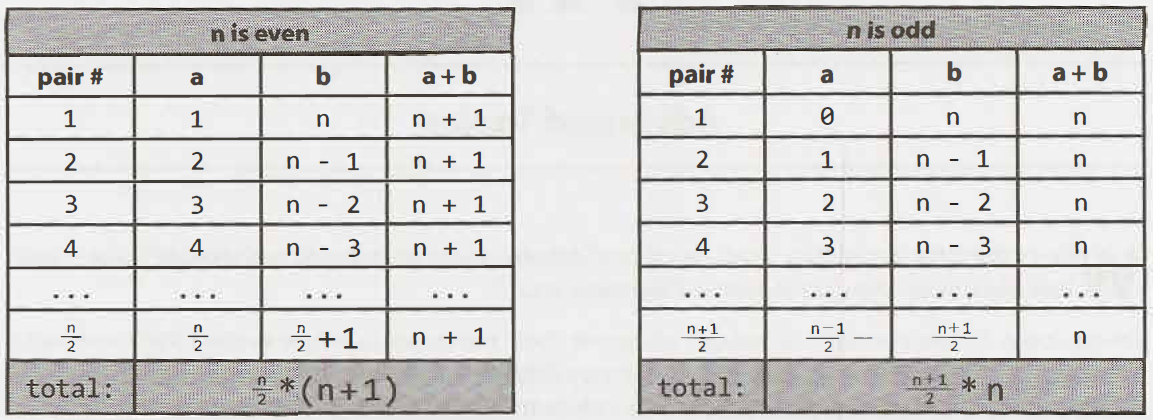
**Useful Math**

* [Sum of integers 1 through N](https://github.com/KiraDiShira/Cracking/blob/master/UsefulMath/readme.md#sum-of-integers-1-through-n)
* [Sum of Powers of 2](https://github.com/KiraDiShira/Cracking/blob/master/UsefulMath/readme.md#sum-of-powers-of-2)
* [Combinatorics](https://github.com/KiraDiShira/Cracking/blob/master/UsefulMath/readme.md#combinatorics)

**Sum of integers 1 through N**

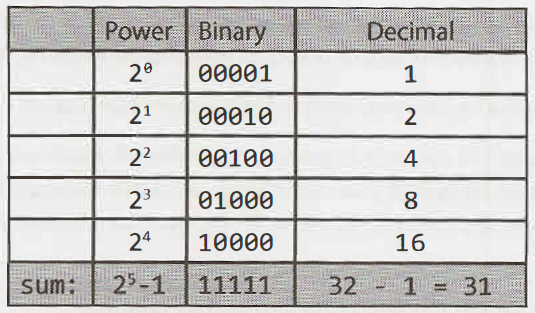
What is 1 + 2 + ... + n? Let's figure it out by pairing up low values with high values.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/um1.PNG)

In either case, the sum is n \* (n +1) / 2

**Sum of Powers of 2**

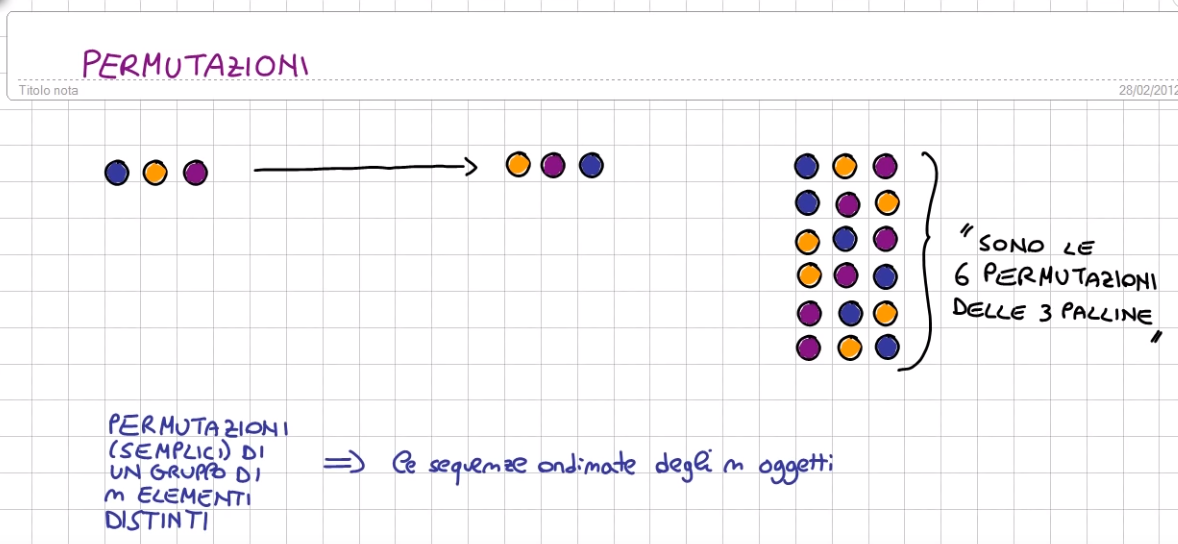
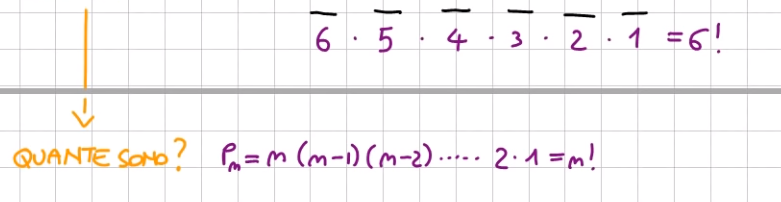
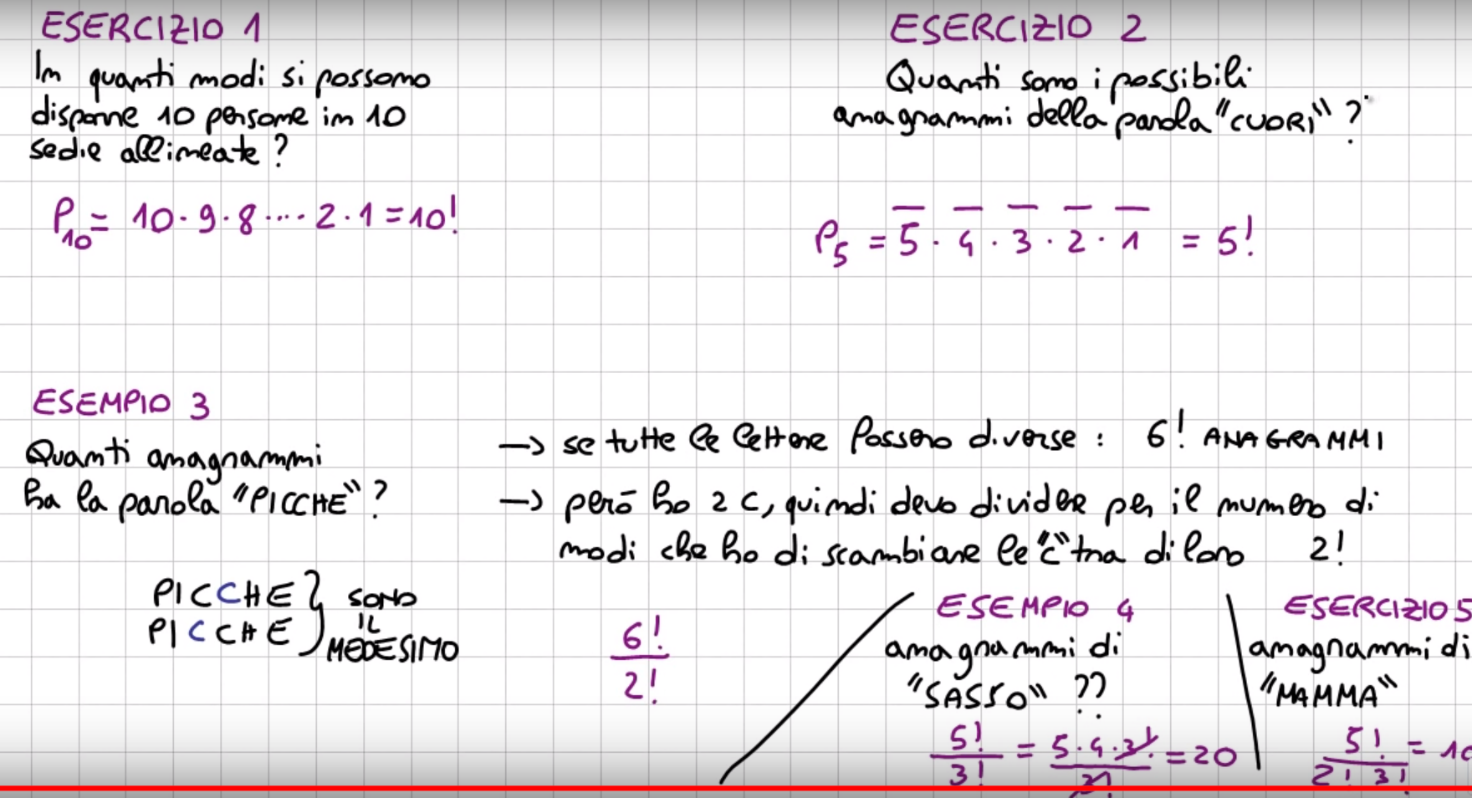
2^0 + 2^1 + 2^2 + ... + 2^n. What is its result?

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/um2.PNG)

2^(n + 1) - 1

**Combinatorics**

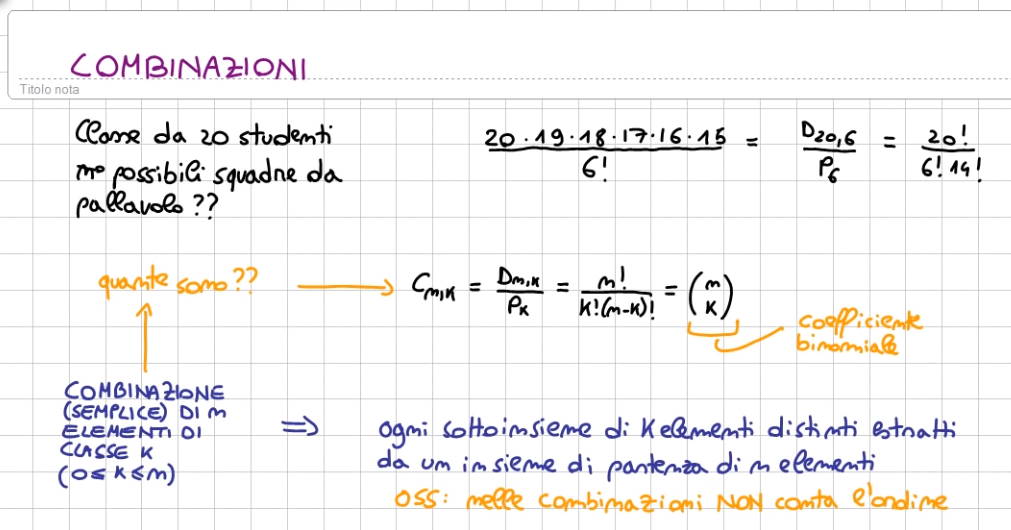
**Permutazioni**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/p1.PNG)[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/p2.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/p3.PNG)

