# Lezione 12

# Grafi

Roberto Trani
roberto.trani@di.unipi.it

Pagina web del corso
http://didawiki.cli.di.unipi.it/doku.php/informatica/all-b/start

# Esercizio 1

## Tabelle Hash: inserimento

Scrivere un programma che legga da tastiera una sequenza di $n$ interi **distinti** e li inserisca in una tabella hash di dimensione $2n$ posizioni utilizzando liste monodirezionali per risolvere eventuali conflitti.

Utilizzare la funzione hash $h(x) = ((ax + b) \ \% \ p) \ \% \ 2n$ dove $p$ è il numero primo 999149 e $a$ e $b$ sono interi positivi minori di 10.000 scelti casualmente.

Una volta inseriti tutti gli interi, il programma deve stampare la lunghezza massima delle liste e il numero totale di conflitti.

Prima di scrivere il programma chiedersi perché la tabella ha dimensione $2n$ e non $n$.

# Esercizio 1

```
typedef struct _node {
  struct _node *next;
  int value;
} node;

int hash(int x, int a, int b, int table_size) {
  int p = 999149;
  return ((a*x + b) % p) % table_size;
}
```

# Esercizio 1

```
 void insert_list(node** list, int x) {
  node* new = (node*)malloc(sizeof(node));
  new->value = x;
  new->next = *list;
  *list=new;
}

void insert(node** ht, int x, int a, int b, int table_size) {
  int index = hash(x, a, b, table_size);
  insert_list(&ht[index], x);
}

int len(node* list) {
  if (list==NULL) return 0;
  return 1+len(list->next);
}
```

# Esercizio 2

## Tabelle Hash: inserimento con rimozione dei duplicati

Scrivere un programma che legga da tastiera una sequenza di $n$ interi **NON distinti** e li inserisca senza duplicati in una tabella hash di dimensione $2n$ posizioni utilizzando liste monodirezionali per risolvere eventuali conflitti.

Utilizzare la funzione hash $h(x) = ((ax + b) \ \% \ p) \ \% \ 2n$ dove $p$ è il numero primo 999149 e $a$ e $b$ sono interi positivi minori di 10.000 scelti casualmente.

Una volta inseriti tutti gli interi, il programma deve stampare il numero totale di conflitti, la lunghezza massima delle liste e il numero di elementi distinti.

# Esercizio 2

```
void insert_list(node** list, int x) {
 while (*list != NULL) {
   if ((*list)->value == x) return;
   list = &((*list)->next);
 }

 node* new = (node*)malloc(sizeof(node));
 new->value = x;
 new->next = NULL;
 *list=new;
}
```

# Esercizio 3

## Liste: cancellazione

Scrivere un programma che legga da tastiera una sequenza di $n$ interi distinti e li inserisca in una lista monodirezionale. Successivamente il programma deve calcolare la media aritmetica dei valori della lista ed eliminare tutti gli elementi il cui valore è inferiore o uguale alla media, troncata all'intero inferiore. Ad esempio:

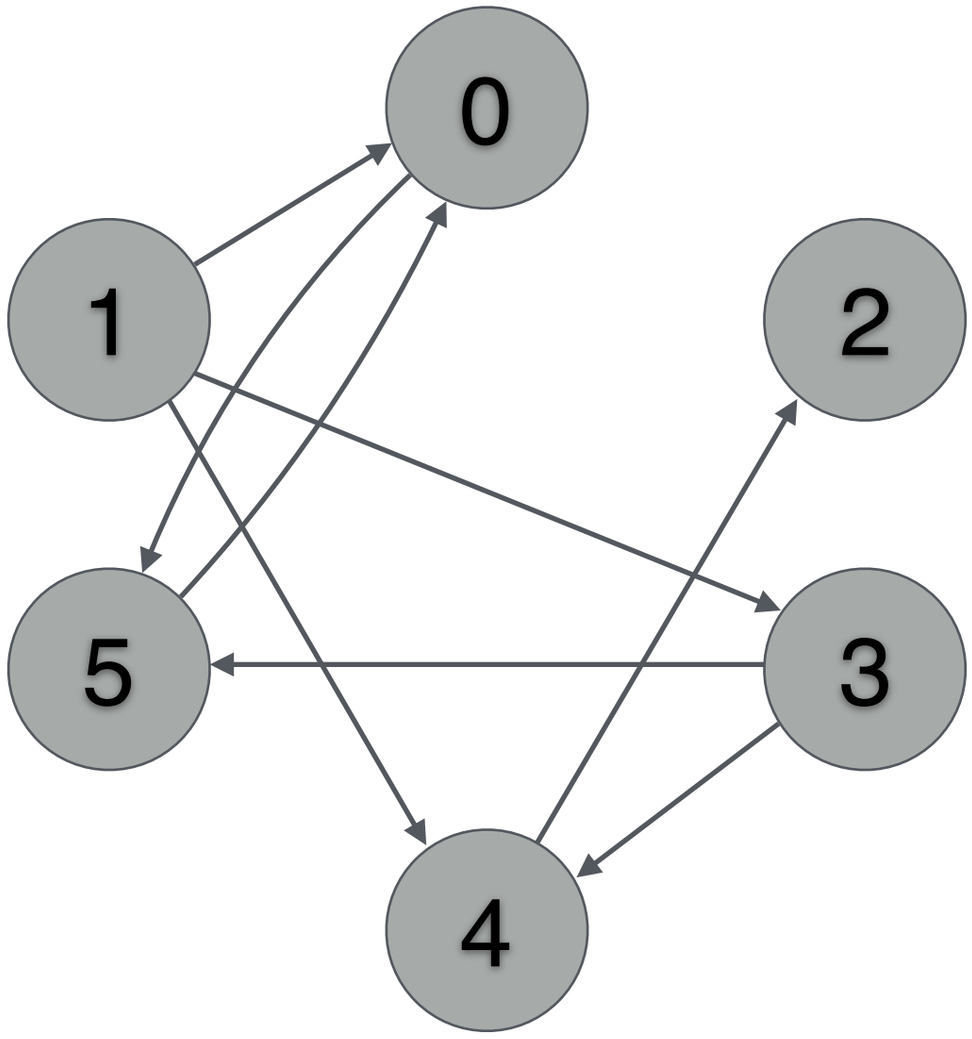
$$avg(1, 2, 4) = 7/3 = 2$$

**IMPORTANTE**: Si abbia cura di liberare la memoria dopo ogni cancellazione.

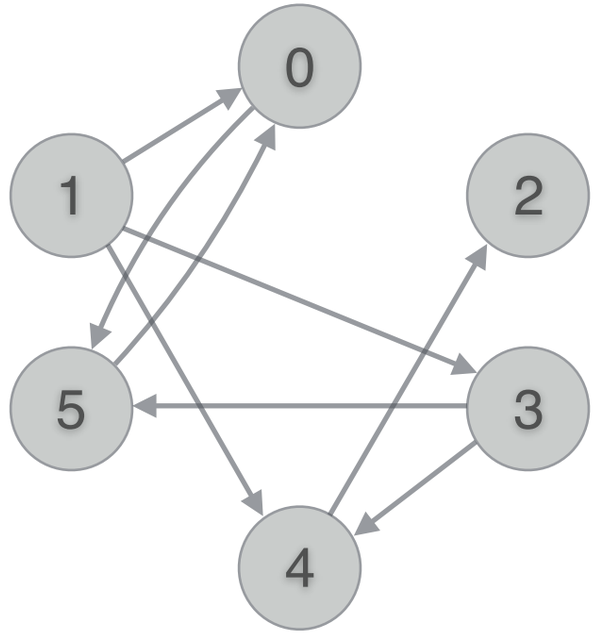
# Esercizio 3

```
int list_average(node* list) {
    int sum = 0, count = 0;
    for (; list != NULL; list = list->next) {
      sum += list->value;
      ++count;
    }
    return (count > 0) ? (sum / count) : 0;
}

void delete_below(node** list, int x) {
  while (*list != NULL) {
    if ((*list)->value > x) {
      list = &(*list)->next;
    } else {
      node* to_delete = *list;
      *list = (*list)->next;
      free(to_delete);
    }
  }
}
```

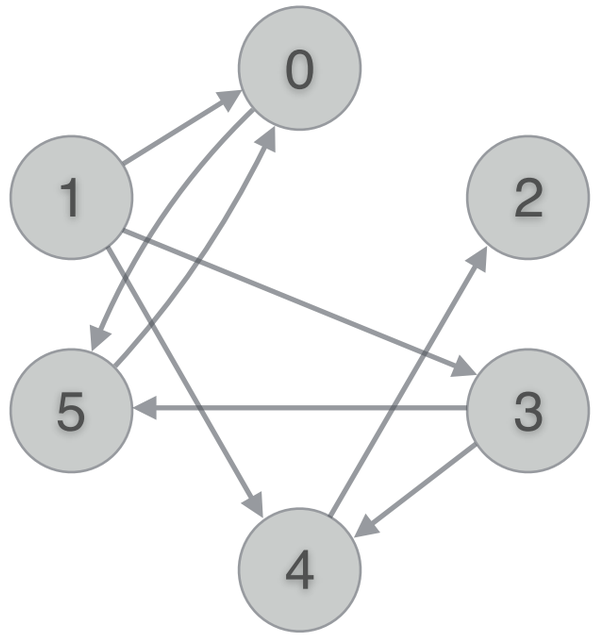
# Grafi



# Matrice di adiacenza



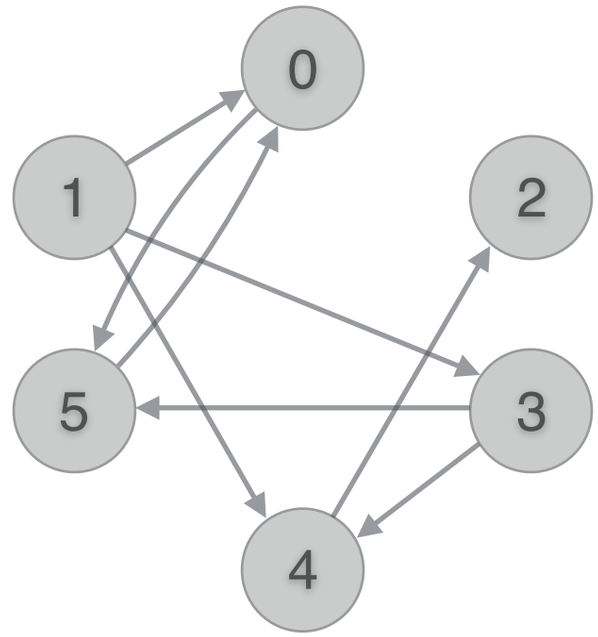
# Matrice di adiacenza




M

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | **1** |
| **1** | 0 | 0 | **1** | **1** | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | **1** | **1** |
| 0 | 0 | **1** | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 |

# Matrice di adiacenza

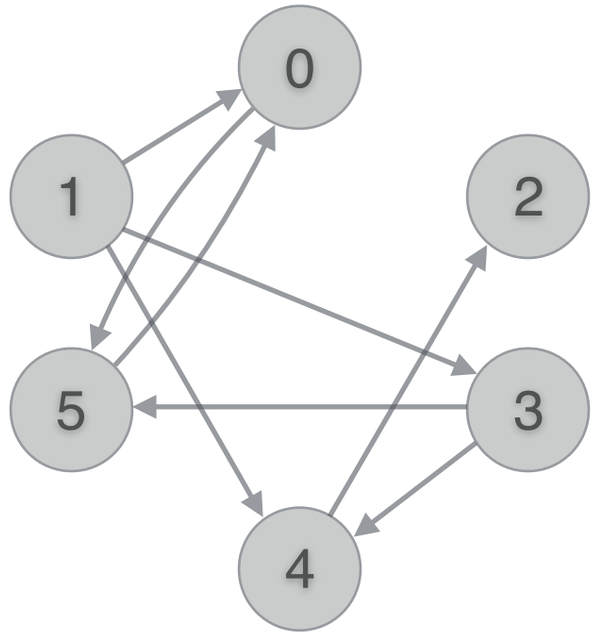



M

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | **1** |
| **1** | 0 | 0 | **1** | **1** | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | **1** | **1** |
| 0 | 0 | **1** | 0 | 0 | |
| **1** | 0 | 0 | 0 | 0 | |

```
int ** M = (int **) malloc(N*sizeof(int *));
for (int i=0; i<N; ++i) {
    M[i] = (int *) malloc(N*sizeof(int));
}
```

# Matrice di adiacenza



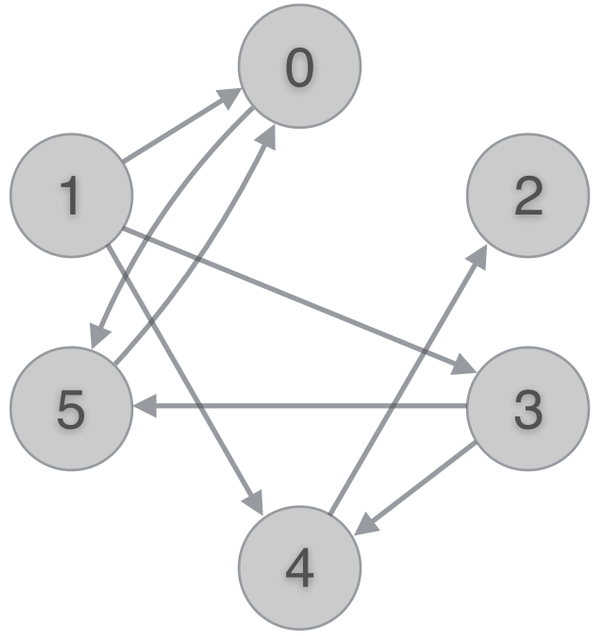

M

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | **1** |
| **1** | 0 | 0 | **1** | **1** | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | **1** | **1** |
| 0 | 0 | **1** | 0 | 0 | |
| **1** | 0 | 0 | 0 | 0 | |

Troppi zeri…

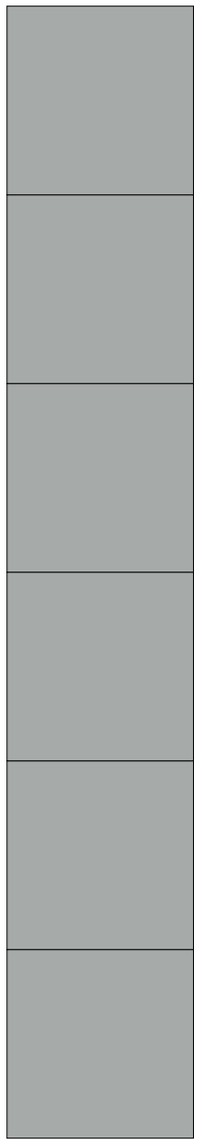
```
int ** M = (int **) malloc(N*sizeof(int *));
for (int i=0; i<N; ++i) {
    M[i] = (int *) malloc(N*sizeof(int));
}
```

