

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 2

ABR: Visita

Scrivere un programma che legga da tastiera una sequenza di N interi distinti e li inserisca in un albero binario di ricerca (senza ribilanciamento). Il programma deve visitare opportunamente l'albero e restituire la sua altezza.

Esercizio 2

```
typedef struct _node {  
    struct _node *left;  
    struct _node *right;  
    int value;  
} node;
```

```
int maxdepth(node *n) {  
    if (n == NULL) return 0;  
    return 1 + max( maxdepth(n->left), maxdepth(n->right) );  
}
```

Esercizio 5

Albero Ternario (prova del 01/02/2012)

Scrivere un programma che riceva in input una sequenza di N interi positivi e costruisca un albero **ternario** di ricerca **non** bilanciato. L'ordine di inserimento dei valori nell'albero deve coincidere con quello della sequenza.

Ogni nodo in un albero ternario di ricerca può avere fino a tre figli: figlio sinistro, figlio centrale e figlio destro. L'inserimento di un nuovo valore avviene partendo dalla radice dell'albero e utilizzando la seguente regola. Il valore da inserire viene confrontato con la chiave del nodo corrente. Ci sono tre possibili casi in base al risultato del confronto:

1. se il valore è minore della chiave del nodo corrente, esso viene inserito ricorsivamente nel sottoalbero radicato nel figlio sinistro;
2. se il valore è **divisibile** per la chiave del nodo corrente, esso viene inserito ricorsivamente nel sottoalbero radicato nel figlio centrale;
3. in ogni altro caso il valore viene inserito ricorsivamente nel sottoalbero radicato nel figlio destro.

Il programma deve stampare il numero di nodi dell'albero che hanno **tre** figli.

Esercizio 5

```
typedef struct __Nodo {  
    int key;  
    struct __Nodo* left;  
    struct __Nodo* central;  
    struct __Nodo* right;  
} Nodo;
```

Esercizio 5

```
void insert(Nodo **t, int k) {
    Nodo *e, *p, *x;

    /* crea il nodo foglia da inserire contenente la chiave */
    e = (Nodo *) malloc(sizeof(Nodo));
    if (e == NULL) exit(-1);
    e->key = k;
    e->left = e->right = e->central = NULL;
    x = *t;

    /* se l'albero è vuoto imposta e come radice dell'albero */
    if (x == NULL) {
        *t = e;
        return;
    }

    continua ...
}
```

Esercizio 5

```
/* altrimenti cerca la posizione della foglia nell'albero */  
while (x != NULL) {  
    p = x;  
    if(k % x->key == 0) x = x->central;  
    else {  
        if (k < x->key) x = x->left;  
        else x = x->right;  
    }  
}
```

```
/* ora p punta al padre del nuovo elemento da inserire in t quindi si procede a  
collegare p ed e */  
if(k % p->key == 0) p->central = e;  
else {  
    if (k < p->key) p->left = e;  
    else p->right = e;  
}
```

Esercizio 5

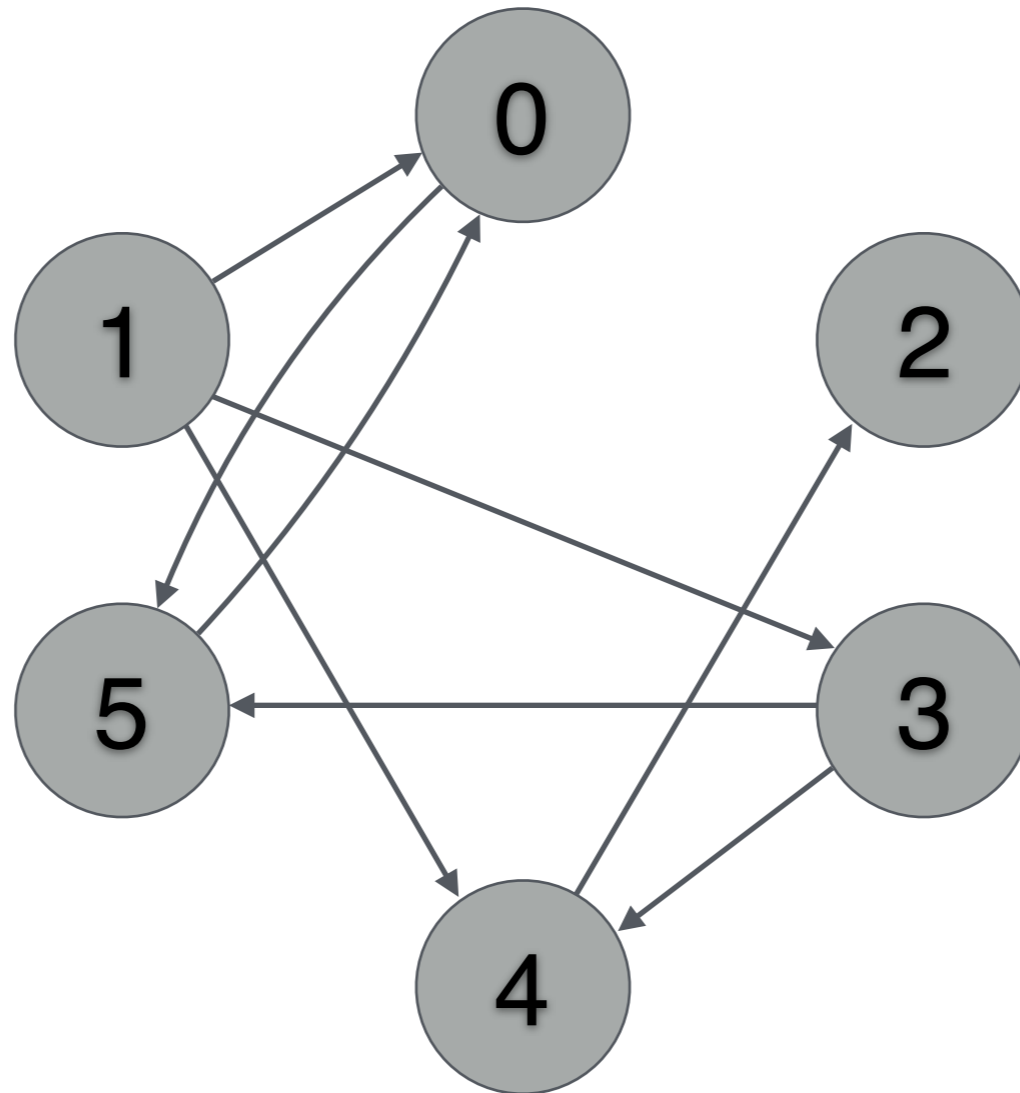
```
int conta(Nodo * node) {
    int r, c, l, curr;

    if (node == NULL) return 0;
    r = c = l = curr = 0;

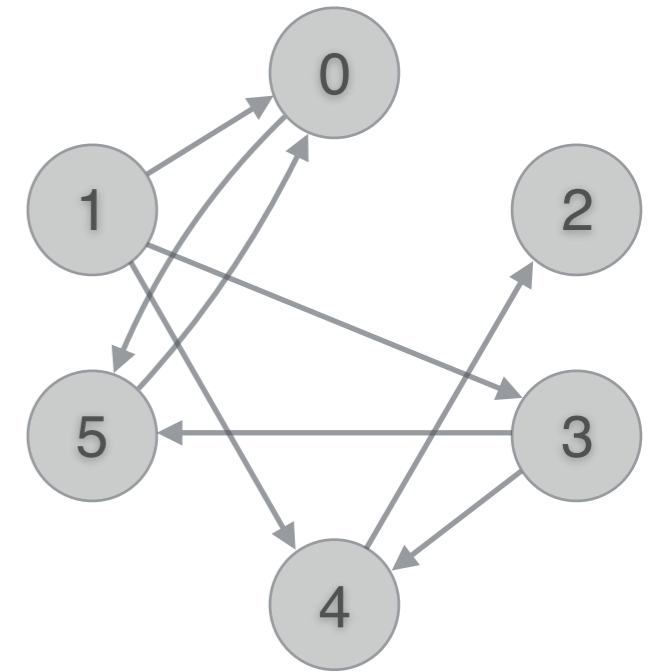
    if (node->right != NULL) { r = conta(node->right); curr++;}
    if (node->left != NULL) { l = conta(node->left); curr++;}
    if (node->central != NULL) { c = conta(node->central); curr++;}

    if (curr == 3) return r+l+c+1;
    else return r+l+c;
}
```


Grafi



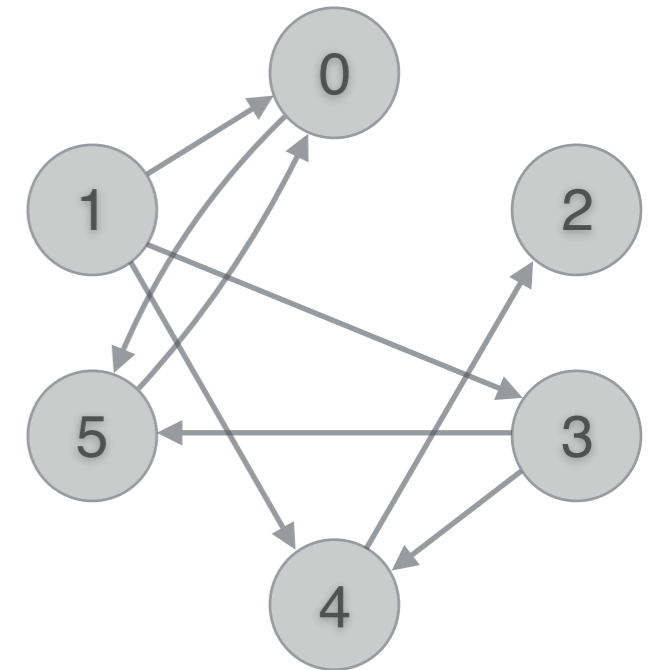
Matrice di adiacenza



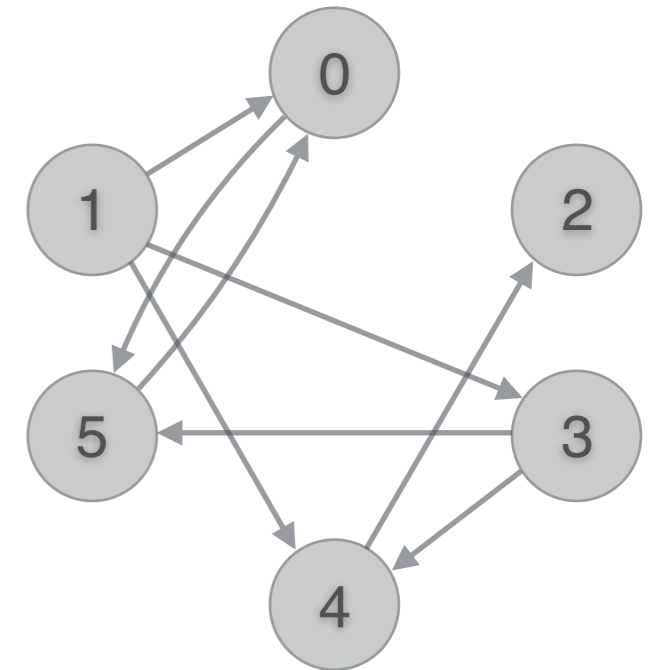
Matrice di adiacenza

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

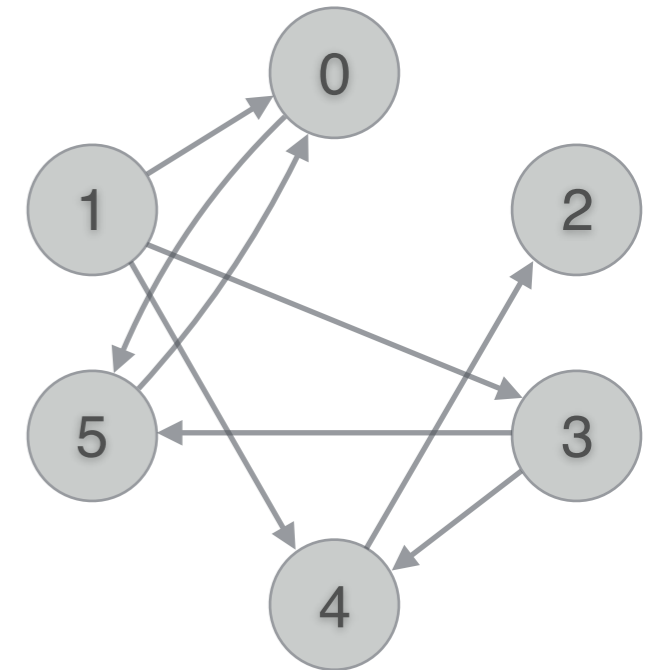


M

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	0
1	0	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