**Disposizioni**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/p4.PNG)

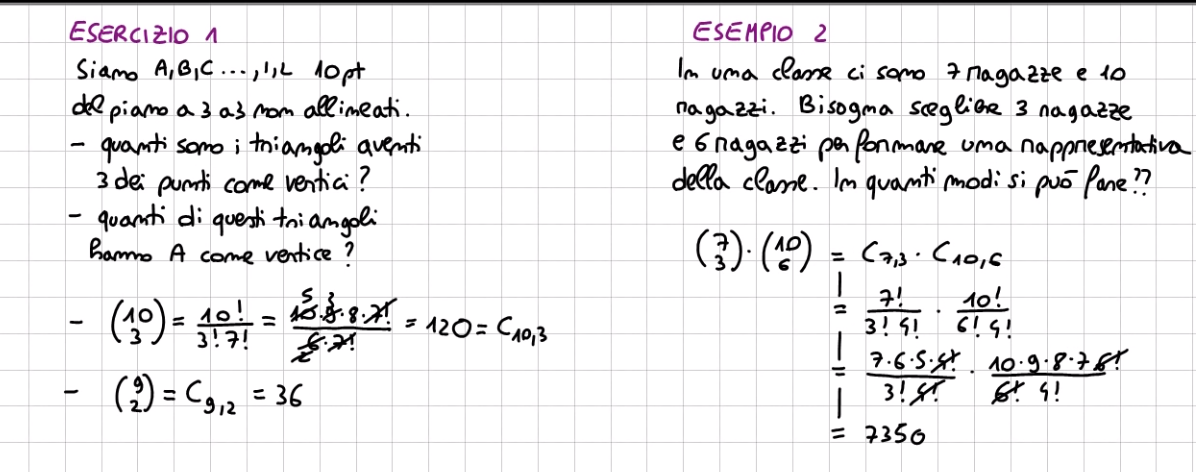
**Combinazioni**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/p5.PNG)

Differenza fra le combinazioni e le disposizioni.

Se voglio fare un gruppetto ordinato di k elementi preso da un gruppo di partenza di n allora dovrò fare le disposizioni. Se mi interessa ordine in cui compaiono: disposizioni.

Se invece mi interessa solo chi c’ è all’interno del gruppo di k elementi parlerò di combinazioni.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/UsefulMath/Images/p6.PNG)

**Big-O Notation**

* [Asymptotic notation](https://github.com/KiraDiShira/Cracking/tree/master/BigONotation#asymptotic-notation)
* [Big-O](https://github.com/KiraDiShira/Cracking/tree/master/BigONotation#big-o)
* [Using Big-O](https://github.com/KiraDiShira/Cracking/tree/master/BigONotation#using-big-o)

**Asymptotic notation**

**Runtimes** is how long takes a program to work;

Computing runtimes is hard:

* depends on details of program
* depends on details of computer

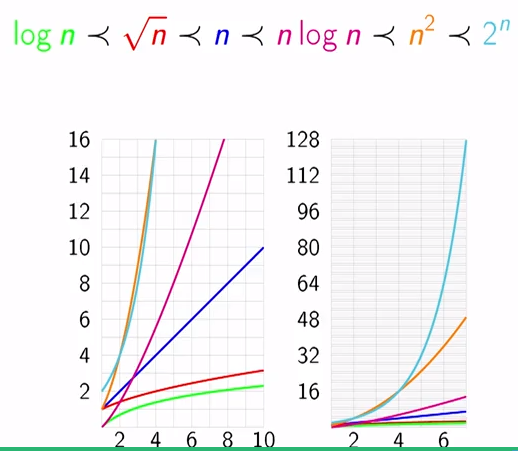
We need something that's a little bit less precise but much easier to work with.

* **Basic idea**: there are lots of factors that have an effect on the final runtime but, most of them will only change the runtimes by a constant. If you're running on a computer that's a hundred times faster, it will take one hundreth of the time, a constant multiple. If your system architecture has multiplications that take three times as long as additions, then if your program is heavy on multiplications instead of additions, it might take three times as long, but it's only a factor of three. If your memory hierarchy is arranged in a different way, you might have to do disk lookups instead of RAM lookups. And those will be a lot slower, but only by a constant multiple.

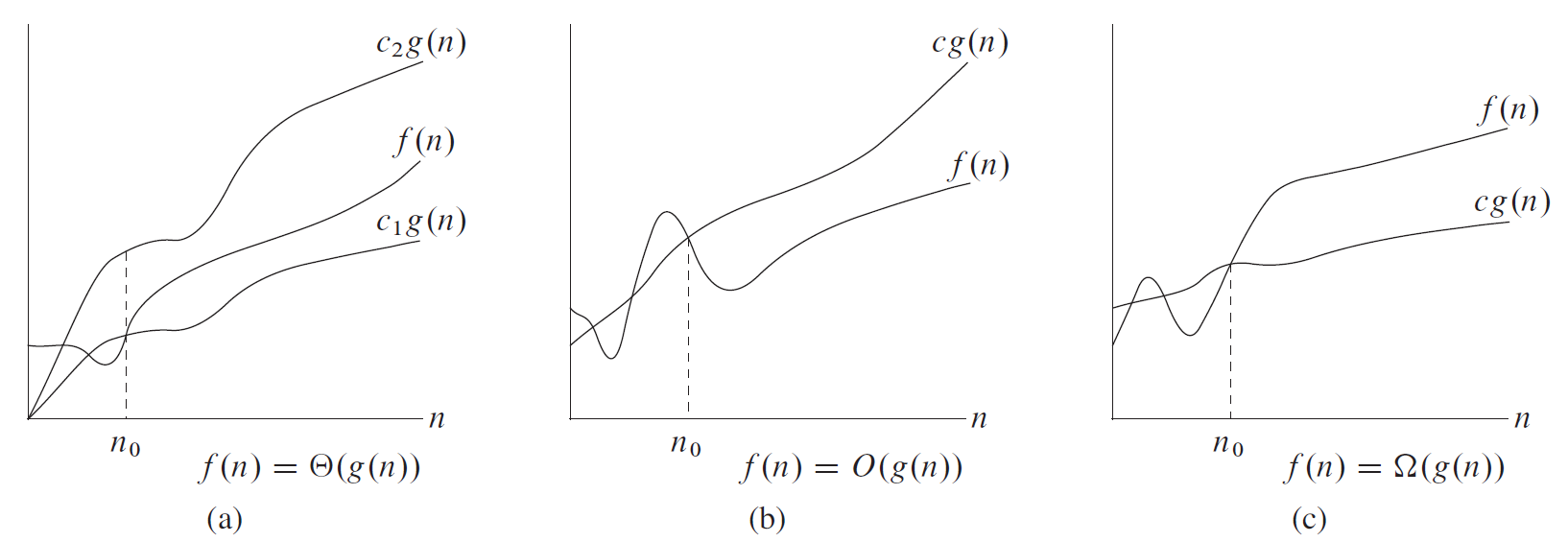
So if we come up with a measure of runtime complexity that ignores all of constant multiples, where running in time n and in running in time 100n are sort of considered to be the same thing, then we don't have to worry about all of these little, bitty details that affect runtime.

* **Problem**: runtimes of 1 second or 1 hour or 1 year, these only differ by constant multiples.
* **Solution**: we're not going to actually consider the runtimes of our programs on any particular input. We're going to look at what are known as **asymptotic runtimes**: how does the runtime scale with input size. As the input size n gets larger, does the output scale proportional to n, maybe proportional to n squared? Is it exponential in n? All these things are different. And in fact they're sort of so different that as long as n is sufficiently large, the difference between n runtime and n squared runtime is going to be worse than any constant multiple.

Only caring about what happens in this sort of long scale behavior, we will be able to do this without seeing these constants, without having to care about these details.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/BigONotation/Images/BigO1.PNG)

**Big-O**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/BigONotation/Images/BigO2.PNG)

Big-O notation is an asymptotic notation.

f (n) = O(g(n)) (f is Big-O of g) or f ⪯ g if there exist constants N and c so that for all n ≥ N, f (n) ≤ c · g(n).

At least for sufficiently large inputs, f is bounded above by some constant multiple of g.

grows no faster than g(x)

Advantages:

* clarifies growth rate
* cleans up notation: O(n^2) vs. 3n^2 + 5n + 2.
* we don't need to know how fast computer is, memory management, system architecture ...

Warnings:

* Using Big-O loses important information about constant multiples.
* Big-O is only asymptotic.

**Other notations:**

For functions f , g : N → R+ we say that:

f (n) = Ω(g(n)) or f ⪰ g if for some c, f (n) ≥ c · g(n) (f grows no slower than g).

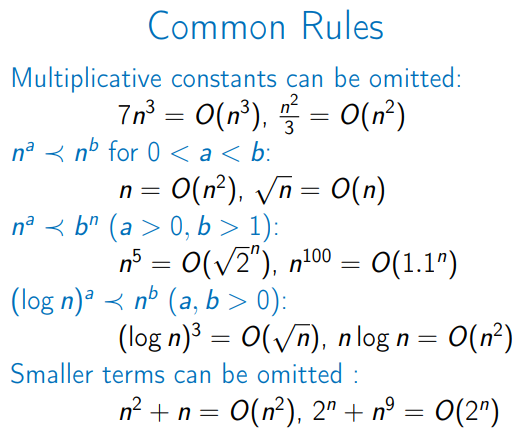
f (n) = Θ(g(n)) or f ≍ g if f = O(g) and f = Ω(g) (f grows at the same rate as g).

f (n) = o(g(n)) or f ≺ g if f (n)/g(n) → 0 as n → ∞ (f grows slower than g).

We use o-notation to denote an upper bound that is not asymptotically tight.

The definitions of O-notation and o-notation are similar. The main difference is that in f (n) = O(g(n)), the bound 0 ≤ f (n) ≤ c · g(n) holds for some constant c > 0, but in f (n) = o(g(n)), the bound 0 ≤ f (n) < c · g(n) holds for all constants c > 0.

**Using Big-O**

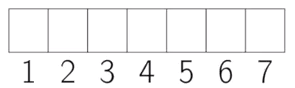
[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/BigONotation/Images/BigO3.PNG)

When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a O(log N) runtime.

When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like O(branches ^ depth), where branches is the number of times each recursive call branches.