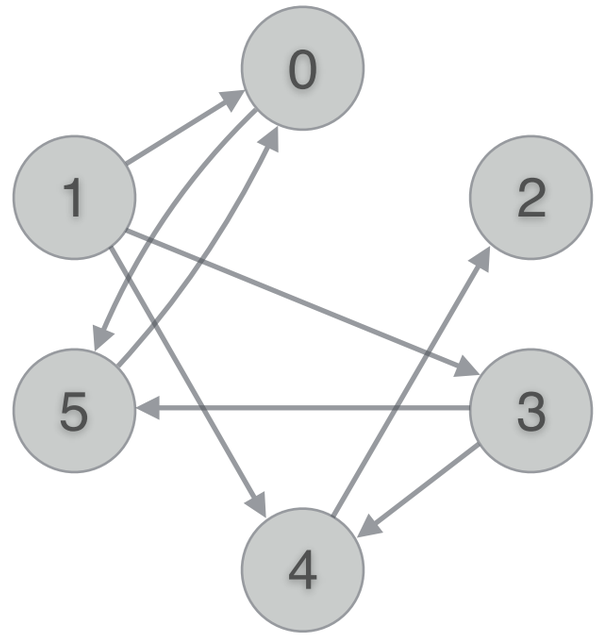
# Liste di adiacenza



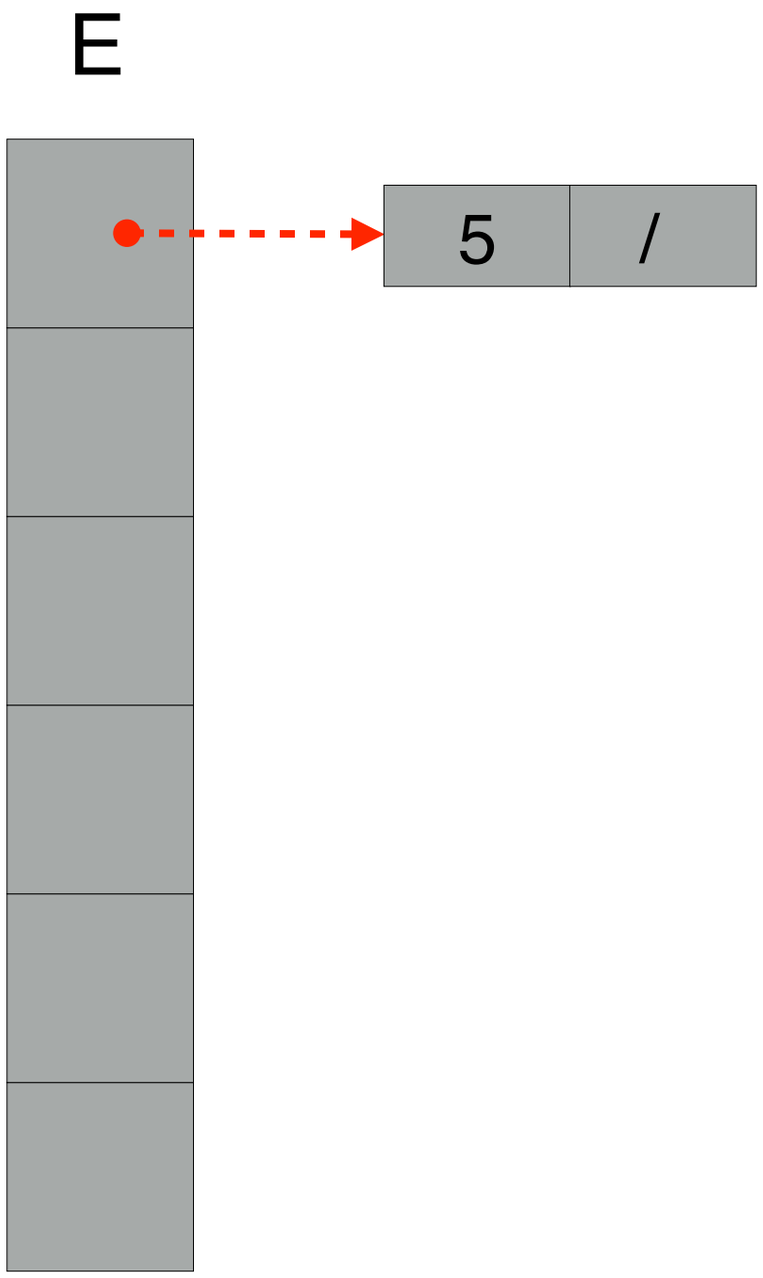
# Liste di adiacenza

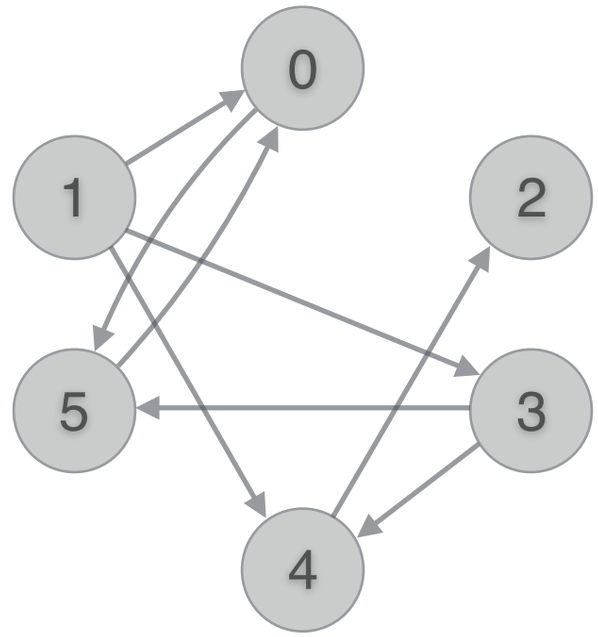
E



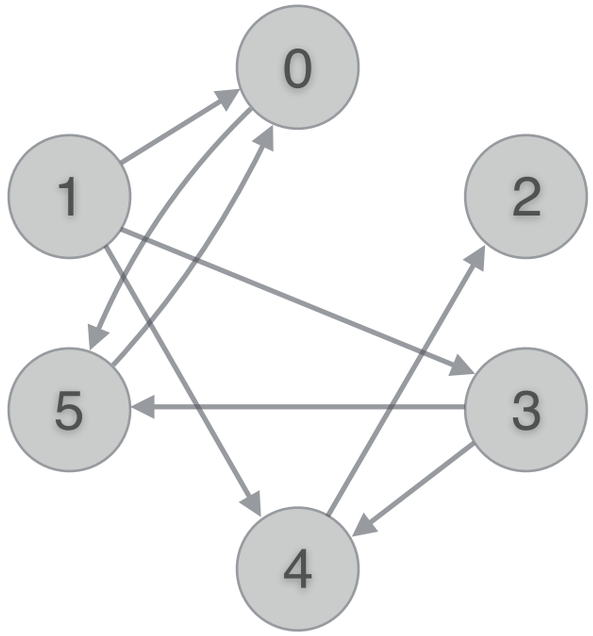


# Liste di adiacenza

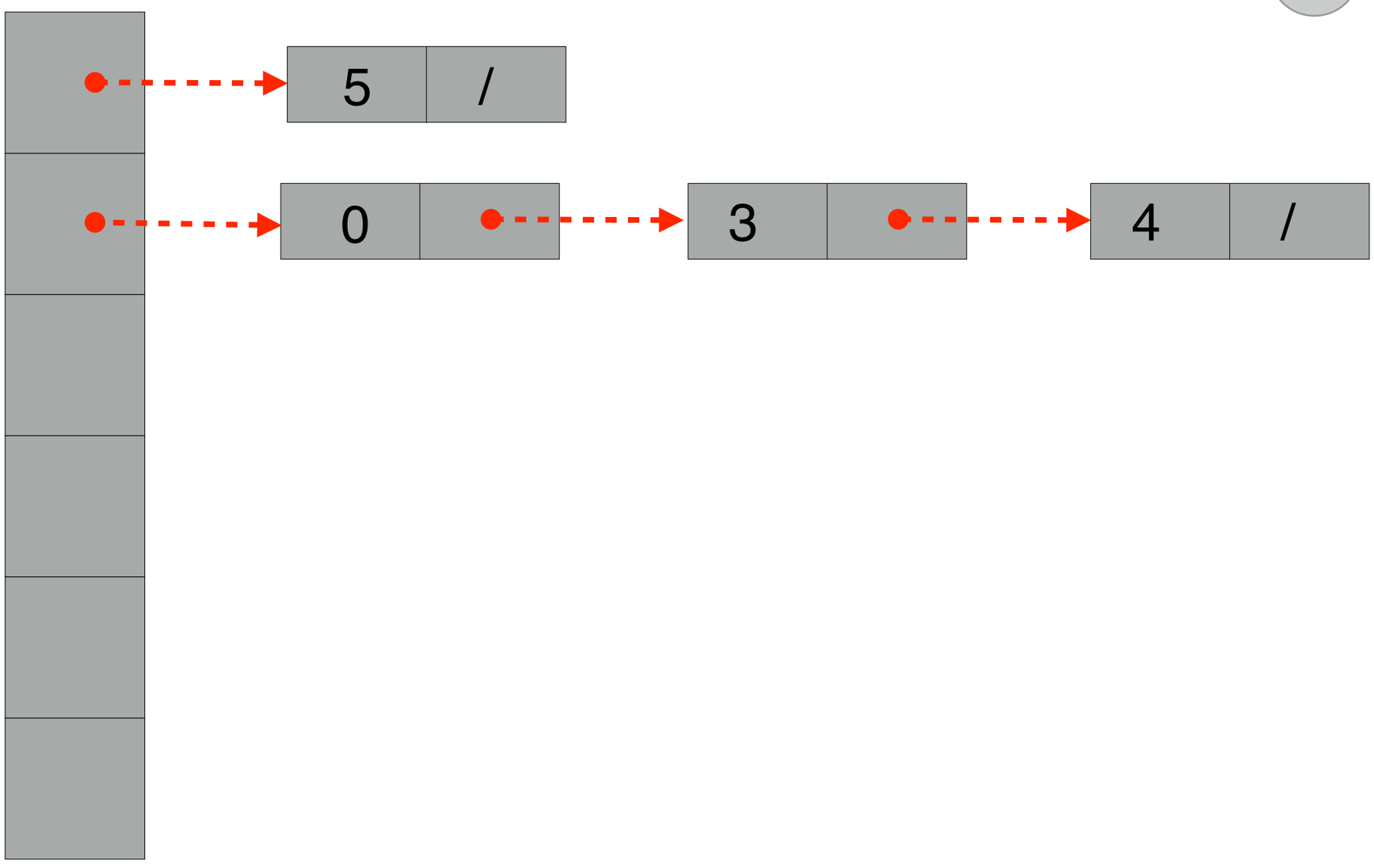


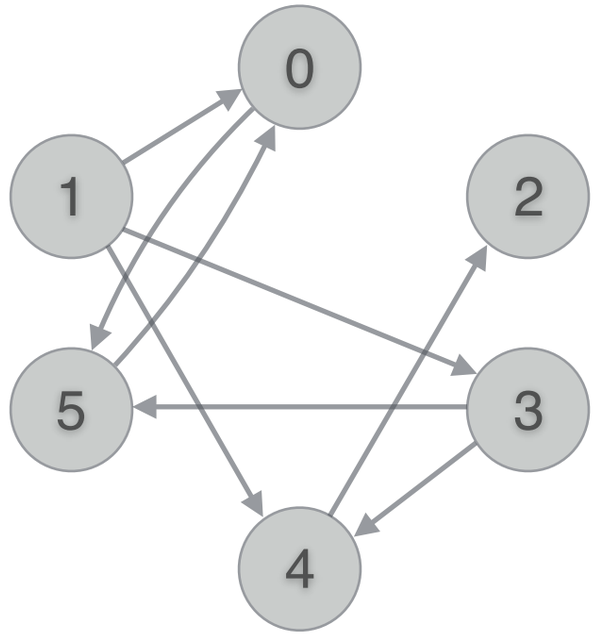
# Liste di adiacenza



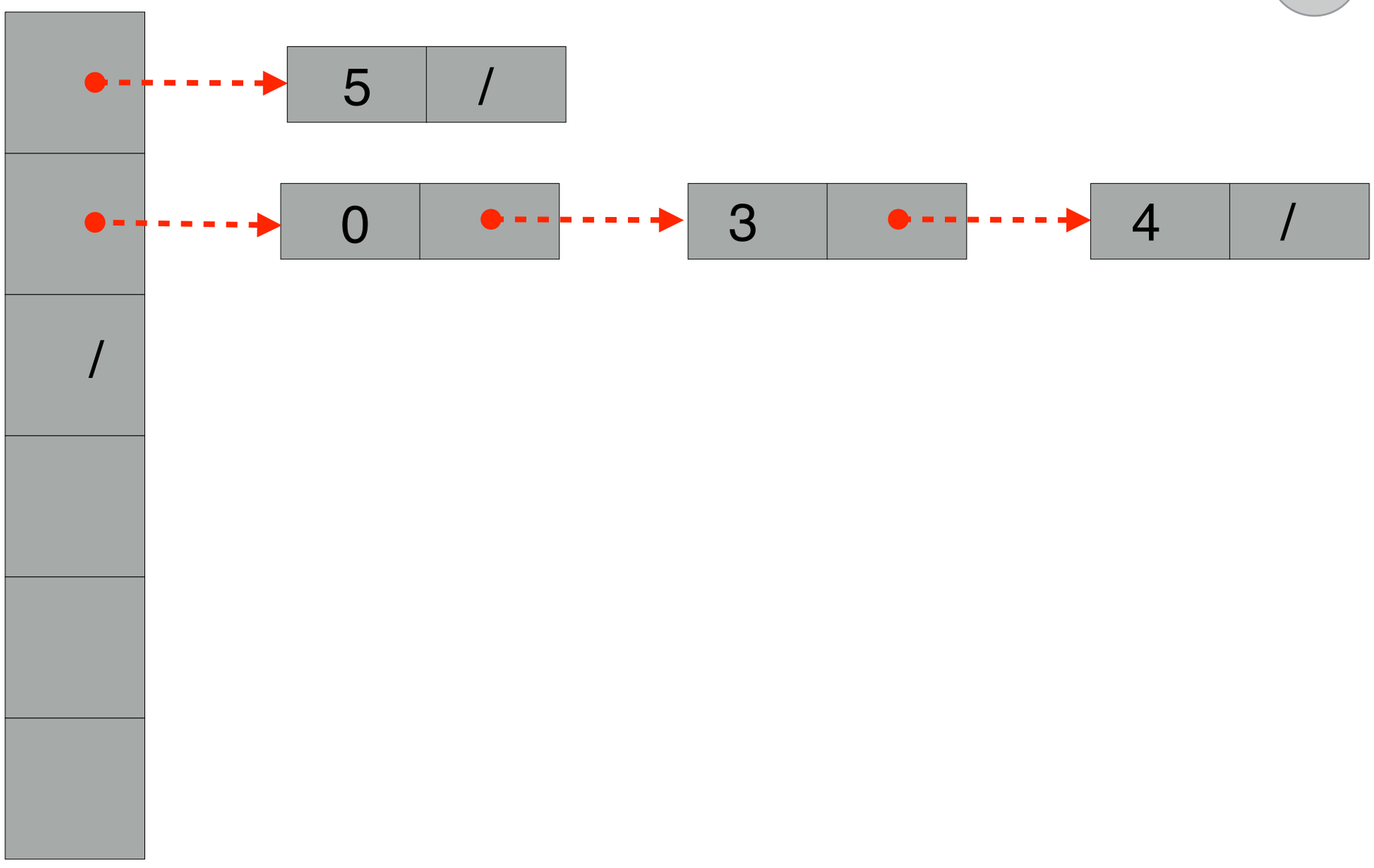

E



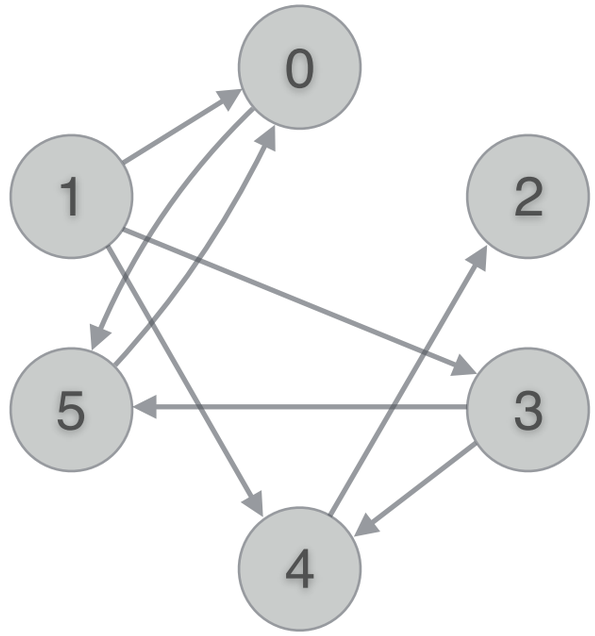
# Liste di adiacenza



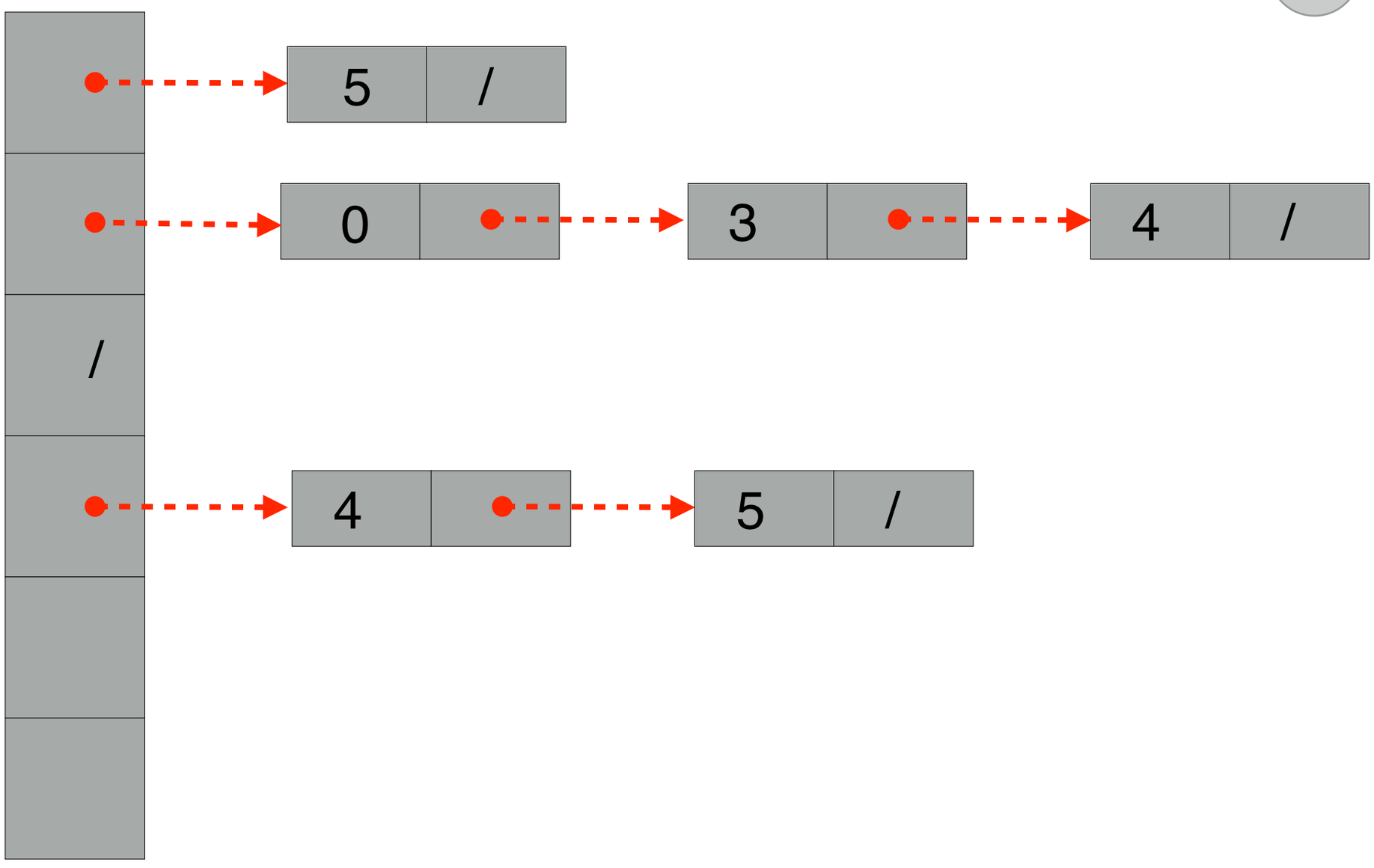

E



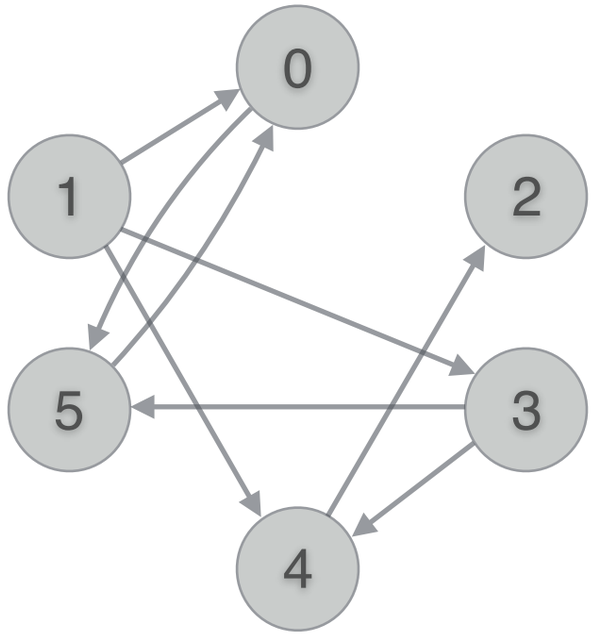
# Liste di adiacenza



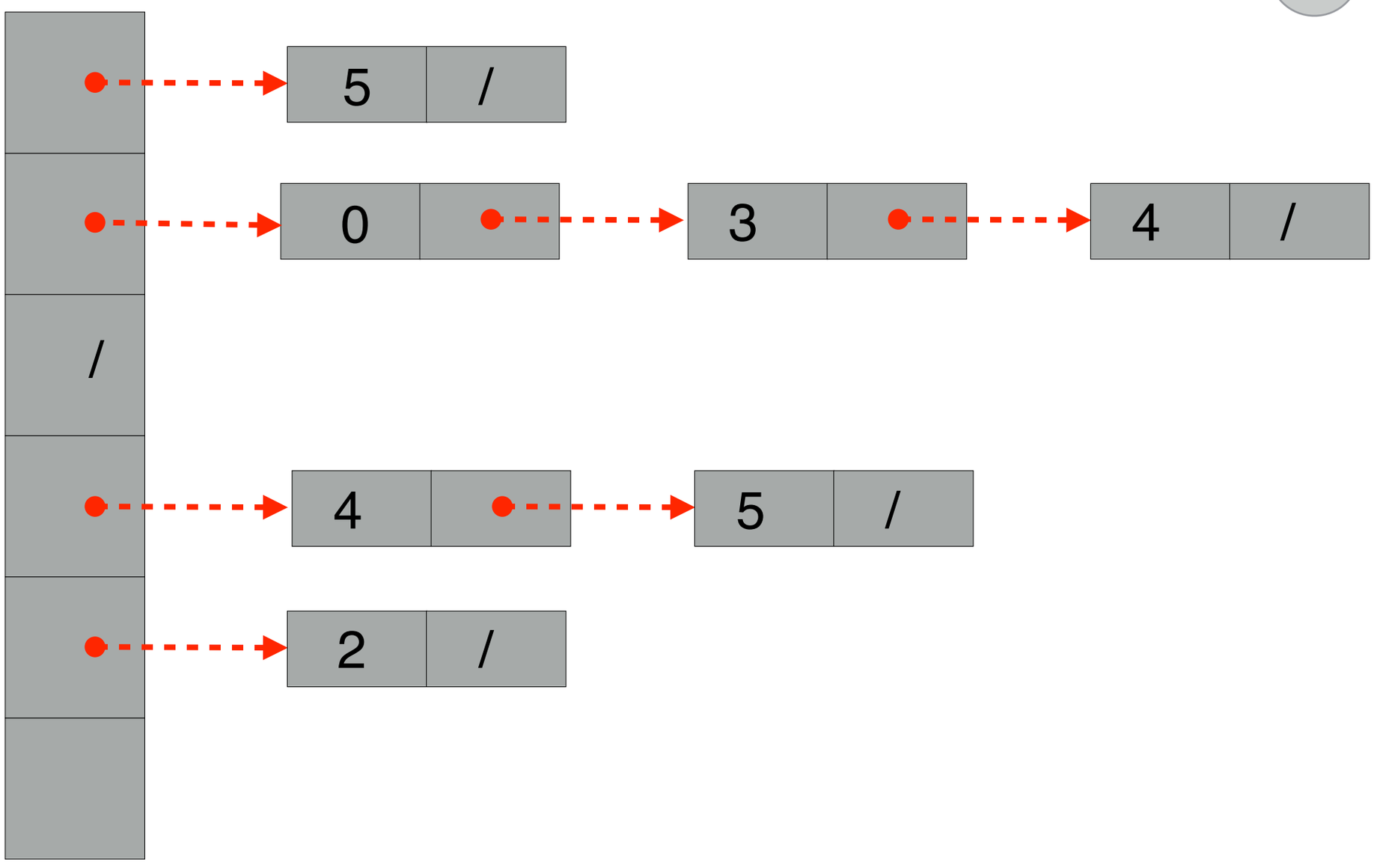

E



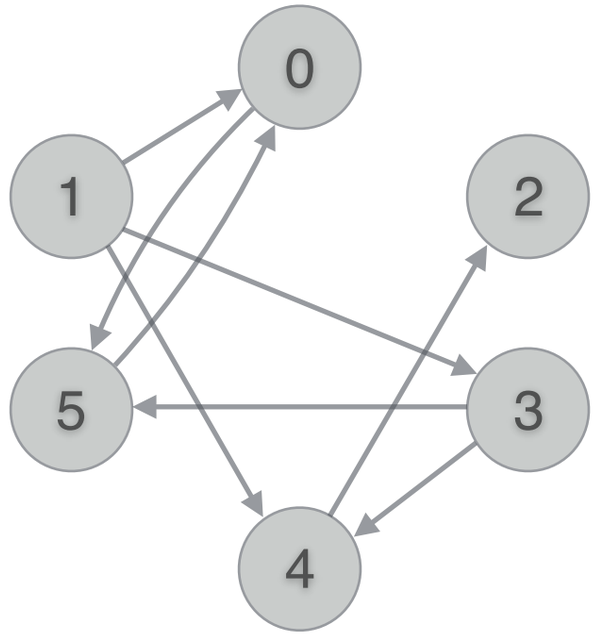