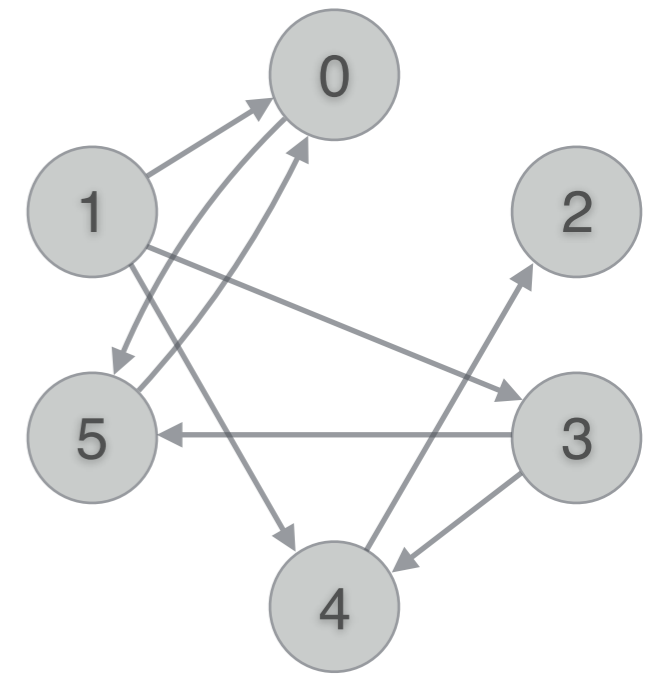
M

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	0
1	0	0	0	0	0

Troppi zeri...

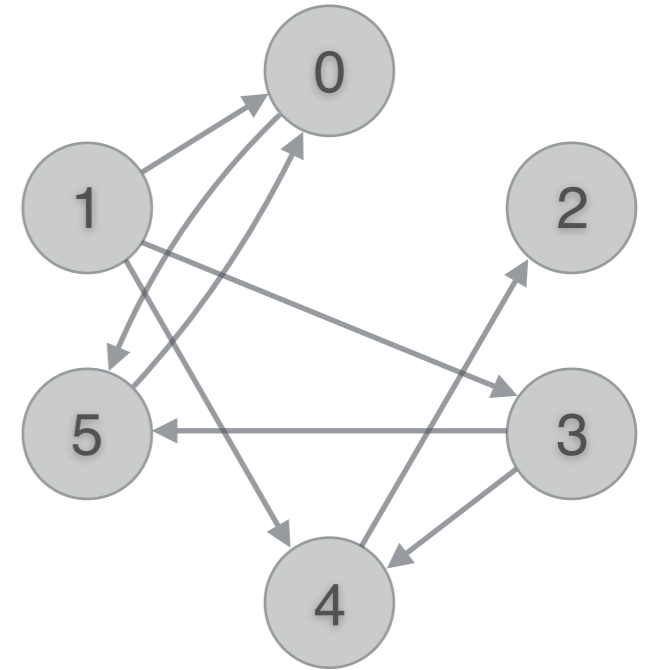
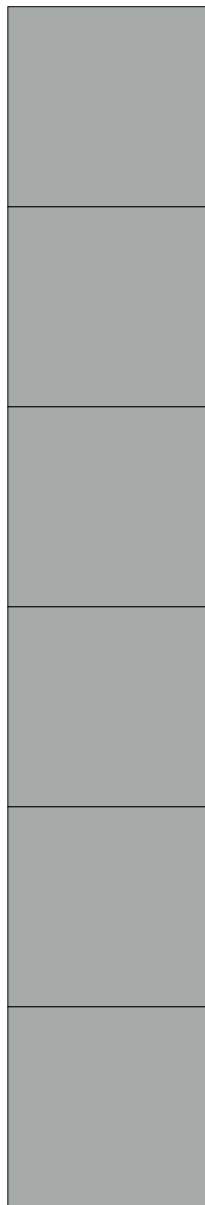
```
int ** M = (int **) malloc(N*sizeof(int *));
for (int i=0; i<N; ++i) {
    M[i] = (int *) malloc(N*sizeof(int));
}
```

Liste di adiacenza

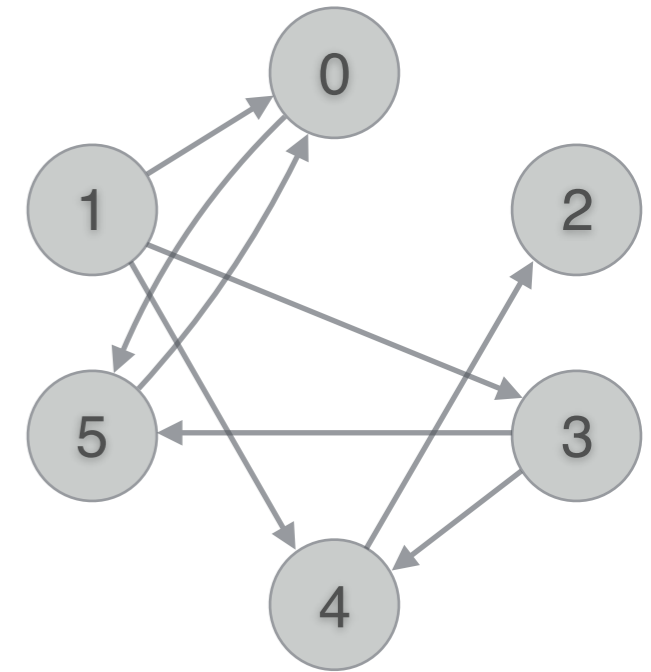
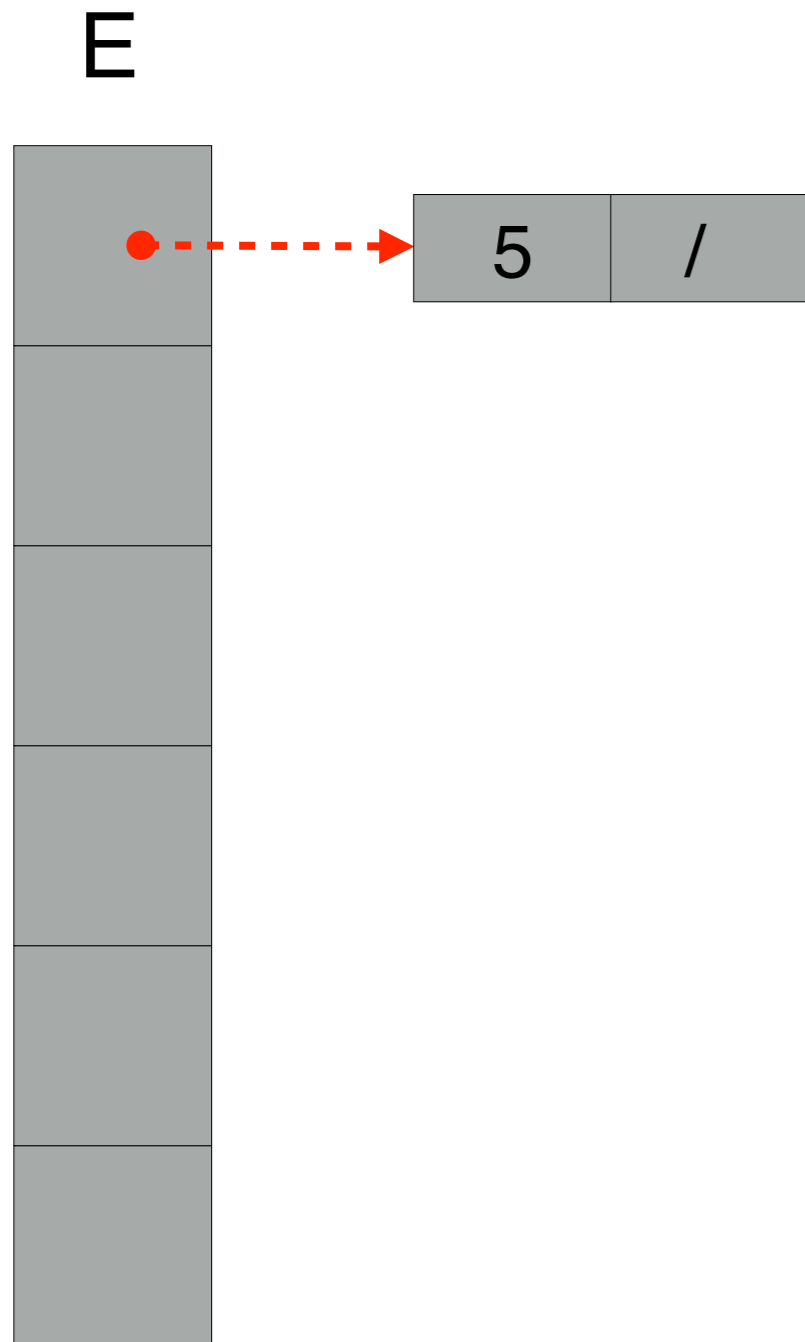


