Università degli Studi di Milano
Corso di Laurea in Informatica, A.A. 2017-2018

Allocazione dinamica della memoria


Homepage del corso

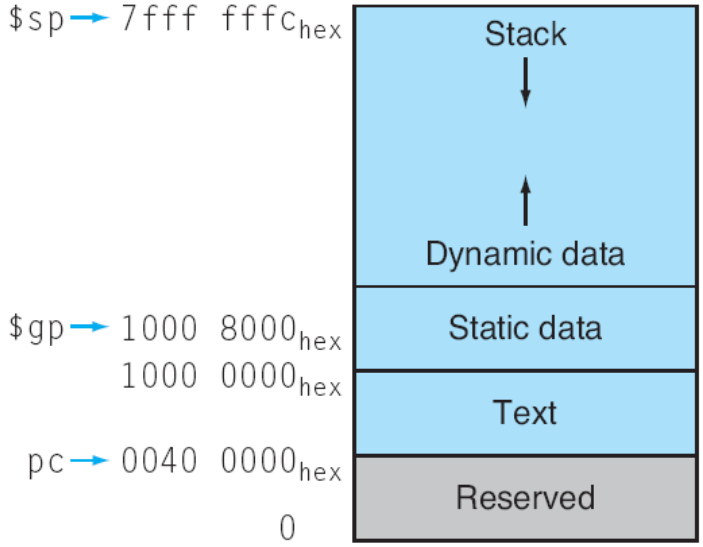
Turno A
Nicola Basilico
Dipartimento di Informatica
Via Comelico 39/41 - 20135 Milano (MI)
Ufficio S242
nicola.basilico@unimi.it
+39 02.503.16294

Turno B
Jacopo Essenziale
Dipartimento di Informatica
Via Celoria 20 - 20133 Milano (MI)
AISLab
jacopo.essenziale@unimi.it
+39 02.503.14010

Utilizzo della memoria

In MIPS la memoria viene divisa in:

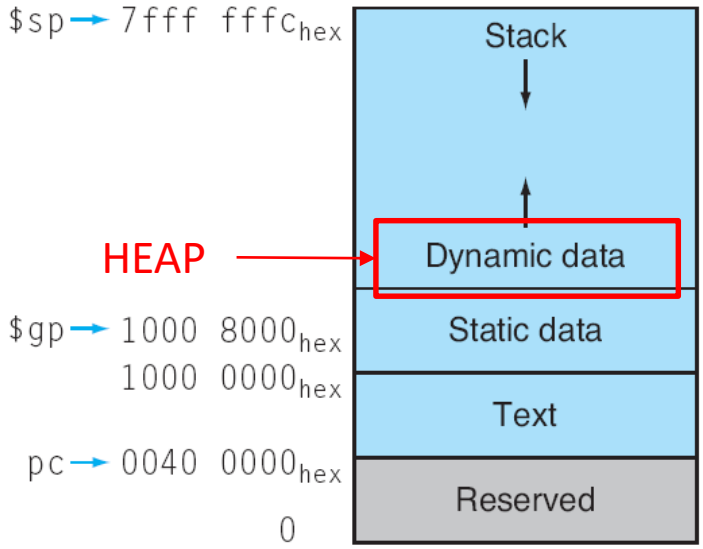
- **Segmento testo**: contiene le **istruzioni** del programma.
- **Segmento dati**:
  - ***dati statici*:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma *(e.g., variabili statiche, costanti, etc.);*
  - ***dati dinamici*:** contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso *(e.g., liste dinamiche, etc.).*
- **Stack**: contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) *(e.g., parametri di una procedura, valori di ritorno, etc.)*.



Utilizzo della memoria

In MIPS la memoria viene divisa in:

- **Segmento testo**: contiene le **istruzioni** del programma.
- **Segmento dati**:
  - ***dati statici*:** contiene dati la cui dimensione è conosciuta a *compile time* e la cui durata coincide con quella del programma *(e.g., variabili statiche, costanti, etc.);*
  - ***dati dinamici*:** contiene dati per i quali lo spazio è allocato dinamicamente a *runtime* su richiesta del programma stesso *(e.g., liste dinamiche, etc.).*
- **Stack**: contiene dati dinamici organizzati secondo una coda LIFO (Last In, First Out) *(e.g., parametri di una procedura, valori di ritorno, etc.)*.