# Liste di adiacenza



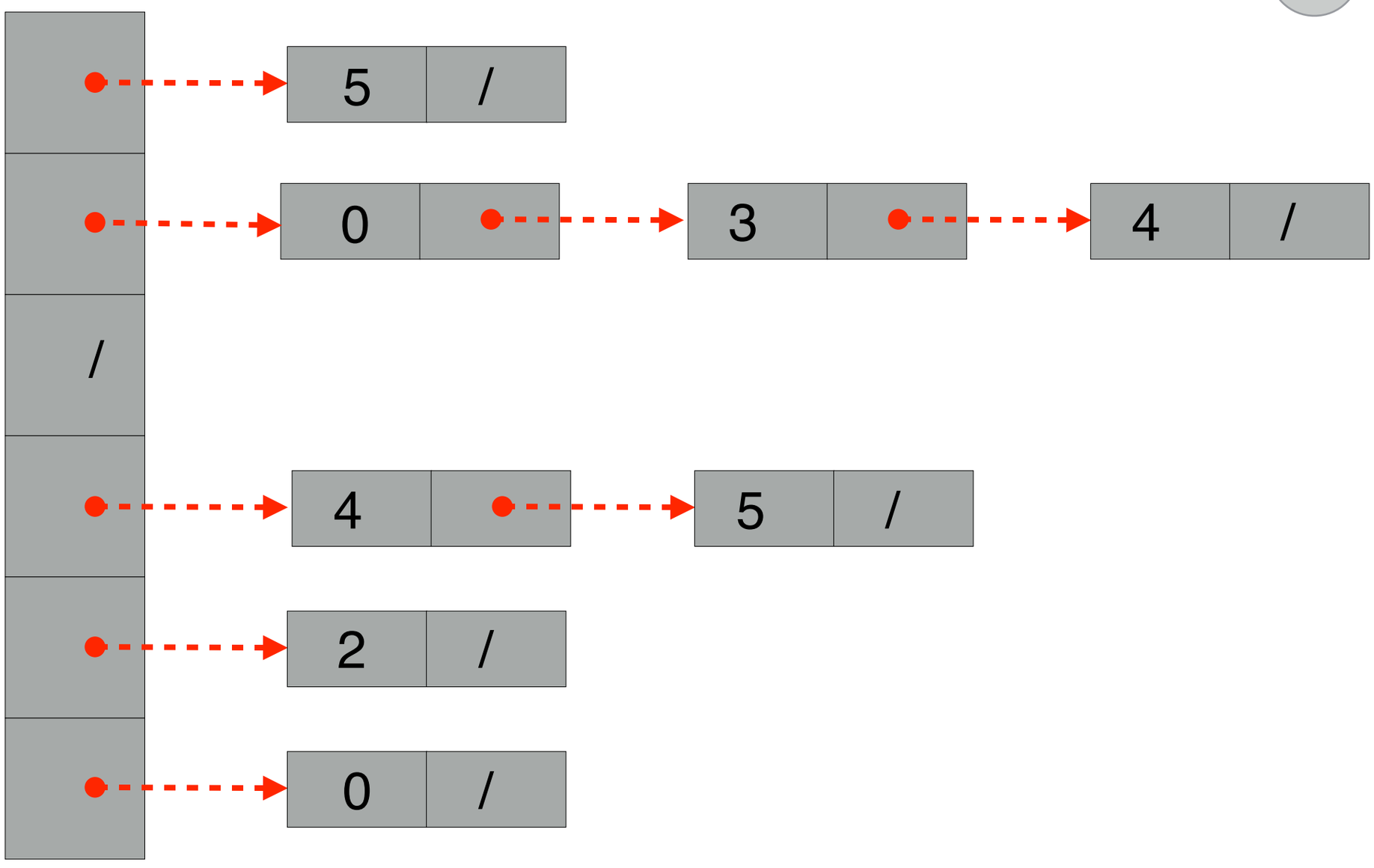

E



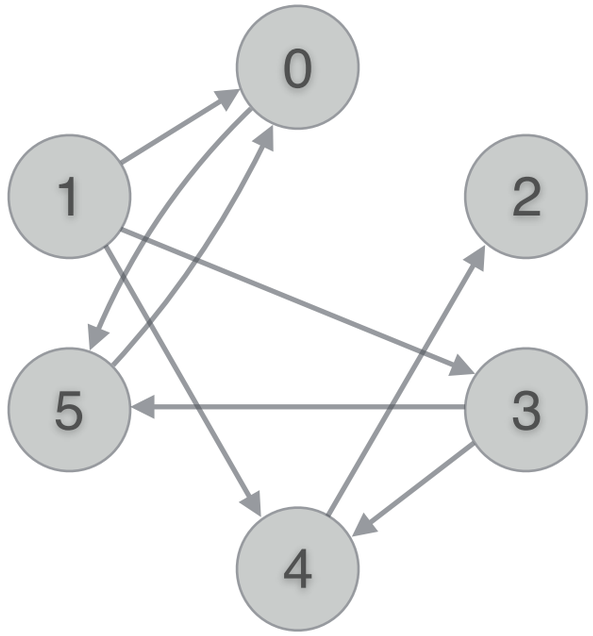
# Liste di adiacenza

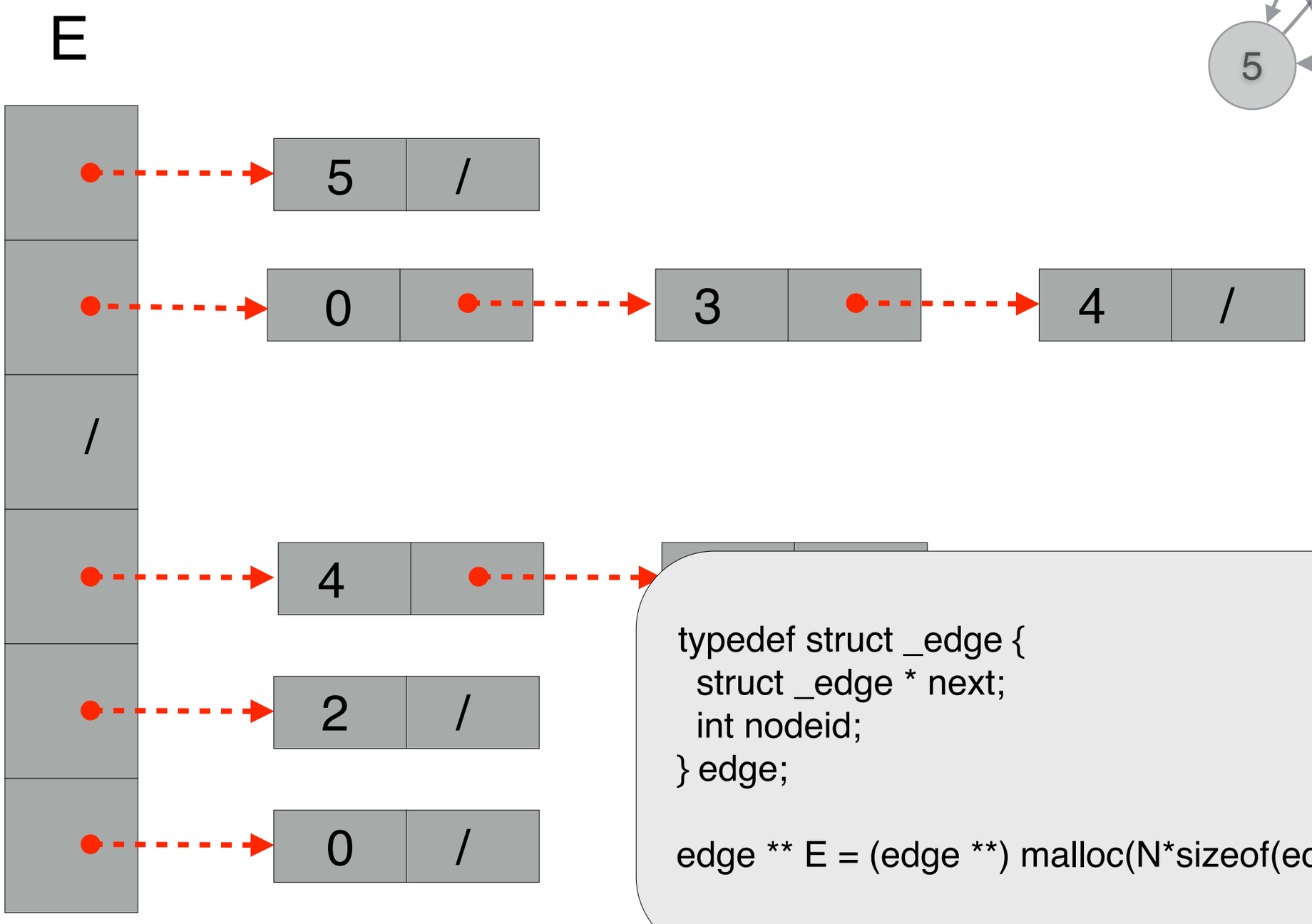



E



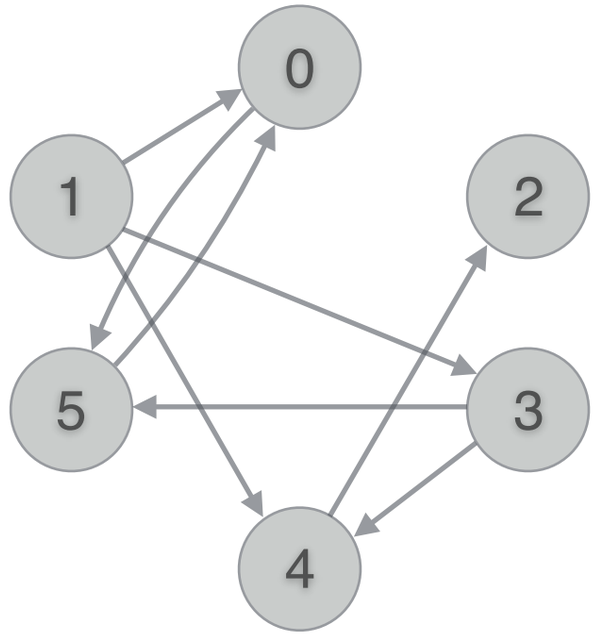
# Liste di adiacenza

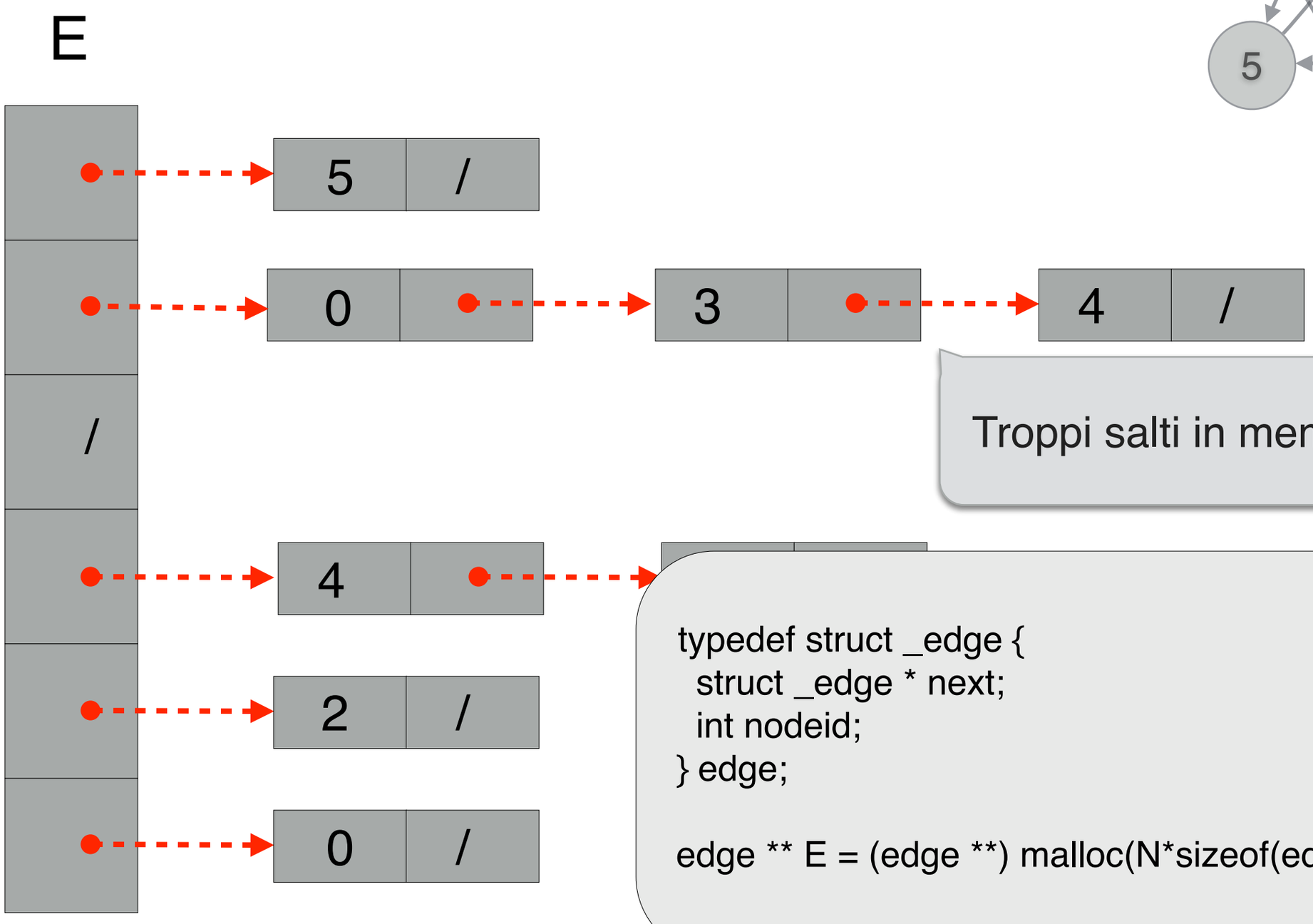



```
typedef struct _edge {
  struct _edge * next;
  int nodeid;
} edge;

edge ** E = (edge **) malloc(N*sizeof(edge *));
```

# Liste di adiacenza

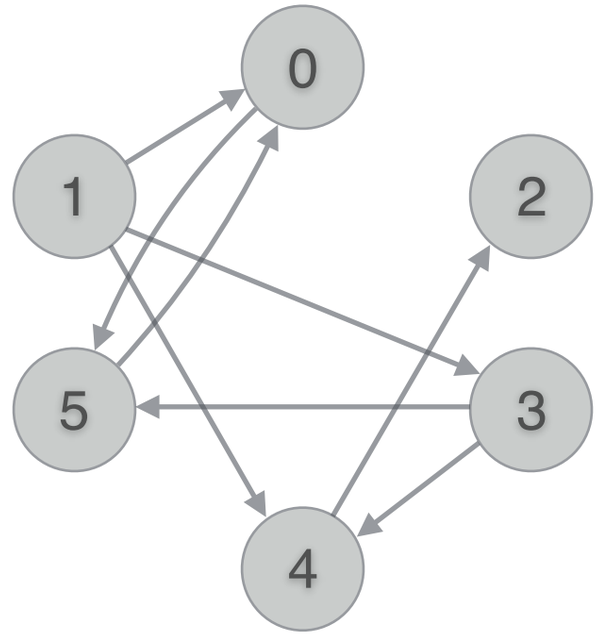


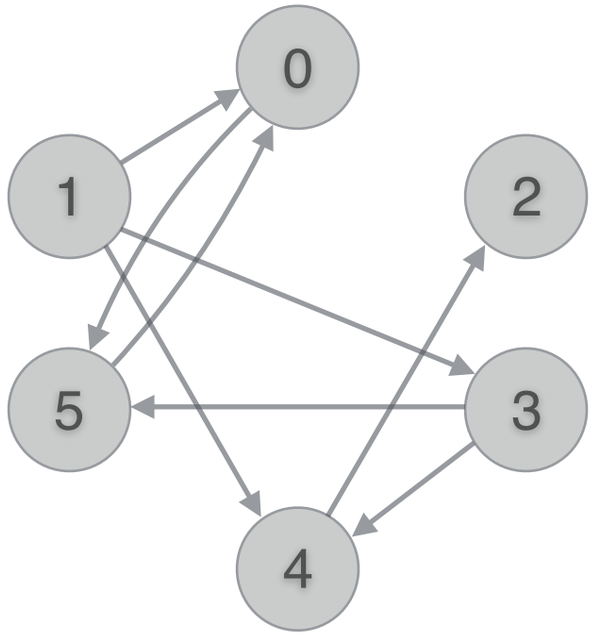

Troppi salti in memoria…

```
typedef struct _edge {
  struct _edge * next;
  int nodeid;
} edge;

edge ** E = (edge **) malloc(N*sizeof(edge *));
```

