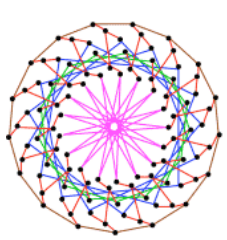
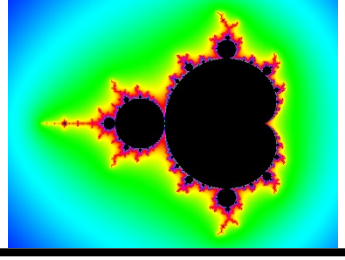
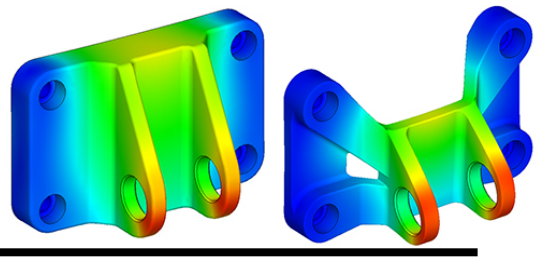




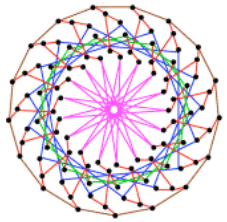
Algoritmos y Estructuras de Datos

Pedro Corcuera

Dpto. Matemática Aplicada y Ciencias de la Computación

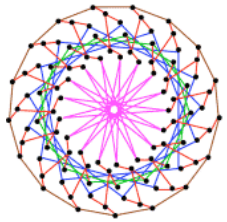
Universidad de Cantabria

corcuerp@unican.es



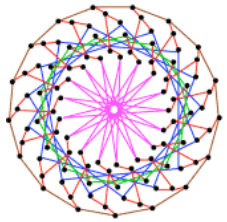
Indice

- Algoritmos
- Estructuras de datos

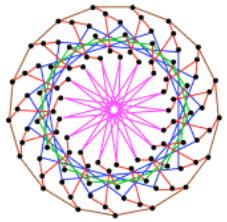


Objetivos

- Introducir los conceptos de algoritmos y estructuras de datos.

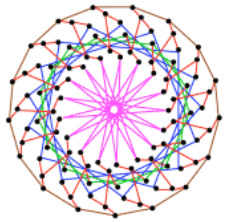


Algoritmos



Definición

- **Algoritmo** es una secuencia ordenada de instrucciones que resuelve un problema concreto
- Ejemplos:
  - Algoritmo de la media aritmética de N valores.
  - Algoritmo para la resolución de una ecuación de segundo grado.
- Niveles de detalle de los algoritmos:
  - Alto nivel: no se dan detalles.
  - Bajo nivel: muchos detalles.



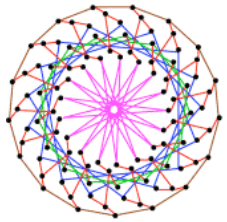
Propiedades

Necesarias o básicas:

- Corrección (sin errores).
- Validez (resuelve el problema pedido)
- Precisión (no puede haber ambigüedad).
- Repetitividad (en las mismas condiciones, al ejecutarlo, siempre se obtiene el mismo resultado).
- Finitud (termina en algún momento). Número finito de órdenes no implica finitud.
- Eficiencia (lo hace en un tiempo aceptable)

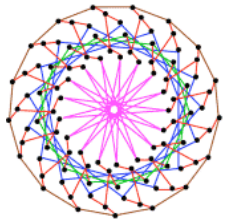
Deseables:

- Generalidad
- Fácil de usar
- Robustez



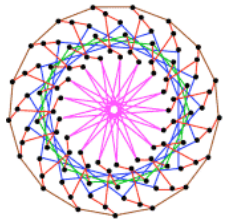
Algoritmos: Métodos de Representación

- Verbal
- Diagramas de flujo
- Diagramas de Bloques, Cajas o de Nassi-Shneiderman
- Gráficos
- Pseudocódigo
- Representaciones algebraicas (fórmulas y expresiones)



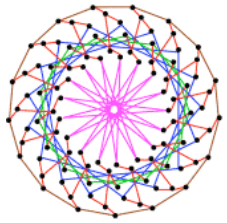
Modificación de algoritmos

- ***Generalización y extensibilidad***: proceso de aplicar el algoritmo a más casos y de incluir más casos dentro del algoritmo
- ***Robustez***: proceso de hacer un algoritmo mas fiable o robusto (se recupera de errores), anticipando errores de entrada u otras dificultades.