Allocazione statica vs. allocazione dinamica

STATICA

- Allocata a **compile time**
- Non può cambiare dimensione
- Usata per costanti, variabili, e array

DINAMICA

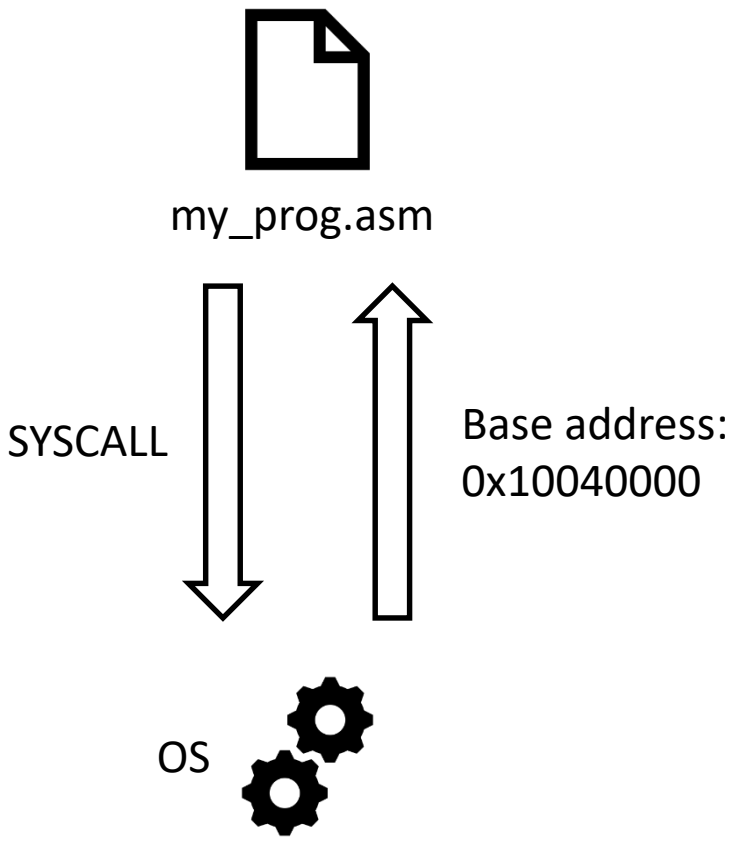
- Allocata a **run time**
- L'allocazione è demandata al programmatore
- Usata per stringhe di lunghezza non determinata, strutture dati dinamiche

Come avviene l'allocazione?

1. Il programma richiede al sistema operativo di allocare un certo numero di byte in memoria tramite una syscall

2. Il sistema operativo verifica l'effettiva disponibilità dello spazio in memoria e, se possibile, alloca lo spazio richiesto e restituisce al programma il base address dell'area di memoria allocata

3. In caso di memoria non allocabile, di norma, il sistema operativo restituirà un codice di errore (tipicamente un numero negativo) al posto del base address permettendoci di gestire il problema



In MARS: la richiesta di allocare più spazio di quanto disponibile, genera un'eccezione a run time

SBRK (segment break)

La syscall utilizzata per allocare spazio dinamicamente sullo heap è la **SBRK** (codice **9**).

- Input: $a0 <- numero di byte da allocare
- Output: $v0 <- base address dell'area di memoria allocata

```
1   .text
2   .globl main
3
4   main:
5
6   li $v0, 9        # SBRK syscall
7   li $a0, 100      # Alloca 100 bytes in memoria
8   syscall
9   # $v0 ora contiene il base address dell'area
10  # di memoria allocata
11  li $t0, 5
12  sw $t0, 0($v0)
13  ...
```

SBRK vs. malloc e free

- La syscall SBRK si occupa solo di «spostare» il puntatore alla fine dello heap più avanti aumentando lo spazio di memoria dinamica disponibile al nostro programma

- non ci permette di gestire come allochiamo la memoria all'interno dello heap, ma solo di richiedere al sistema operativo di darci più memoria

