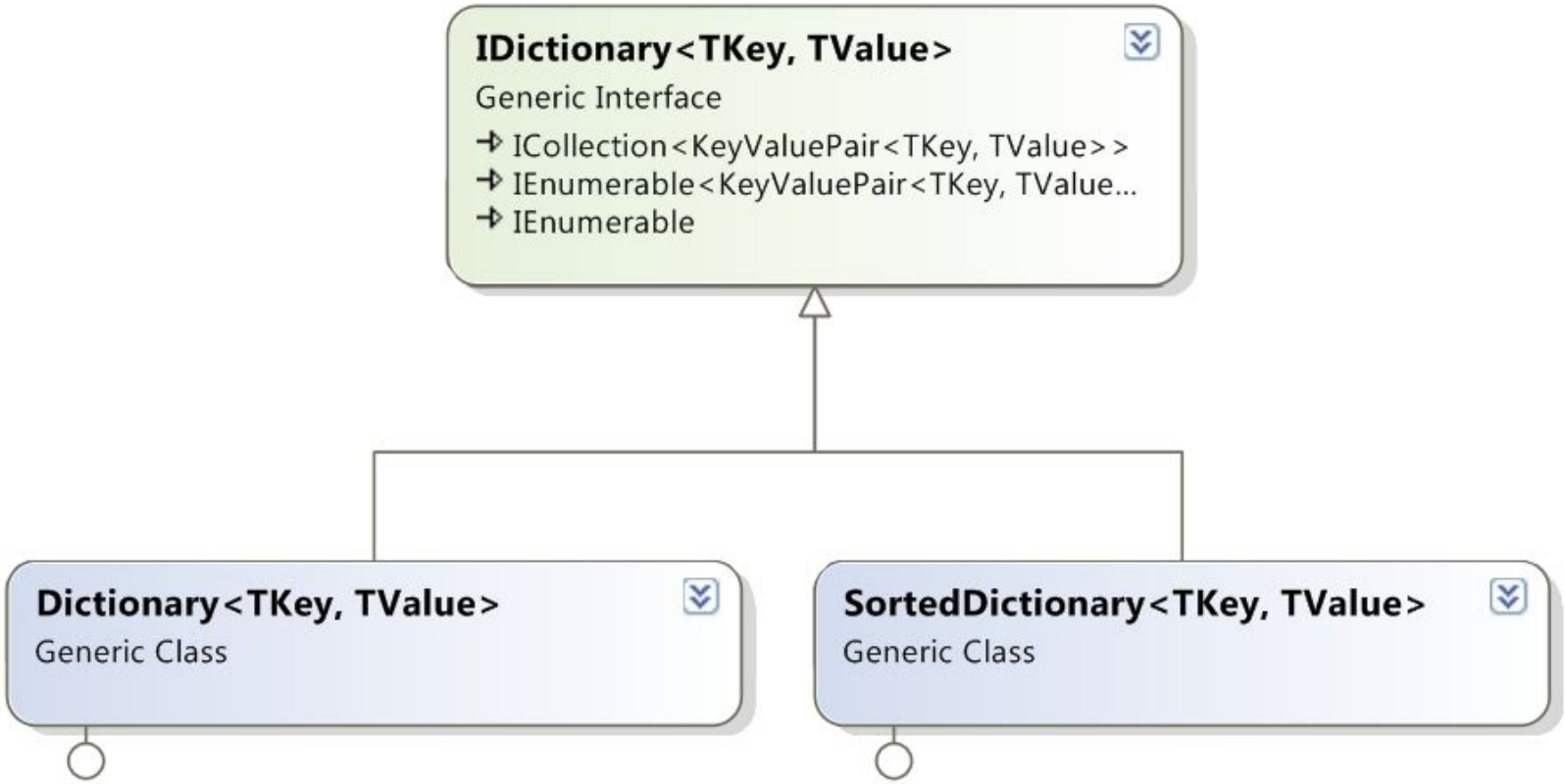
collision



Hash Table

- Hash tables are the most efficient implementation of abstract data type "dictionary"
- Add / Find / Delete take just few primitive operations
  - Speed does not depend on the size of the hash-table (constant time)
  - Example: finding an element in a hash-table with 1000000 elements, takes just few steps
    - Finding an element in array of 1000000 elements takes average 500000 steps

Dictionary


IDictionary<TKey, TValue>
Generic Interface
ICollection<KeyValuePair<TKey, TValue>>
IEnumerable<KeyValuePair<TKey, TValue...
IEnumerable
Dictionary<TKey, TValue>
Generic Class
SortedDictionary<TKey, TValue>
Generic Class

Dictionary

- Implements the abstract data type dictionary as hash table
  - Size is dynamically increased as needed
  - Contains a collection of key-value pairs
  - Collisions are resolved by chaining
  - Elements have almost random order
    - Ordered by the hash code of the key
- Dictionary<TKey, TValue> relies on
  - `Object.Equals()` – for comparing the keys
  - `Object.GetHashCode()` – for calculating the hash codes of the keys