Algoritmos alternativos y equivalentes

- Pueden haber muchas formas de llevar a cabo un algoritmo.
- En esos casos la elección se basa en la eficiencia (memoria y velocidad).
- El *análisis de algoritmos* estudia la cantidad de recursos que demanda la ejecución de un algoritmo.
- Preocupa más el tiempo de ejecución de un algoritmo: *Complejidad del algoritmo*

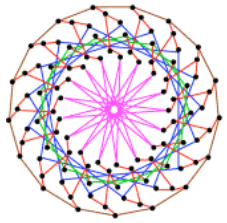


Programación estructurada

- Método para construir algoritmos a partir de un número pequeño de bloques básicos.
- Formas fundamentales:
  - ***Secuencia***: indica secuencia temporal lineal de las acciones a realizarse.

    A
    B
  - ***Selección***: especifica una condición que determina la acción a realizarse.

```
if C
        D
else
        E
```



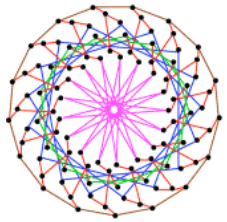
Programación estructurada

- ***Repetición***: indica que una o más acciones deben repetirse un determinado número de veces.

while G do
H

- ***Invocación***: corresponde al grupo de acciones agrupadas bajo un nombre.

Calcula_promedio



Pseudocódigo

- Lectura o entrada de datos

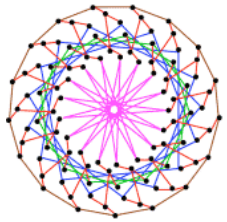
  Input

- Repetición

  while expr
  instrucción
  endwhile

  for i = 1 to m
  instrucción
  endfor

  do
  instrucción
  while expr



Pseudocódigo

- Decisión

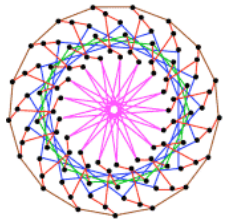
  if expr

  instrucción

  endif

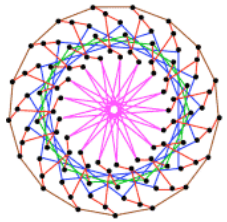
- Escritura o salida de datos

  Output



Análisis de Algoritmos: Complejidad

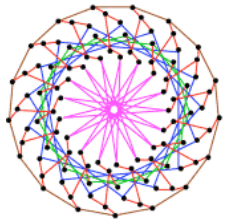
- Para comparar algoritmos se pueden estudiar desde dos puntos de vista:
  - el tiempo que consume un algoritmo para resolver un problema (complejidad temporal) ⟵ más interés
  - la memoria que necesita el algoritmo (complejidad espacial).
- Para analizar la complejidad se cuentan los pasos del algoritmo en función del tamaño de los datos y se expresa en unidades de tiempo utilizando la notación asíntotica “O- Grande” (complejidad en el peor caso).



Análisis de Algoritmos: Complejidad

Problema: Buscar el mayor valor en una lista de números desordenados (array)

```
Algoritmo: (n = número de elementos)
1   max = s1
2   i = 2
3   while i <= n
4     if si > max then
5               max = si
6      i = i + 1
7   endwhile
```



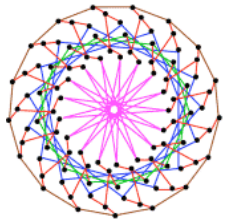
Análisis de Algoritmos: Complejidad

Número de operaciones realizadas (unid):

Línea	Operaciones	Tiempo
1	indexado y asignación	2
2	asignación	1
3	comparación	1
4,5,6	2 indexado, comparación, 2 asignación, suma	6

Tiempo total:

$t(n) = 2 + 1 + (n - 1) + 6 \cdot (n - 1) = 3 + 7 \cdot (n - 1) = \mathbf{7n - 4}$



Notación asintótica

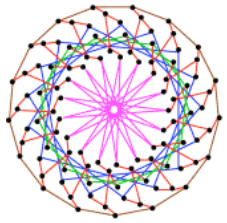
- Es útil concentrarse en la tasa de crecimiento del tiempo de ejecución t como función del tamaño de la entrada *n*.
- Se usa la notación **Big O** (**O grande** cota superior al ritmo de crecimiento de un algoritmo):