- Le funzioni «malloc» e «free» appartenenti alla libreria stdlib distribuita con libc (installata praticamente su tutti i sistemi operativi moderni) servono per gestire in maniera più fine lo heap e sono implementate «sopra» a SBRK, permettendo in maniera automatica di allocare e deallocare spazio per le nostre strutture dati dinamiche, gestendo automaticamente problemi di frammentazione della memoria e minimizzando le chiamate a SBRK utilizzando opportuni buffer di memoria

- In MARS e in SPIM non abbiamo accesso alle funzioni malloc e free, che sono disponibili solo su sistemi «reali», per cui o ne implementiamo una nostra versione oppure ne facciamo a meno

Strutture Dati

Una struttura dati rappresenta un modo particolare di **organizzare i dati** in memoria **in modo da massimizzare** l'**efficienza** di esecuzione di **determinate operazioni** su di essi (accesso, ricerca di elementi, ecc.)

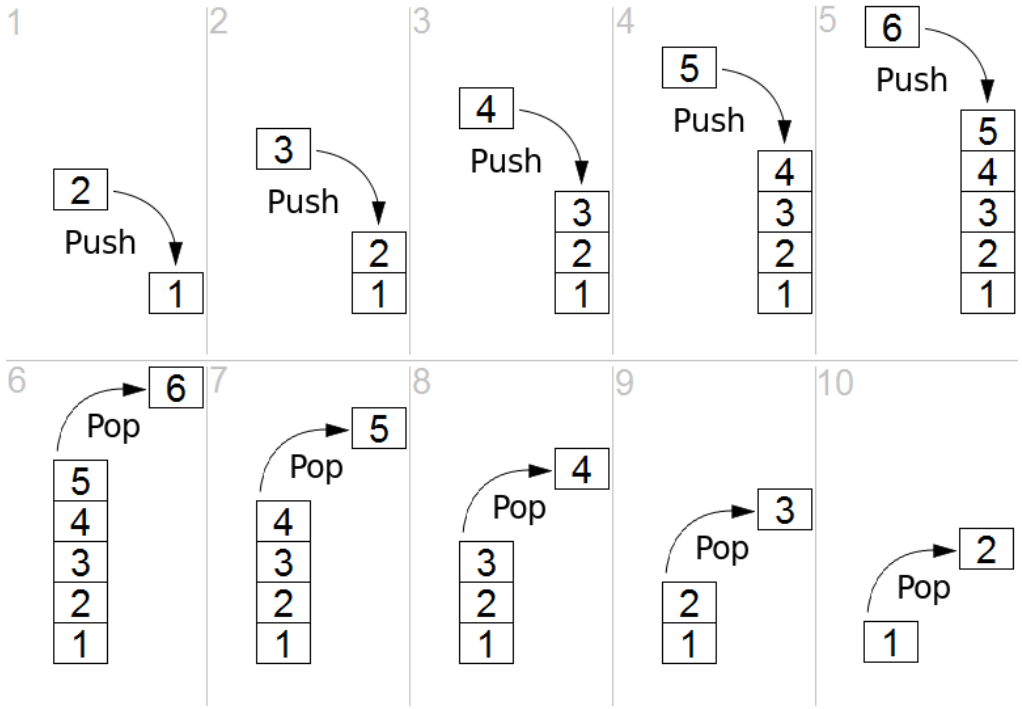
Una struttura dati contiene le seguenti informazioni:

- I **valori** dei dati in essa contenuti (interi, stringhe, strutture …)
- Le eventuali **relazioni** tra i dati in essa contenuti (ordine …)
- Le **funzioni** e le **operazioni** che possono essere applicate sui vari dati in essa contenuti (cancellazione, inserimento, ricerca …)

«stack» o «pila»

- Lo stack è un tipo particolare di struttura dati, appartenente alla categoria **LIFO** (Last In First Out)

- Il suo scopo è quello di massimizzare l'efficienza nelle operazioni di salvataggio di un nuovo elemento e di accesso all'ultimo elemento in esso inserito



«stack» o «pila»
implementazione

Obiettivo: implementare uno stack nel segmento dati dinamico:

1. Identificare i dati con cui lavoreremo e come li rappresenteremo in memoria
2. Stabilire come collegare fra di loro questi dati
3. Implementare le funzioni di PUSH e POP dando così un’interfaccia all’utente per utilizzare la struttura dati