Liste di adiacenza

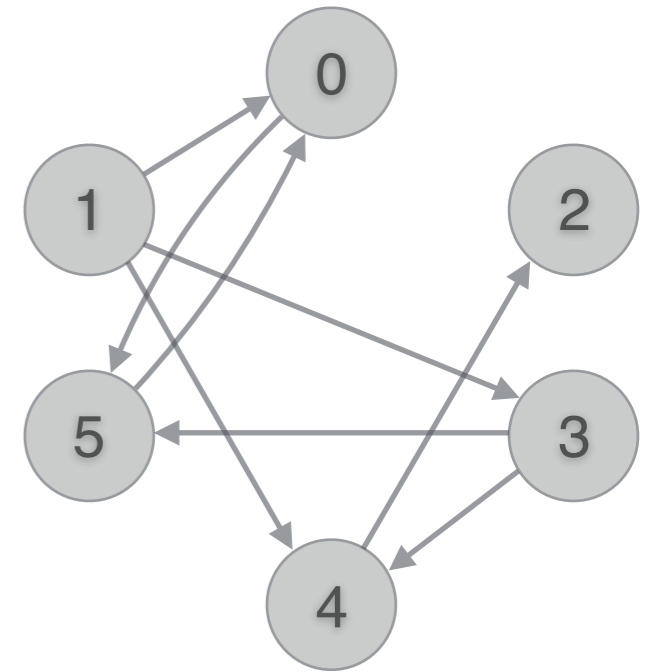
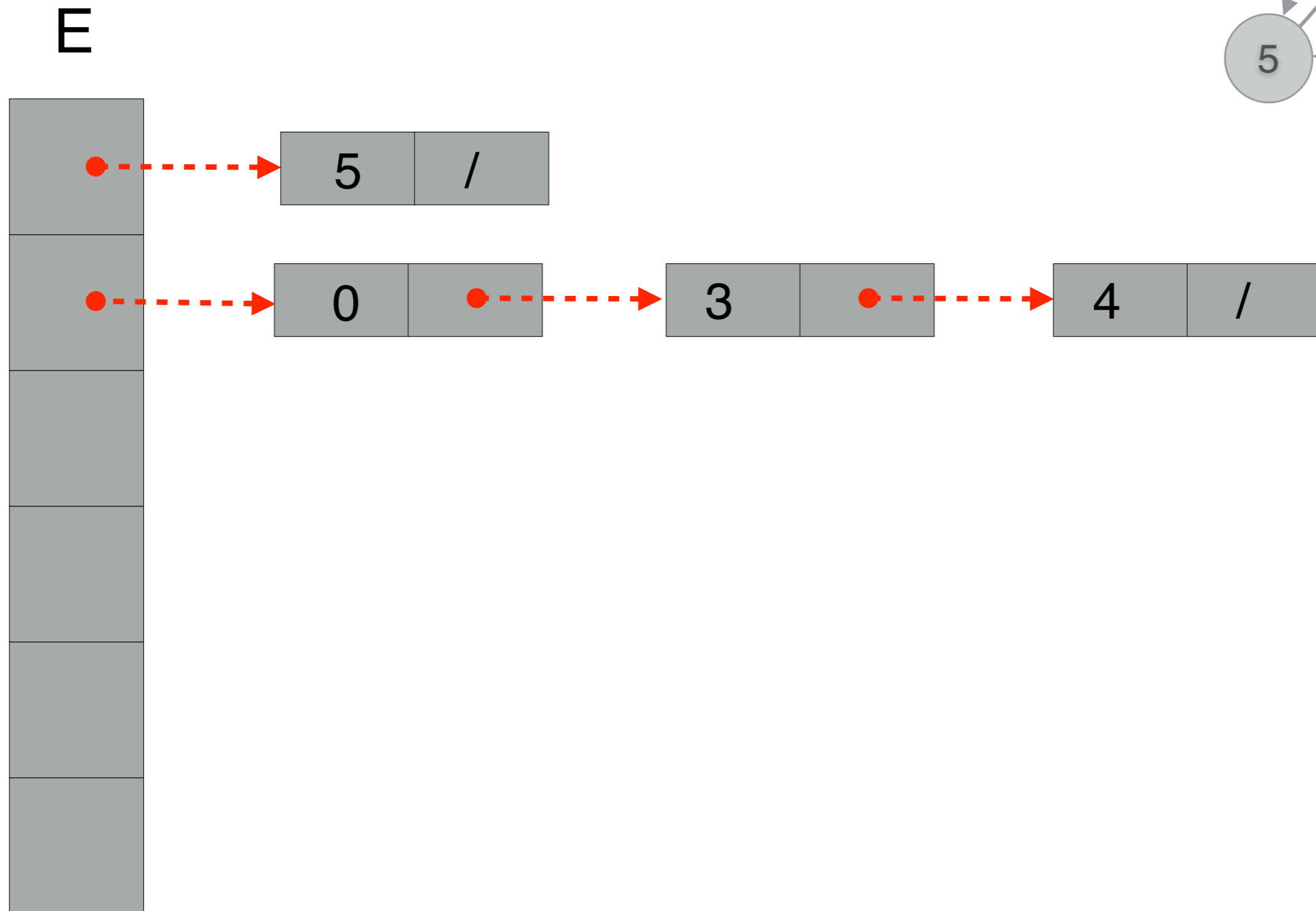
E



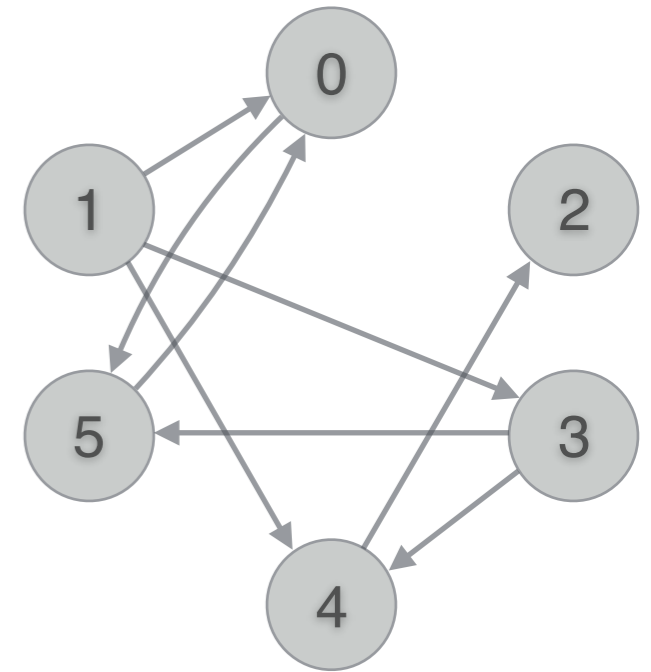
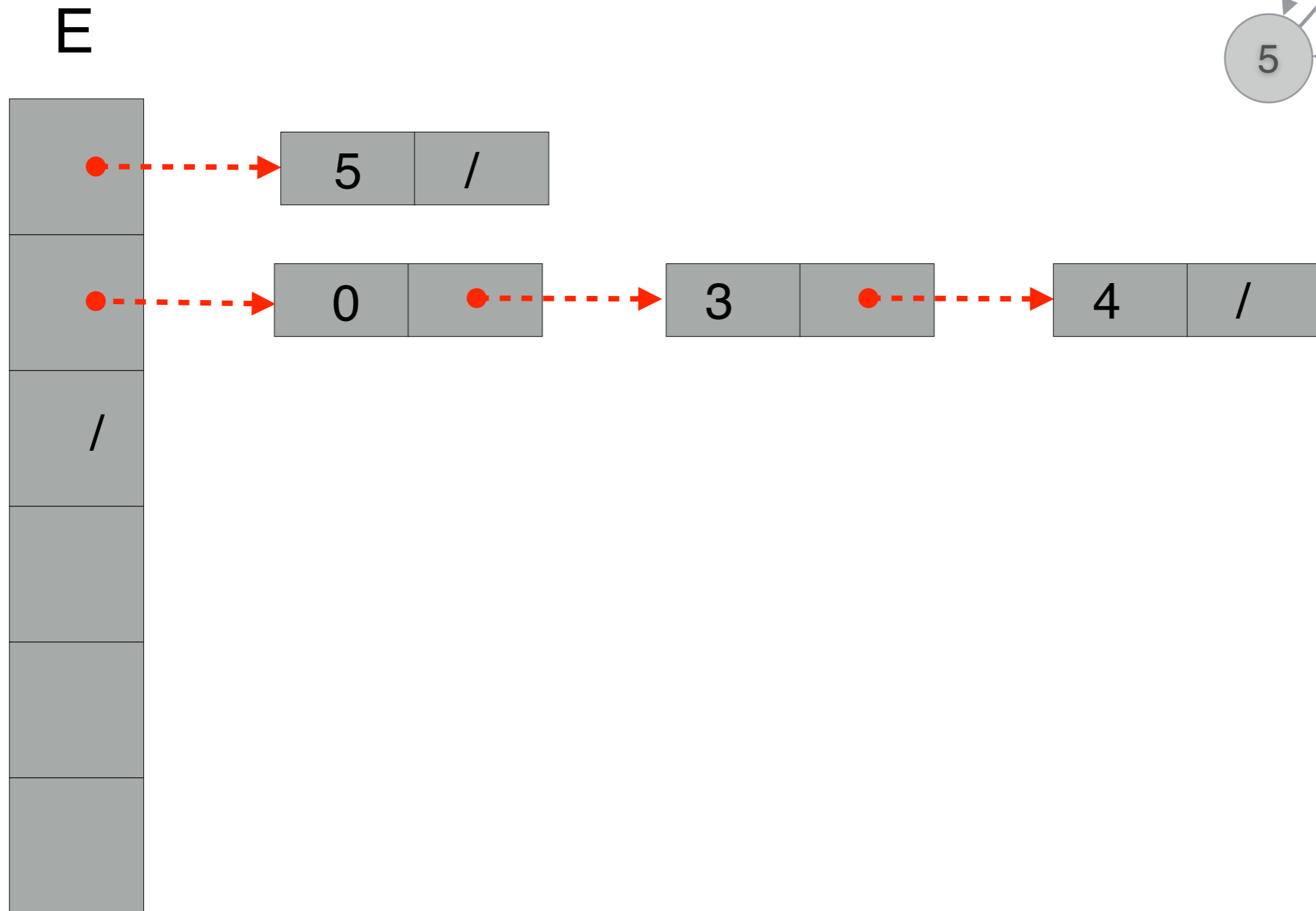
Liste di adiacenza



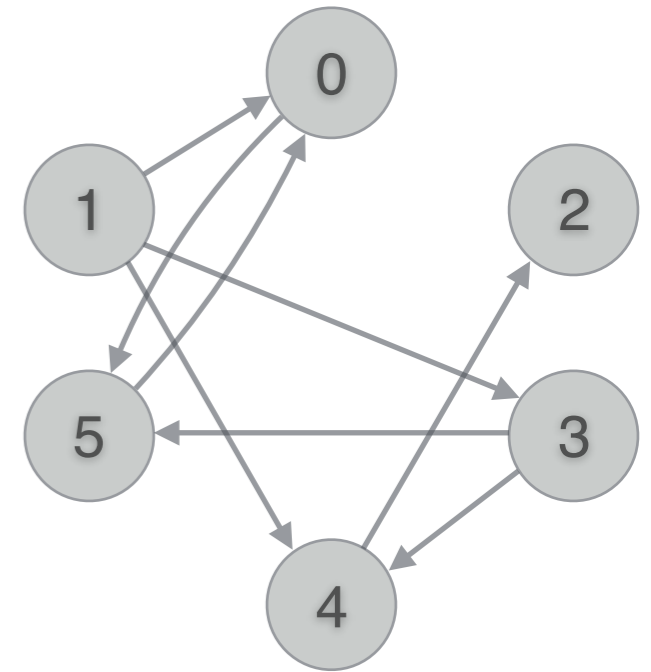
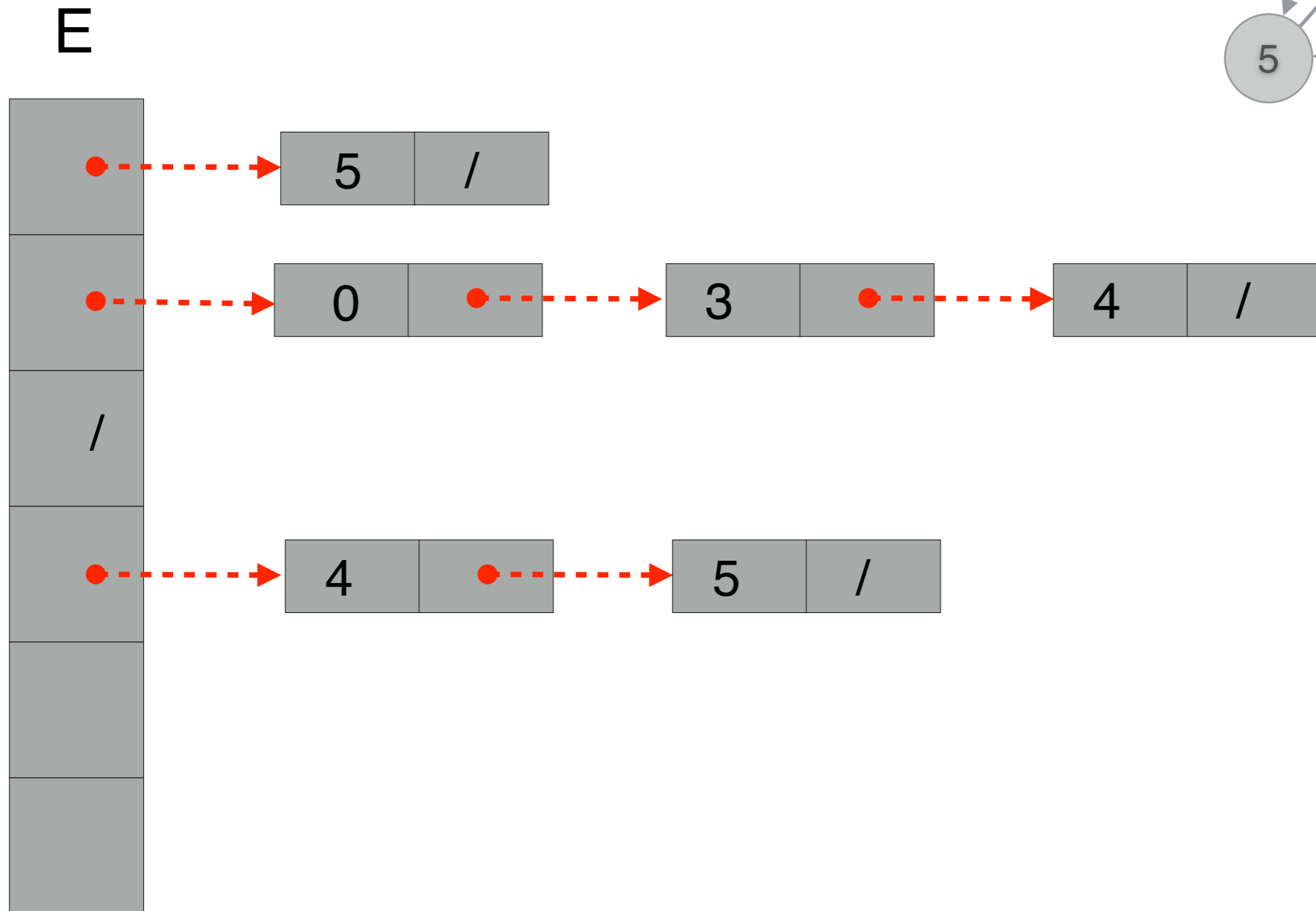
Liste di adiacenza