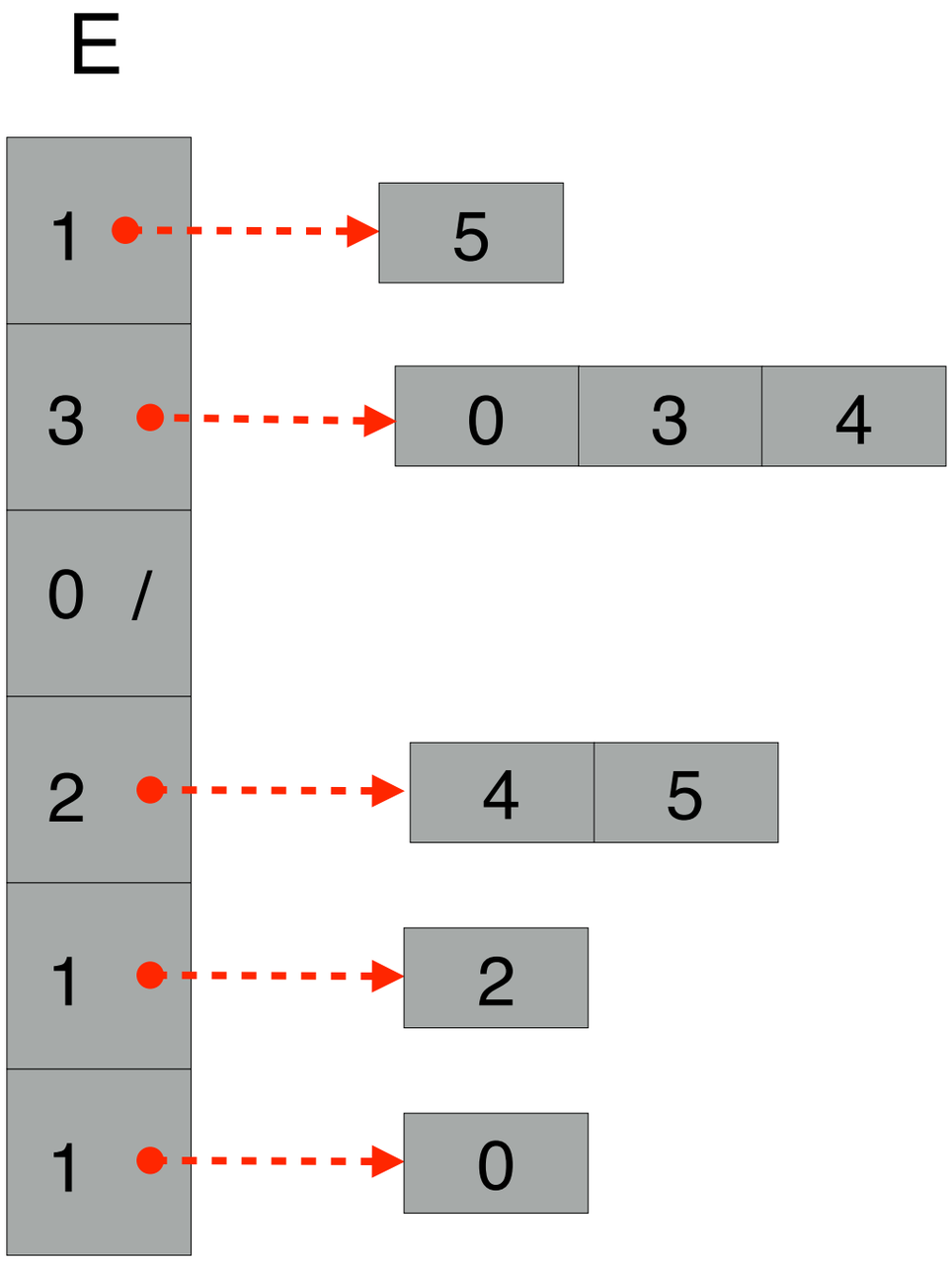
# Liste di adiacenza compatte



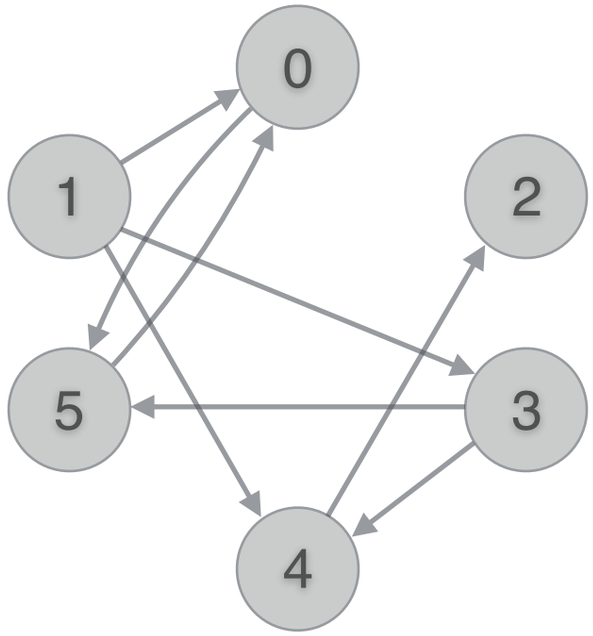
# Liste di adiacenza compatte



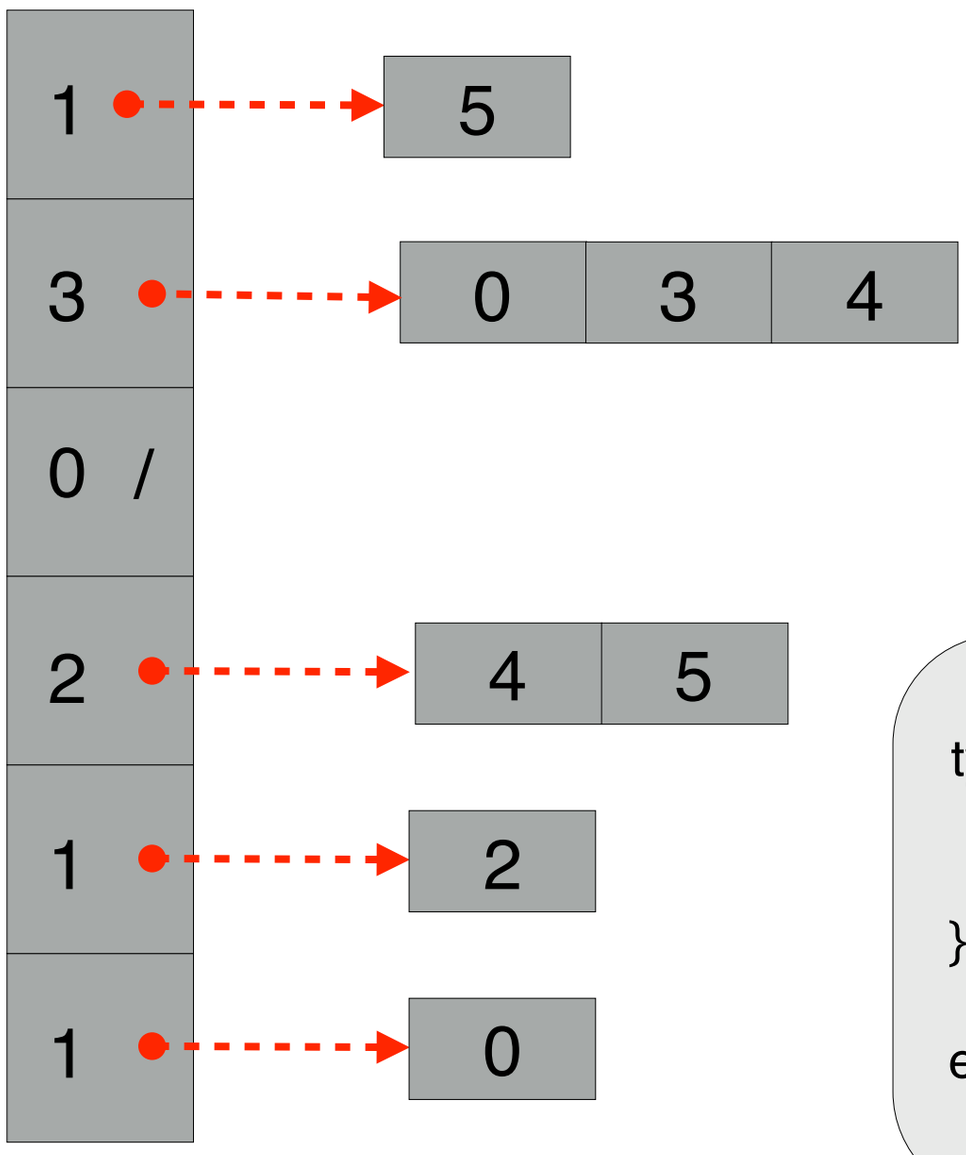

E



# Liste di adiacenza compatte

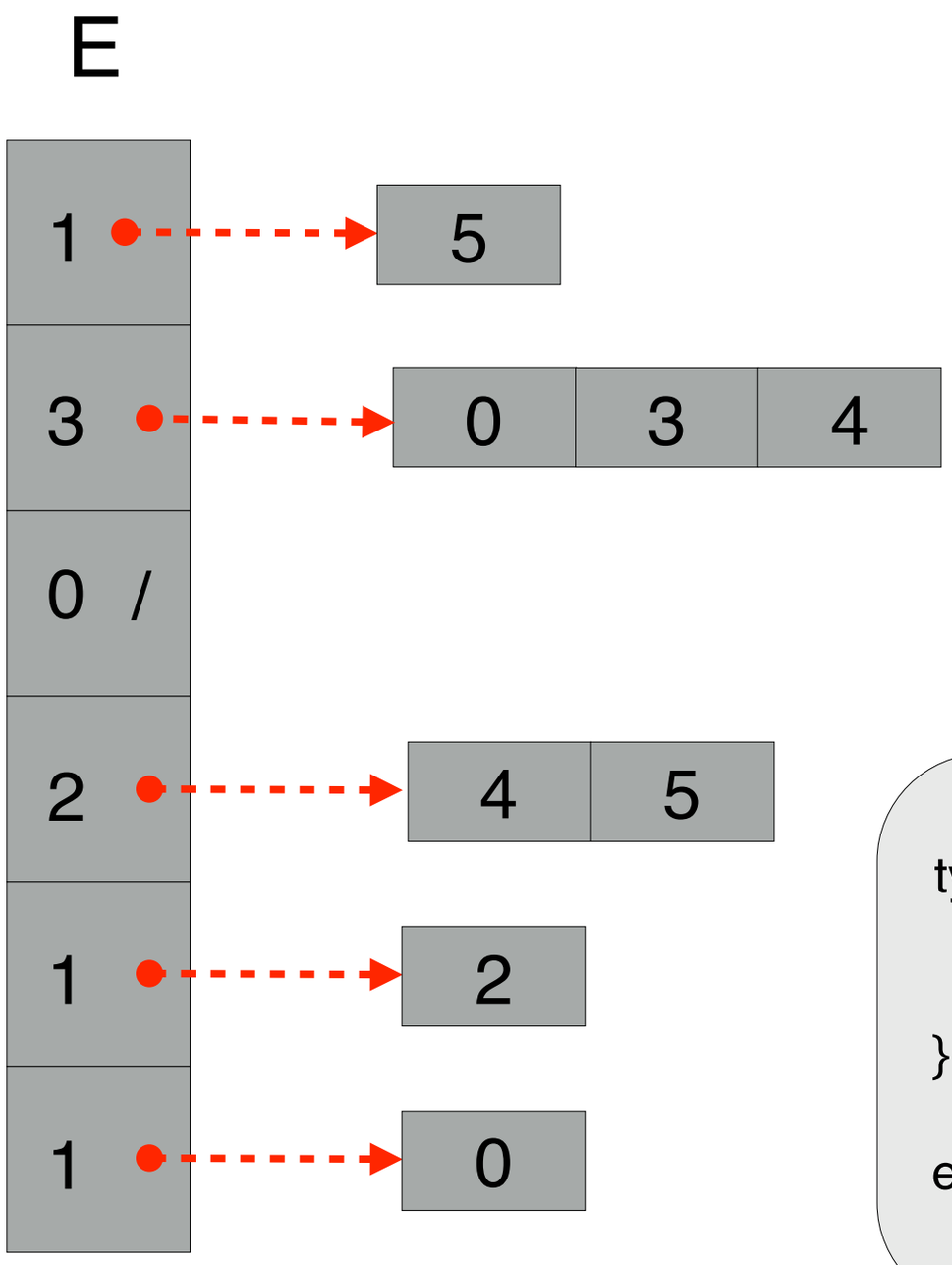


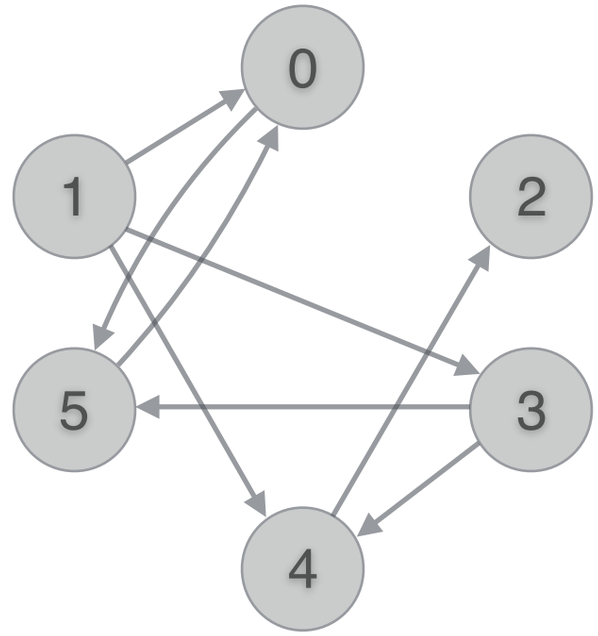

E




```
typedef struct _edges {
  int num;
  int * edges;
} edges;

edges *E = (edges *) malloc(N*sizeof(edges));
```

# Liste di adiacenza compatte




I grafi dinamici possono essere realizzati usando vettori dinamici, o ridimensionabili — algoritmo dimezza e raddoppia… —

```
typedef struct _edges {
  int num;
  int * edges;
} edges;

edges *E = (edges *) malloc(N*sizeof(edges));
```

# Lettura grafo da input

# Lettura grafo da input

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

# Lettura grafo da input

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input

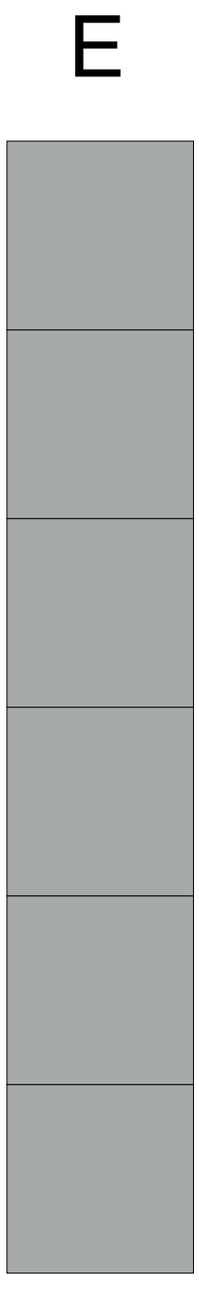
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