  Sean f(*n*) y g(*n*) funciones no negativas, f(*n*) es O(g(*n*)) si hay un valor c > 0 y

  $n_0 \geq 1$ tal que $f(n) \leq cg(n)$ para $n \geq n_0$
- Se dice que f(*n*) **es de orden** g(*n*)

  Ej: $7n - 4$ es O(*n*) si c=7 y $n_0 = 1$
- Generalmente para cualquier polinomio

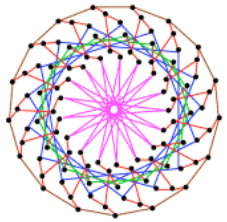
  $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$ es $O(n^k)$



Eficiencia de un algoritmo

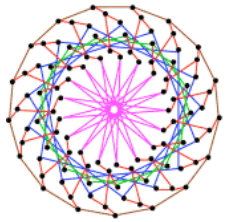
- Definición de eficiencia: Un algoritmo es eficiente si tiene un tiempo de ejecución polinómico.
- Tiempos de ejecución (redondeados) de diferentes complejidades de algoritmos en entradas de tamaños crecientes, para un procesador que realiza un *millón de instrucciones de alto nivel por segundo*. Si el tiempo de ejecución > 10^{25} años, se considera que el algoritmo toma un tiempo "very long".

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long



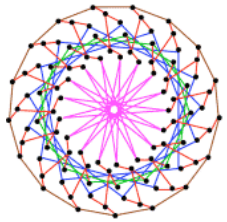
Desarrollo de algoritmos

- Análisis del problema
  - Dominio del problema
  - Modelo
- Diseño del algoritmo
  - Refinamiento sucesivo
  - Top down o botton up
- Análisis del algoritmo
  - Cuánto tarda en dar una solución? Se puede modificar para aumentar la eficiencia?
  - Análisis de la Complejidad
- Verificación del algoritmo
  - Comprobar que es correcto



Algoritmos iterativos

- Muchos algoritmos se basan en ***ciclos o bucles***, es decir en la ejecución de una serie de pasos repetitivos.
- **Iteración** significa hacer algo de forma repetida.

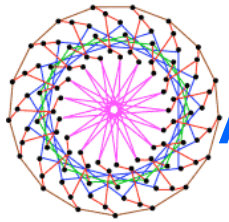


Algoritmos recursivos

- Son algoritmos que expresan la solución de un problema en términos de una llamada a sí mismo (llamada recursiva o recurrente)
- Ejemplo típico: Factorial (n!) de un número

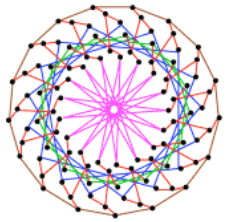
$$n! = \begin{cases} 1 & si\ n = 0 \\ n \cdot (n-1)! & si\ n > 0 \end{cases}$$

- Son más ineficientes que los iterativos pero más simples y elegantes
- Todo algoritmo recursivo tiene su equivalente iterativo



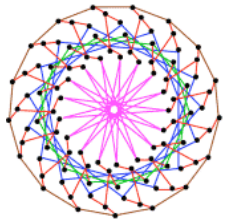
Algoritmos recursivos – definición y diseño

- Un método recursivo es un método que se llama a sí mismo dentro del cuerpo del método.
- Para diseñar correctamente un algoritmo recursivo, es necesario:
  - Establecer correctamente la ley de recurrencia.
  - Definir el procedimiento de finalización del algoritmo recursivo (normalmente con el valor o valores iniciales).



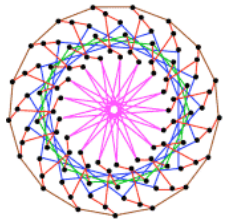
Algoritmos recursivos – Verificación

- Para verificar funciones recursivas se aplica el método de las tres preguntas:
  - *pregunta Caso-Base*: Hay una salida no recursiva de la función, y la rutina funciona correctamente para este caso "base"?
  - *pregunta Llamador-Más Pequeño*: Cada llamada recursiva a la función se refiere a un caso más pequeño del problema original?
  - *pregunta Caso-General*: Suponiendo que las llamadas recursivas funcionan correctamente, funciona correctamente toda la función?