«stack» o «pila»
rappresentazione dei dati

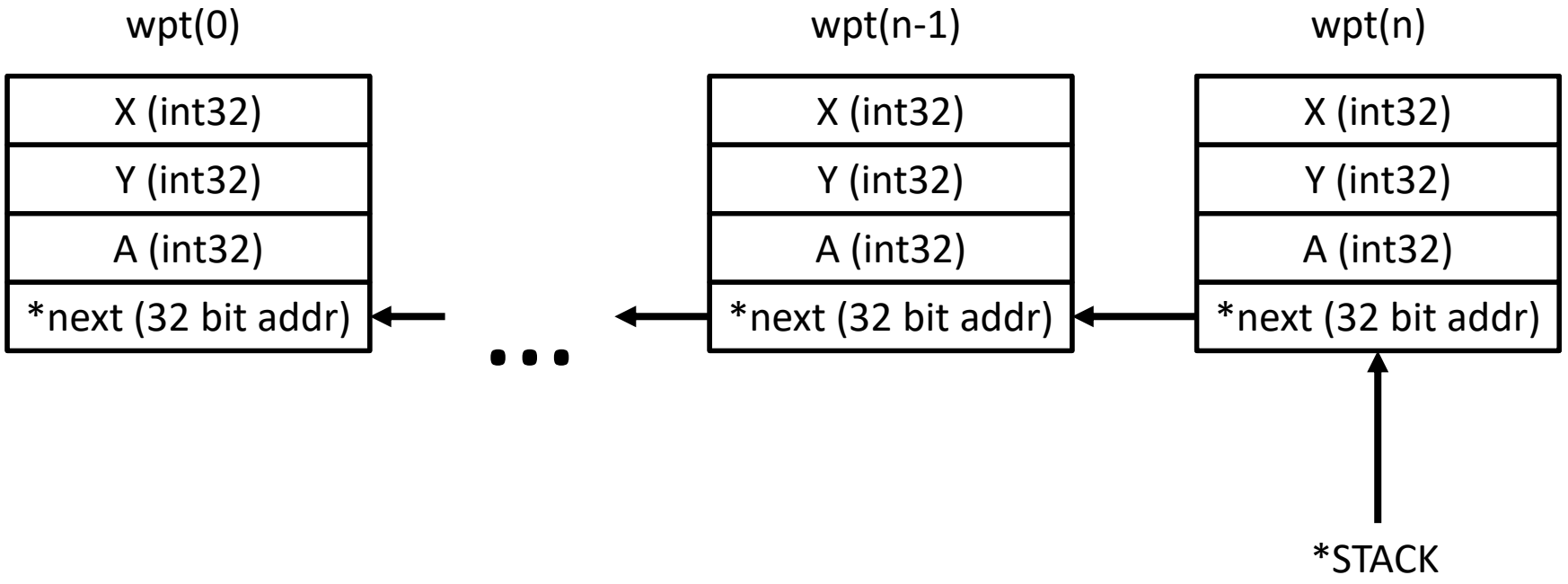
Supponiamo di volere memorizzare nello stack dei waypoint di un percorso che verranno «consumati» da un robot in ordine inverso a come sono stati inseriti (il primo waypoint visitato dal robot sarà l'ultimo inserito)

- Un waypoint è definito come una posizione su una mappa bidimensionale (x,y) e un angolo di rotazione (a) espresso in gradi
- Per semplicità trattiamo le coordinate come numeri interi a 32 bit

```
1 struct{
2     int x;      /* X position on the map */
3     int y;      /* Y position on the map */
4     int a;      /* Expected rotation at waypoint */
5     int *next;  /* A pointer to next element in stack */
6 };
```

«stack» o «pila»
collegamento fra i dati

Possiamo rappresentare lo stack, come una «lista linkata», ovvero come una lista in cui ogni elemento punta all'elemento adiacente



«stack» o «pila»
implementazione funzione push (1)

Push(x,y,a)