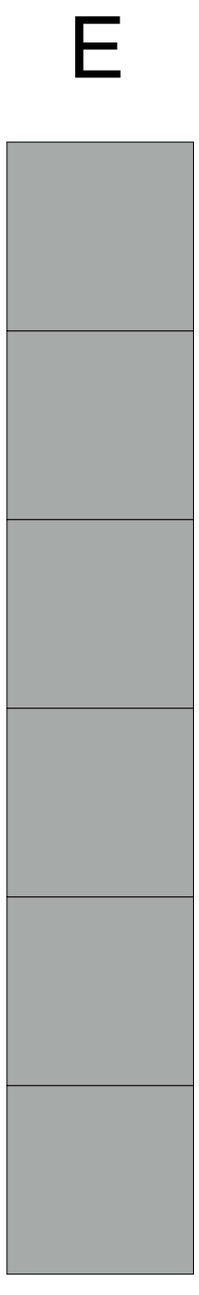

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



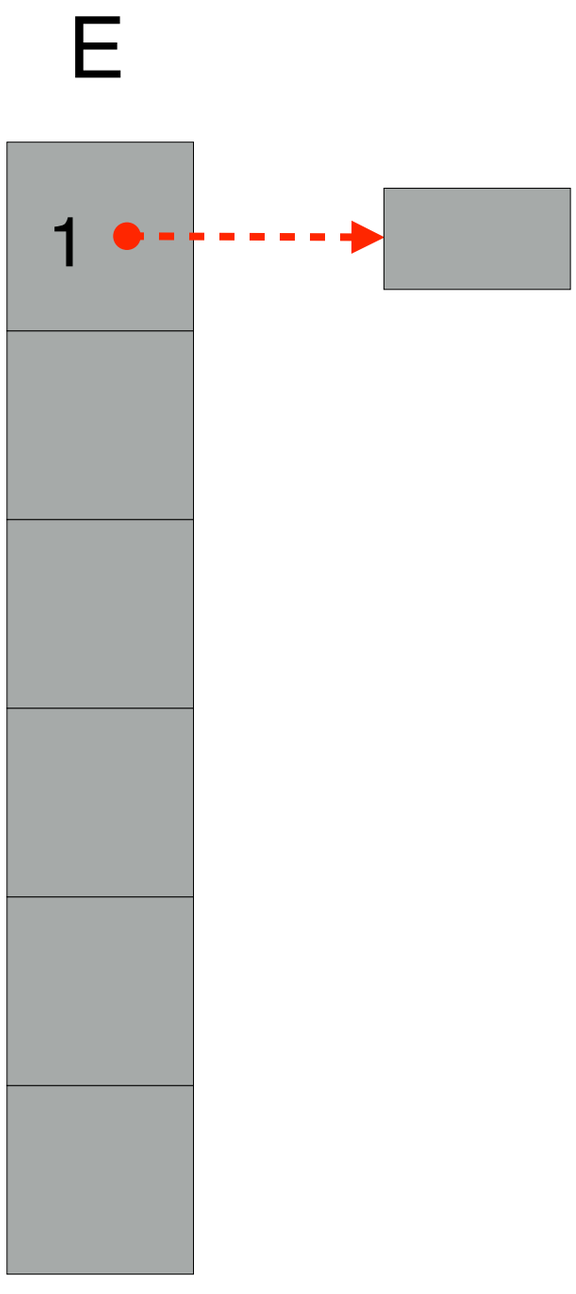

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



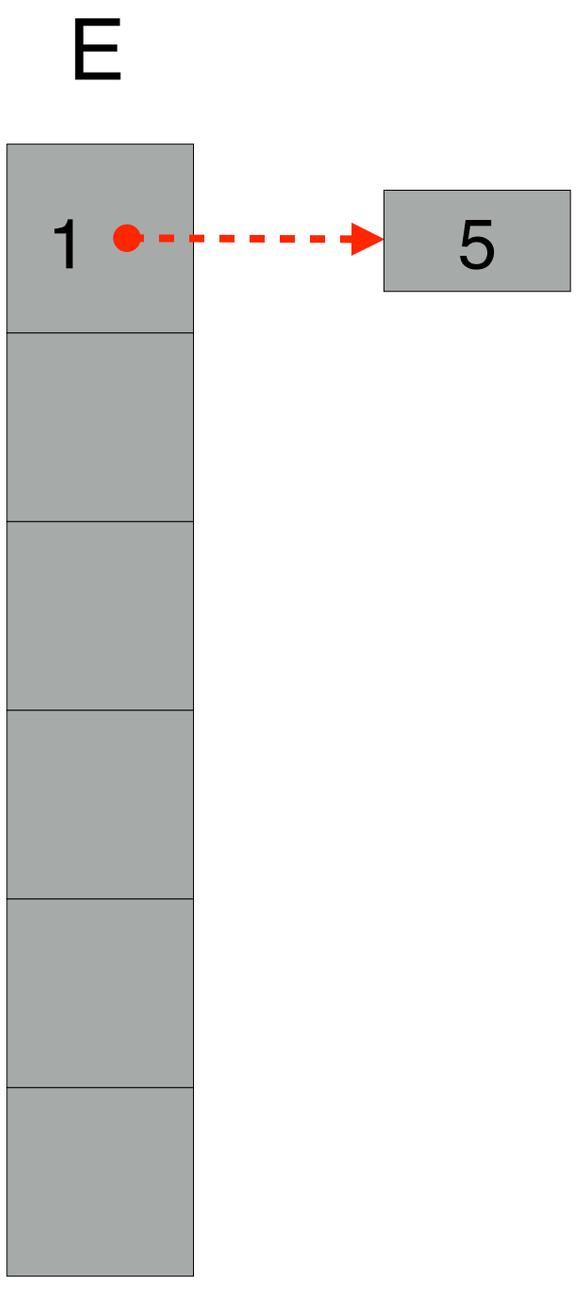

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



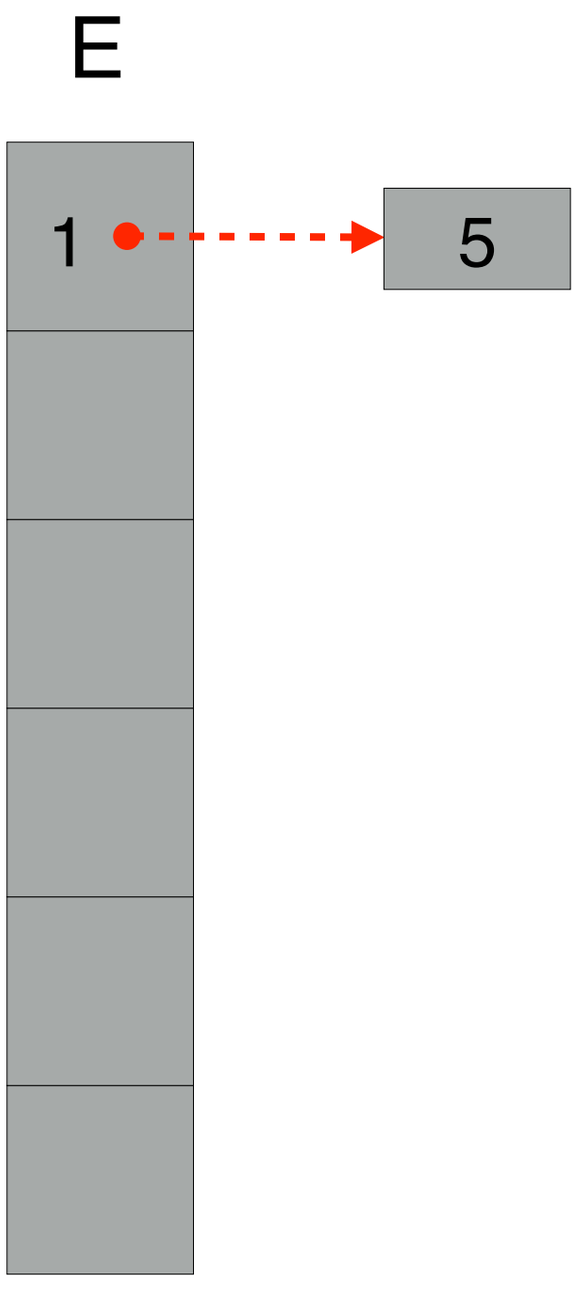

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



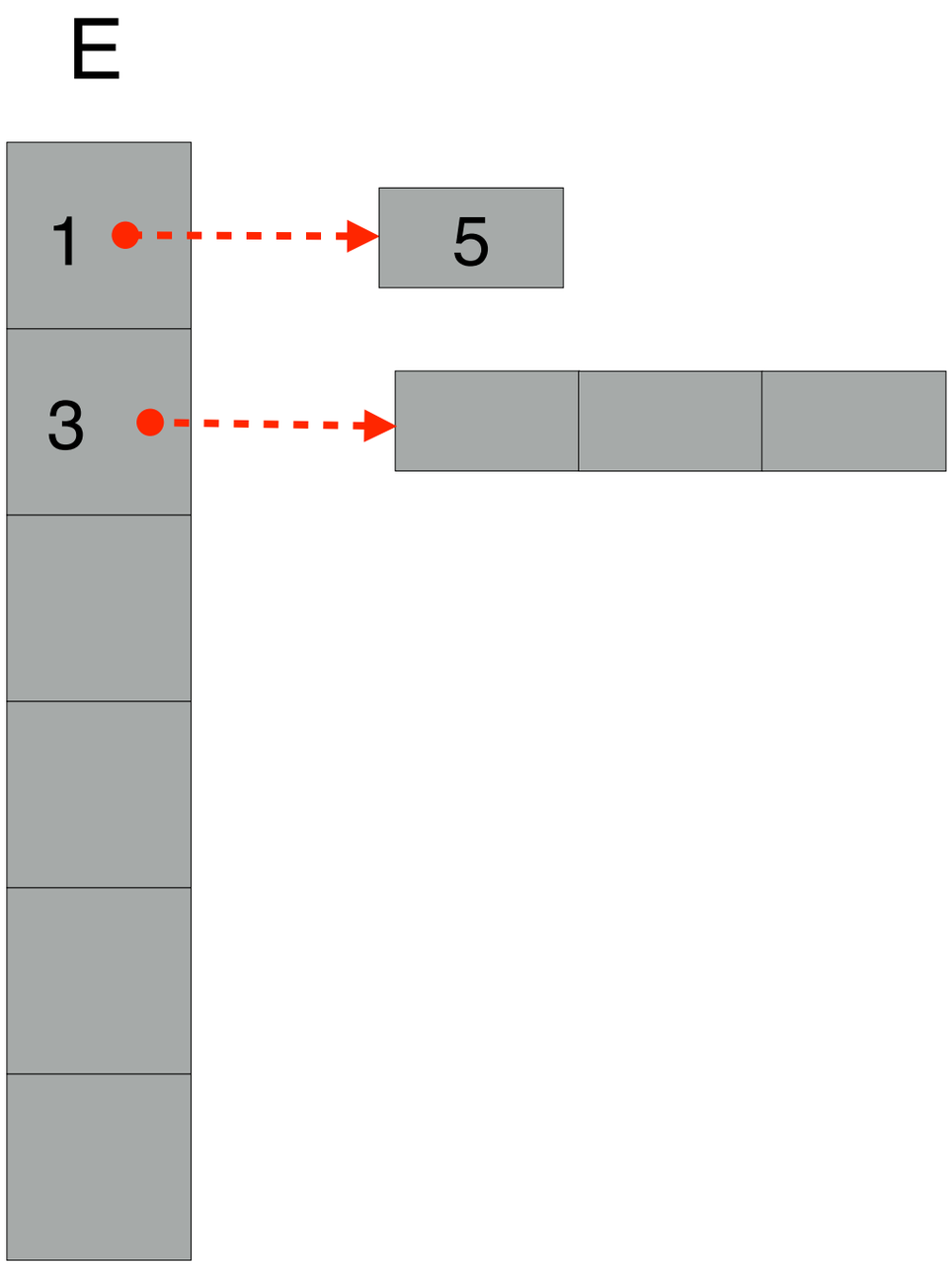

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



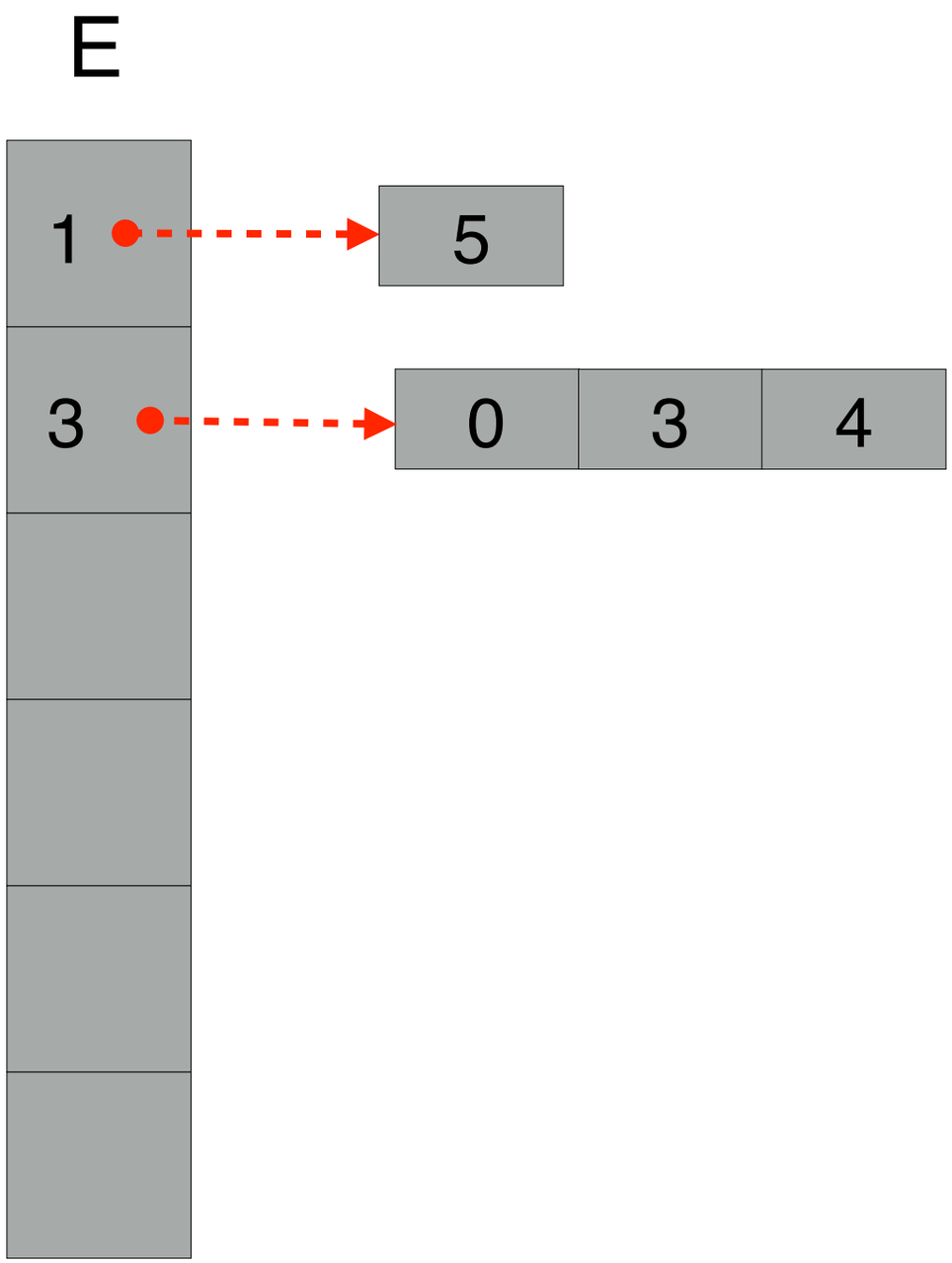

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



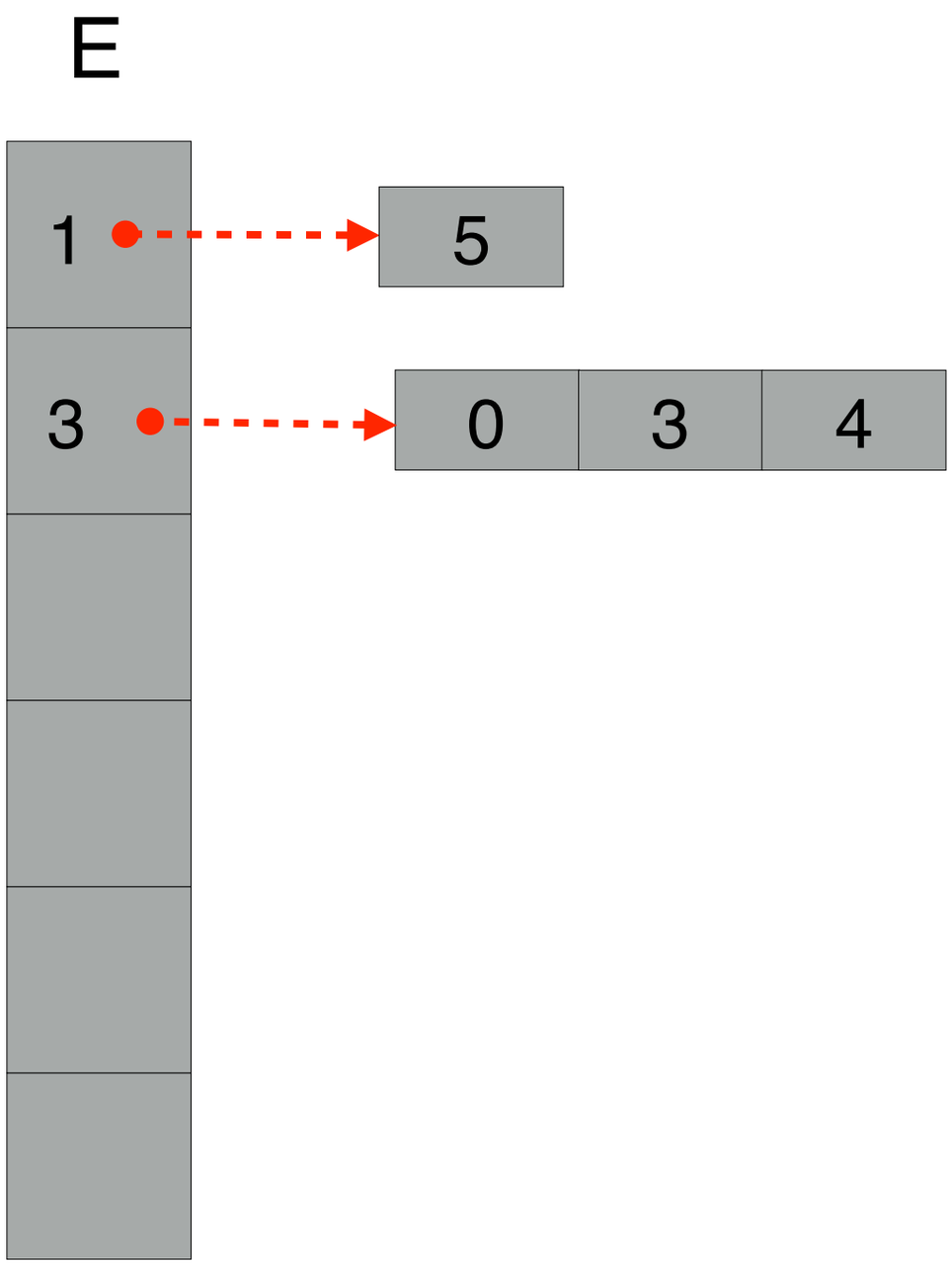

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



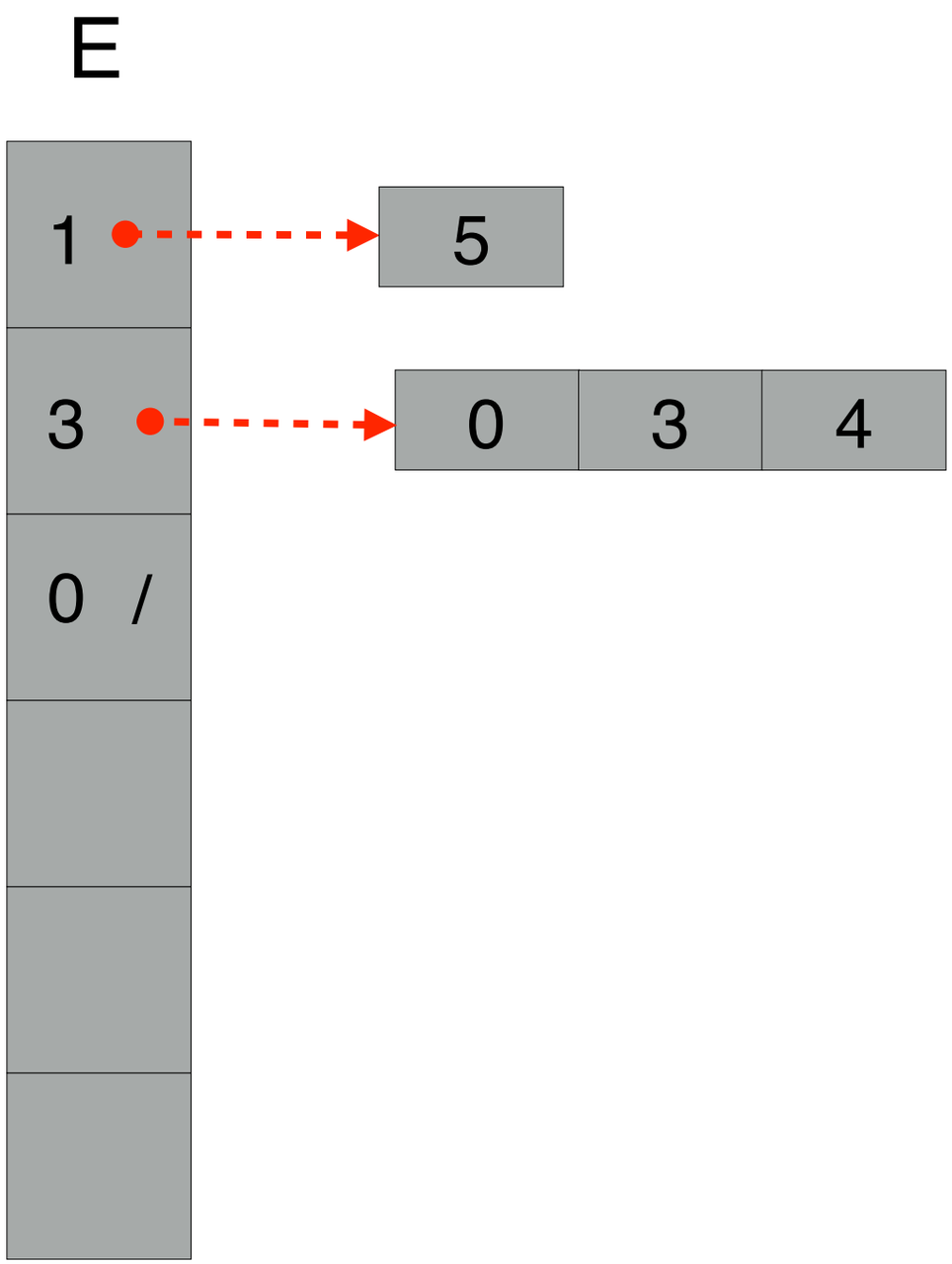

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



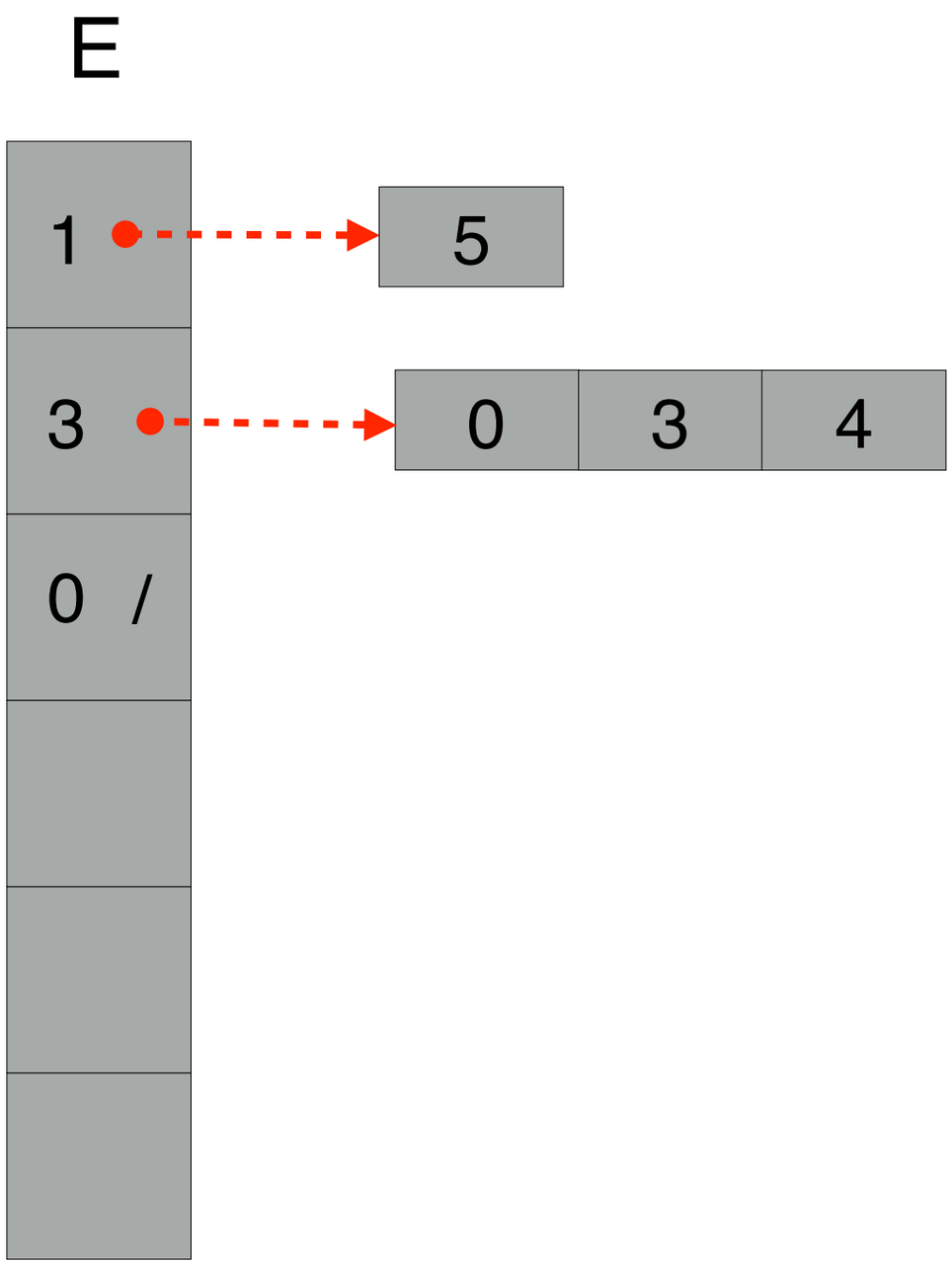

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



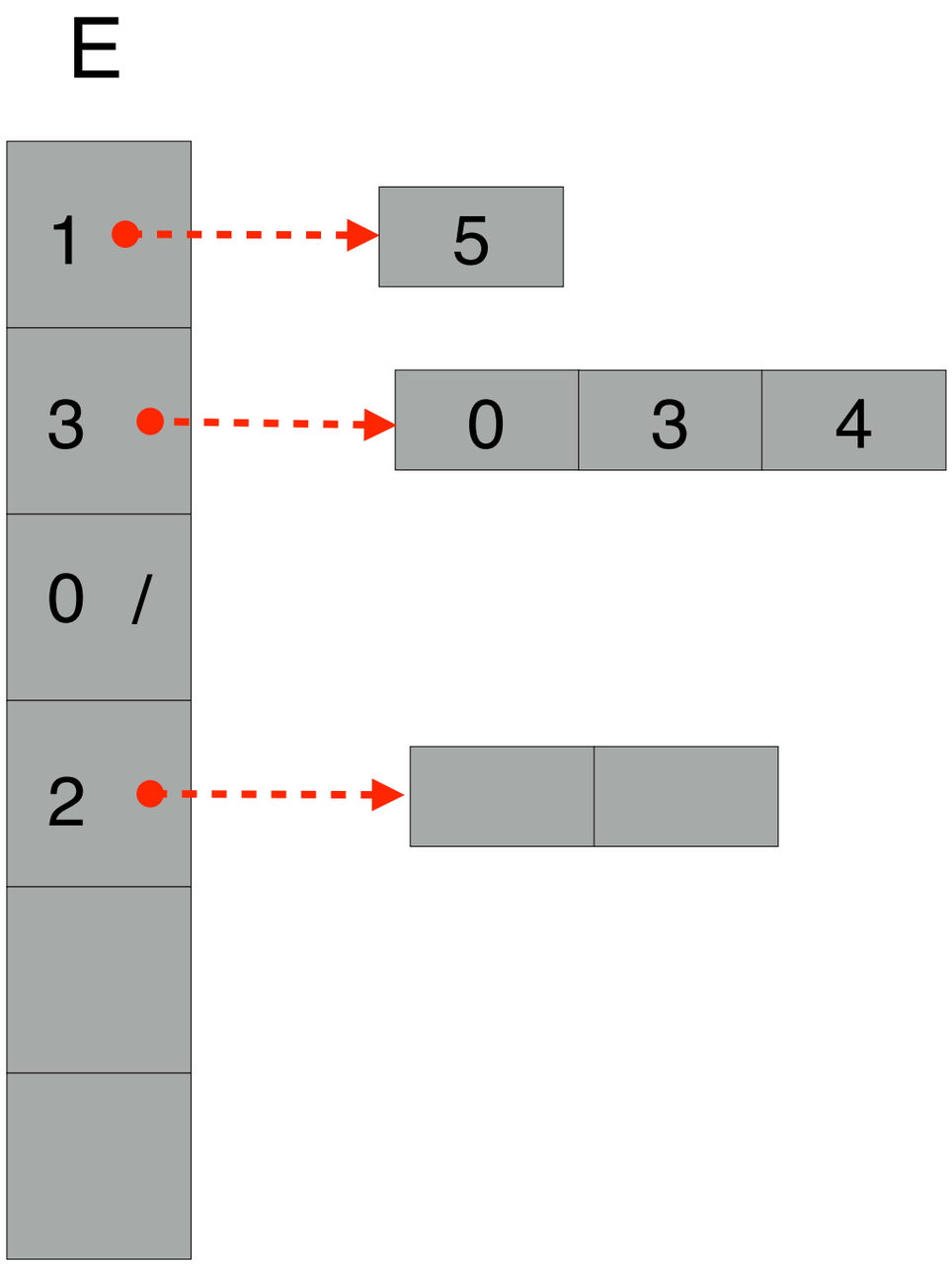

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



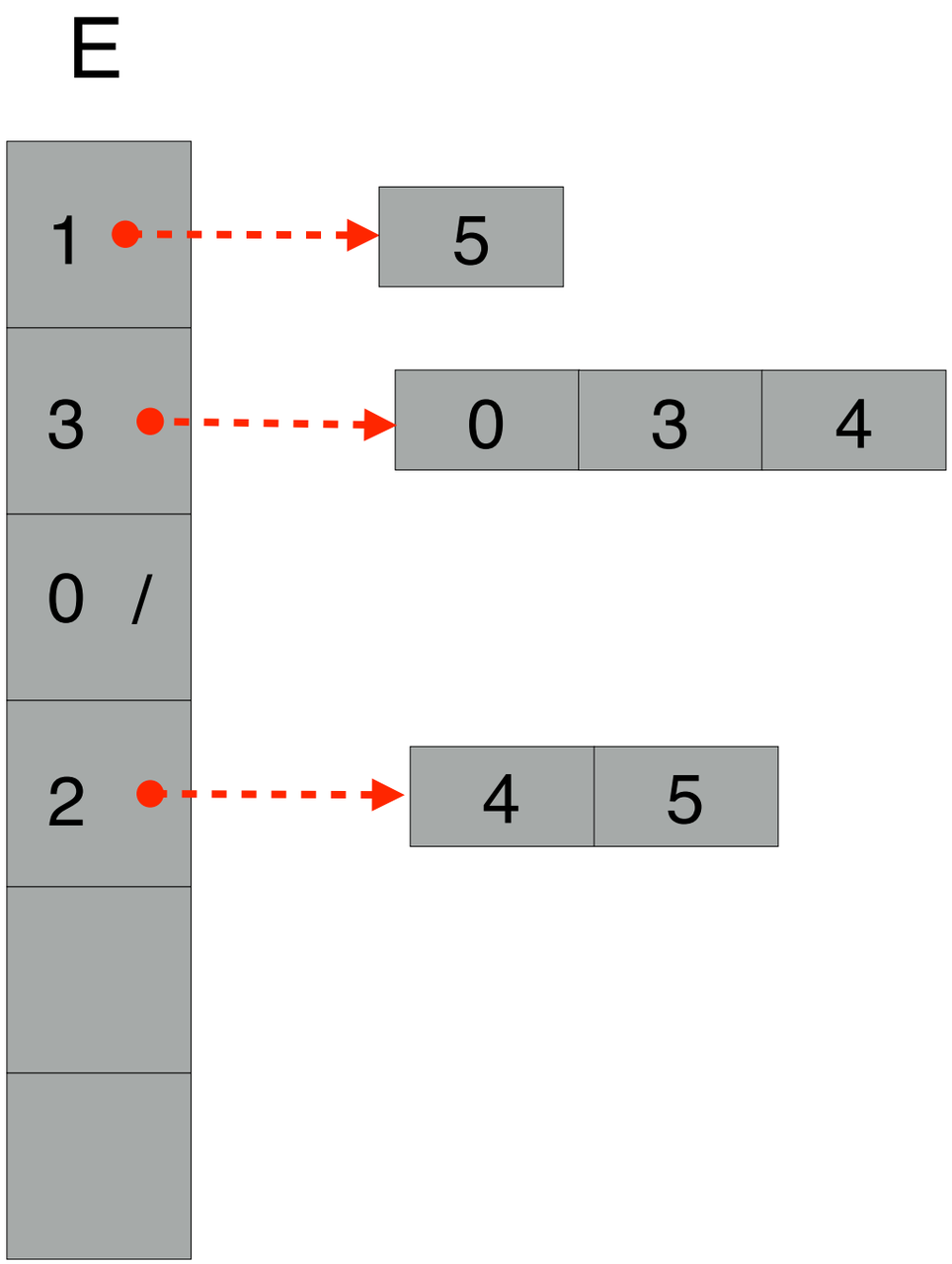

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