Computabilidad

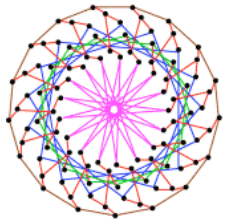
- **Algoritmo**: Procedimiento sistemático que permite resolver un problema en un número finito de pasos, cada uno de ellos especificado de manera efectiva y sin ambigüedad.
- **Función computable**: Aquella que puede ser calculada mediante un dispositivo, dado un tiempo y espacio de almacenamiento ilimitado (pero finito)
- No importa la eficiencia, sino la posibilidad de ser calculada.



Computabilidad

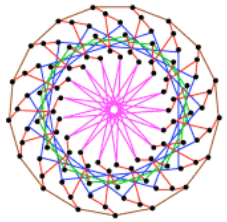
- Hay problemas que *no son computables o intratables*: ningún computador pueden resolverlos sin entrar en ciclos infinitos (nunca paran) para ciertas entradas.
- Un ejemplo de problema intratable es el problema del vendedor viajero:

  un vendedor tiene que visitar un cierto número de ciudades para lo que tiene un presupuesto fijo y conoce lo que le cuestan los viajes entre todas las ciudades. La cuestión es si puede encontrar un recorrido, que no exceda el presupuesto que tiene, que parta y termine en la ciudad en que vive y que visite una sola vez cada una de las ciudades requeridas.
- Existen técnicas que permiten obtener soluciones parciales o aproximadas para problemas no computables.



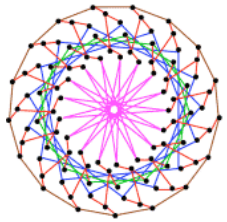
Técnicas de diseño de algoritmos

- Algoritmos voraces (greedy): seleccionan los elementos más prometedores del conjunto de candidatos hasta encontrar una solución. En la mayoría de los casos la solución no es óptima.
- Divide y vencerás: dividen el problema en subconjuntos disjuntos obteniendo una solución de cada uno de ellos para después unirlas, logrando así la solución al problema completo.
- Programación dinámica: intenta resolver problemas disminuyendo su coste computacional aumentando el coste espacial.
- Vuelta Atrás (Backtracking): se construye el espacio de soluciones del problema en un árbol que se examina completamente, almacenando las soluciones menos costosas.



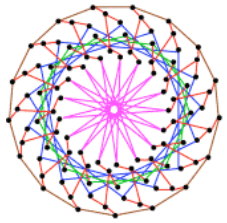
Técnicas de diseño de algoritmos

- Ramificación y acotación: se basa en la construcción de las soluciones al problema mediante un árbol implícito que se recorre de forma controlada encontrando las mejores soluciones.
- Metaheurísticas: encuentran soluciones aproximadas (no óptimas) a problemas basándose en un conocimiento anterior (a veces llamado experiencia) de los mismos.
- Algoritmos determinísticos: El comportamiento del algoritmo es lineal: cada paso del algoritmo tiene únicamente un paso sucesor y otro ancesor.
- Algoritmos probabilísticos: algunos de los pasos de este tipo de algoritmos están en función de valores pseudoaleatorios

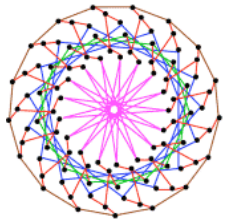


Técnicas de diseño de algoritmos

- Algoritmos no determinísticos: El comportamiento del algoritmo tiene forma de árbol y a cada paso del algoritmo puede bifurcarse a cualquier número de pasos inmediatamente posteriores, además todas las ramas se ejecutan simultáneamente.
- Algoritmos paralelos: permiten la división de un problema en subproblemas de forma que se puedan ejecutar de forma simultánea en varios procesadores.