Dictionary

```
string text = "a text, some text, just some text";
IDictionary<string, int> wordsCount =
  new Dictionary<string, int>();

string[] words = text.Split(' ', ',', '.');
foreach (string word in words)
{
  int count = 1;
  if (wordsCount.ContainsKey(word))
    count = wordsCount[word] + 1;
  wordsCount[word] = count;
}

foreach(var pair in wordsCount)
{
  Console.WriteLine("{0} -> {1}", pair.Key, pair.Value);
}
```

Dictionary

- Major operations:
  - `Add(TKey,TValue)` – adds an element with the specified key and value
  - `Remove(TKey)` – removes the element by key
  - `this[]` – get/add/replace of element by key
  - `Clear()` – removes all elements
  - `Count` – returns the number of elements
  - `Keys` – returns a collection of the keys
  - `Values` – returns a collection of the values

Dictionary

- Major operations:
  - `ContainsKey(TKey)` – checks whether the dictionary contains given key
  - `ContainsValue(TValue)` – checks whether the dictionary contains given value
    - Warning: slow operation!
  - `TryGetValue(TKey, out TValue)`
    - If the key is found, returns it in the `TValue`
    - Otherwise returns `false`

Dictionary

```
Dictionary<string, int> studentsMarks =
  new Dictionary<string, int>();
studentsMarks.Add("Ivan", 4);
studentsMarks.Add("Peter", 6);
studentsMarks.Add("Maria", 6);
studentsMarks.Add("George", 5);

int peterMark = studentsMarks["Peter"];
Console.WriteLine("Peter's mark: {0}", peterMark);
Console.WriteLine("Is Peter in the hash table: {0}",
  studentsMarks.ContainsKey("Peter"));

Console.WriteLine("Students and grades:");
foreach (var pair in studentsMarks)
{
  Console.WriteLine("{0} --> {1}", pair.Key, pair.Value);
}
```

SortedDictionary

- `SortedDictionary<TKey,TValue>` implements the abstract data type "dictionary" as self-balancing search tree
  - Elements are arranged in the tree ordered by key
  - Traversing the tree returns the elements in order
  - Add / Find / Delete perform $\log_2(n)$ operations
- Use `SortedDictionary<TKey,TValue>` when you need the elements sorted
  - Otherwise use `Dictionary<TKey,TValue>` – it has better performance

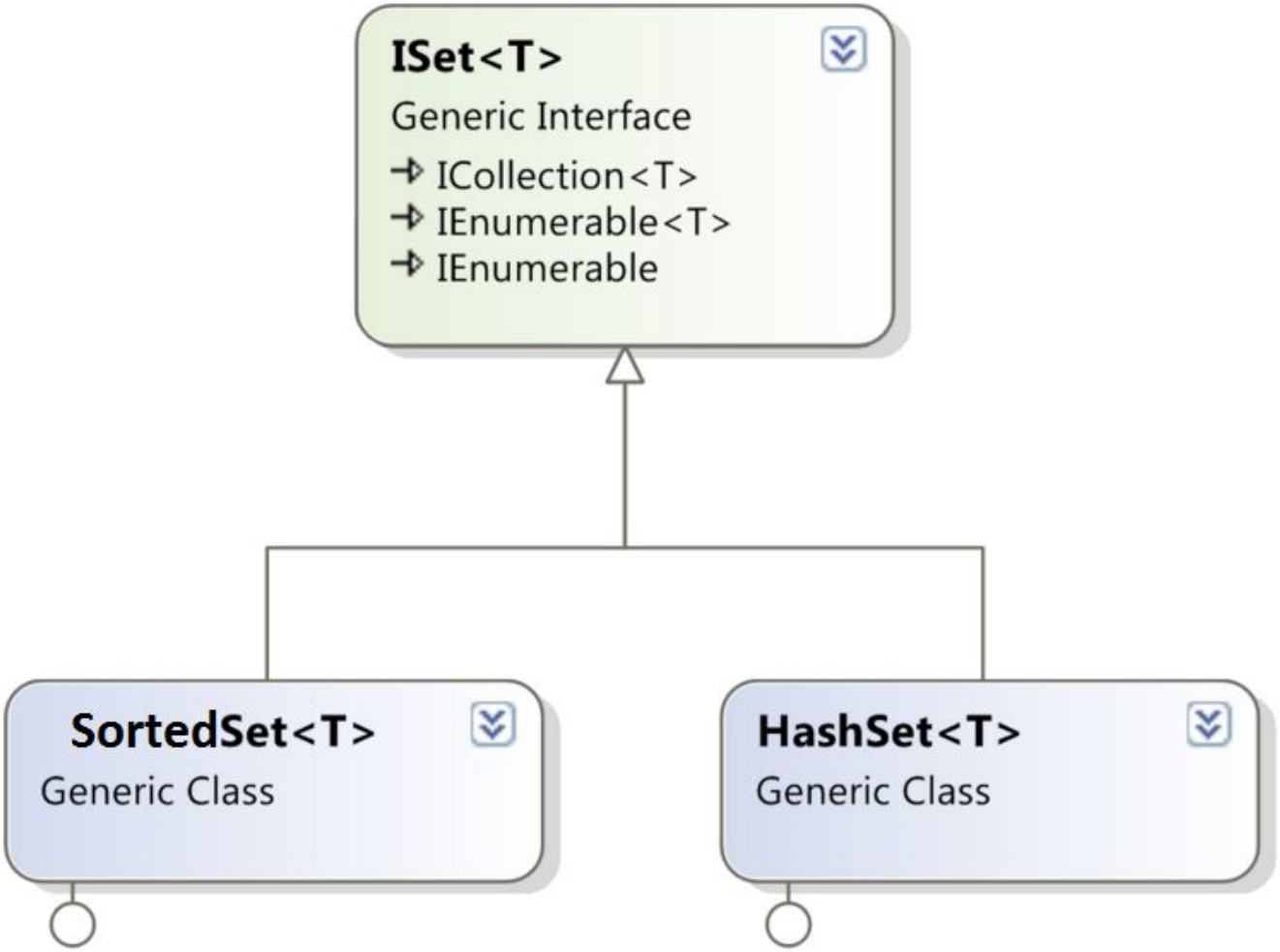
Dictionary

```
string text = "a text, some text, just some text";
IDictionary<string, int> wordsCount =
  new SortedDictionary<string, int>();

string[] words = text.Split(' ', ',', '.');
foreach (string word in words)
{
  int count = 1;
  if (wordsCount.ContainsKey(word))
    count = wordsCount[word] + 1;
  wordsCount[word] = count;
}

foreach(var pair in wordsCount)
{
  Console.WriteLine("{0} -> {1}", pair.Key, pair.Value);
}
```