# Array

**Array**: contiguous area of memory consisting of equal-size elements indexed by countiguos integers.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Array/Images/arr1.PNG)

What's special about arrays? Constant-time access: O(1)

array\_addr + elem\_size \* (i - first\_index)

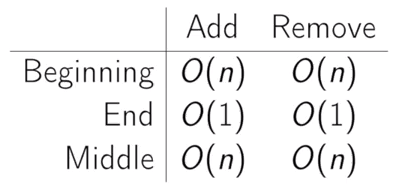
Constant-time access also for **multidimensional arrays**:

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Array/Images/arr2.PNG)

We need to skip the full rows that we are not using ((3 - 1) \* 6), and then the situation is like for mono dimensional arrays:

array\_addr + elem\_size \* ((3 - 1) \* 6 + (4 - 1))

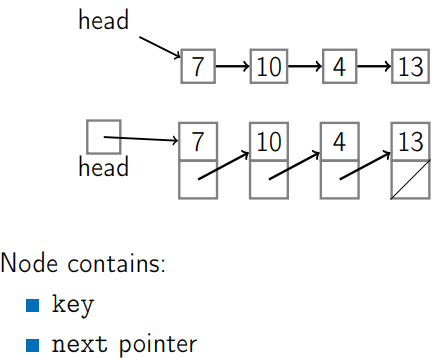
Time for common operations:

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Array/Images/arr4.PNG)

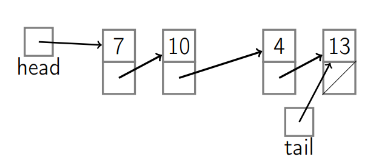
**Singly-Linked Lists**

* [Definition](https://github.com/KiraDiShira/Cracking/tree/master/SinglyLinkedList#definition)
* [List Api](https://github.com/KiraDiShira/Cracking/tree/master/SinglyLinkedList#list-api)
* [Times for Some Operations](https://github.com/KiraDiShira/Cracking/tree/master/SinglyLinkedList#times-for-some-operations)
* [Other operations](https://github.com/KiraDiShira/Cracking/tree/master/SinglyLinkedList#other-operations)
* [Summary](https://github.com/KiraDiShira/Cracking/tree/master/SinglyLinkedList#summary)

**Definition**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll1b.PNG)

It's possible to have also a **Tail**: a pointer to the last node.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/Tail.PNG)

public class Node

{

public object Key { get; set; }

public Node Next { get; set; }

}

public class SinglyLinkedList

{

public Node Head { get; set; }

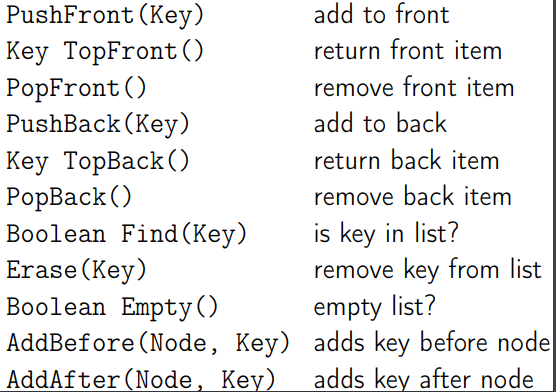
public Node Tail { get; set; }

public void PushFront(object key)

{

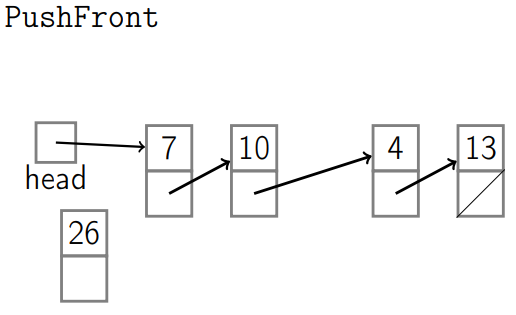
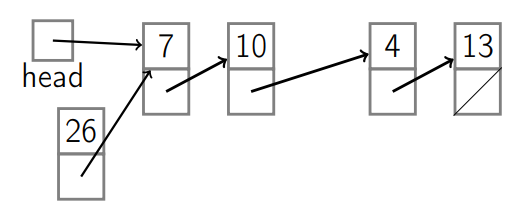
...

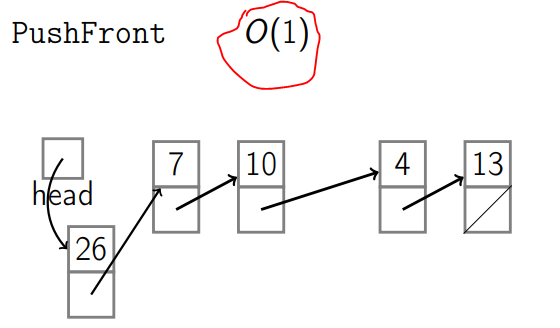
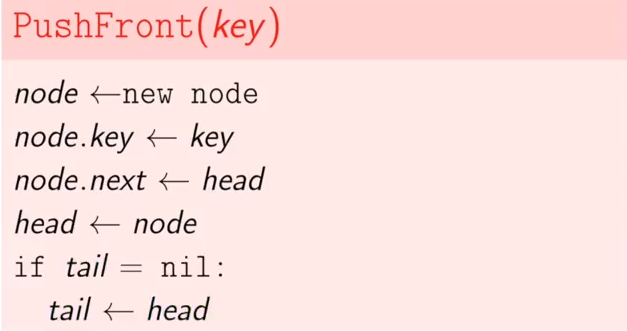
**List API**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll2b.PNG)

**Times for Some Operations**

**PushFront**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll3.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll4.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll5.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/codice1.PNG)

public void PushFront(object key)

{

var node = new Node()

{

Key = key,

Next = Head

};

Head = node;

//Se la coda non punta ad alcuna parte vuol dire che prima la lista era vuota e

//il nuovo nodo è il primo nodo, quindi faccio puntare la tail all'unico elemento inserito

if (Tail == null)

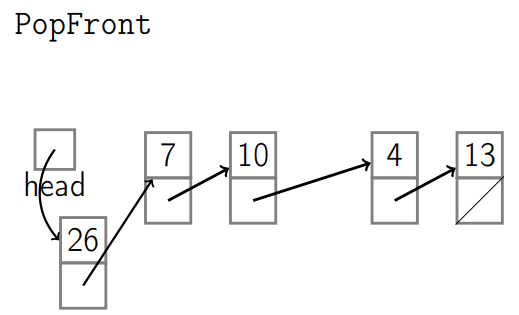
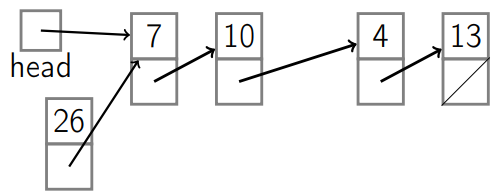
{

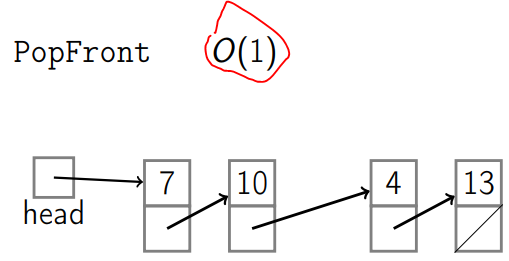
Tail = Head;

}

}

**PopFront**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll6.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll7.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll8.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/codice2.PNG)

public void PopFront()

{

if (Head == null)

{

throw new Exception("ERROR: empty list");

}

Head = Head.Next;

if (Head == null) //significa che la lista è vuota, indi setto a null anche la tail

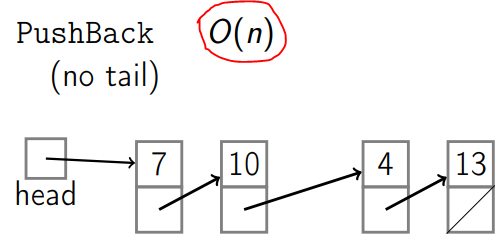
{

Tail = null;

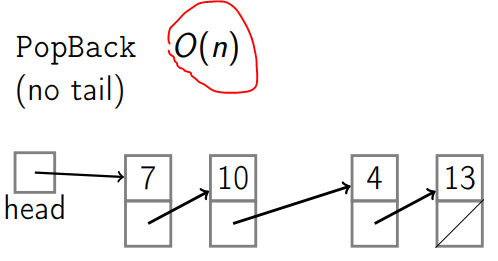
}

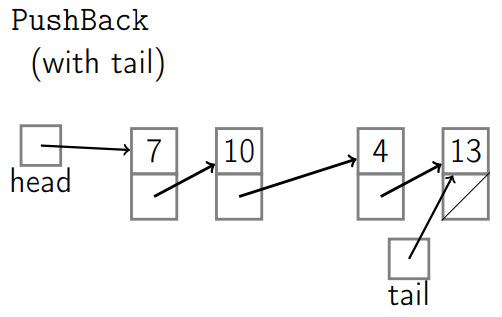
}

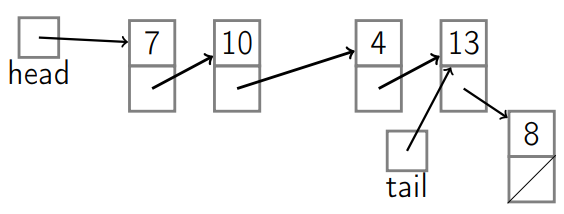
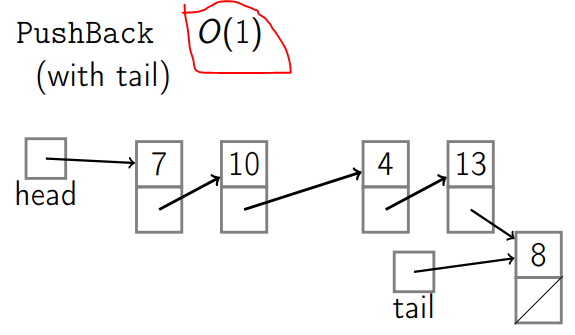
**Times for PushBack (no tail)**

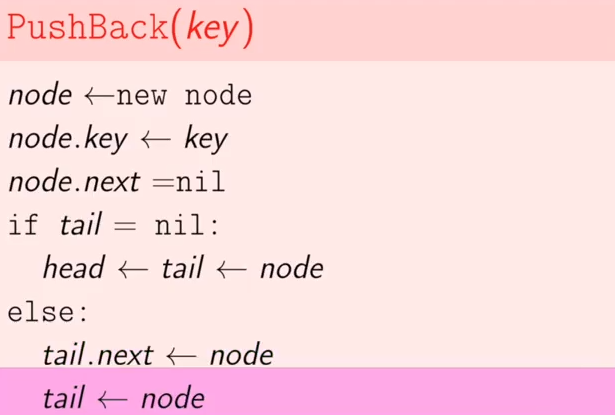
[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll9.PNG)

**Times for PopBack (no tail)**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll10.PNG)

**Times for PushBack (with tail)**[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll11.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll12.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll13.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll14.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/codice3.PNG)

public void PushBack(object key)

{

var node = new Node()

{

Key = key,

Next = null

};

if (Tail == null)

{

Head = node;

Tail = node;

}

else

{

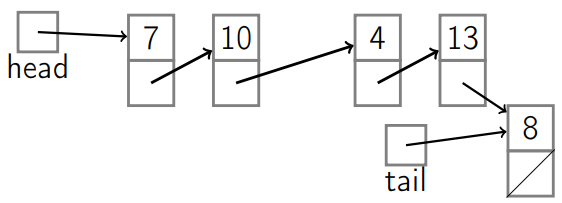
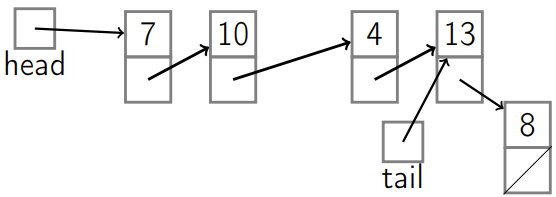
Tail.Next = node;