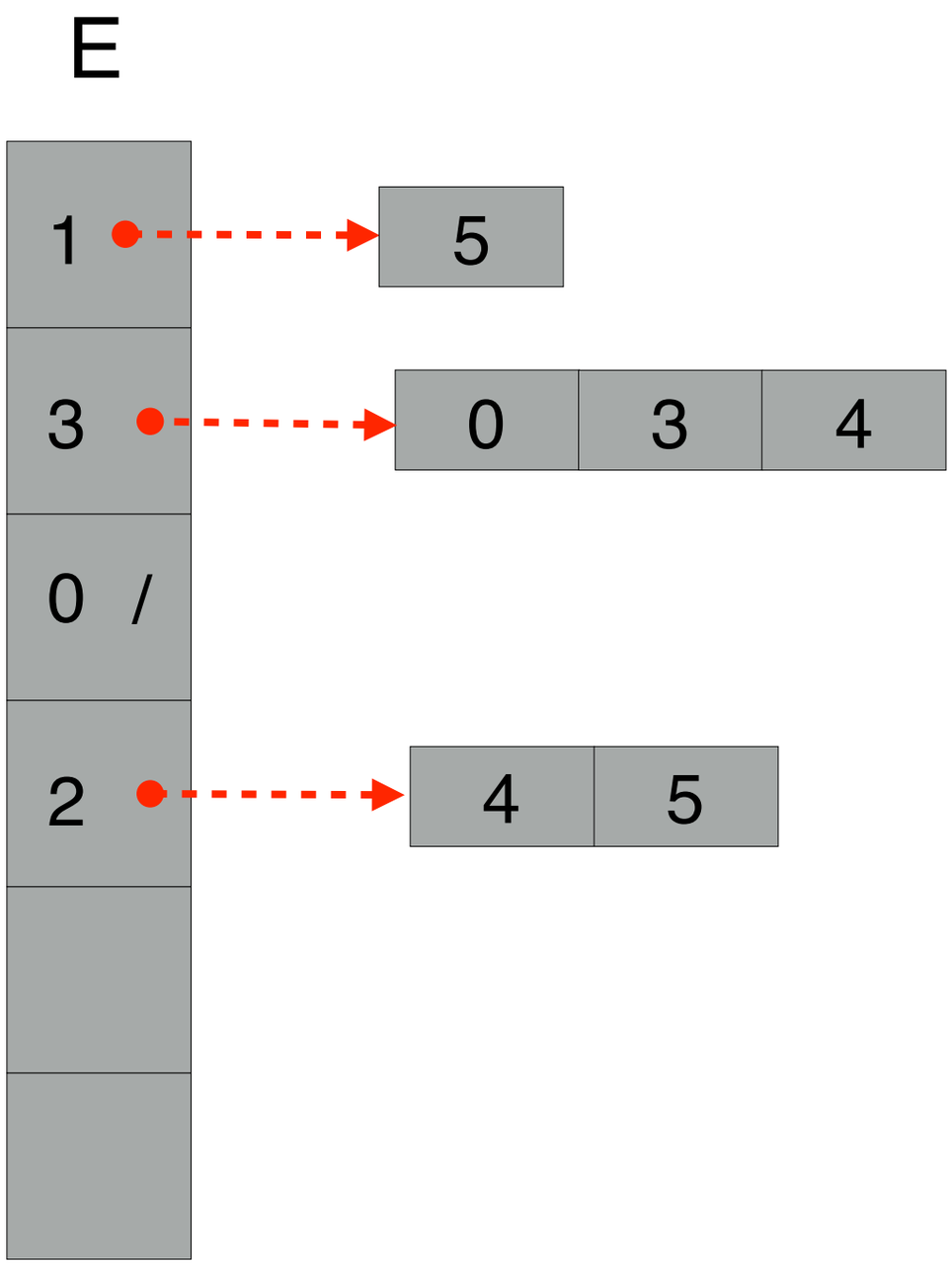

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



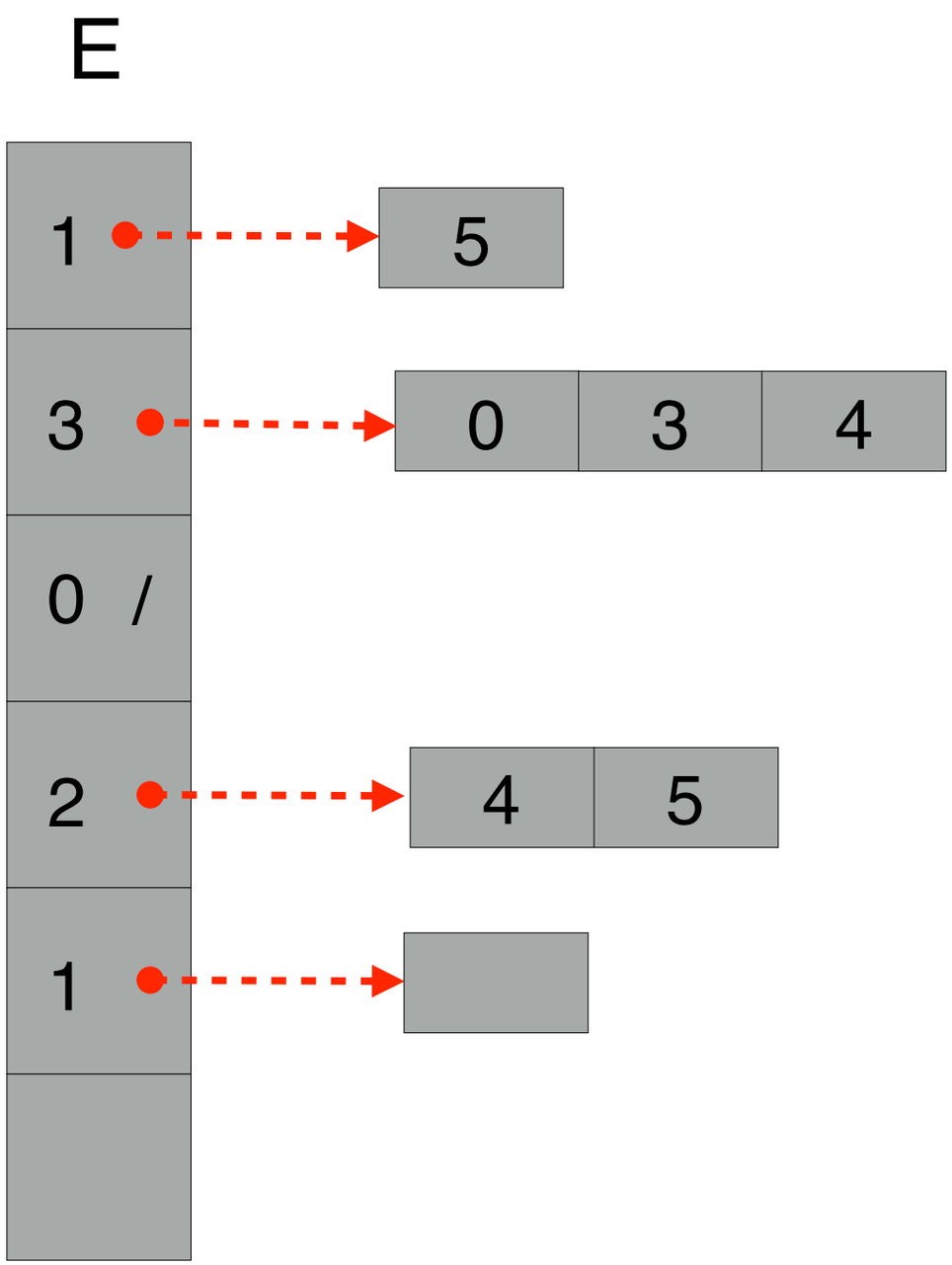

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



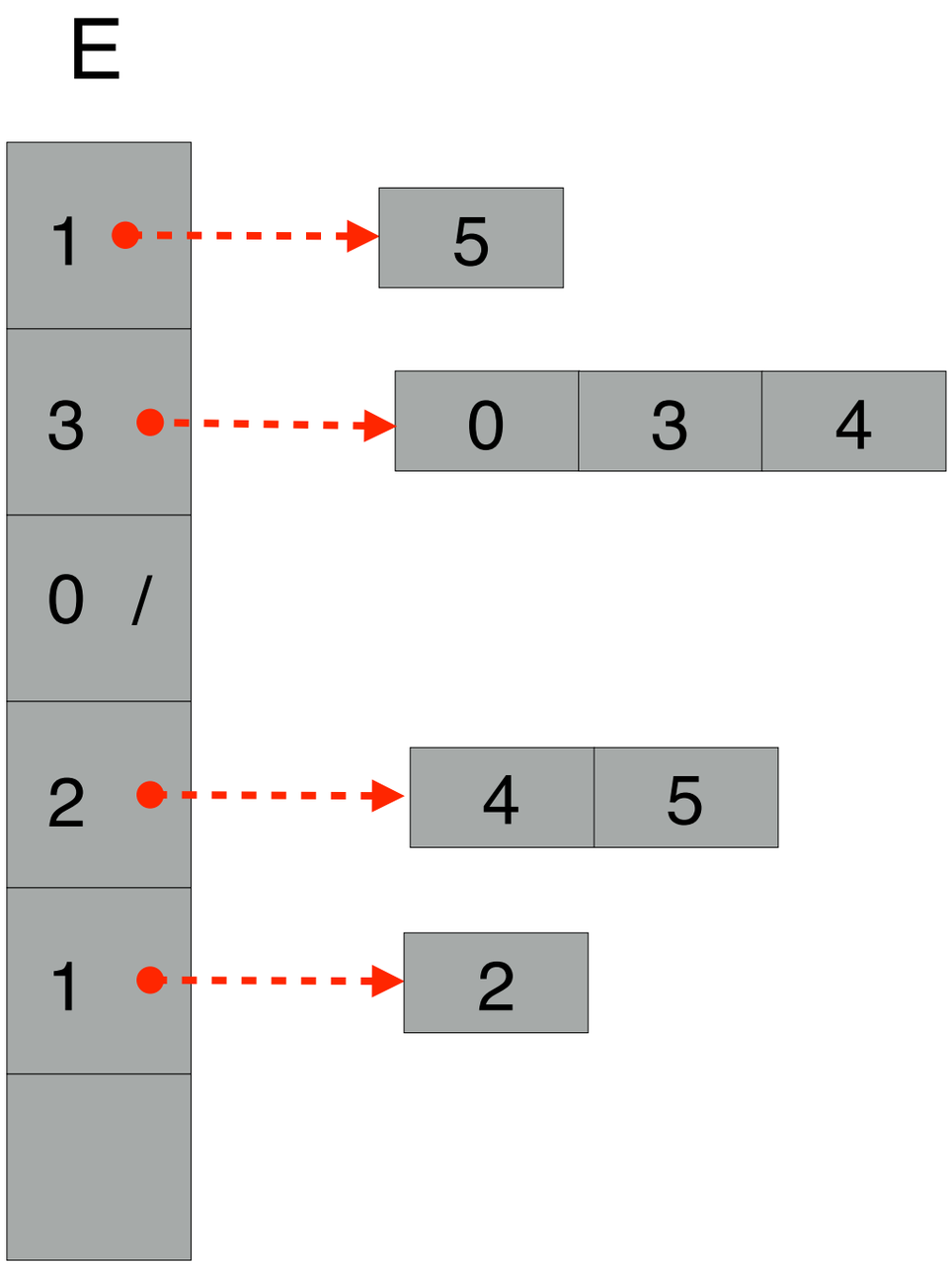

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



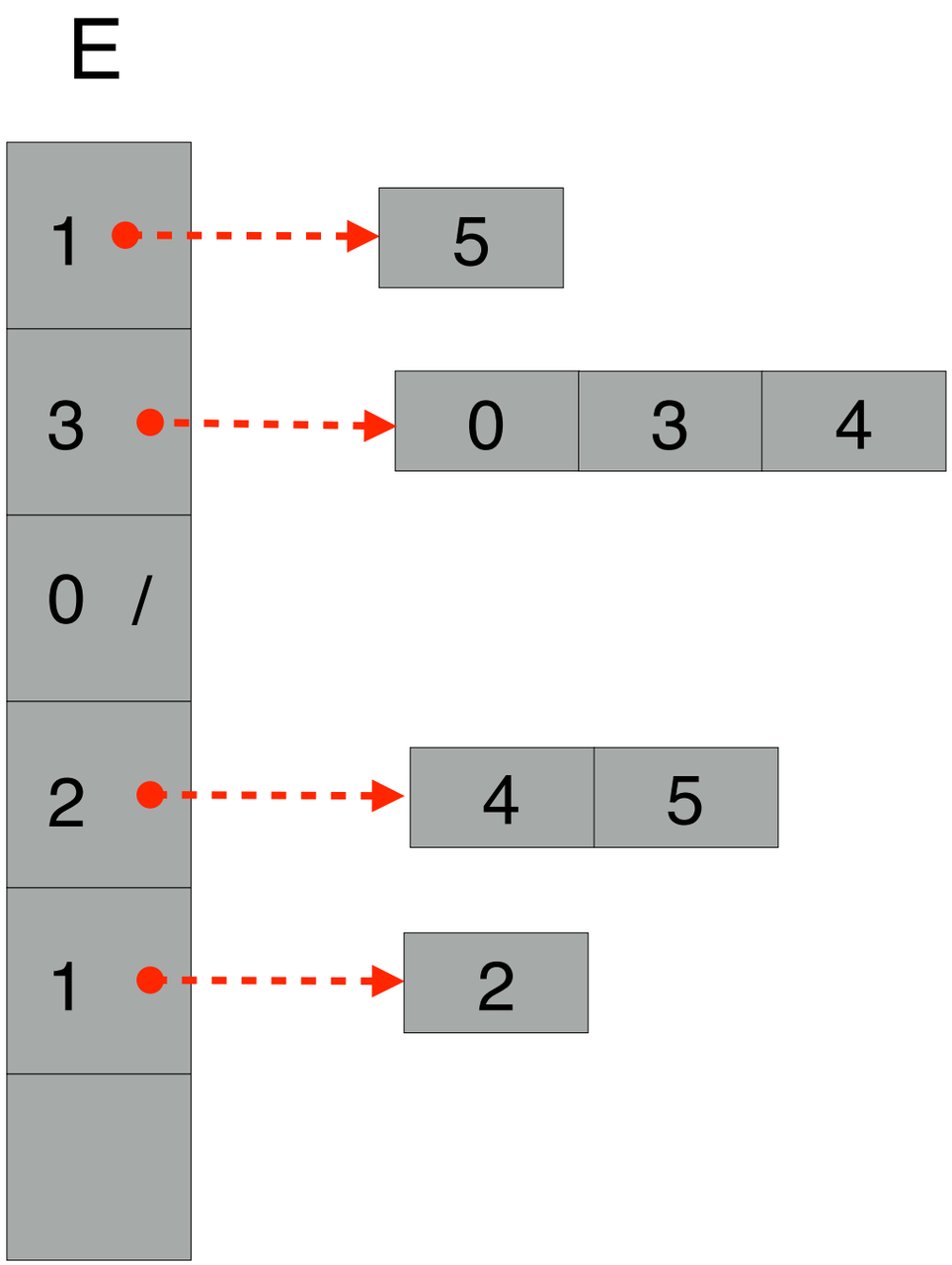

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



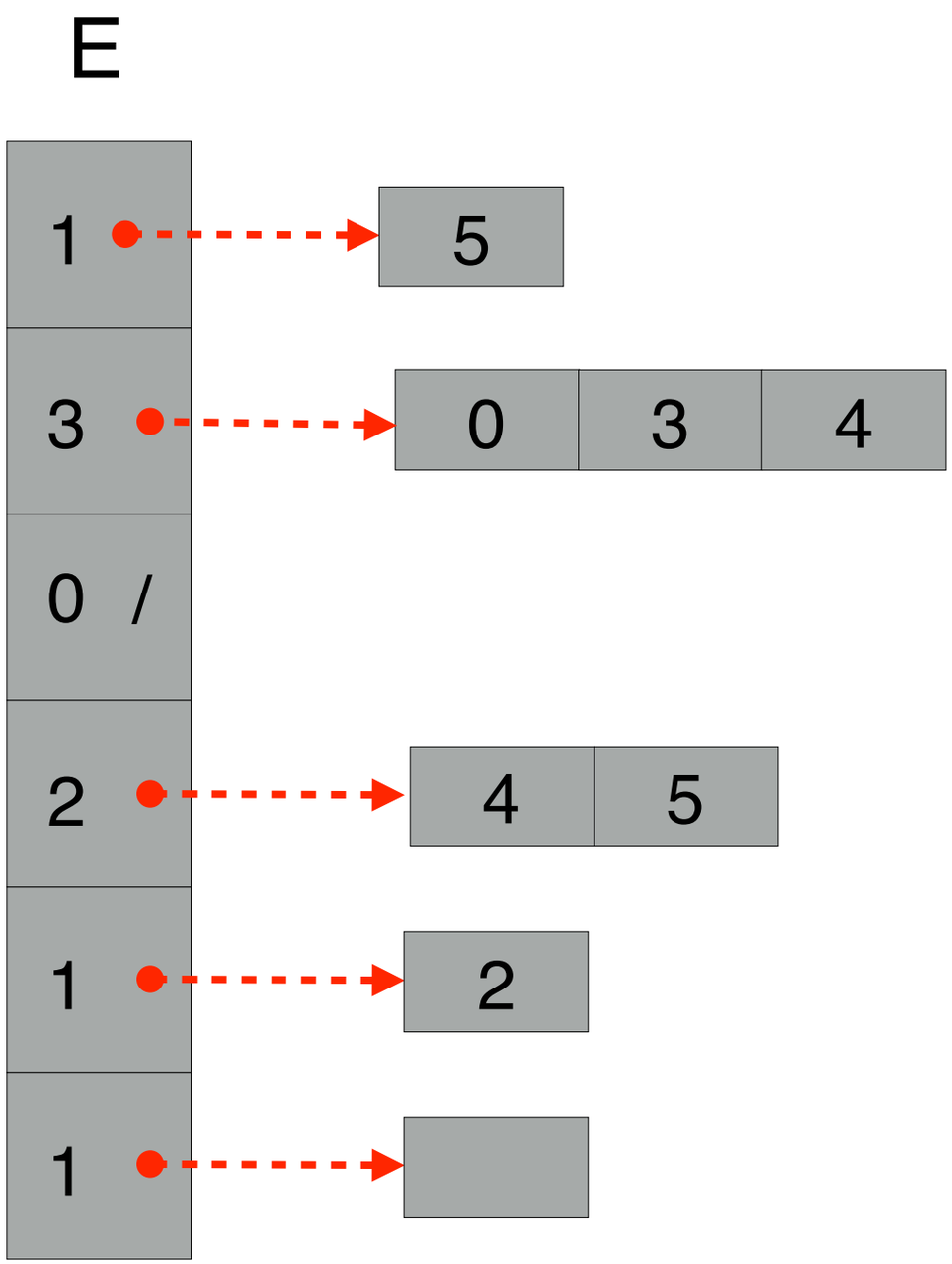

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input



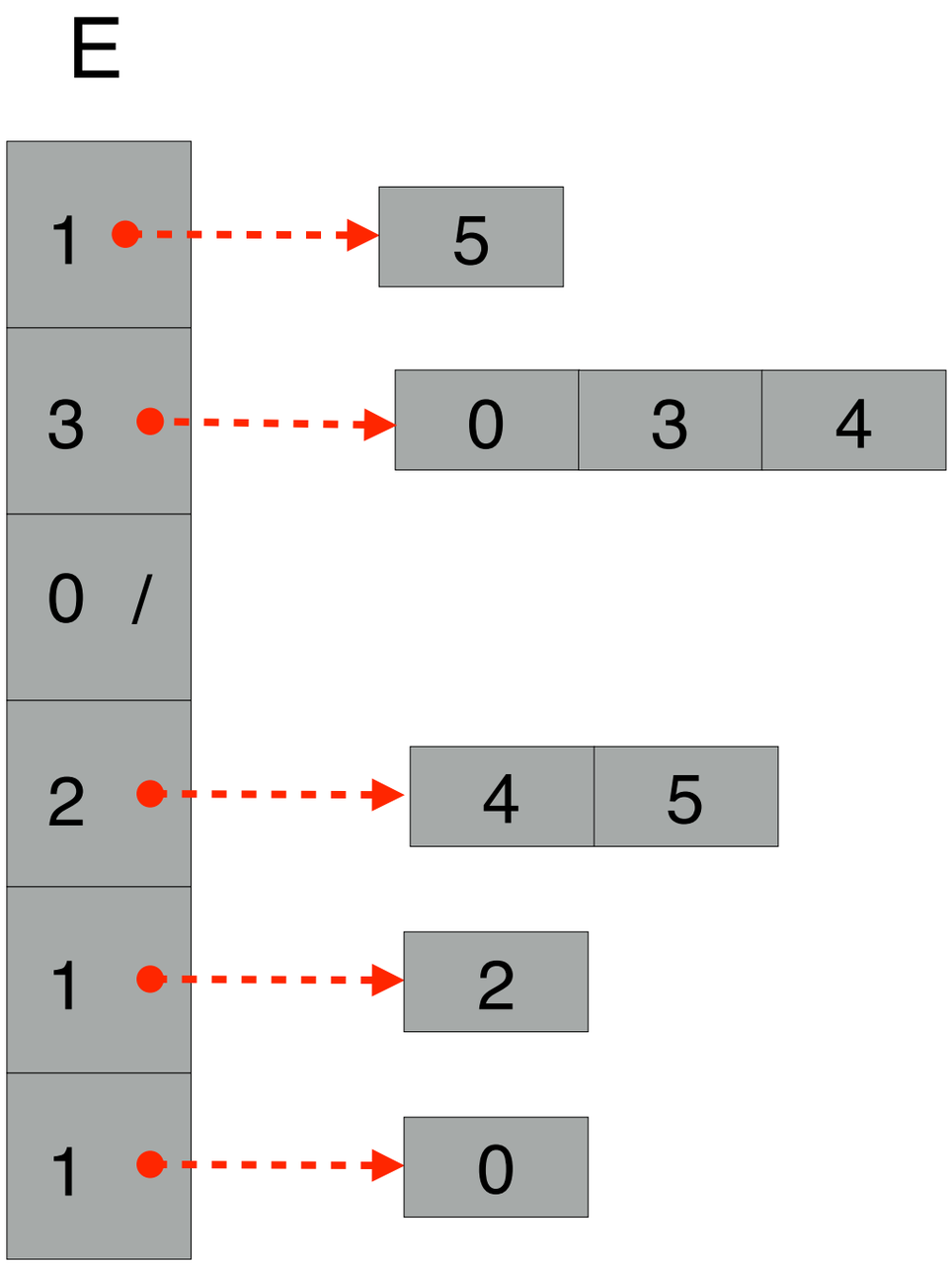

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input




Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:

```
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0
```

# Lettura grafo da input

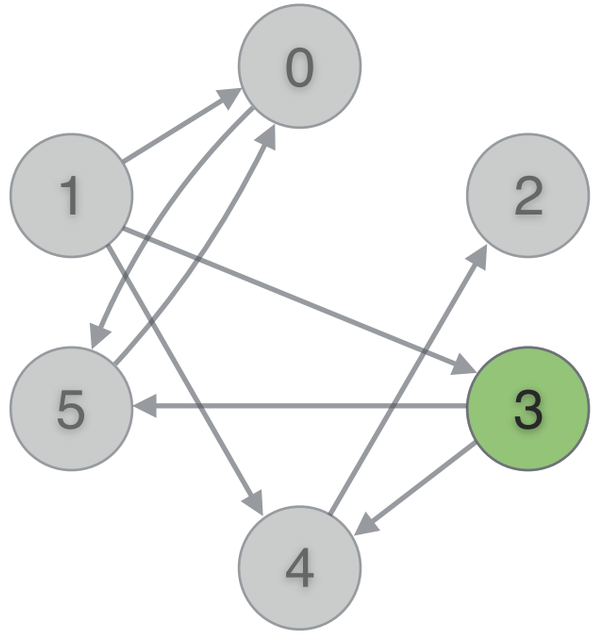
```
edges * read_graph() {
   edges * E;
   int n, i, j;

   scanf("%d", &n);
   E = (edges *) malloc(sizeof(edges) * n);
   for (i=0; i < n) {
      scanf("%d", &(E[i].num));
      E[i].edges = (int *) malloc(sizeof(int) *
      E[i].num);
      for (j=0; j < E[i].num; ++j) {
         scanf("%d", E[i].edges + j);