INPUT: x,y,a

1. Alloca spazio per un nuovo elemento e lo inizializza con i valori passati come argomento alla funzione.

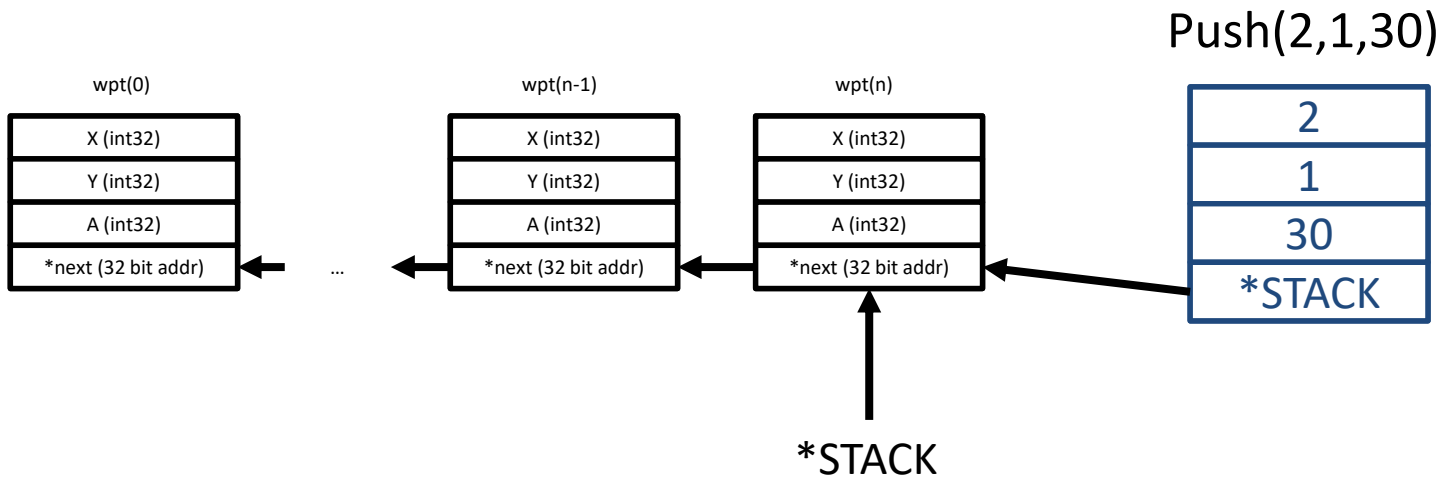
Push(2,1,30)

«stack» o «pila»
implementazione funzione push (2)

Push(x,y,a)

INPUT: x,y,a

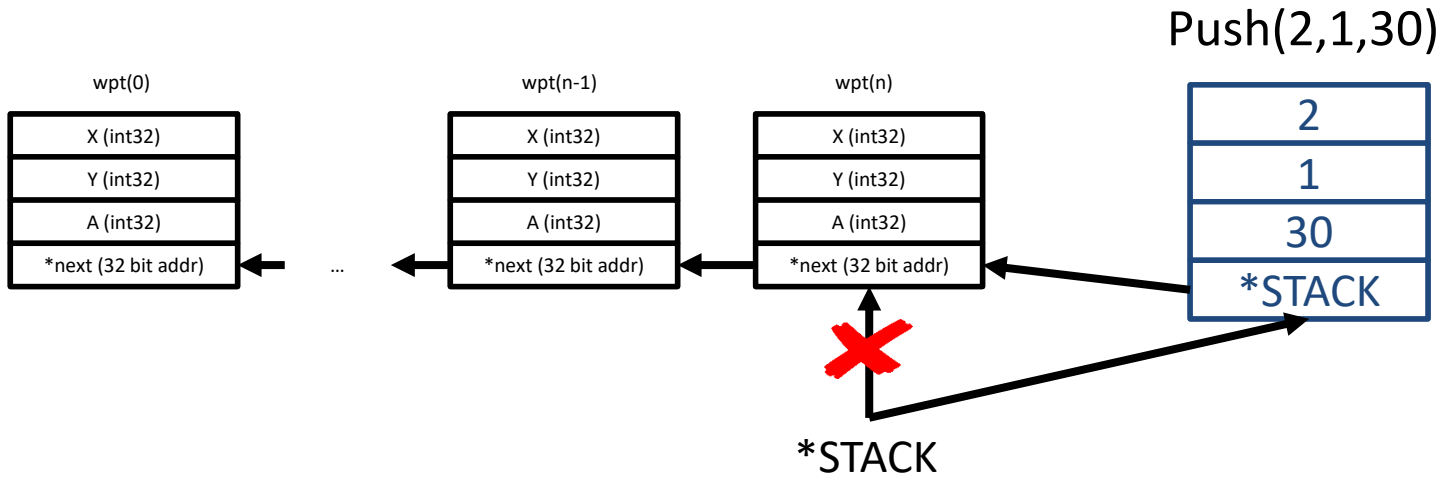
2. Inizializza il puntatore al prossimo elemento con il valore corrente del puntatore allo stack