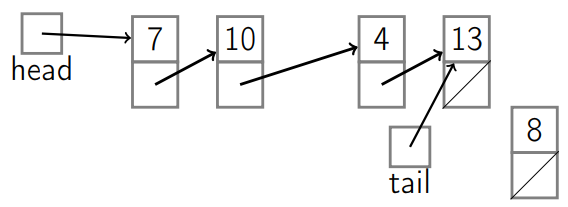
Tail = node;

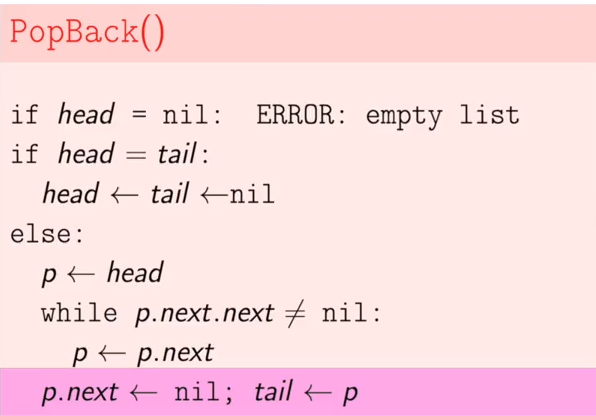
}

}

**Times for PopBack (with tail)**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll15.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll16.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll17.PNG) [](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/sll18.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/codice4.PNG)

public void PopBack()

{

if (Head == null)

{

throw new Exception("ERROR: empty list");

}

if (Head == Tail)

{

Head = null;

Tail = null;

}

else

{

Node currentNode = Head;

//mi trova il penultimo elemento

while (currentNode.Next.Next != null)

{

currentNode = currentNode.Next;

}

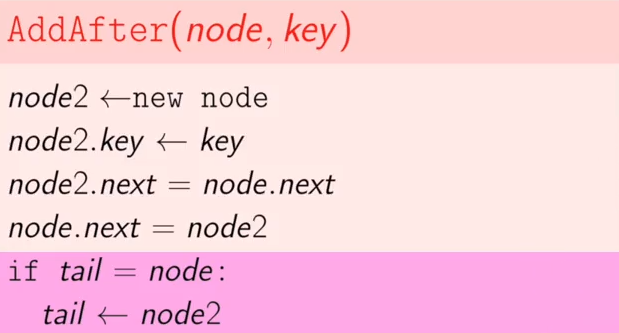
currentNode.Next = null;

Tail = currentNode;

}

}

**Add After**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/codice5.PNG)

public void AddAfter(Node node, object key)

{

var newNode = new Node()

{

Key = key,

Next = node.Next

};

node.Next = newNode;

if (Tail == node)

{

Tail = newNode;

}

}

**Other operations**

public object TopFront()

{

if (Head == null)

{

throw new Exception("ERROR: empty list");

}

return Head.Key;

}

public object TopBack()

{

if (Head == null)

{

throw new Exception("ERROR: empty list");

}

return Tail.Key;

}

public bool Find(object key)

{

if (Head == null)

{

return false;

}

Node currentNode = Head;

if (currentNode.Key.Equals(key))

{

return true;

}

//mi trova il ultimo elemento

while (currentNode.Next != null)

{

currentNode = currentNode.Next;

if (currentNode.Key.Equals(key))

{

return true;

}

}

return false;

}

public void Erase(object key)

{

if (Head == null)

{

throw new Exception("cannot delete");

}

if (Head.Key.Equals(key))

{

Head = Head.Next;

if (Head == null)

{

Tail = null;

}

return;

}

Node currentNode = Head;

Node previousNode = null;

while (currentNode != null && !currentNode.Key.Equals(key))

{

previousNode = currentNode;

currentNode = currentNode.Next;

}

if (currentNode == null) throw new Exception("cannot delete");

//delete cur node

previousNode.Next = currentNode.Next;

}

public bool Empty()

{

if (Tail == null)

{

return true;

}

return false;

}

public void AddBefore(Node node, object key)

{

if (Head == null)

{

return;

}

if (Head.Key.Equals(key))

{

PushFront(key);

return;

}

Node currentNode = Head;

Node previousNode = null;

while (currentNode != null && !currentNode.Key.Equals(key))

{

previousNode = currentNode;

currentNode = currentNode.Next;

}

if (currentNode == null)

{

return;

}

var newNode2 = new Node()

{

Key = key,

Next = currentNode

};

previousNode.Next = newNode2;

}

public override string ToString()

{

if (Head == null)

{

return "NULL";

}

string toString = Head.Key + "->";

Node curr = Head;

while (curr.Next != null)

{

curr = curr.Next;

toString += curr.Key;

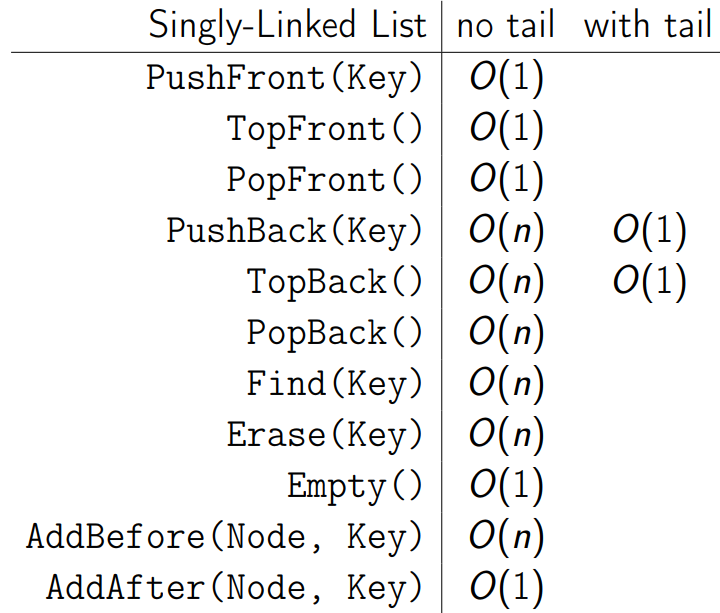
toString += "->";

}

return toString;

}

**Summary**

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/SinglyLinkedList/Images/codice6.PNG)

If I need a reference to a node different from the Head or the Tail the running time is O(n).

**PopBack** is O(n) because I need a reference to last minus one node.

**Find** and **Erase** are linear searching operation so they are O(n)

**AddBefore** is O(n) because I need a reference to the node previuous to my target node.

# Doubly-Linked Lists

* [Definition](https://github.com/KiraDiShira/Cracking/blob/master/DoublyLinkedList/Readme.md#definition)
* [Times for Some Operations](https://github.com/KiraDiShira/Cracking/blob/master/DoublyLinkedList/Readme.md#times-for-some-operations)
* [Summary](https://github.com/KiraDiShira/Cracking/blob/master/DoublyLinkedList/Readme.md#summary)

## Definition

There is a way to make popping the back and adding before cheap: we need a reference to the previous node.

public class Node

{

public object Key { get; set; }

public Node Next { get; set; }

public Node Prev { get; set; }

}

public class DoublyLinkedList

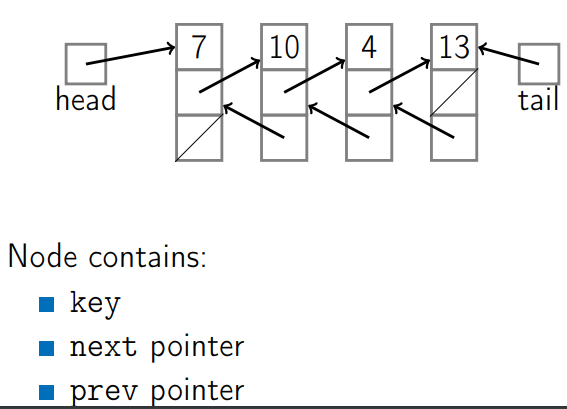
{

public Node Head { get; set; }

public Node Tail { get; set; }

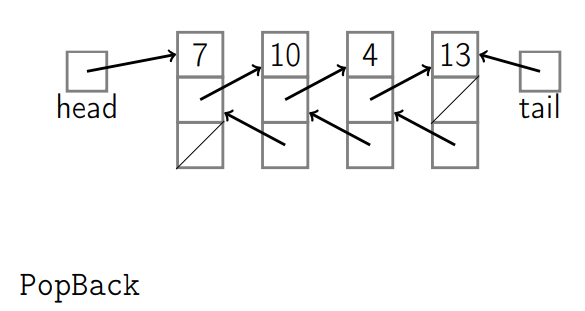
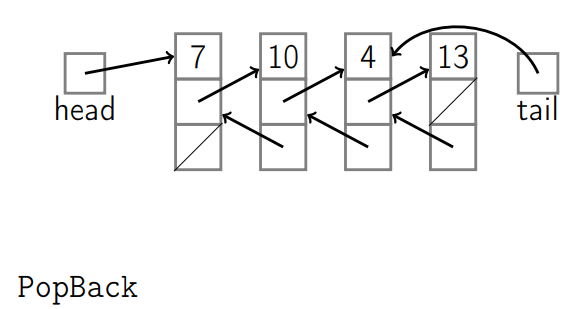
...

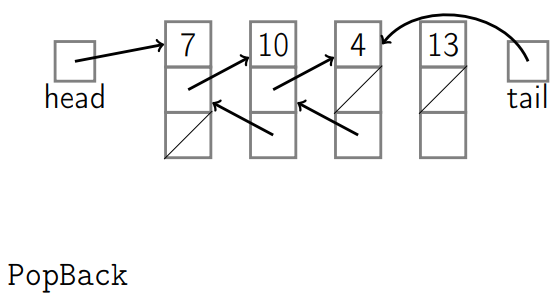
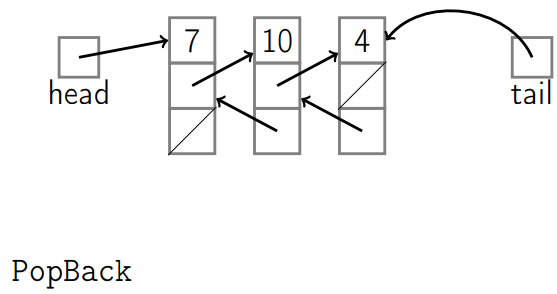
**Doubly-Linked Lists**:

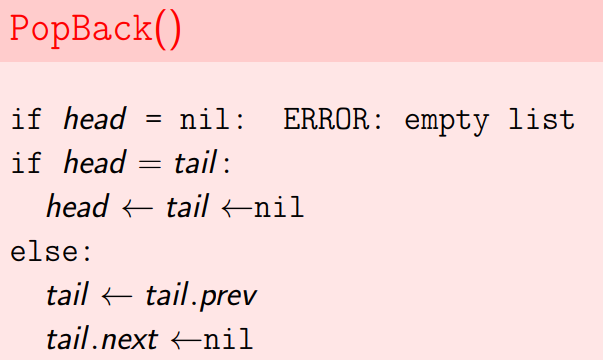
[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll1.PNG)