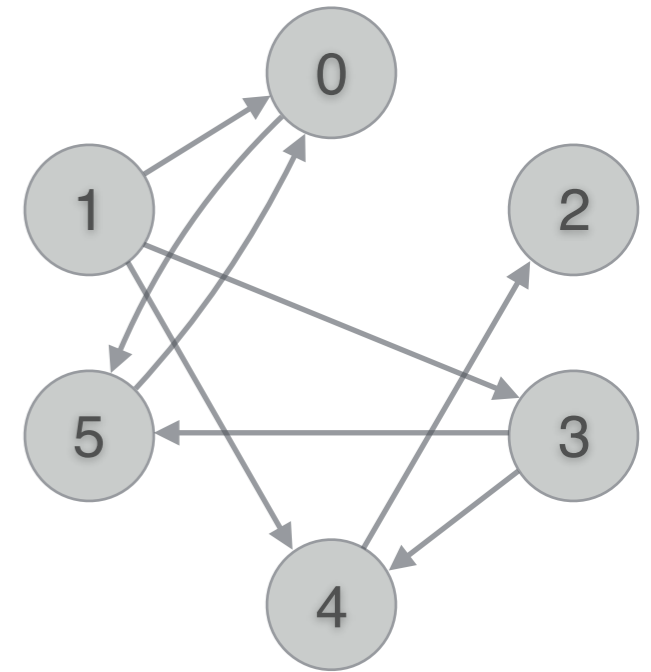
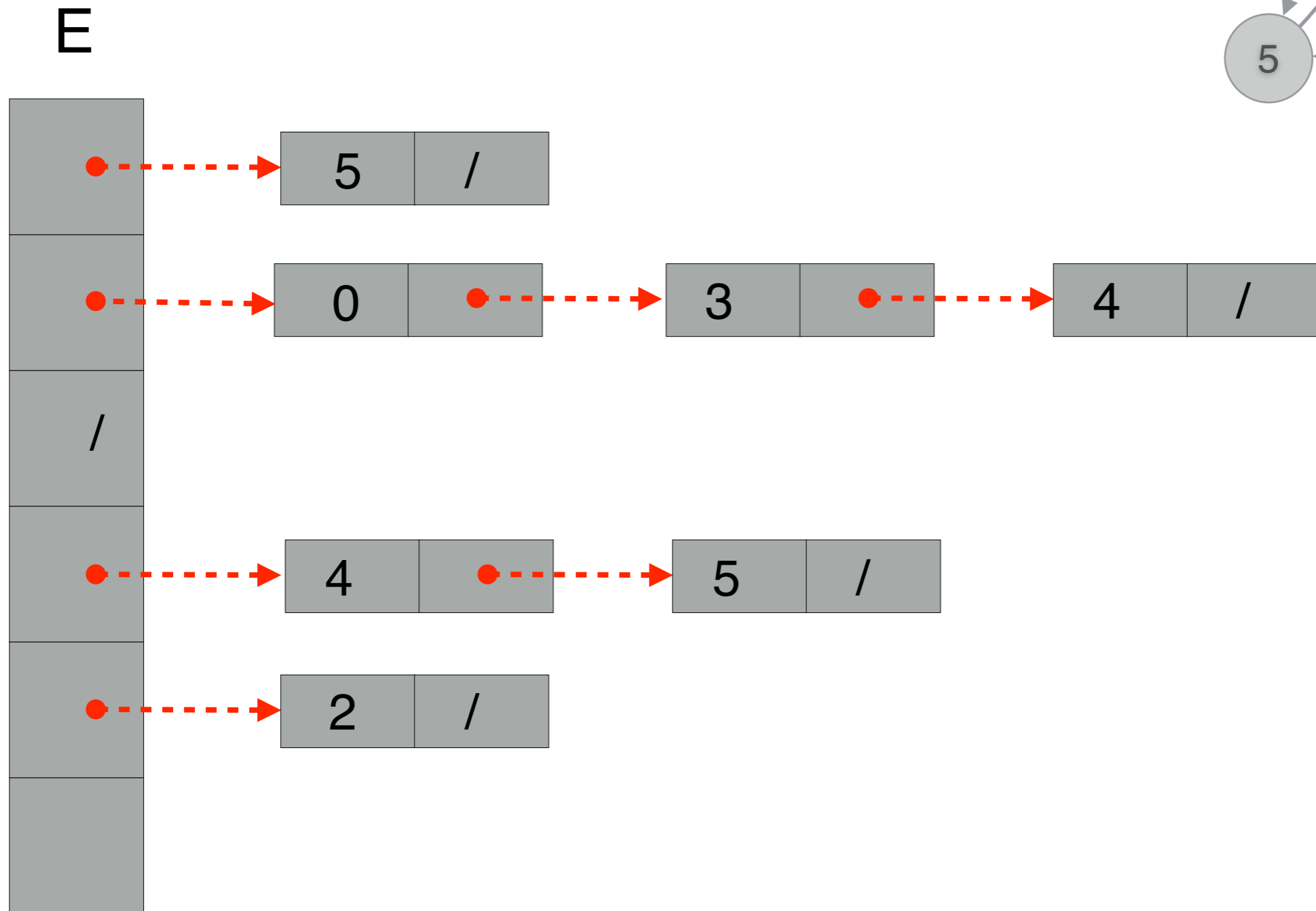
Liste di adiacenza



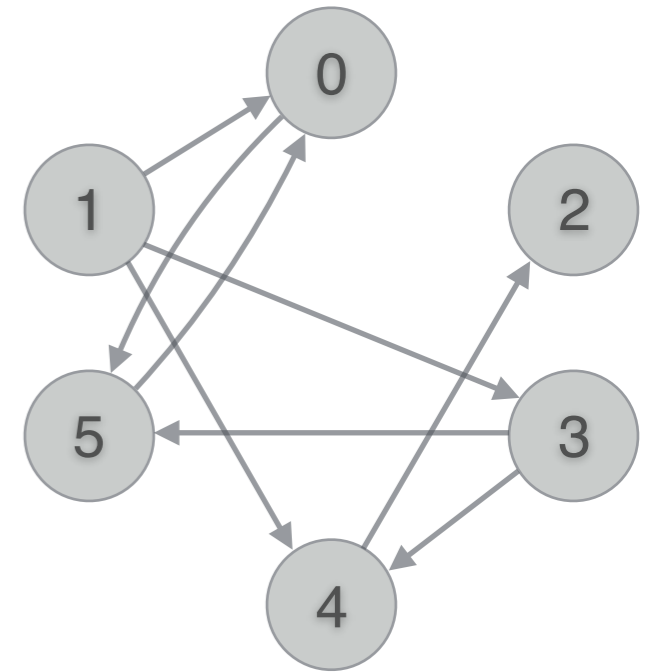
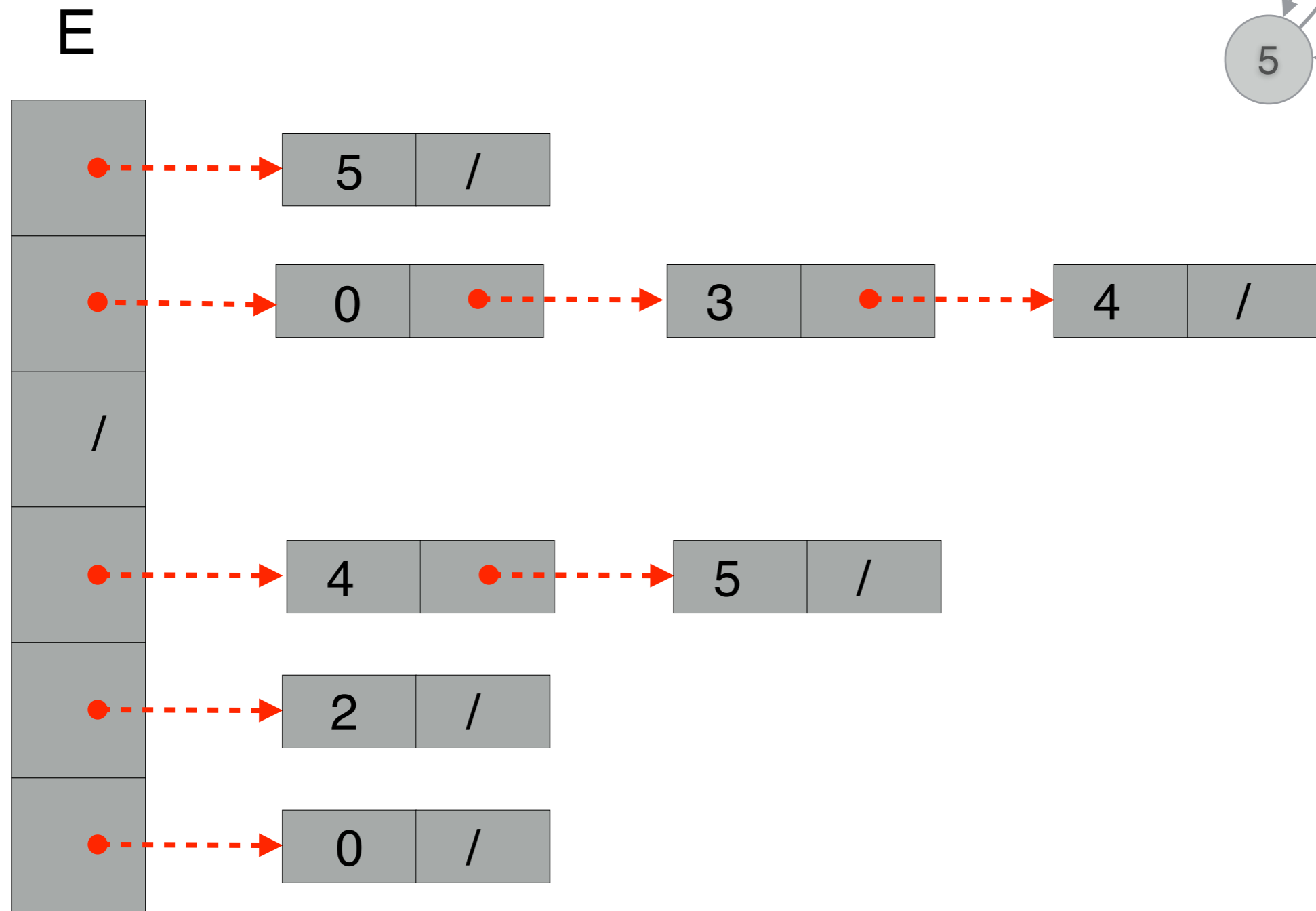
Liste di adiacenza