      }
   }
   return E;
}
```

Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

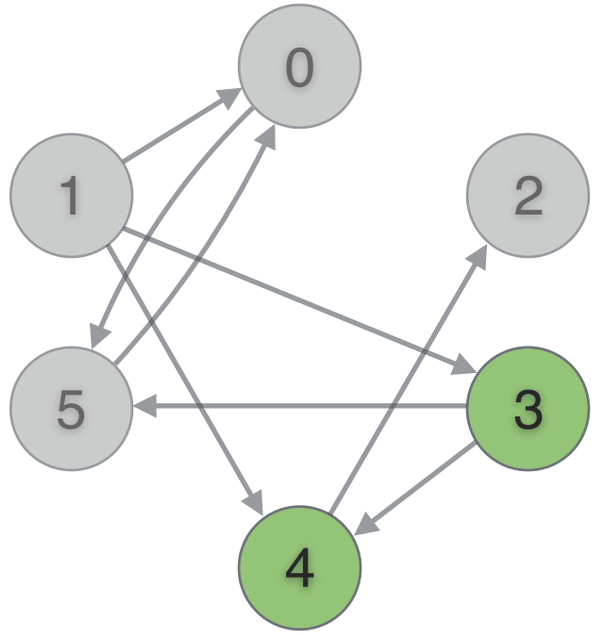
- una riga contenente il numero $n$ di nodi del grafo;
- $n$ righe, una per ciasun nodo $i$, con $i \in [0, n)$, nel seguente formato:
  - numero $n_i$ di archi uscenti da $i$;
  - lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Esempio:
6
1 5
3 0 3 4
0
2 4 5
1 2
1 0

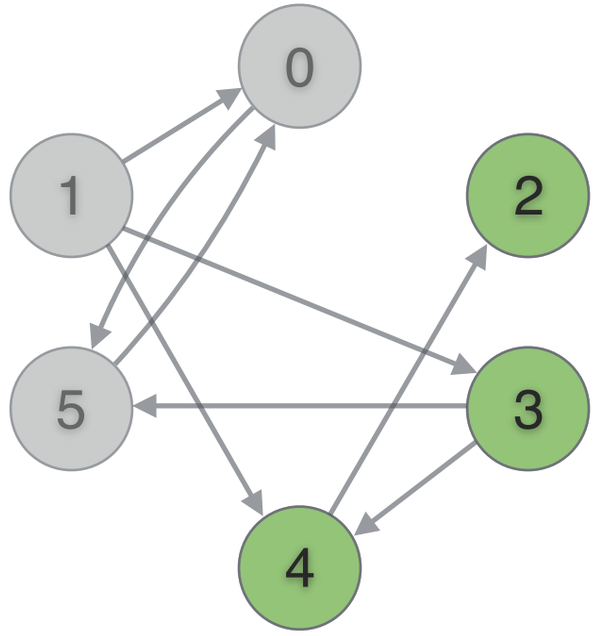
# Visita in profondità



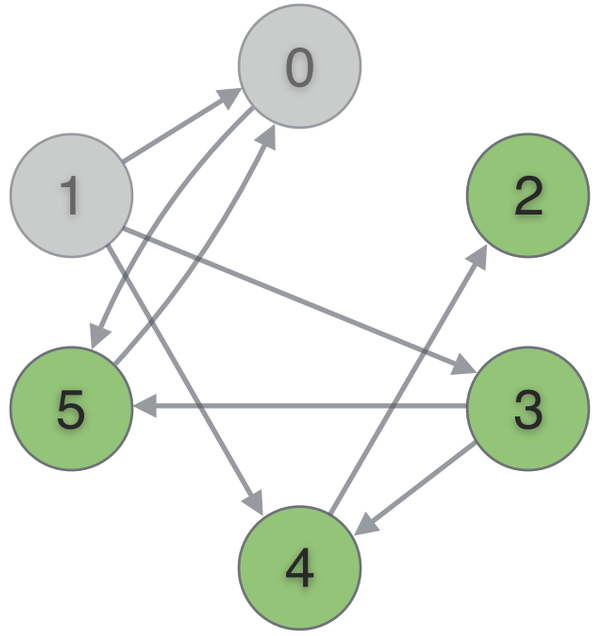
# Visita in profondità



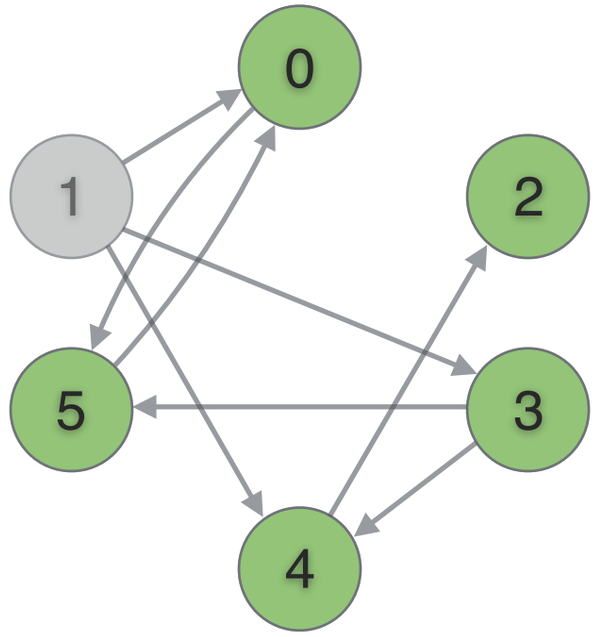
# Visita in profondità



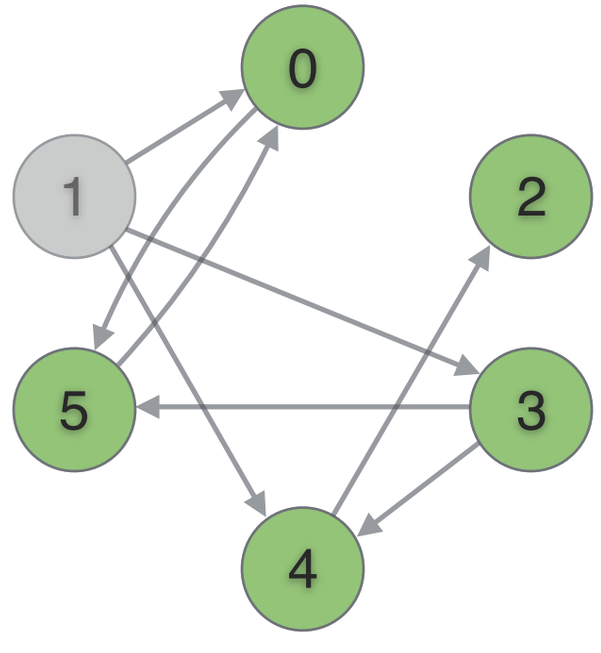
# Visita in profondità



# Visita in profondità



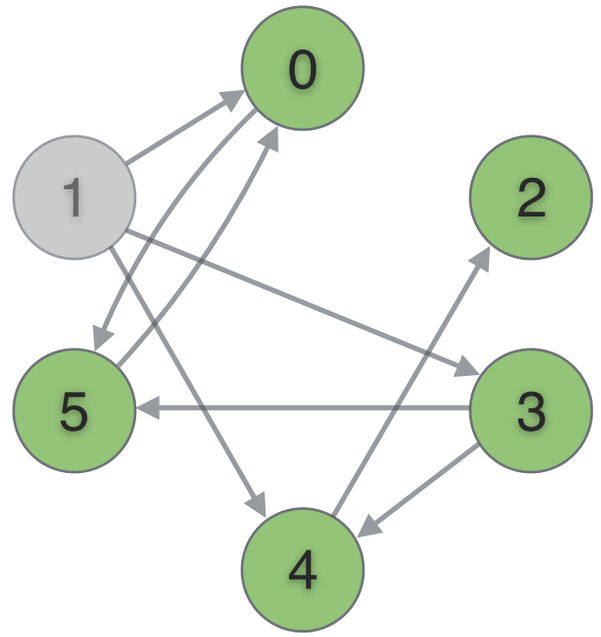
# Visita in profondità




```
void recursive_dfs(
      int src, edges *E, int *colors
) {
      int dest;
      for (int i=0; i < E[src].num; ++i) {
            dest = E[src].edges[i];
            if (!colors[dest]) {
                  colors[dest] = 1;
                  recursive_dfs(dest, E, colors);
            }
      }
 }
```