## Times for some operations

### PopBack

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll2.PNG)[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll3.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll4.PNG)[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll5.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll6.PNG)

public void PopBack()

{

if (Head == null)

{

throw new Exception("ERROR: empty list");

}

if (Head == Tail)

{

Head = null;

Tail = null;

return;

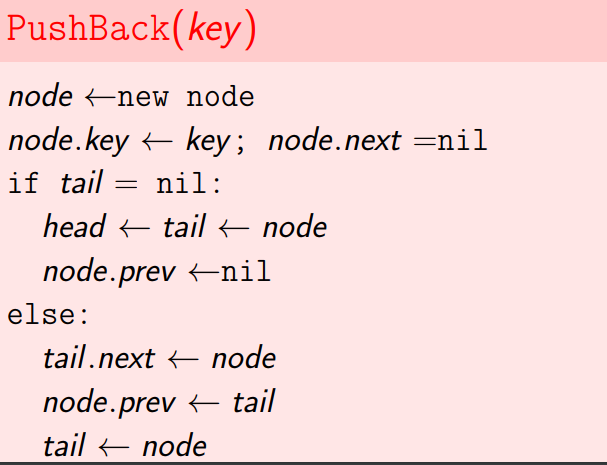
}

Tail = Tail.Prev;

Tail.Next = null;

}

### PushBack

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll7.PNG)

public void PushBack(object key)

{

var node = new Node()

{

Key = key,

Next = null

};

if (Tail == null)

{

Head = node;

Tail = node;

node.Prev = null;

}

else

{

node.Prev = Tail;

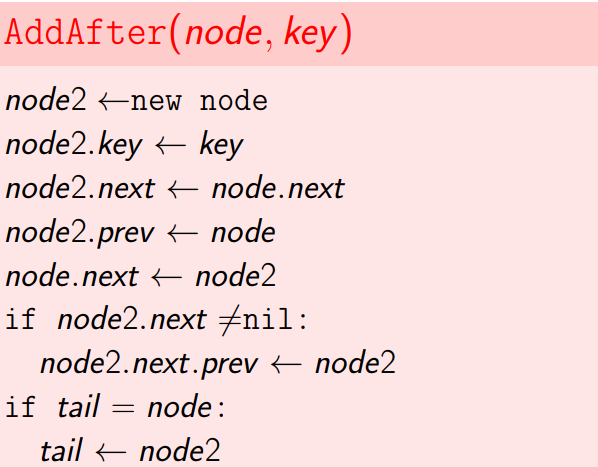
Tail.Next = node;

Tail = node;

}

}

### AddAfter

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll8.PNG)

public void AddAfter(Node node, object key)

{

var newNode = new Node()

{

Key = key,

Next = node.Next,

Prev = node

};

node.Next = newNode;

//devo aggiornare il nodo precedente a quello successivo a newnode

if (newNode.Next != null)

{

newNode.Next.Prev = newNode;

}

if (Tail == node)

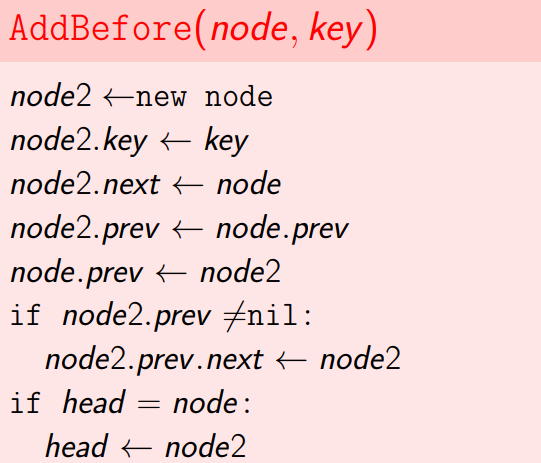
{

Tail = newNode;

}

}

### AddBefore

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll9.PNG)

public void AddBefore(Node node, object key)

{

if (Head == null)

{

return;

}

var newNode = new Node()

{

Key = key,

Next = node,

Prev = node.Prev

};

node.Prev = newNode;

if (newNode.Prev != null)

{

newNode.Prev.Next = newNode;

}

if (Head == node)

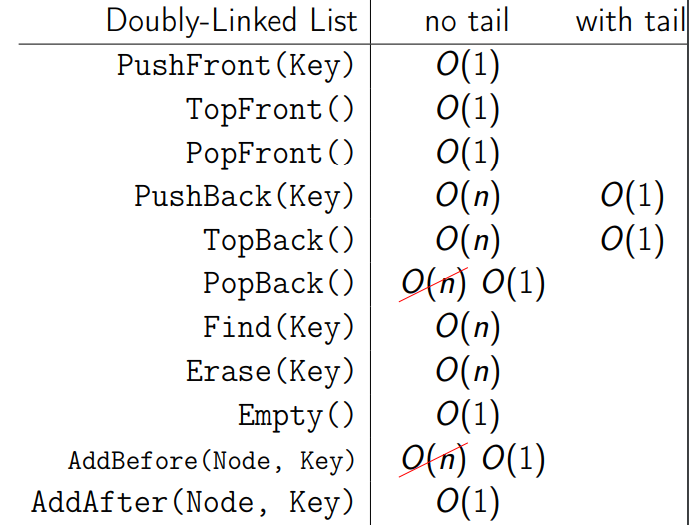
{

Head = newNode;

}

}

### Summary

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DoublyLinkedList/Images/dll10.PNG)

# Stack

* [Definition](https://github.com/KiraDiShira/Cracking/tree/master/Stack#definition)
* [Balanced Brackets example](https://github.com/KiraDiShira/Cracking/tree/master/Stack#balanced-brackets-example)
* [Stack Implementation with Array](https://github.com/KiraDiShira/Cracking/tree/master/Stack#stack-implementation-with-array)
* [Stack Implementation with Linked List](https://github.com/KiraDiShira/Cracking/tree/master/Stack#stack-implementation-with-linked-list)
* [Summary](https://github.com/KiraDiShira/Cracking/tree/master/Stack#summary)

## Definition

**Stack**: Abstract data type with the following operations:

* Push(Key): adds key to collection
* Key Top(): returns most recently-added key
* Key Pop(): removes and returns most recently-added key
* Boolean Empty(): are there any elements?

Stack is useful when you need to be keep track of what has happened in a particular order.

Stacks can be implemented with either an **array** or a **linked list**.

Stacks are ocassionaly known as **LIFO queues**.

Each stack operation is O(1): Push, Pop, Top, Empty.

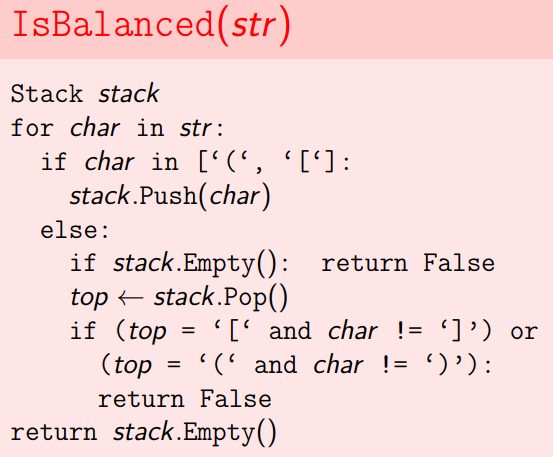
## Balanced Brackets example

**Balanced**:

* ([])[]()
* ((([([])]))())

**Unbalanced**:

* ([]]()
* ][

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Stack/Images/st1.PNG)

private static IDictionary<char, char> \_brackets = new Dictionary<char, char>()

{

{ '(', ')'},

{ '[', ']'},

{ '{', '}'},

};

static bool IsBalanced(string source)

{

Stack<char> stack = new Stack<char>();

foreach (char character in source)

{

if (IsOpenBracket(character))

{

stack.Push(character);

}

else

{

if (IsStackEmpty(stack))

{

return false;

}

char top = stack.Pop();

if (\_brackets.ContainsKey(top) && character != \_brackets[top])

{

return false;

}

}

}

return IsStackEmpty(stack);

}

private static bool IsStackEmpty(Stack<char> stack)

{

return stack.Count == 0;

}

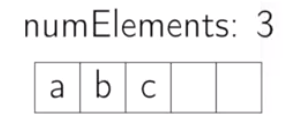
private static bool IsOpenBracket(char character)

{

return \_brackets.ContainsKey(character);

}

## Stack Implementation with Array

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Stack/Images/arr1.PNG)

public class MyStack<T>

{

private int \_index;

private readonly T[] \_array;

public MyStack(int size)

{

\_array = new T[size];

\_index = 0;

}

public void Push(T item)

{

if (\_index >= \_array.Length)

{

throw new IndexOutOfRangeException("pushing index error");

}

\_array[\_index] = item;

\_index++;

}

public T Peek()

{

return \_array[\_index - 1];

}

public T Pop()

{

if (\_index - 1 < 0)

{

throw new IndexOutOfRangeException("popping index error");

}

T item = \_array[\_index - 1];

\_index--;

return item;

}

public bool IsEmpty()

{

if (\_index == 0)

{

return true;

}

return false;

}

public override string ToString()

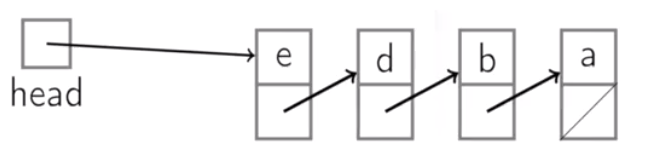
{

return String.Join(",", \_array.Take(\_index));

}

}

## Stack Implementation with Linked List

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Stack/Images/sst1.PNG)

One limitation of the array is that we have a maximum size, based on the array we initially allocated.

The other potential problem is that we have potentially wasted space. So if we allocated a very large array, to allow a possibly large stack, we didn't actually use much of it, all the rest of it is wasted.

If we have a linked list, there's no a priori limit as to the number of elements you can add. As long as you have available memory, you can keep adding. There's an overhead though, like in the array, we have each element size, is just big enough to store our key. Here we've got the overhead of storing a pointer as well. On the other hand there's no wasted space in terms of allocated space that isn't actually being used.

public class LinkedListStack<T>

{

private readonly SinglyLinkedList<T> \_singlyLinkedList;

public LinkedListStack()

{

\_singlyLinkedList = new SinglyLinkedList<T>();

}

public void Push(T item)

{

\_singlyLinkedList.PushFront(item);

}

public T Peek()

{

return \_singlyLinkedList.TopFront();

}

public T Pop()

{

T top = \_singlyLinkedList.TopFront();

\_singlyLinkedList.PopFront();

return top;

}

public bool IsEmpty()

{

return \_singlyLinkedList.Empty();

}

public override string ToString()

{

return \_singlyLinkedList.ToString();

}

}

# Queues

* [Definition](https://github.com/KiraDiShira/Cracking/blob/master/Queues/Readme.md#definition)
* [Queue Implementation with Linked List](https://github.com/KiraDiShira/Cracking/blob/master/Queues/Readme.md#queue-implementation-with-linked-list)
* [Queue Implementation with Array](https://github.com/KiraDiShira/Cracking/blob/master/Queues/Readme.md#queue-implementation-with-array)

## Definition

**Queue**: Abstract data type with the following operations:

* Enqueue(Key): adds key to collection
* Key Dequeue(): removes and returns least recently-added key
* Boolean Empty(): are there any elements?