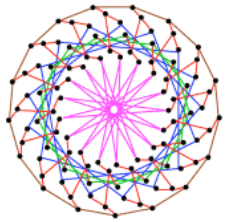


Estructuras de Datos
Data Structures



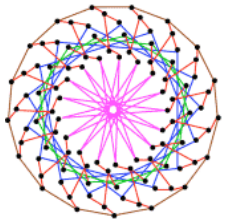
Abstract Data Types (ADTs)

- A set of objects and a set of operations to manipulate them
- Separate the notions of specification and implementation:
  - Specification: “what does an operation do?”
  - Implementation: “how is it done?”
- Benefits:
  - Simplicity: code is easier to understand
  - Encapsulation: details are hidden
  - Modularity: an ADT can be changed without modifying the programs that use it
  - Reuse: it can be used by other programs

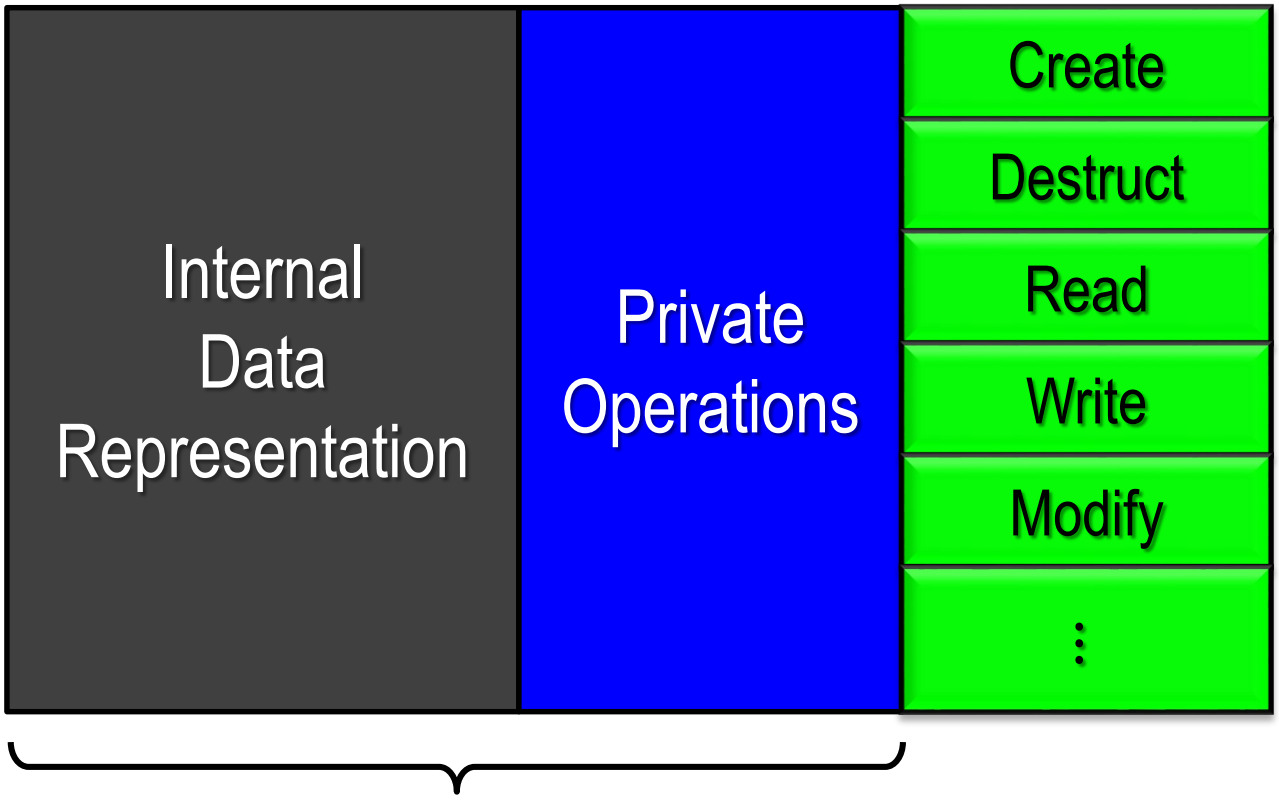


Abstract Data Types (ADTs)

- An ADT has two parts:
  - Public or external: abstract view of the data and operations (methods) that the user can use.
  - Private or internal: the actual implementation of the data structures and operations.
- Operations:
  - Creation/Destruction
  - Access
  - Modification

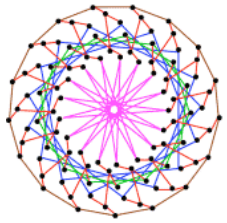


Abstract Data Types (ADTs)



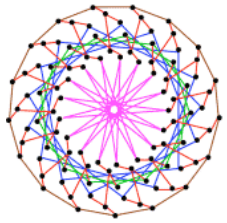
User Interface (API*)