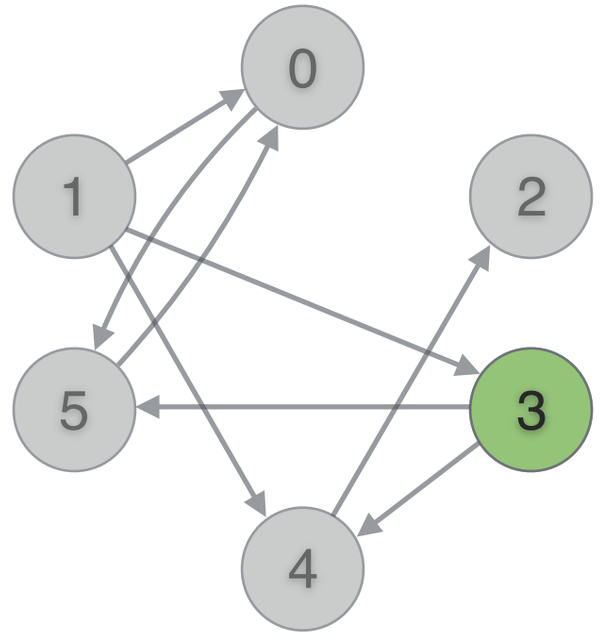
# Visita in profondità

```
int * dfs(int src, edges *E, int n) {
        int * colors = (int *) malloc(sizeof(int) * n);
        int * stack = (int *) malloc(sizeof(int) * n);
        int stack_size, src, dest, i;
        // inizializzo i colori
        for (int i=0; i < n; ++i) colors[i] = 0;
        colors[src] = 1;
        // inizializzo lo stack
        stack[0] = src; stack_size = 1;
        // loop fino a terminazione dello stack
        while (stack_size) {
                src = stack[--stack_size];
                for (i=0; i < E[src].num; ++i) {
                        dest = E[src].edges[i];
                        if (!colors[dest]) {
                                colors[dest] = 1;
                                stack[stack_size++] = dest;
                        }
                }
        }
        // libero la memoria
        free(stack);
        return colors;
}
```

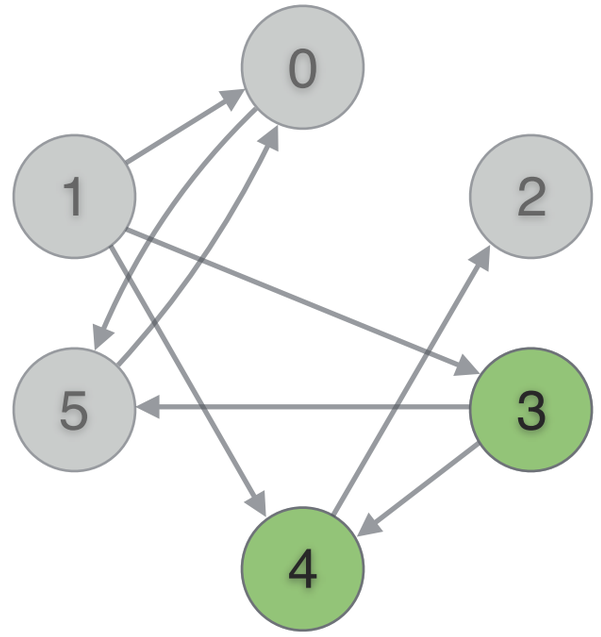



```
void recursive_dfs(
        int src, edges *E, int *colors
) {
        int dest;
        for (int i=0; i < E[src].num; ++i) {
                dest = E[src].edges[i];
                if (!colors[dest]) {
                        colors[dest] = 1;
                        recursive_dfs(dest, E, colors);
                }
        }
}
```

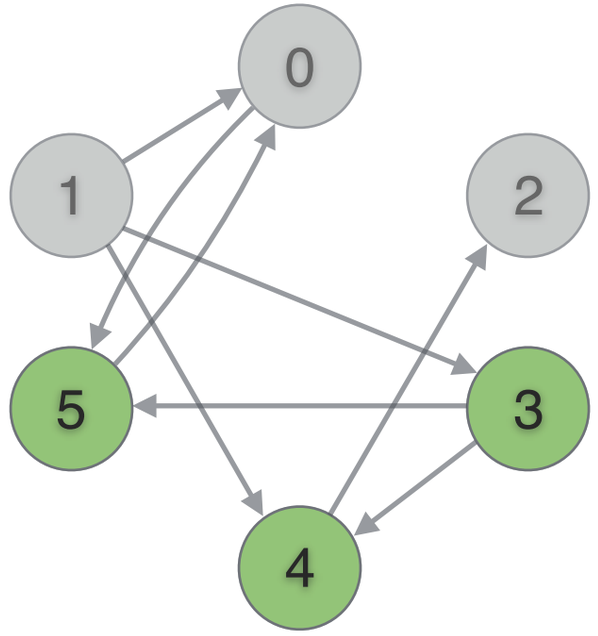
# Visita in ampiezza



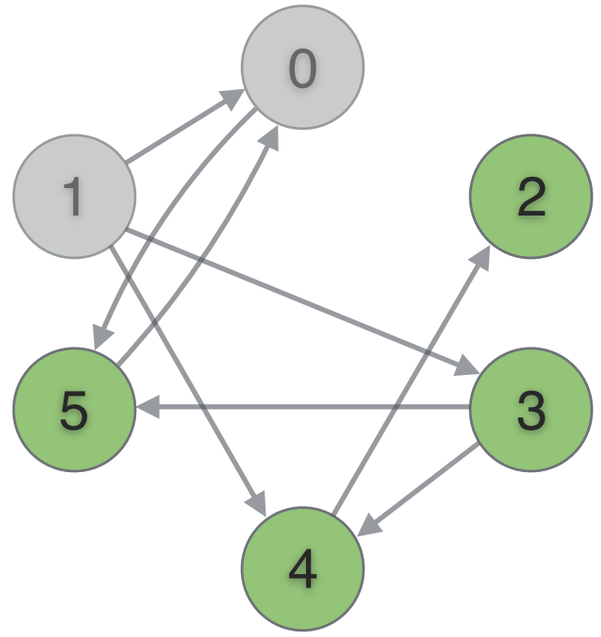
# Visita in ampiezza



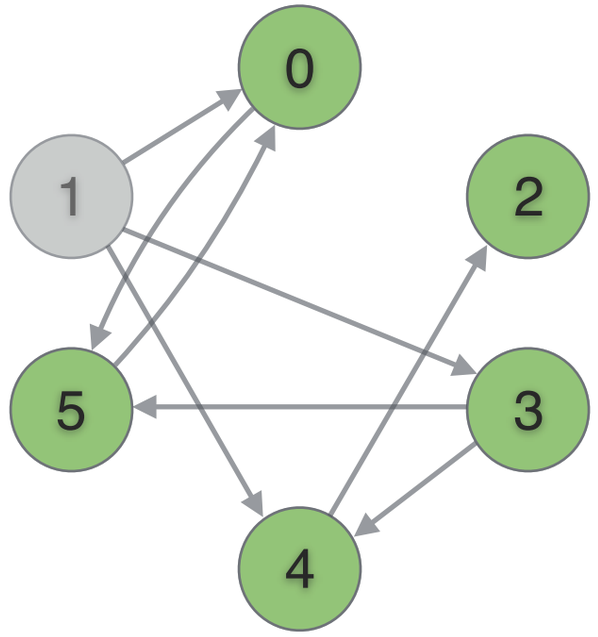
# Visita in ampiezza



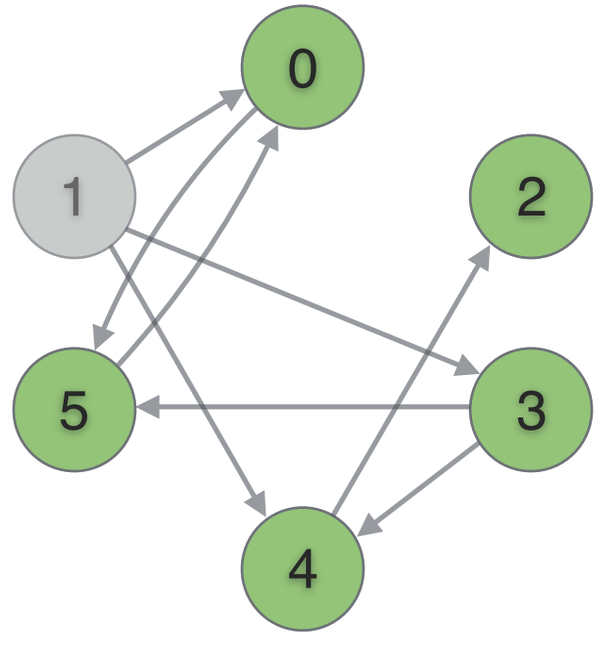
# Visita in ampiezza



# Visita in ampiezza



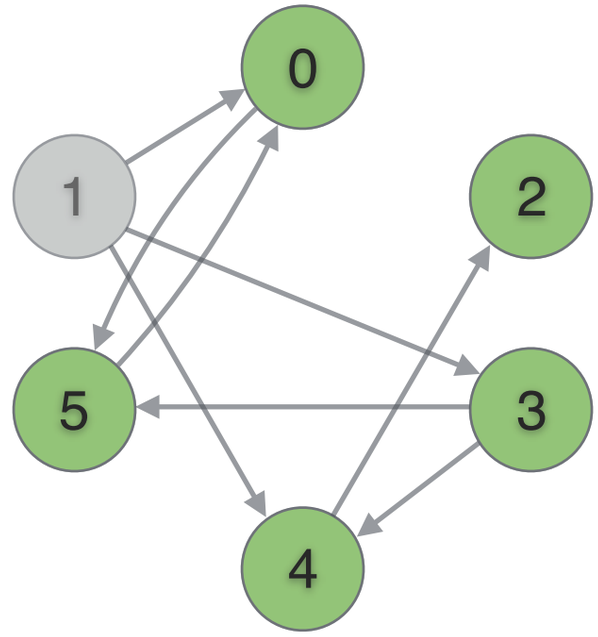
# Visita in ampiezza




```
typedef struct _queue {
  int * elements;
  int size;
  int head;
  int tail;
} queue;

void init_queue(queue * Q, int size);
void deinit_queue(queue * Q);
void enqueue(queue * Q, int element);
int dequeue(queue * Q);
```

# Visita in ampiezza

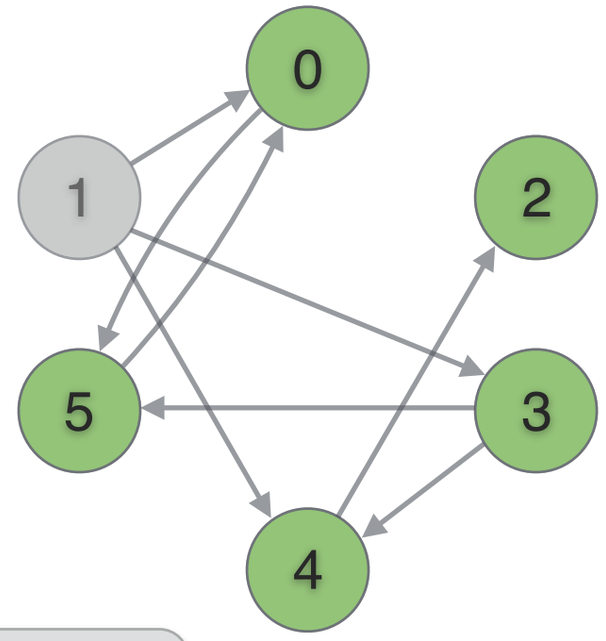
```
int * bfs(int src, edges *E, int n) {
      int * colors = (int *) malloc(sizeof(int) * n);
      queue q;
      int src, dest, i;
      // inizializzo i colori
      for (int i=0; i < n; ++i) colors[i] = 0;
      colors[src] = 1;
      // inizializzo la coda
      init_queue(&q, n); enqueue(&q, src);
      // loop fino a terminazione della coda
      while (queue.size) {
            src = dequeue(&q);
            for (i=0; i < E[src].num; ++i) {
                  dest = E[src].edges[i];
                  if (!colors[dest]) {
                        colors[dest] = 1;
                        enqueue(&q, dest);
                  }
            }
      }
       // libero la memoria
      deinit_queue(&q);
      return colors;
}
```




```
typedef struct _queue {
  int * elements;
  int size;
  int head;
  int tail;
} queue;

void init_queue(queue * Q, int size);
void deinit_queue(queue * Q);
void enqueue(queue * Q, int element);
int dequeue(queue * Q);
```

# Visita in ampiezza

```
int * bfs(int src, edges *E, int n) {
      int * colors = (int *) malloc(sizeof(int) * n);
      queue q;
      int src, dest, i;
      // inizializzo i colori
      for (int i=0; i < n; ++i) colors[i] = 0;
      colors[src] = 1;
      // inizializzo la coda
      init_queue(&q, n); enqueue(&q, src);
      // loop fino a terminazione della coda
      while (queue.size) {
            src = dequeue(&q);
            for (i=0; i < E[src].num; ++i) {
                  dest = E[src].edges[i];
                  if (!colors[dest]) {
                        colors[dest] = 1;
                        enqueue(&q, dest);
                  }
            }
      }
      // libero la memoria
      deinit_queue(&q);
      return colors;
}
```




Da implementare…

```
typedef struct _queue {
  int * elements;
  int size;
  int head;
  int tail;
} queue;

void init_queue(queue * Q, int size);
void deinit_queue(queue * Q);
void enqueue(queue * Q, int element);
int dequeue(queue * Q);
```

# Esercizio 1

## Grafo bipartito

Scrivere un programma che legga da tastiera un grafo indiretto e stampi `1` se il grafo è bipartito, `0` altrimenti. Il grafo è rappresentato nel seguente formato: la prima riga contiene il numero $n$ di nodi, le successive $n$ righe contengono, per ciascun nodo $i$, con $0 \leq i < n$, il numero $n_i$ di archi uscenti da $i$ seguito da una lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$. Si assuma che l'input contenga un grafo indiretto, e quindi che per ciascun arco da $i$ a $j$ esista anche l'arco da $j$ ad $i$.

Un grafo bipartito è un grafo tale che l'insieme dei suoi vertici si può partizionare in due sottoinsiemi in cui ogni vertice è collegato solo a vertici appartenenti alla partizione opposta.

Suggerimento: un grafo è bipartito se e solo se è possibile colorarlo usando due colori. Colorare il grafo corrisponde ad assegnare a ciascun vertice un colore diverso da quello dei suoi vertici adiacenti.

# Esercizio 2

## Grafo connesso

Scrivere un programma che legga da tastiera un grafo indiretto e stampi `1` se il grafo è connesso, `0` altrimenti. Il grafo è rappresentato nel seguente formato: la prima riga contiene il numero $n$ di nodi, le successive $n$ righe contengono, per ciascun nodo $i$, con $0 \leq i < n$, il numero $n_i$ di archi uscenti da $i$ seguito da una lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$. Si assuma che l'input contenga un grafo indiretto, e quindi che per ciascun arco da $i$ a $j$ esiste anche l'arco da $j$ ad $i$.

Un grafo è connesso quando esiste un percorso tra due vertici qualunque del grafo. Il programma deve eseguire una visita DFS (a partire da un nodo qualunque, perché?) del grafo per stabilire se questo è connesso.

# Esercizio 3

## Percorso minimo

Scrivere un programma che legga da tastiera un grafo diretto, una sequenza di $m$ query composte da due indici ciascuna e stampi, per ciascuna query, la lunghezza del percorso minimo che collega i rispettivi due nodi della query. Il grafo è rappresentato nel seguente formato: la prima riga contiene il numero $n$ di nodi, le successive $n$ righe contengono, per ciascun nodo $i$, con $0 \leq i < n$, il numero $n_i$ di archi uscenti da $i$ seguito da una lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Il percorso minimo dal nodo $i$ al nodo $j$ è il percorso che porta da $i$ a $j$ avente il minor numero di nodi. A tale scopo si esegua una visita BFS del grafo a partire dal nodo $i$ per stabilire il percorso minimo che porta al nodo $j$, qualora questo esista.

# Esercizio 4

## Diametro grafo

Scrivere un programma che legga da tastiera un grafo diretto e stampi il diametro del grafo. Il grafo è rappresentato nel seguente formato: la prima riga contiene il numero $n$ di nodi, le successive $n$ righe contengono, per ciascun nodo $i$, con $0 \leq i < n$, il numero $n_i$ di archi uscenti da $i$ seguito da una lista di $n_i$ nodi destinazione, rappresentati con i numeri $[0, n)$.

Il diametro di un grafo è la lunghezza del "più lungo cammino minimo" fra tutte le coppie di nodi. Il programma deve eseguire una visita BFS a partire da ciascun nodo $i$ del grafo per stabilire il cammino minimo più lungo a partire da $i$, e quindi stampare il massimo tra tutti questi.

# Puzzle

## Formiche in riga

Una colonia di $n$ formiche è disposta in linea retta su una corda lunga esattamente $n$ segmenti. Le formiche possono muoversi solo in orizzontale, in entrambi i versi, ma non possono passare una sopra l'altra. Le formiche una volta decisa una direzione non la cambiano più fino a che non si scontrano con un'altra formica che andava in direzione opposta. Le formiche si muovono tutte insieme a step regolari, di un segmento alla volta nelle rispettive direzioni. Quando due formiche si scontrano cambiano direzione e tornano al segmento che occupavano (ma questa volta andando in direzione opposta). Quando una formica fa un passo oltre il primo o l'ultimo dei segmenti cade dalla corda.

Inizialmente ogni formica occupa un segmento distinto della corda, ma con una direzione iniziale a noi sconosciuta. Individuare il numero di step necessari affinché al caso pessimo tutte le formiche cadano dalla corda.