Queues are known as **FIFO queues**.

Queue is useful when you need to be keep track of what has happened in a particular order.

Queues can be implemented with either an **array** or a **linked list**.

Each Queue operation is O(1): Enqueue, Dequeue, Empty.

One distinction between the array and the linked list implementation, is that in the array implementation, we have a maximum size that the queue can grow to. Maybe you want that in which case it's fine, but if you don't know a priori how long the queue you need is going to be an array is a bad choice. And any amount that is unused is wasted space. In a queue that's implemented with a linked list, it can get arbitrarily large as long as there's available memory. The downside is, every element you have to pay for another pointer.

## Queue Implementation with Linked List

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Queues/q1.PNG)

With a linked list, where you have a head and a tail pointer.

Enqueue: we are going to push to the back of the linked list.

Dequeue: that's just an implementation of popping from the front.

public class LinkedListQueue<T>

{

private readonly SinglyLinkedList<T> \_singlyLinkedList;

public LinkedListQueue()

{

\_singlyLinkedList = new SinglyLinkedList<T>();

}

public void Enqueue(T key)

{

\_singlyLinkedList.PushBack(key);

}

public T Dequeue()

{

T key = \_singlyLinkedList.TopFront();

\_singlyLinkedList.PopFront();

return key;

}

public bool IsEmpty()

{

return \_singlyLinkedList.Empty();

}

public override string ToString()

{

return \_singlyLinkedList.ToString();

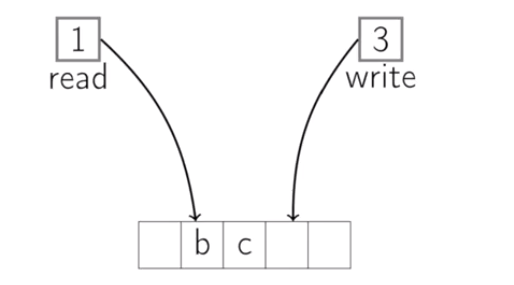
}

}

What would happen if we use List.PushFront to implement Enqueue and List.TopBack and List.PopBack to implement Dequeue?

If the linked-list is implemente as a doubly-linked-list there is no problem. If the linked-list is implemente as a singly-linked-list, Dequeue would be O(n).

## Queue Implementation with Array

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Queues/q2.PNG)

We could add at the end and then pop from the front. Then enqeueing is easy, but dequeuing would be an expensive O(n)operation. And we want enqeueing to be O(1). We can do that, in a fashion I'll show you right now which is basically keeping track of sort of the array as a circular array: so we need a write index and a reading index.

public class ArrayQueue<T>

{

private readonly T[] \_array;

private int \_readingIndex;

private int \_writingIndex;

public ArrayQueue(int size)

{

\_readingIndex = 0;

\_writingIndex = 0;

\_array = new T[size];

}

public void Enqueue(T key)

{

\_array[\_writingIndex] = key;

\_writingIndex++;

if (\_writingIndex == \_array.Length)

{

\_writingIndex = 0;

}

if (\_writingIndex == \_readingIndex)

{

throw new Exception("\_writing index is equal to \_reading index: I can't know if queue is empty");

}

}

public T Dequeue()

{

T key = \_array[\_readingIndex];

\_readingIndex++;

if (\_readingIndex == \_array.Length)

{

\_readingIndex = 0;

}

return key;

}

public bool IsEmpty()

{

return \_readingIndex == \_writingIndex;

}

public override string ToString()

{

if (\_writingIndex >= \_readingIndex)

{

return String.Join(",", \_array.SubArray(\_readingIndex, \_writingIndex - \_readingIndex));

}

return String.Join(",", \_array.SubArray(\_readingIndex, \_array.Length - \_readingIndex)

.Concat(\_array.SubArray(0, \_writingIndex)));

}

}

# Tree

* [Definition](https://github.com/KiraDiShira/Cracking/blob/master/Tree/Readme.md#definition)
* [Depth-first](https://github.com/KiraDiShira/Cracking/blob/master/Tree/Readme.md#depth-first)
* [Breath-first](https://github.com/KiraDiShira/Cracking/blob/master/Tree/Readme.md#breath-first)

## Definition

A **Tree** is:

* empty, or
* a node with:
  + a key, and
  + a list of child trees.
  + (optional) a parent

For binary tree, node contains:

* key
* left
* right
* (optional) parent

public class Tree<T>

{

public T Key { get; set; }

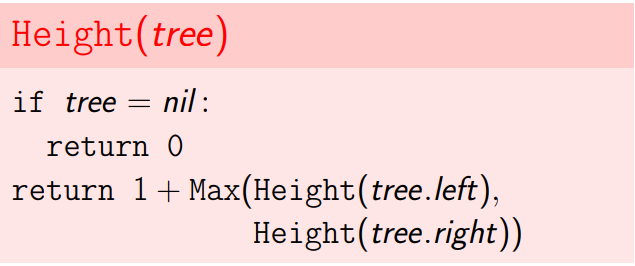
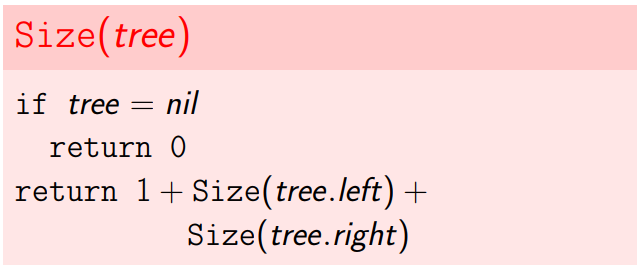
public Tree<T> Left { get; set; }

public Tree<T> Right { get; set; }

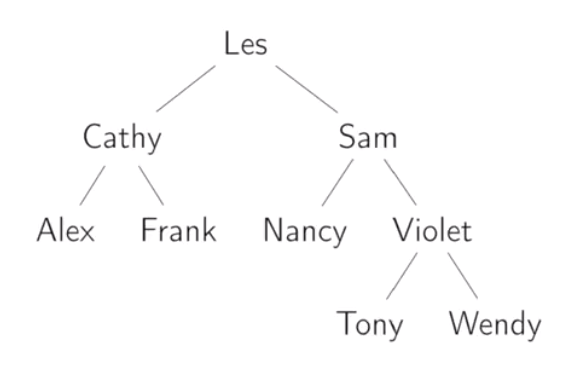
}

**Height**: maximum depth of subtree node and farthest leaf

**Size**: number of nodes

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree1.PNG)[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree2.PNG)

### Binary Search Tree

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree2_b.PNG)

The binary search tree it's binary: it has at most two children at each node.

And we have the property that at the root node, the value of that root node is greater than or equal to all of the nodes in the left child, and it's less than the nodes in the right child.

The binary search tree allows you to search quickly. For instance, if we wanted to search in this tree for Tony, we could start at Les. Notice that we are greater than Les, so therefore, we're going to go right. We're greater than Sam so we'll go right. We're less than Violet so we'll go left and then we find Tony. And we do that in just four comparisons. It's a lot like a binary search in a sorted array.

## Tree Traversal

Often we want to visit the nodes of a tree in a particular order. For example, print the nodes of the tree.

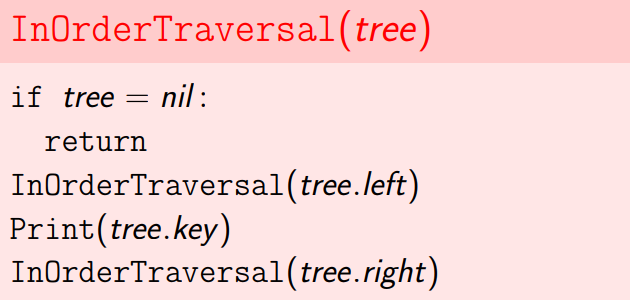
**Depth-first**: We completely traverse one sub-tree before exploring a sibling sub-tree.

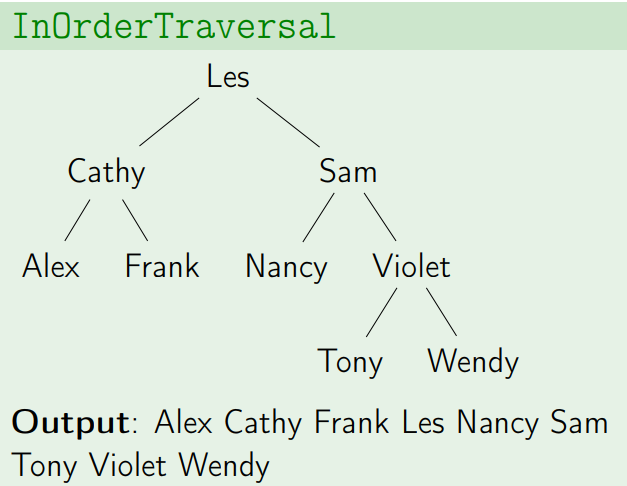
**Breadth-first**: We traverse all nodes at one level before progressing to the next level.

### Depth-first

In order traversal has a meaning only for binary tree, because every tree has maximum two child and between them I can insert the print procedure.

Depth-first search now is implemented in a recursive way, but it can also be implemented with an iterative alghoritm and a stack.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree3.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree4.PNG)

public void InOrderRecursiveTraversal(Tree<T> tree)

{

if (tree == null)

{

return;

}

InOrderRecursiveTraversal(tree.Left);

Console.WriteLine(tree.Key);

InOrderRecursiveTraversal(tree.Right);

}

public void InOrderIterativeTraversal(Tree<T> tree)

{

if (tree == null)

{

return;

}

var stack = new Stack<Tree<T>>();

stack.Push(tree);

Tree<T> actualTree = tree;

while (stack.Count != 0)

{

if (actualTree.Left != null)

{

stack.Push(actualTree.Left);

actualTree = actualTree.Left;

}

else

{

actualTree = stack.Pop();

Console.WriteLine(actualTree.Key);

if (actualTree.Right != null)

{

stack.Push(actualTree.Right);

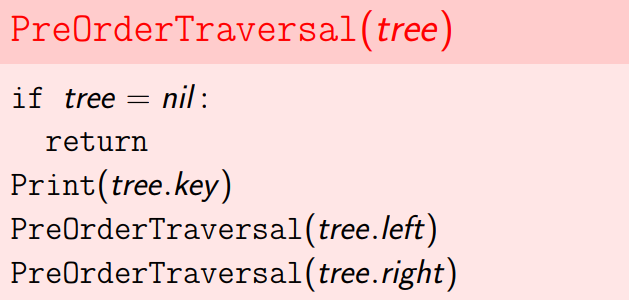
actualTree = actualTree.Right;