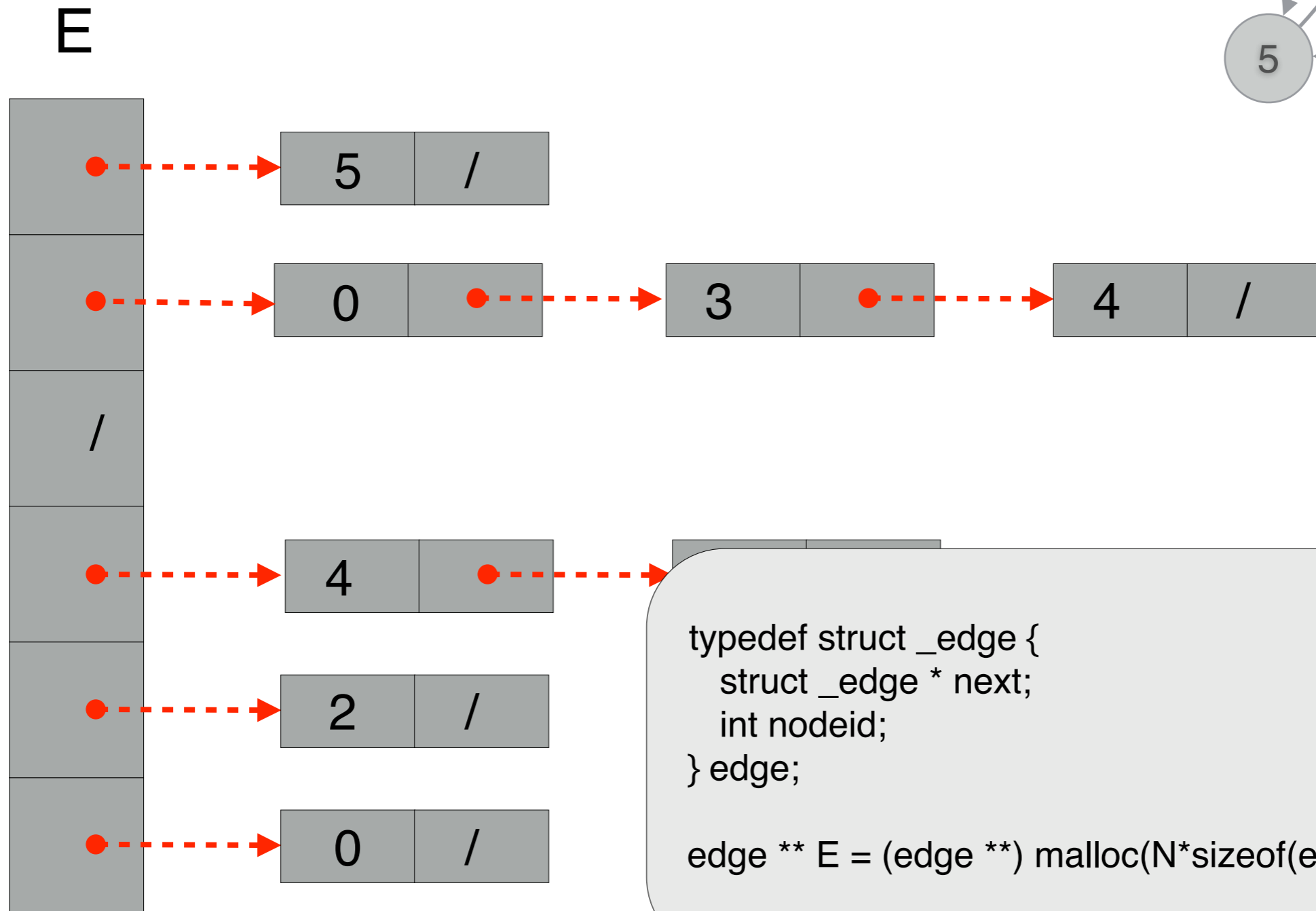
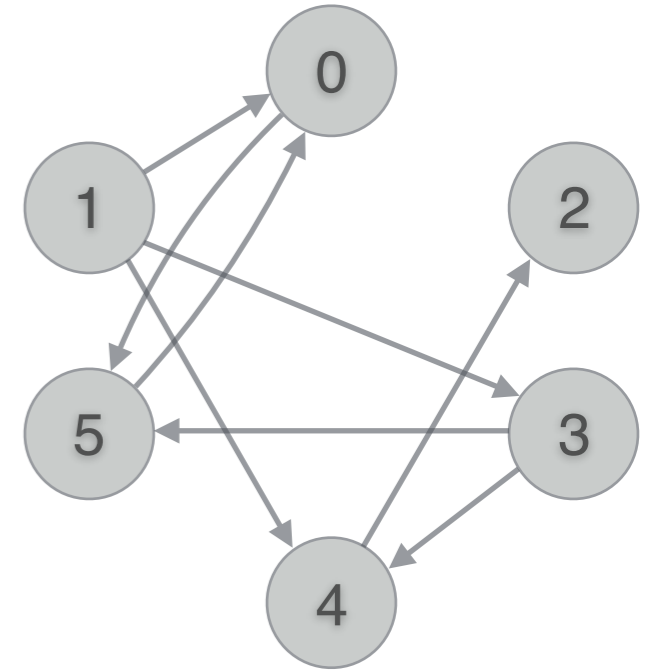
Liste di adiacenza



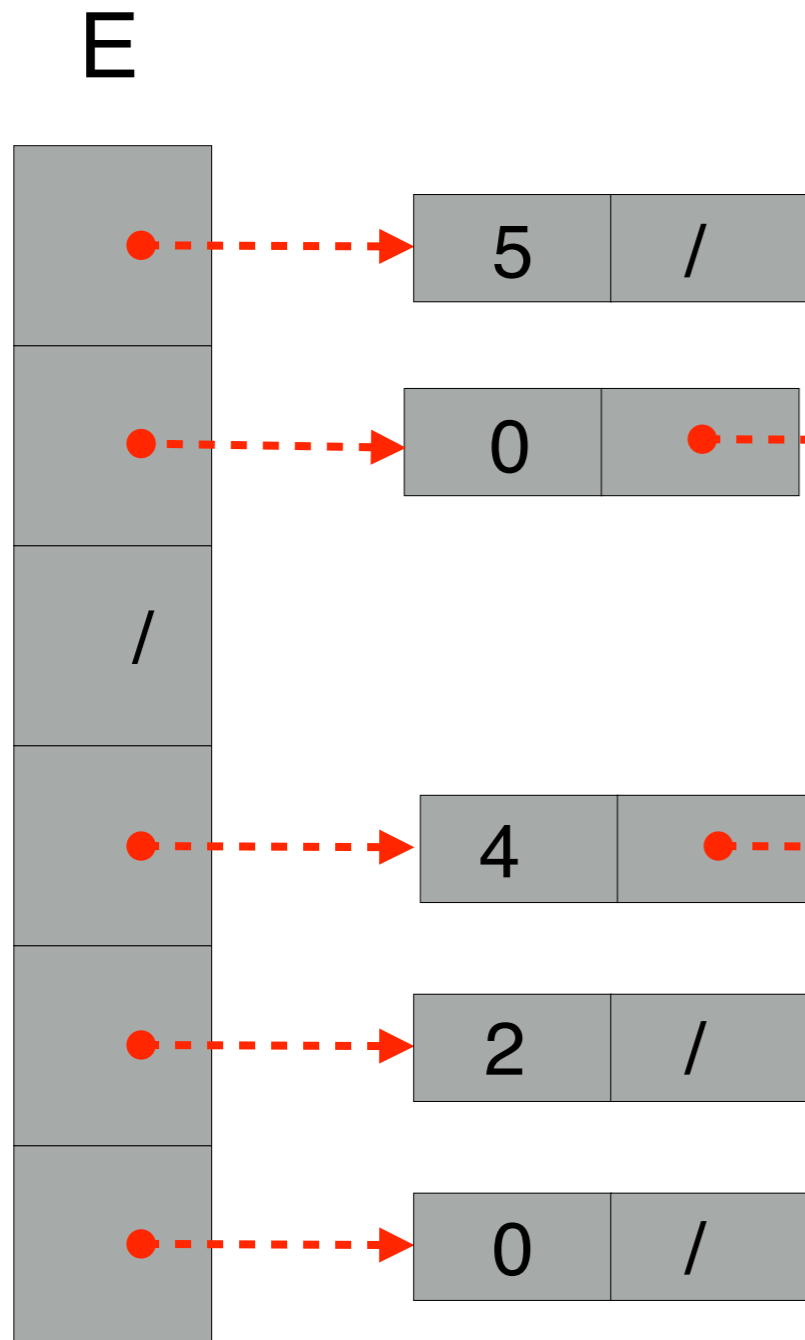
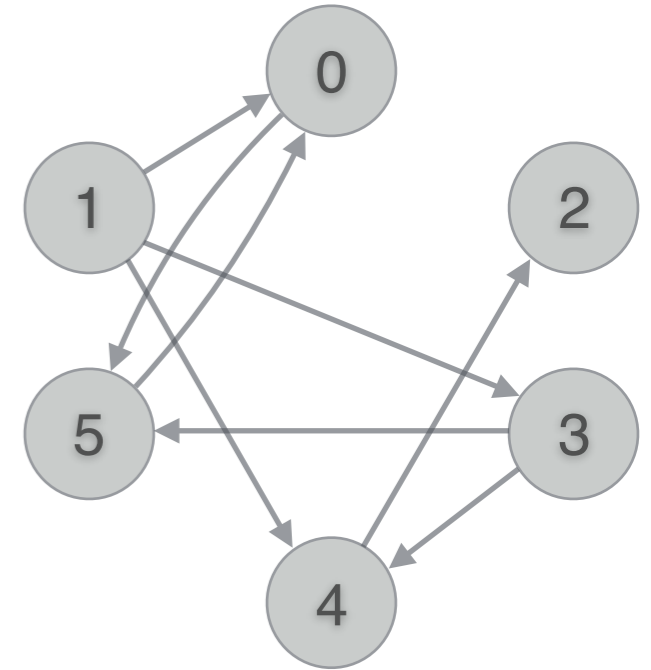
Liste di adiacenza



Liste di adiacenza



Liste di adiacenza

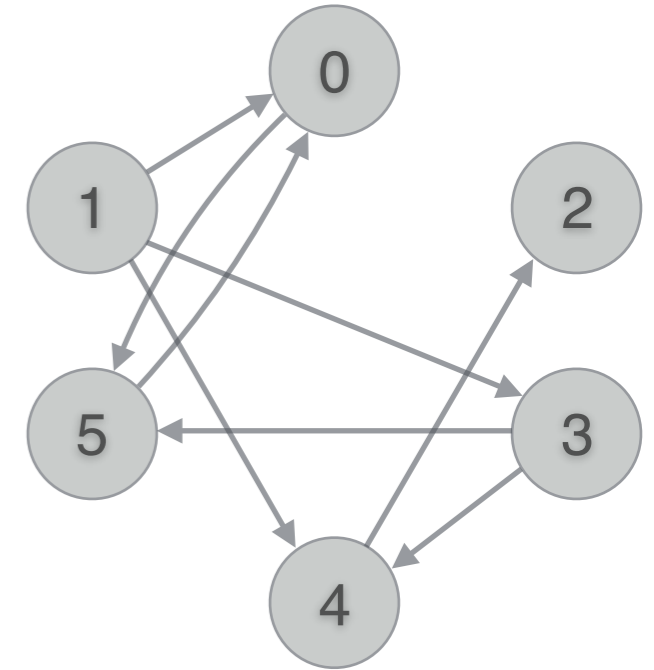


Troppi salti in memoria...