*API: Application Programming Interface



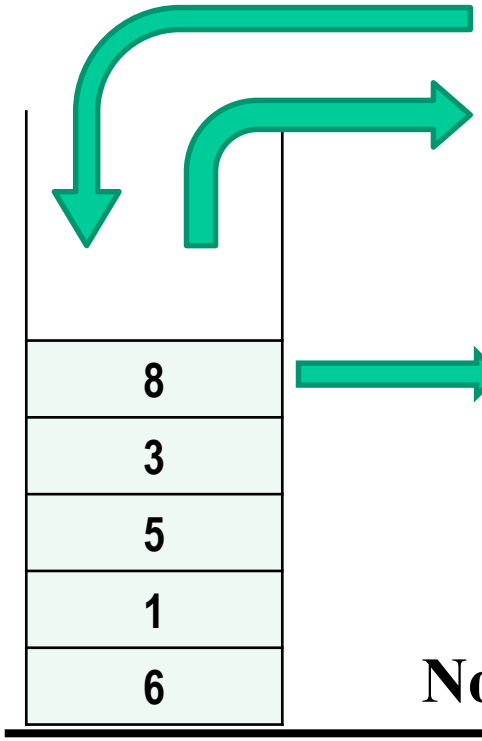
ADTs and Object-Oriented Programming

- OOP is a programming paradigm: a program is a set of objects that interact with each other.
- An object has:
  - fields (or attributes) that contain data
  - functions (or methods) that contain code
- Objects (variables) are instances of classes (types). A class is a template for all objects of a certain type.
- In OOP, a class is the natural way of implementing an ADT.



The Stack ADT

- A stack is a list of objects in which insertions and deletions can only be performed at the top of the list.
- Also known as LIFO (Last In, First Out)



push (insert an element at the top)

pop (delete the most recently inserted element)

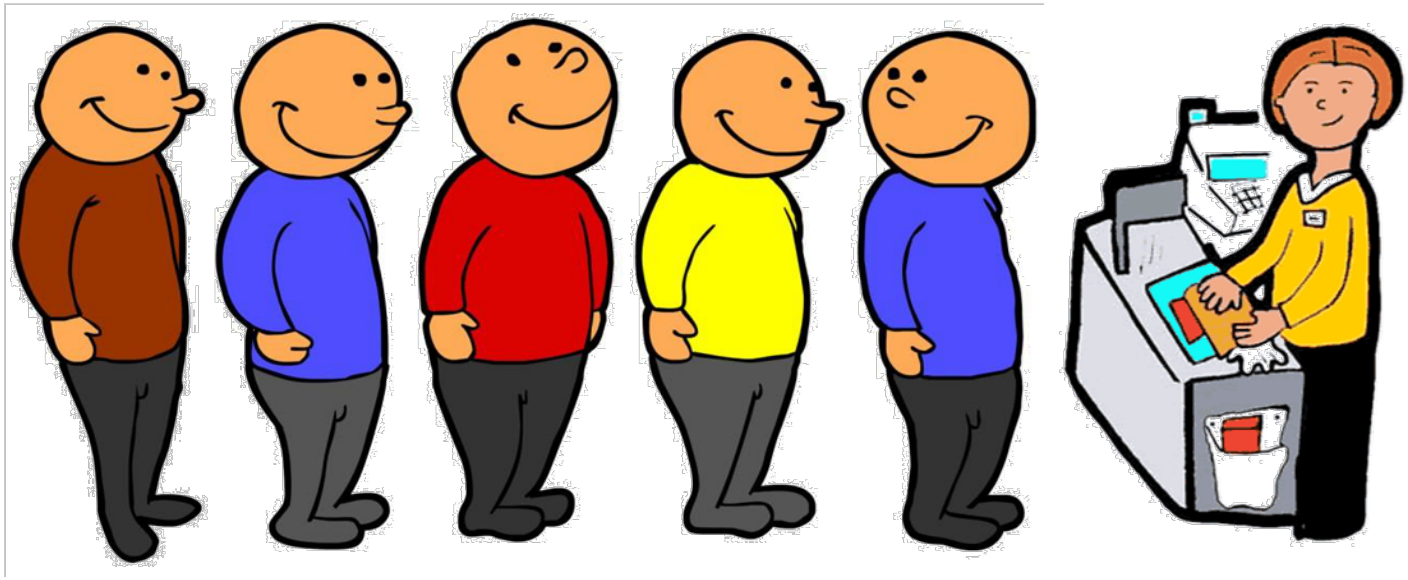
top (the most recently inserted element)

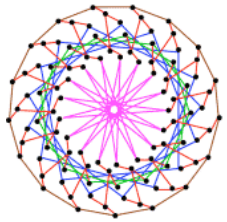
empty (is there any element?)