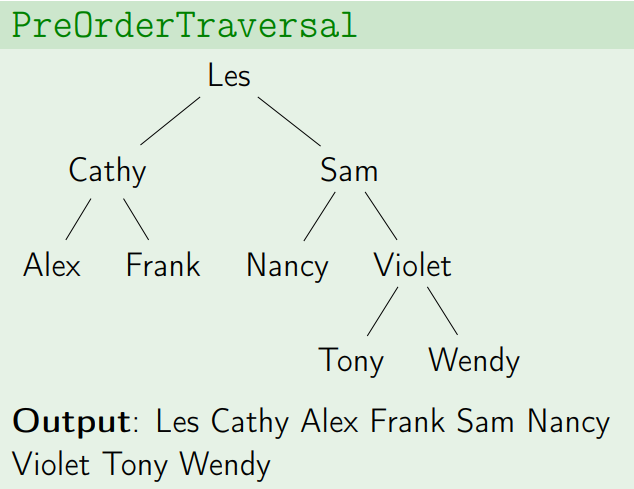
}

}

}

}

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree5.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree6.PNG)

public void PreOrderRecursiveTraversal(Tree<T> tree)

{

if (tree == null)

{

return;

}

Console.WriteLine(tree.Key);

PreOrderRecursiveTraversal(tree.Left);

PreOrderRecursiveTraversal(tree.Right);

}

public void PreOrderIterativeTraversal(Tree<T> tree)

{

if (tree == null)

{

return;

}

var stack = new Stack<Tree<T>>();

stack.Push(tree);

while (stack.Count != 0)

{

Tree<T> actualTree = stack.Pop();

Console.WriteLine(actualTree.Key);

if (actualTree.Right != null)

{

stack.Push(actualTree.Right);

}

if (actualTree.Left != null)

{

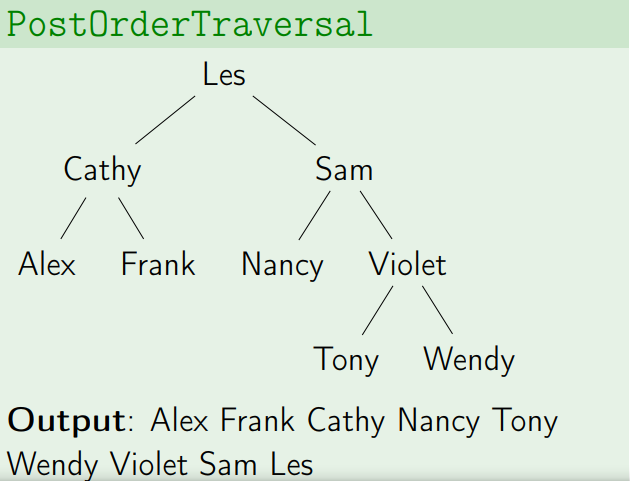
stack.Push(actualTree.Left);

}

}

}

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree7.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree8.PNG)

public void PostOrderRecursiveTraversal(Tree<T> tree)

{

if (tree == null)

{

return;

}

PostOrderRecursiveTraversal(tree.Left);

PostOrderRecursiveTraversal(tree.Right);

Console.WriteLine(tree.Key);

}

public void PostOrderIterativeTraversal(Tree<T> tree)

{

if (tree == null)

{

return;

}

var reverseStack = new Stack<Tree<T>>();

reverseStack.Push(tree);

var postOrderStack = new Stack<T>();

while (reverseStack.Count != 0)

{

Tree<T> actualTree = reverseStack.Pop();

postOrderStack.Push(actualTree.Key);

if (actualTree.Left != null)

{

reverseStack.Push(actualTree.Left);

}

if (actualTree.Right != null)

{

reverseStack.Push(actualTree.Right);

}

}

while (postOrderStack.Count != 0)

{

Console.WriteLine(postOrderStack.Pop());

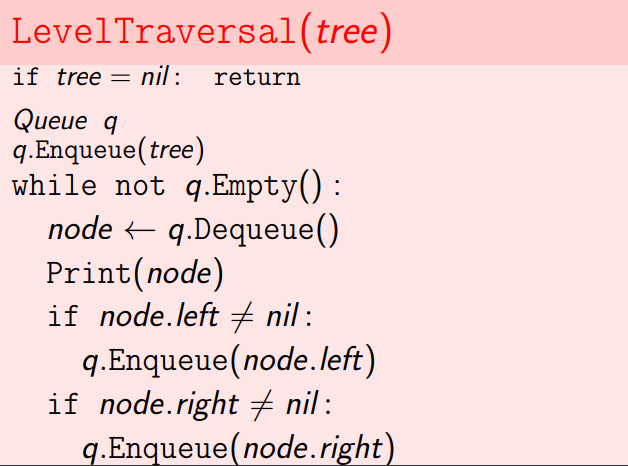
}

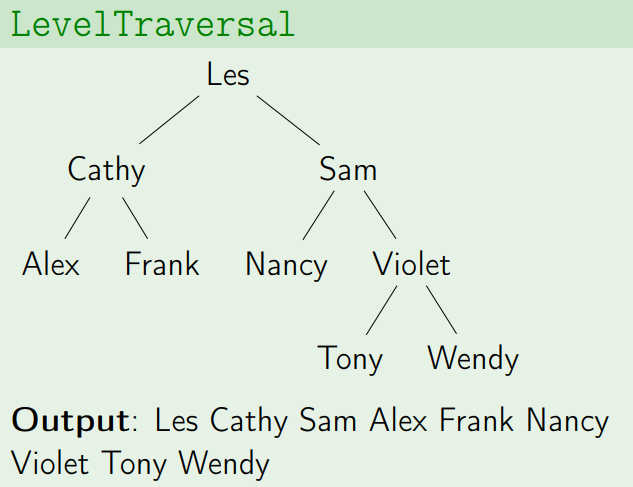
}

There is also a PostOrder iterative single stack implementation:

<https://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/>

### Breath-first

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree9.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/Tree/Images/tree10.PNG)

public void BreathFirst(Tree<T> tree)

{

if (tree == null)

{

return;

}

Queue<Tree<T>> queue = new Queue<Tree<T>>();

queue.Enqueue(tree);

while (queue.Count != 0)

{

Tree<T> actualTree = queue.Dequeue();

Console.WriteLine(actualTree.Key);

if (actualTree.Left != null)

{

queue.Enqueue(actualTree.Left);

}

if (actualTree.Right != null)

{

queue.Enqueue(actualTree.Right);

}

}

}

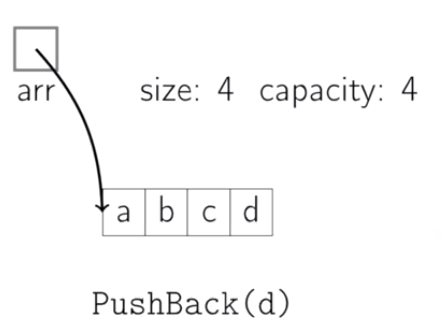
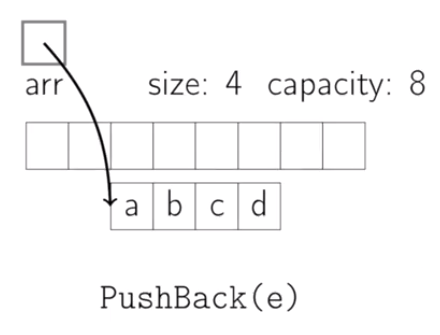
# Dynamic Arrays and Amortized Analysis

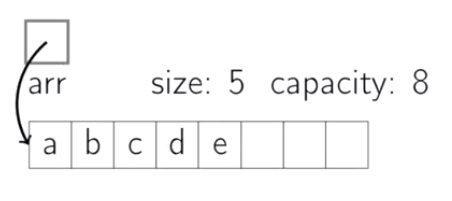
* [Dynamic Arrays](https://github.com/KiraDiShira/Cracking/blob/master/DynamicArraysandAmortizedAnalysis/Readme.md#dynamic-arrays)
  + [Implementation](https://github.com/KiraDiShira/Cracking/blob/master/DynamicArraysandAmortizedAnalysis/Readme.md#implementation)
  + [Runtimes](https://github.com/KiraDiShira/Cracking/blob/master/DynamicArraysandAmortizedAnalysis/Readme.md#runtimes)
* Amortized Analysis
  + [Aggregate Method](https://github.com/KiraDiShira/Cracking/blob/master/DynamicArraysandAmortizedAnalysis/Readme.md#aggregate-method)
  + [Online resource](http://www.cs.cornell.edu/courses/cs3110/2013sp/lectures/lec21-amortized/lec21.html)
* [Summary](https://github.com/KiraDiShira/Cracking/blob/master/DynamicArraysandAmortizedAnalysis/Readme.md#summary)

## Dynamic Arrays

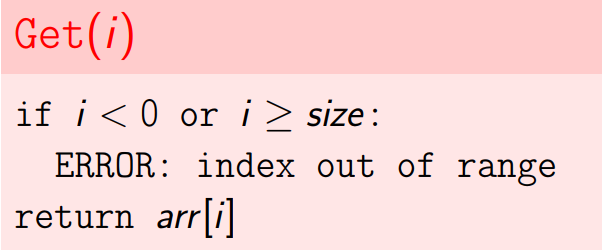
Unlike static arrays, dynamic arrays can be resized. **Solution**: dynamic arrays (also known as resizable arrays are an abstract data type with the following operations (at a minimum):

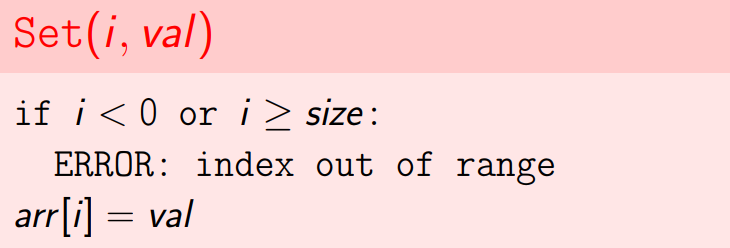
* Get(i): returns element at location i. Must must be constant time
* Set(i, val): Sets element i to val. Must be constant time
* PushBack(val): Adds val to the end
* Remove(i): Removes element at location i
* Size(): the number of elements

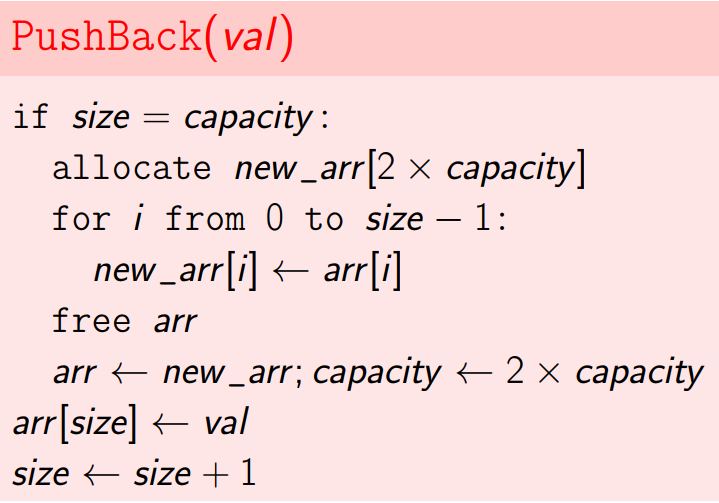
[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/pb1.PNG)[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/pb2.PNG)