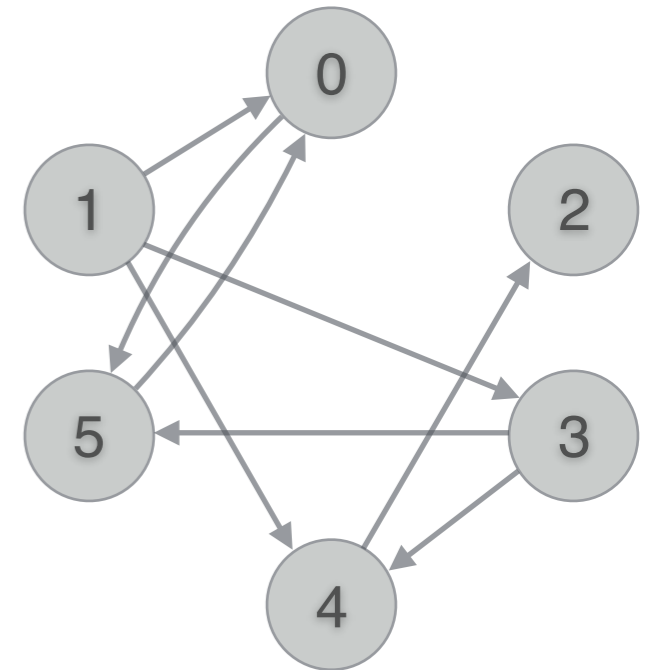
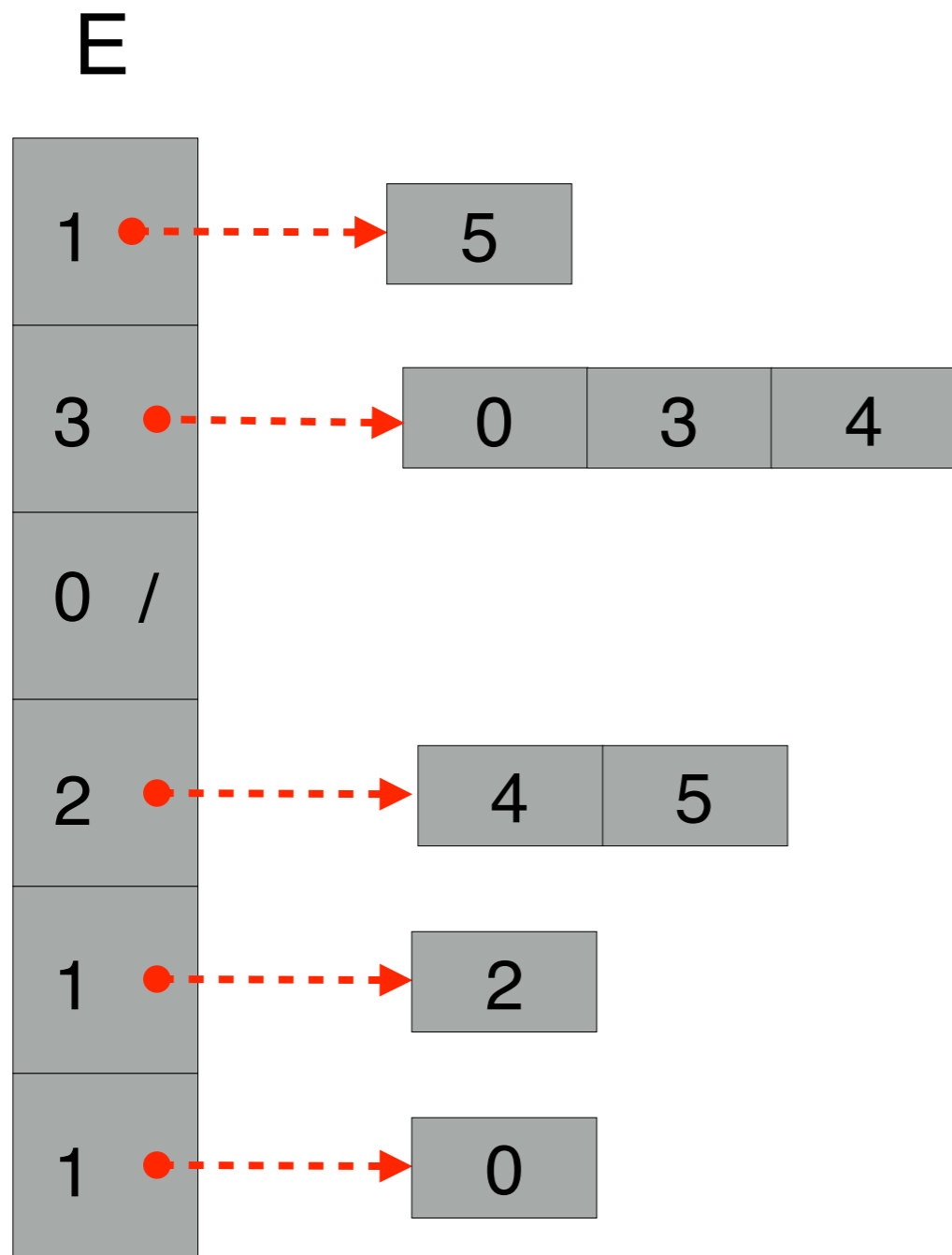
```
typedef struct _edge {
    struct _edge * next;
    int nodeid;
} edge;

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

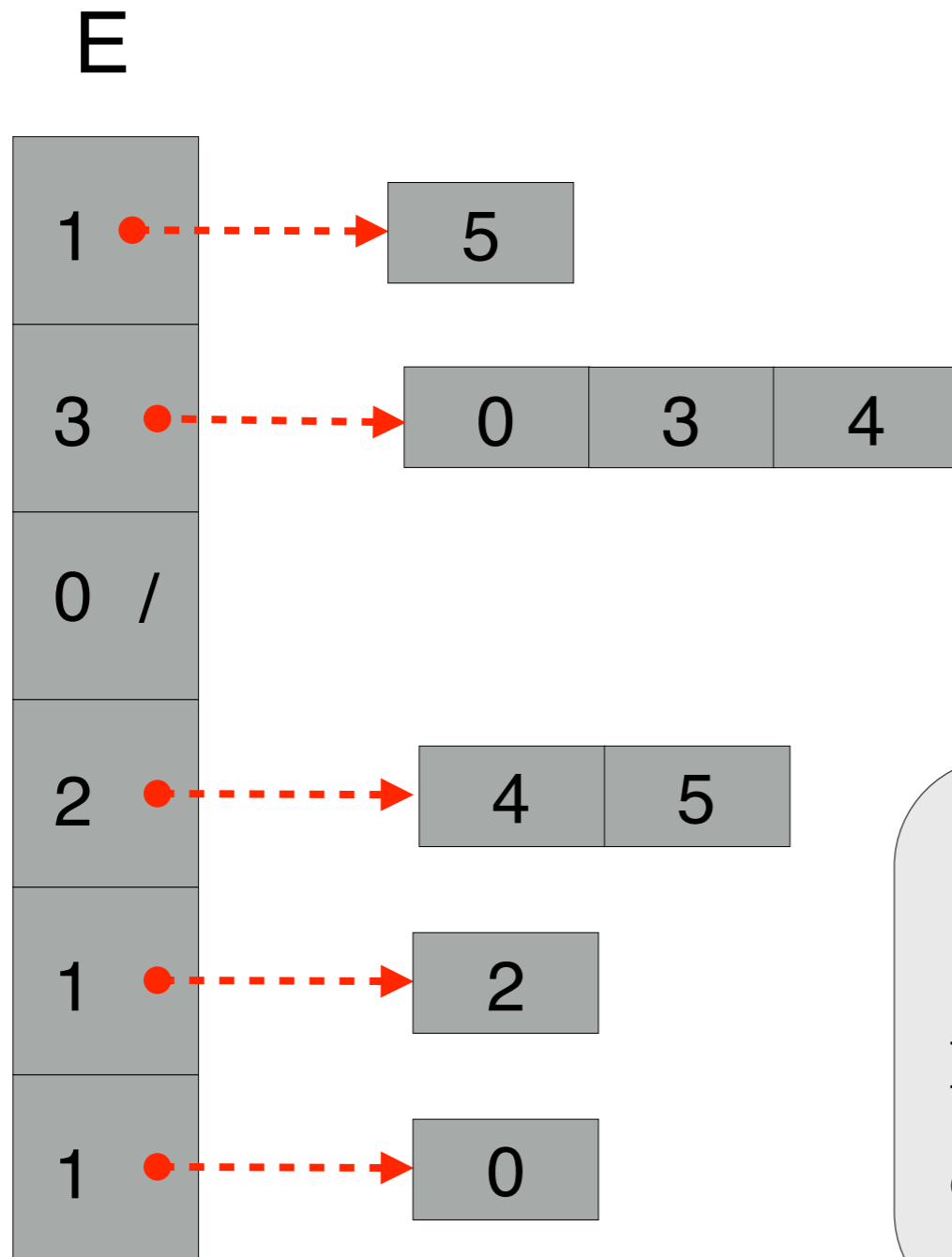
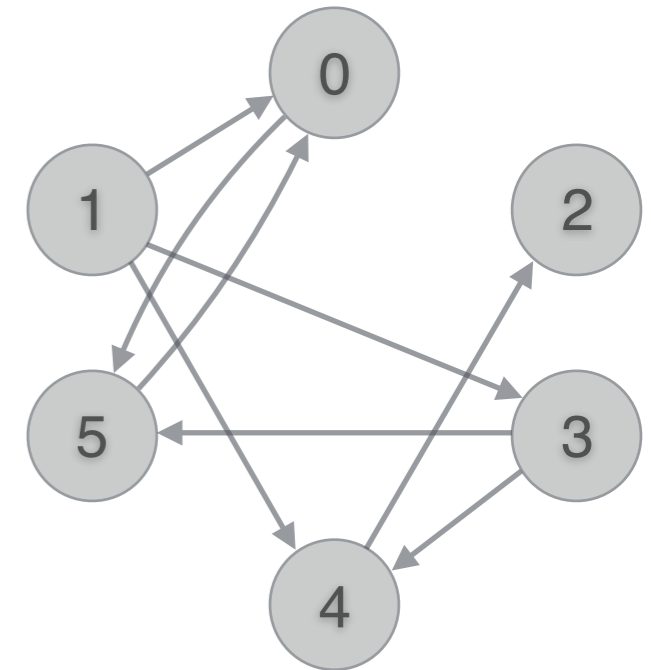
Liste di adiacenza compatte



Liste di adiacenza compatte



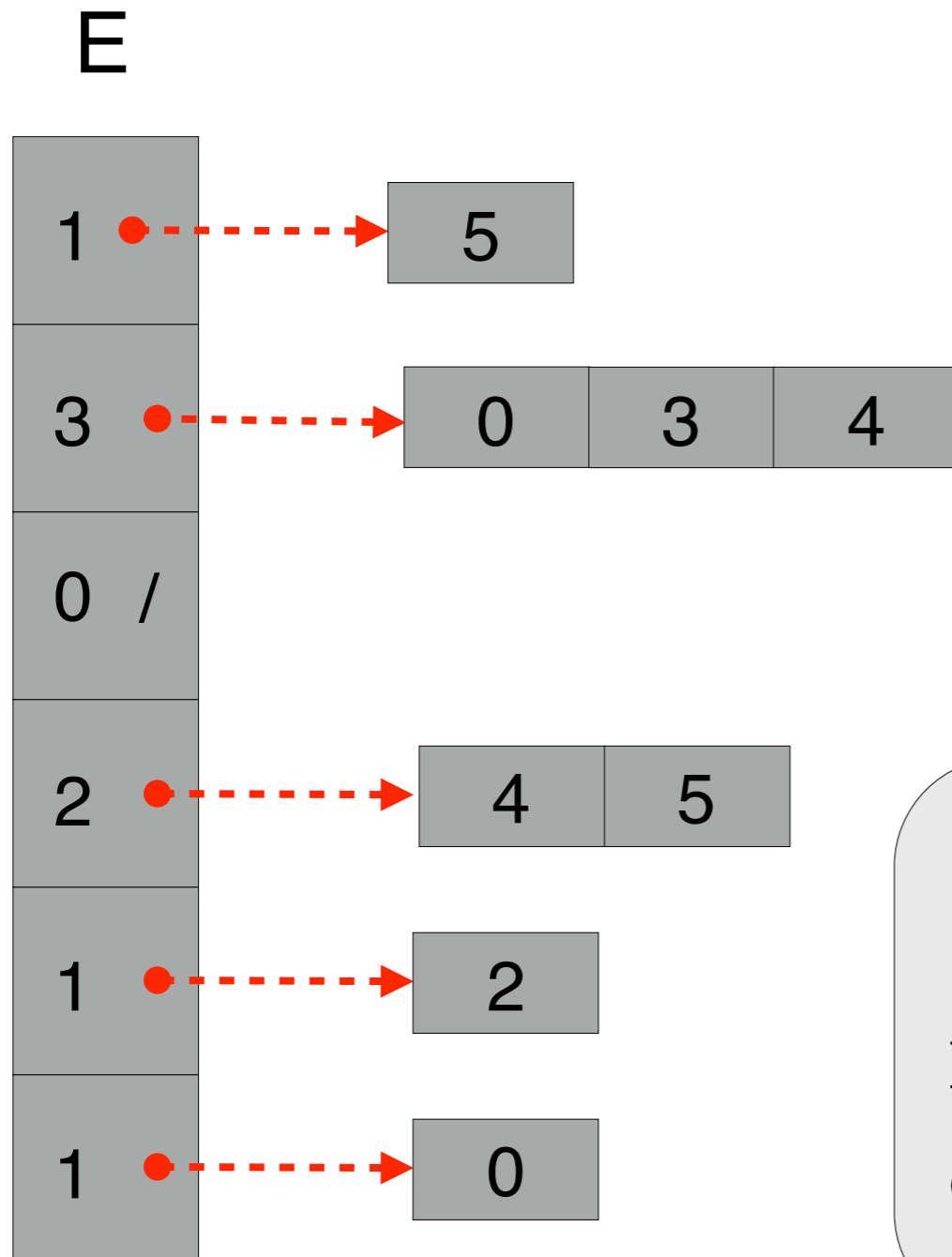
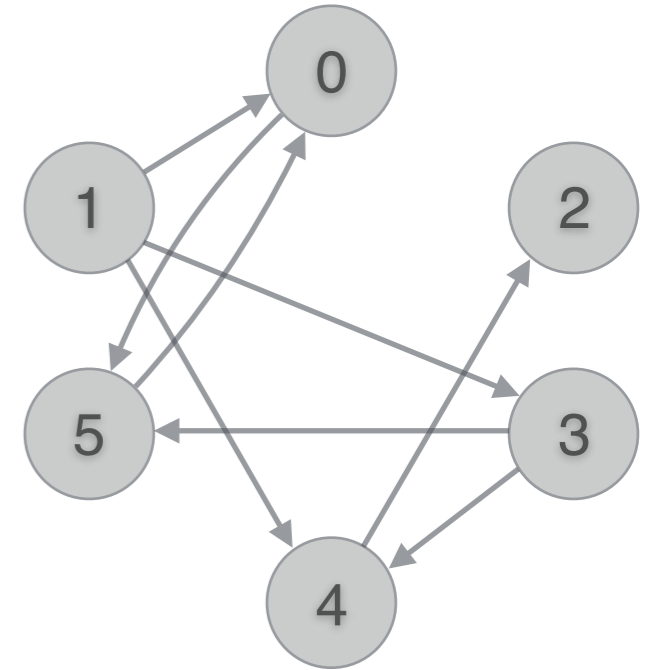
Liste di adiacenza compatte