«stack» o «pila»
implementazione funzione push (3)

Push(x,y,a)
INPUT: x,y,a

3. Aggiorna il puntatore allo stack facendolo puntare all'elemento appena creato



Funzione PUSH

in MIPS

```
.data

stack :   .word 0   # Puntatore all'ultimo elemento inserito nello stack
stack_n : .word 0   # Nummero di elementi attualmente nello stack

.text
.globl stack_push

# Input :
# a0 : x value
# a1 : y value
# a2 : angle value
# Output:
# ---------
stack_push:             # Inserisce un elemento nello stack
    move $t0, $a0
    move $t1, $a1
    move $t2, $a2

    la $t8, stack
    lw $t7, 0($t8)
    la $t9, stack_n

    li $a0, 16          # sbrk(4*4 bytes)
    li $v0, 9
    syscall

    sw $t0, 0($v0)      # inizializza nuovo nodo
    sw $t1, 4($v0)
    sw $t2, 8($v0)
    sw $t7, 12($v0)     # next = *old_stack

    lw $t3, 0($t9)      # stack_n++
    addi $t3, $t3, 1

    sw $v0, 0($t8)      # aggiorna lo stack pointer al nodo appena creato
    sw $t3, 0($t9)      # aggiorna il contatore di elementi

    jr $ra              # ritorna al chiamante
```

Esercizio 9.1

- Nome del file sorgente: *logicalstack.asm*

- Partendo dal codice della funzione PUSH presente sul repository, si completi l'implementazione della struttura dati stack per strutture di tipo waypoint (x,y,a), scrivendo le funzioni:
  - STACK_POP: rimuove un elemento dallo stack e restituisce il suo indirizzo al chiamante
  - STACK_PRINT: stampa l'intero contenuto dello stack partendo dall'ultimo elemento inserito fino al primo

- Scrivere un main per il programma in cui, in un loop infinito, si richieda all'utente di scegliere se inserire o rimuovere un elemento nello stack, oppure stamparne il contenuto. Il programma dovrà anche fornire un'opzione per terminarne l'esecuzione

- **SUGGERIMENTO**: per entrambe le funzioni è necessario gestire il caso in cui lo stack sia ancora vuoto quando vengono richiamate, il campo next del primo elemento inserito nello stack può puntare a un valore convenzionale che sappiamo non essere un indirizzo valido (ad esempio 0)

Esercizio 9.2

- Nome del file sorgente: *logicalstack_extended.asm*
- Partendo dal codice dell'esercizio 9.1, estendere le funzionalità dello stack, implementando una funzione **in_stack**, che prenda come parametri gli elementi della tupla (x,y,a) e restituisca 1 se l'elemento è presente nello stack, 0 altrimenti
- Estendere il main, fornendo un'opzione per testare anche questa funzione
- **NOTA**: la struttura dati stack classica, non prevede per sua natura funzionalità di ricerca. Questo esercizio mostra come mantenendo la stessa organizzazione dei dati, sia possibile estenderne le funzionalità semplicemente implementando funzioni extra, ottenendo così una struttura dati più complessa, come ad esempio una lista

Università degli Studi di Milano
Laboratorio di Architettura degli Elaboratori II
Corso di Laurea in Informatica, A.A. 2017-2018

Jacopo Essenziale
Dipartimento di Informatica
Via Celoria 20 - 20133 Milano (MI)
AISLab
jacopo.essenziale@unimi.it
+39 02.503.14010

Hanno contribuito alla realizzazione di queste slides:

- Nicola Basilico