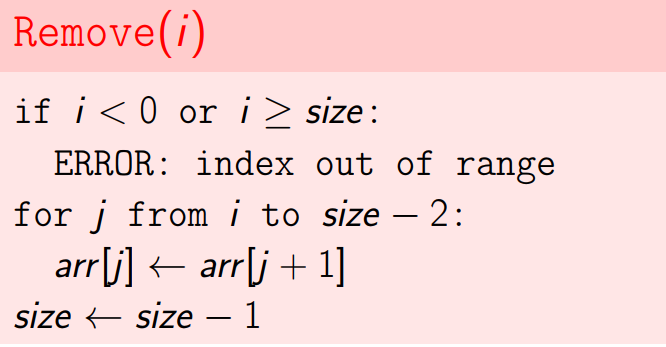
[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/pb3.PNG)

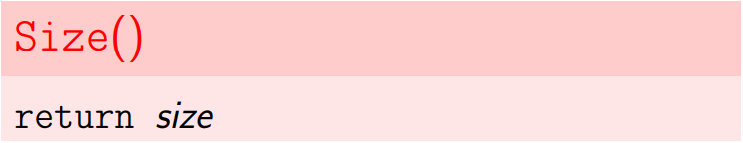
### Implementation

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa1.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa2.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa3.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa4.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa5.PNG)

public class DynamicArray<T>

{

private T[] \_array;

private int \_capacity;

public DynamicArray(int size)

{

Size = 0;

\_array = new T[size];

\_capacity = size;

}

public int Size { get; set; }

public T Get(int index)

{

if (index < 0 || index >= Size)

{

throw new Exception("index out of range");

}

return \_array[index];

}

public void Set(int index, T value)

{

if (index < 0 || index >= Size)

{

throw new Exception("index out of range");

}

\_array[index] = value;

}

public void PushBack(T value)

{

if (Size == \_capacity)

{

T[] newArray = new T[\_capacity \* 2];

\_array.CopyTo(newArray, 0);

\_array = newArray;

\_capacity = 2 \* \_capacity;

}

\_array[Size] = value;

Size++;

}

public void Remove(int index)

{

if (index < 0 || index >= Size)

{

throw new Exception("index out of range");

}

for (int j = index; j < Size - 1; j++)

{

\_array[j] = \_array[j + 1];

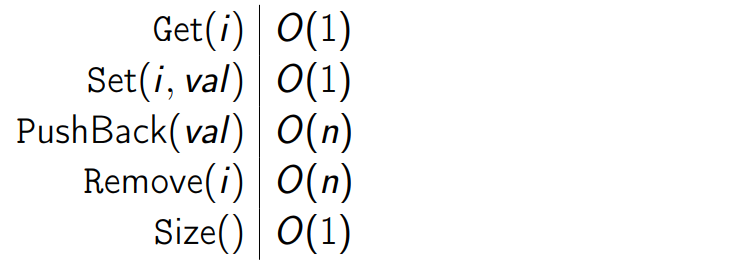
}

Size--;

}

}

### Runtimes

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa6.PNG)

## Aggregate Method

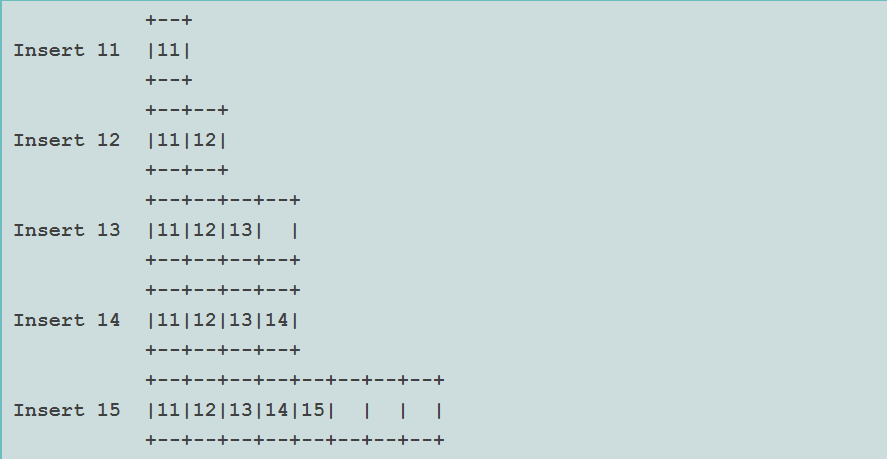
Sometimes, we're looking at an individual worst case and that may be too severe. In particular we may want to know the total worst case for a sequence of operations and it may be some of those operations are cheap, while only certain of them are expensive. So if we look at the worst case operation for any one and multiply that by the total, it may be overstating the total cost.

As an example, for a dynamic array, we only resize every so often. If we do our running time analysis for a sequence of n insert operations we would have n \* O(n) = O(n^2). Most of the time, we're doing a constant time operation, just adding an element. It's only when we fully reach the capacity, that we have to resize. So the question is, what's the total cost if you have to insert a bunch of items?

So here's the definition of **amortized cost**. You have a sequence of n operations, the amortized cost is the cost of those n operations divided by n.

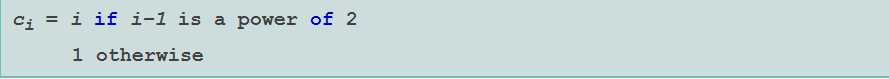
Amortized Cost = Cost(n operations) / n

For instance, consider the following sequence of insertions, starting with an array of length 1:

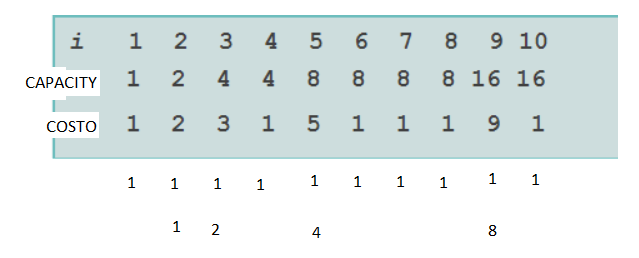
[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa7.PNG)

The table is doubled in the second, third, and fifth steps. As each insertion takes O(n) time in the worst case, a simple analysis would yield a bound of O(n^2) time for n insertions. But it is not this bad.

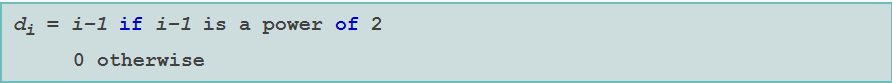
Let c\_i be the cost of the i-th insertion:

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa8.PNG)

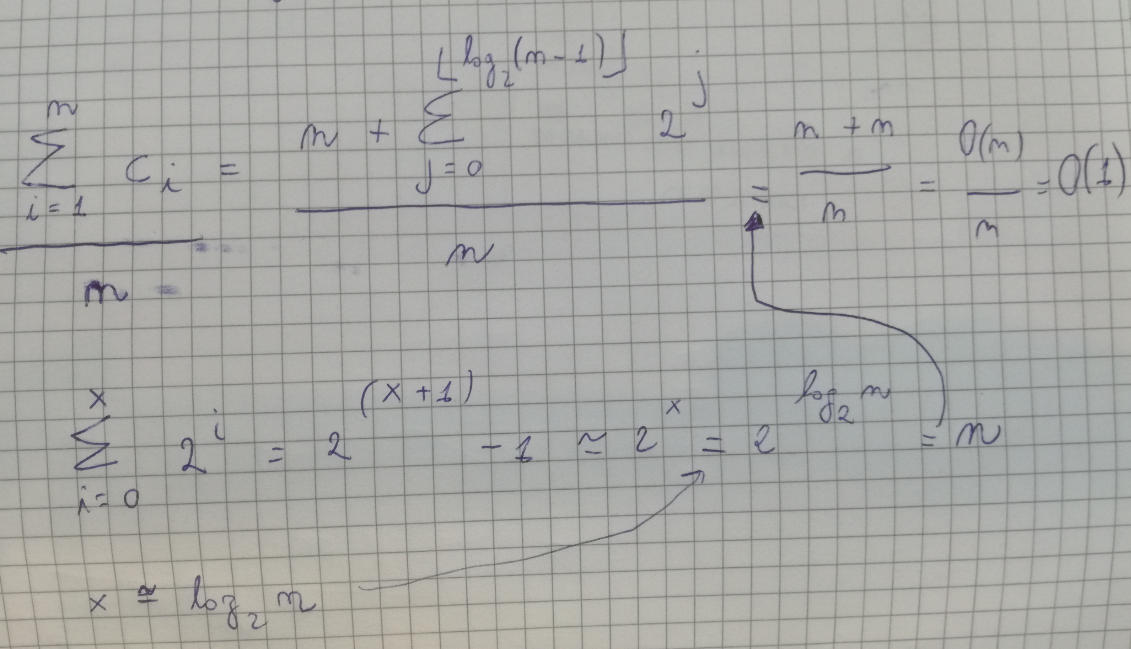
Let's consider the capacity of the table and the cost ci for the first few insertions in a sequence:

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/pb4.PNG)

Alteratively we can see that c\_i=1+d\_i where d\_i is the cost of doubling the table size.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa10.PNG)

Then summing over the entire sequence, all the 1's sum to O(n), and all the di also sum to O(n). That is,

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/pb5.PNG)

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa11.PNG)

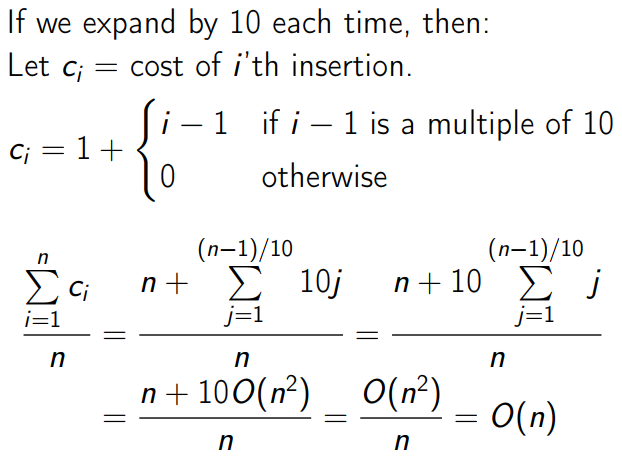
So we've got n plus something no more than 2n, that's clearly O(n) divided by n, and that's just O(1). So what we've determined then is that we have a amortized cost for each insertion of order 1.

Our worst case cost is still order n, so if we want to know how long it's going to take in the worst case for any particular insertion is O(n), but the amortized cost is O(1).

## Summary

Can we use a constant amount (+10 for instance), intead of a constant factor (\* 2) for growing our dynamic array?

So this shows that if we use a constant amount to grow the dynamic array each time that we end up with an amortized cost for push back of O(n) rather than O(1). So it's extremely important to use a constant factor.

[](https://raw.githubusercontent.com/KiraDiShira/Cracking/master/DynamicArraysandAmortizedAnalysis/Images/daaa22.PNG)

So this shows that if we use a constant amount to grow the dynamic array each time that we end up with an amortized cost for push back of O(n) rather than O(1). So it's extremely important to use a constant factor.