```
typedef struct _edges {  
    int num_edges;  
    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_edges;  
    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 ciascun 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 ciascun 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 ciascun 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

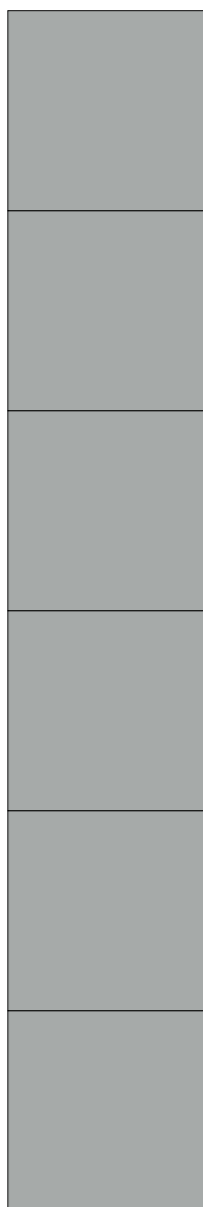
2 4 5

1 2

1 0

Lettura grafo da input

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 ciascun 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

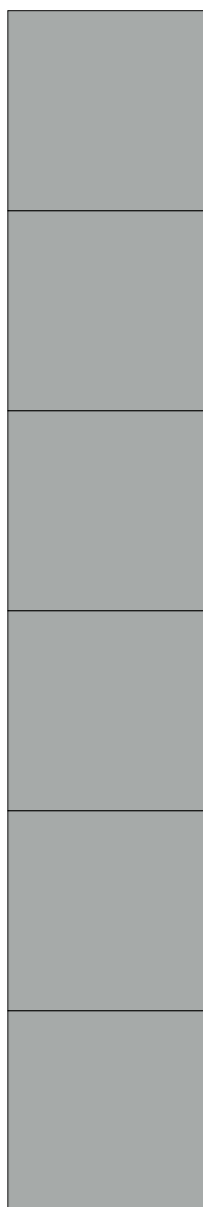
2 4 5

1 2

1 0

Lettura grafo da input

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 ciascun 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

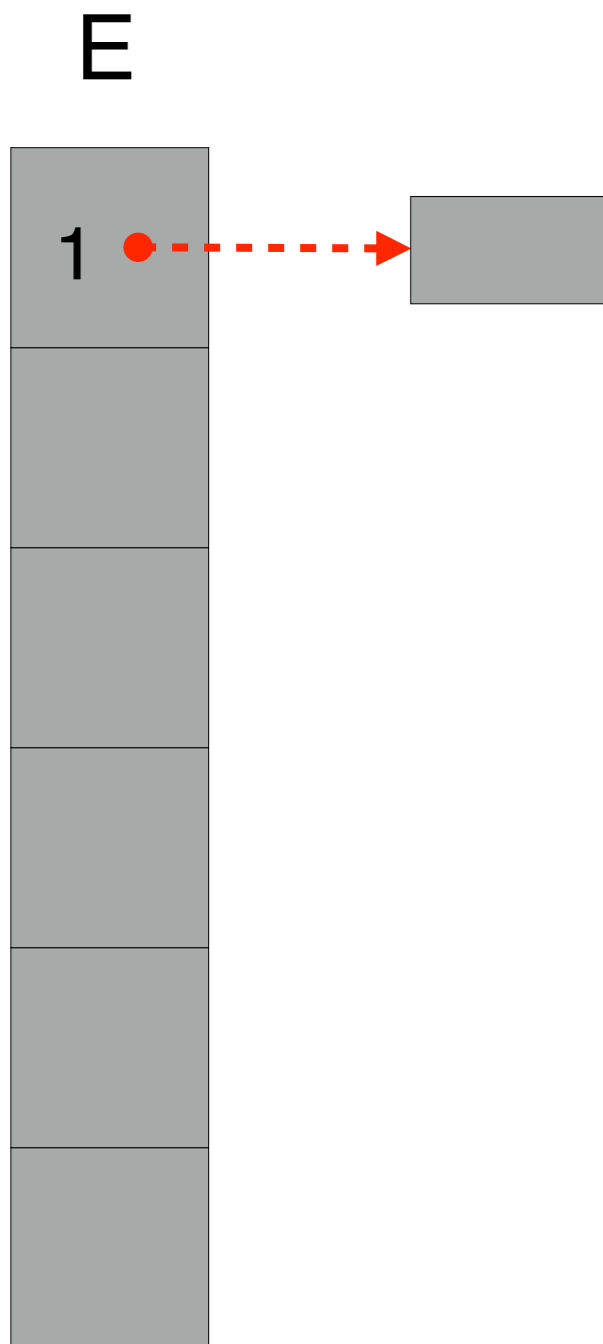
0

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 ciascun 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

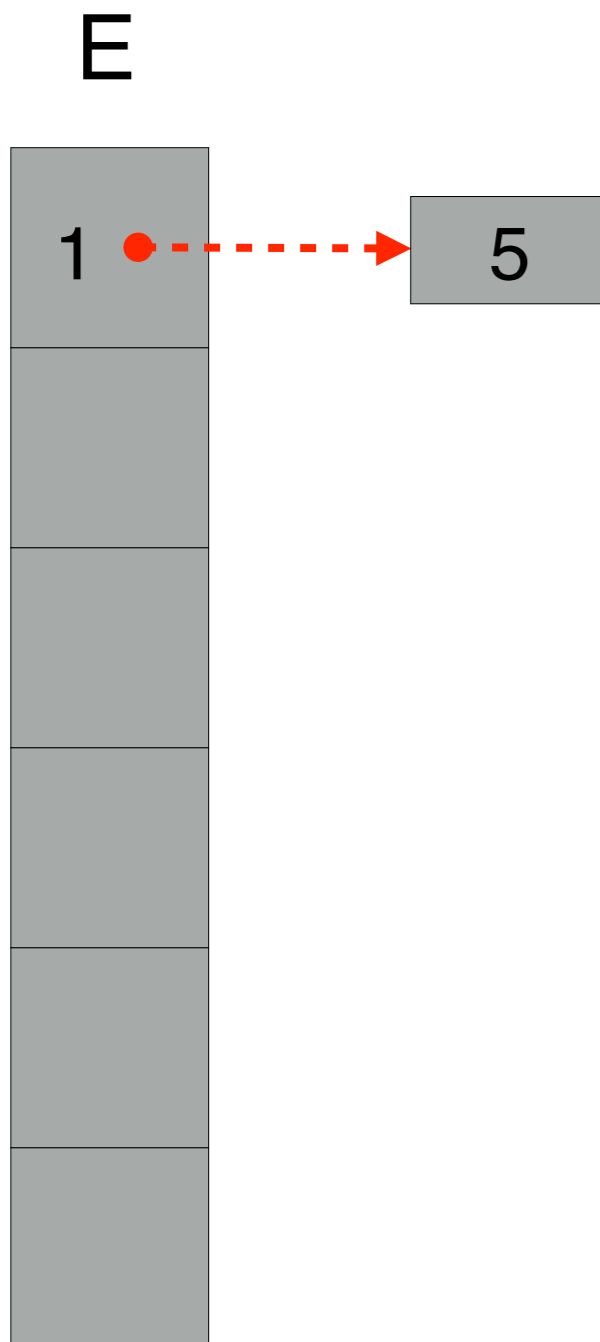
0

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 ciascun 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

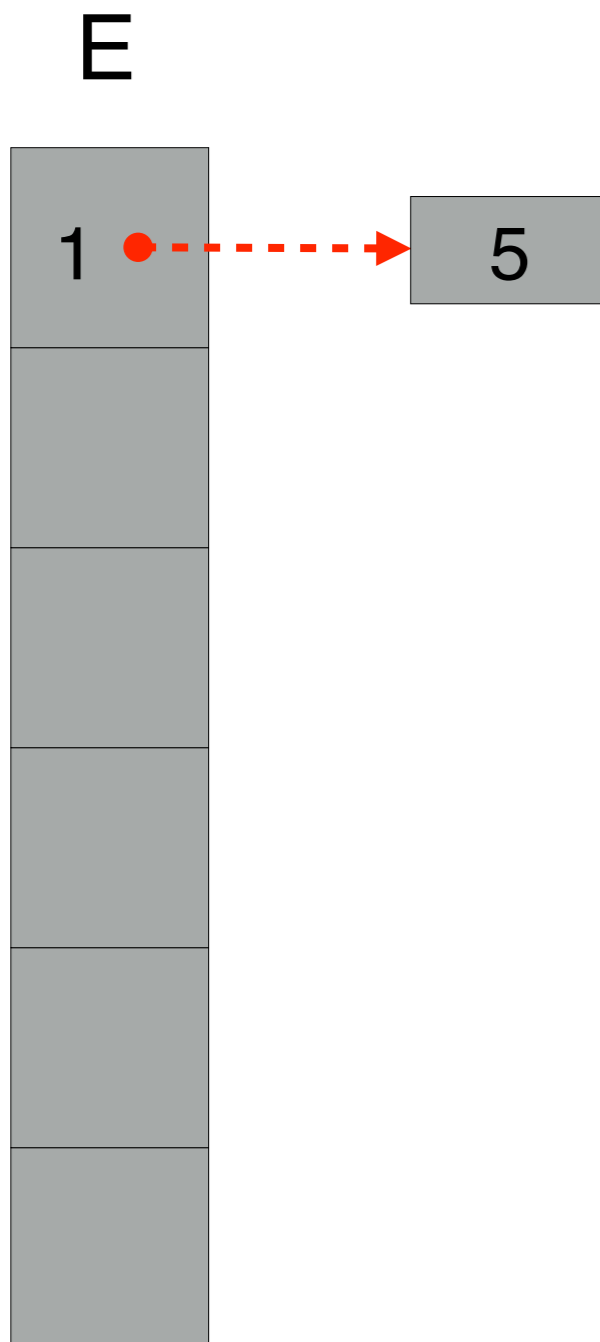
0

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 ciascun 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