Sets

- The abstract data type "set" keeps a set of elements with no duplicates
- Sets with duplicates are also known as abstract data type "bag"
- Set operations:
  - `Add(element)`
  - `Contains(element)` → `true / false`
  - `Delete(element)`
  - `Union(set)`
  - `Intersect(set)`
- Sets can be implemented in several ways
  - List, array, hash table, balanced tree, ...

Sets


ISet<T>
Generic Interface
ICollection<T>
IEnumerable<T>
IEnumerable
SortedSet<T>
Generic Class
HashSet<T>
Generic Class

HashSet

- `HashSet<T>` implements abstract data type set by hash table
  - Elements are in no particular order
- All major operations are fast:
  - `Add(element)` – appends an element to the set
    - Does nothing if the element already exists
  - `Remove(element)` – removes given element
  - `Count` – returns the number of elements
  - `UnionWith(set)` – performs union with another set
  - `IntersectWith(set)` – performs intersection with another set

HashSet

```
ISet<string> firstSet = new HashSet<string>(
    new string[] { "SQL", "Java", "C#", "PHP"  });
ISet<string> secondSet = new HashSet<string>(
    new string[] { "Oracle", "SQL", "MySQL"  });

ISet<string> union = new HashSet<string>(firstSet);
union.UnionWith(secondSet);
PrintSet(union); // SQL Java C# PHP Oracle MySQL

private static void PrintSet<T>(ISet<T> set)
{
    foreach (var element in set)
    {
        Console.Write("{0} ", element);
    }
    Console.WriteLine();
}
```

SortedSet

- SortedSet<T> implements abstract data type set by balanced search tree
  - Elements are sorted in increasing order
- Example:

```
ISet<string> firstSet = new SortedSet<string>(
  new string[] { "SQL", "Java", "C#", "PHP" });
ISet<string> secondSet = new SortedSet<string>(
  new string[] { "Oracle", "SQL", "MySQL" });
ISet<string> union = new HashSet<string>(firstSet);
union.UnionWith(secondSet);
PrintSet(union); // C# Java PHP SQL MySQL Oracle
```

Școala informală de IT

Dictionaries and Sets

Collection	Ordering	Contiguous Storage?	Access	Notes
Dictionary	Unordered	Yes	Via Key	Best for high performance lookups.
SortedDictionary	Sorted	No	Via Key	Compromise of Dictionary speed and ordering, uses binary search tree.
SortedList	Sorted	Yes	Via Key	Very similar to SortedDictionary, except tree is implemented in an array, so has faster lookup on preloaded data, but slower loads.
List	Precise control over element ordering	Yes	Via Index	Best for smaller lists where direct access required and no sorting.
LinkedList	Precise control over element ordering	No	No	Best for lists where inserting/deleting in middle is common and no direct access required.
HashSet	Unordered	Yes	Via Key	Unique unordered collection, like a Dictionary except key and value are same object.
SortedSet	Sorted	No	Via Key	Unique sorted collection, like SortedDictionary except key and value are same object.
Stack	LIFO	Yes	Only Top	Essentially same as List<T> except only process as LIFO
Queue	FIFO	Yes	Only Front	Essentially same as List<T> except only process as FIFO