Note: pop and top generate an error on an empty stack

Queue

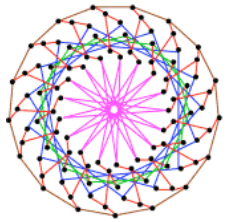
- A container in which insertion is done at one end (the tail) and deletion is done at the other end (the head).
- Also called FIFO (First-In, First-Out)





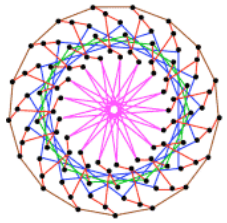
List

- List: a container with sequential access.
- It allows to insert/erase elements in the middle of the list in constant time.
- A list can be considered as a sequence of elements with one or several cursors (iterators) pointing at internal elements.



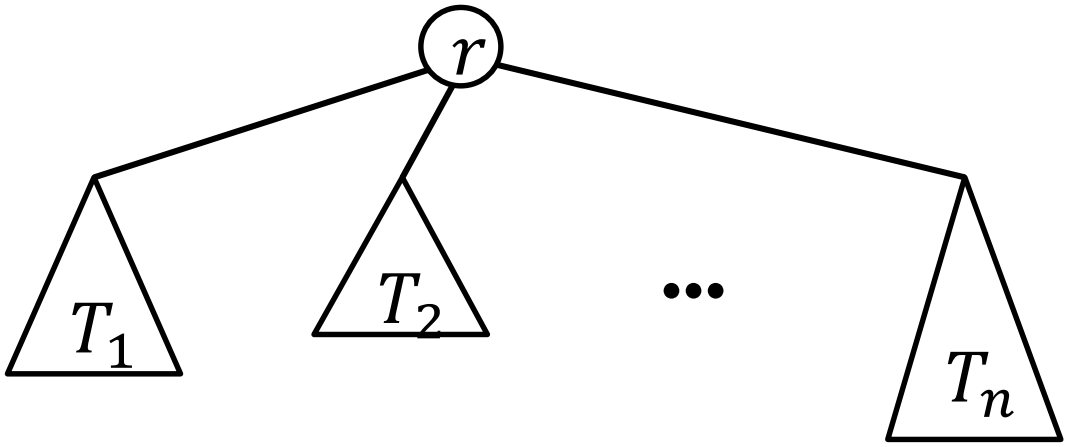
A priority queue

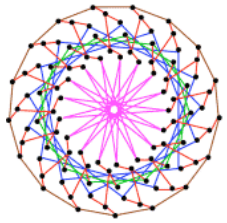
- A priority queue is a queue in which each element has a priority.
- Elements with higher priority are served before elements with lower priority.
- It can be implemented as a vector or a linked list. For a queue with n elements:
  - Insertion is $\mathrm{O}(n)$.
  - Extraction is $\mathrm{O}(1)$.
- A more efficient implementation can be proposed in which insertion and extraction are $\mathrm{O}(\log n)$: ***binary heap***.



Trees

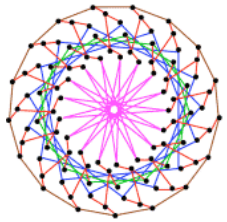
- Graph theory: a tree is an undirected graph in which any two vertices are connected by exactly one path.
- Recursive definition (CS). A non-empty tree T consists of:
  - a root node r
  - a list of trees $T_1, T_2, \ldots, T_n$ that hierarchically depend on r.





Sets and Dictionaries

- A set: a collection of items. The typical operations are:
  - Add/remove one element
  - Does it contain an element?
  - Size?, Is it empty?
  - Visit all items
- A dictionary (map): a collection of key-value pairs. The typical operations are:
  - Put a new key-value pair
  - Remove a key-value pair with a specific key
  - Get the value associated to a key
  - Does it contain a key?
  - Visit all key-value pairs



Graph

- A graph consists of a finite (and possibly mutable) set of **vertices** (also called nodes or points), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as **edges** (also called links or lines), and for a directed graph are also known as arrows.
- A graph may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).
- Link to a detailed description of graph DS and algorithms.