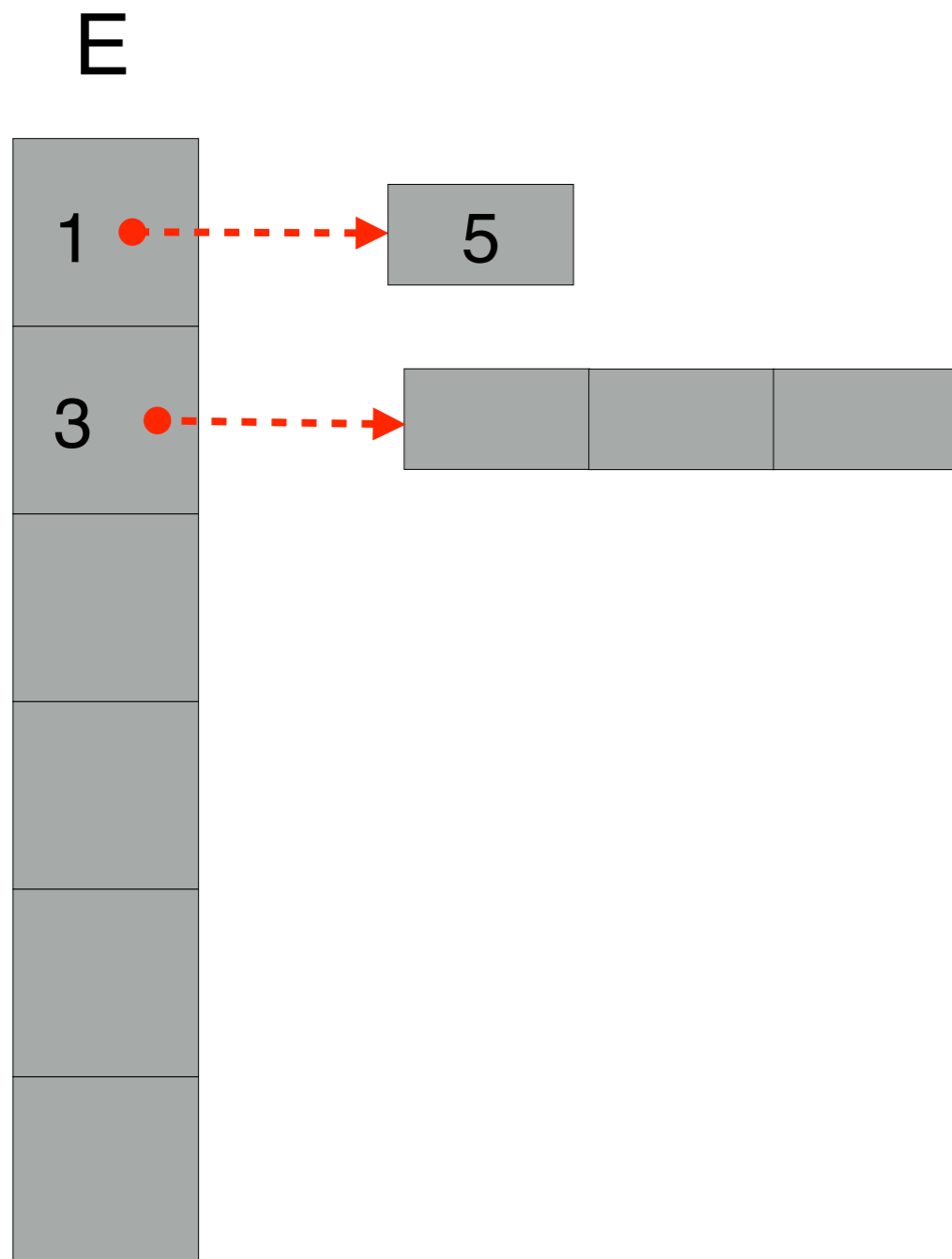
0

2 4 5

1 2

1 0

Lettura grafo da input



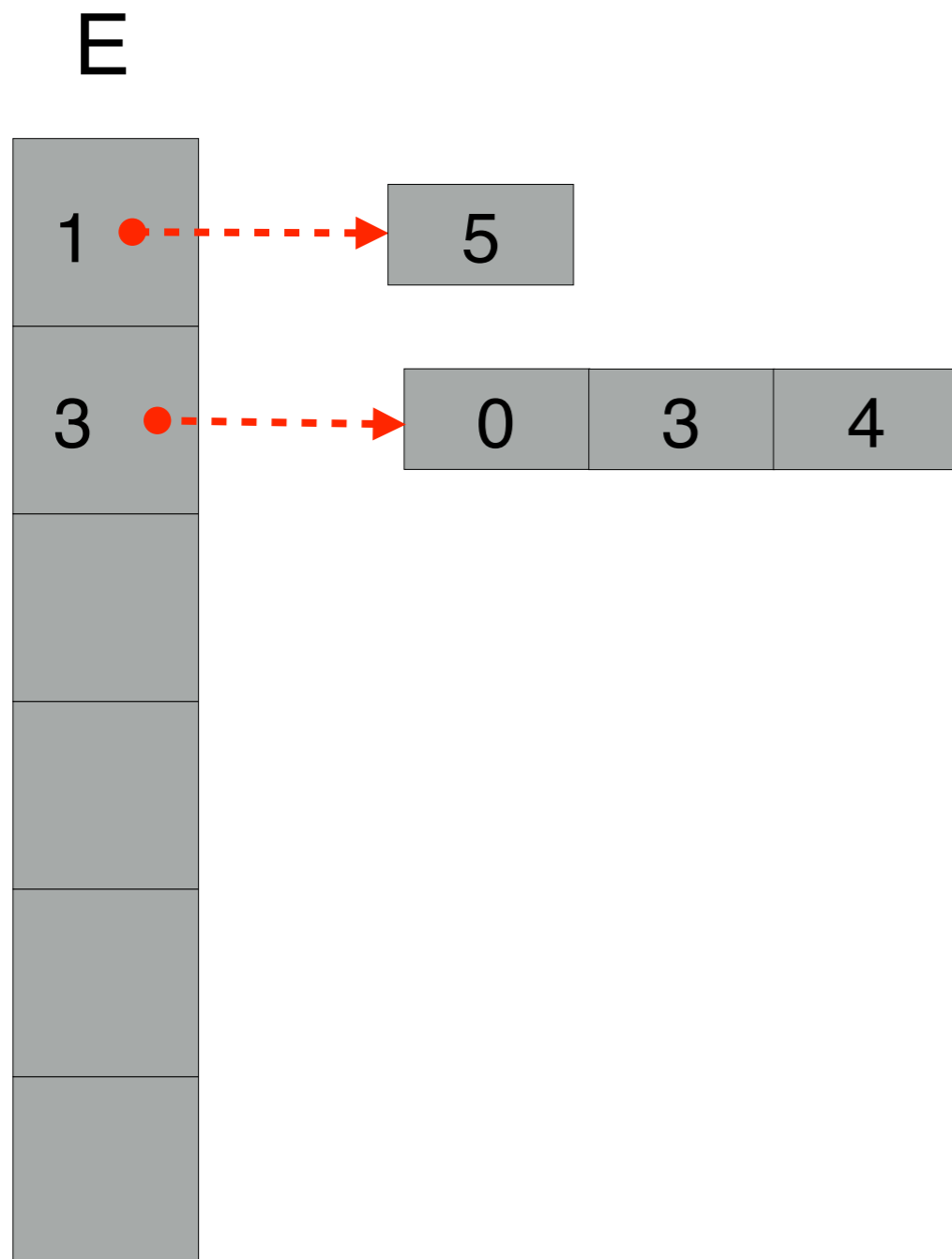
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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



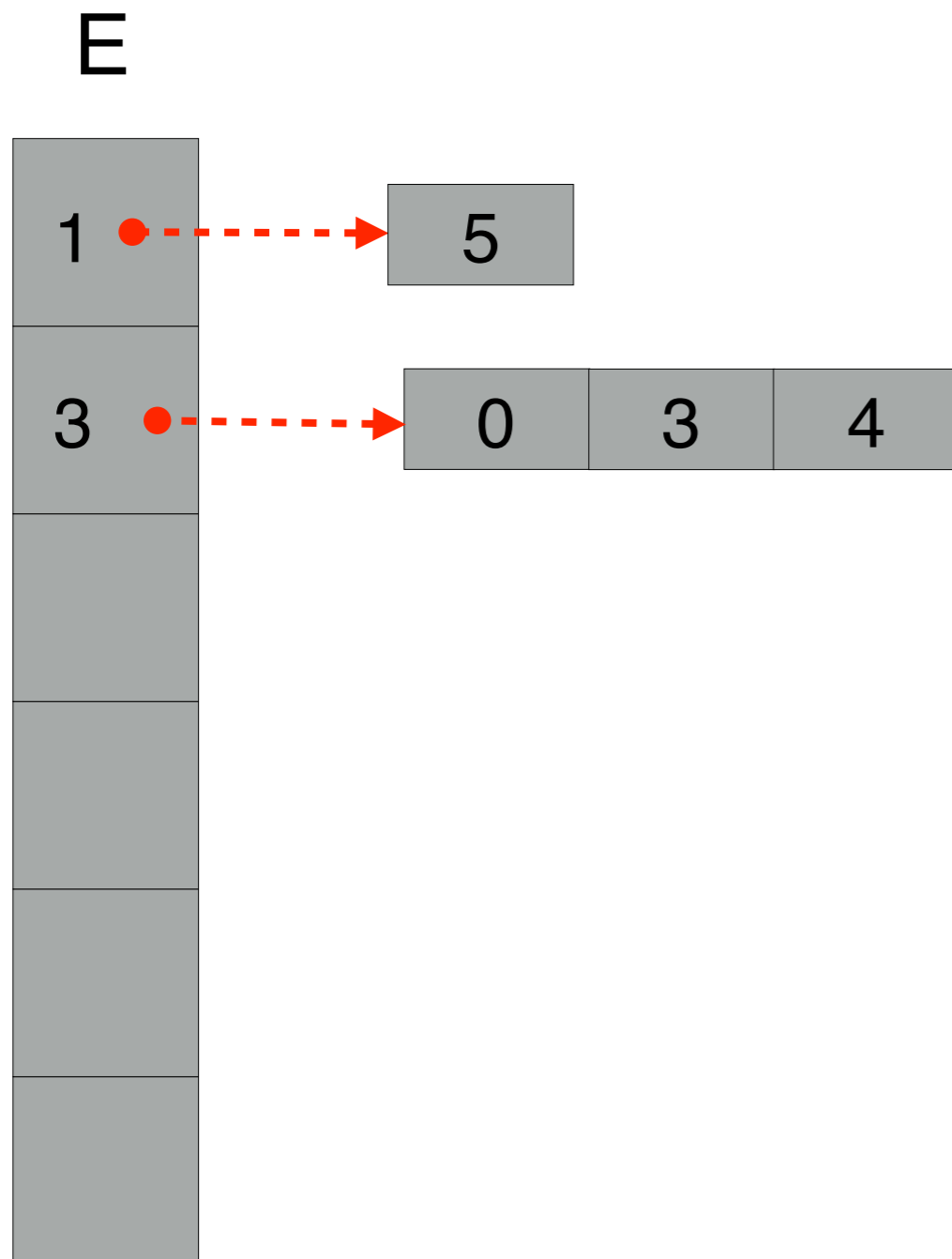
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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



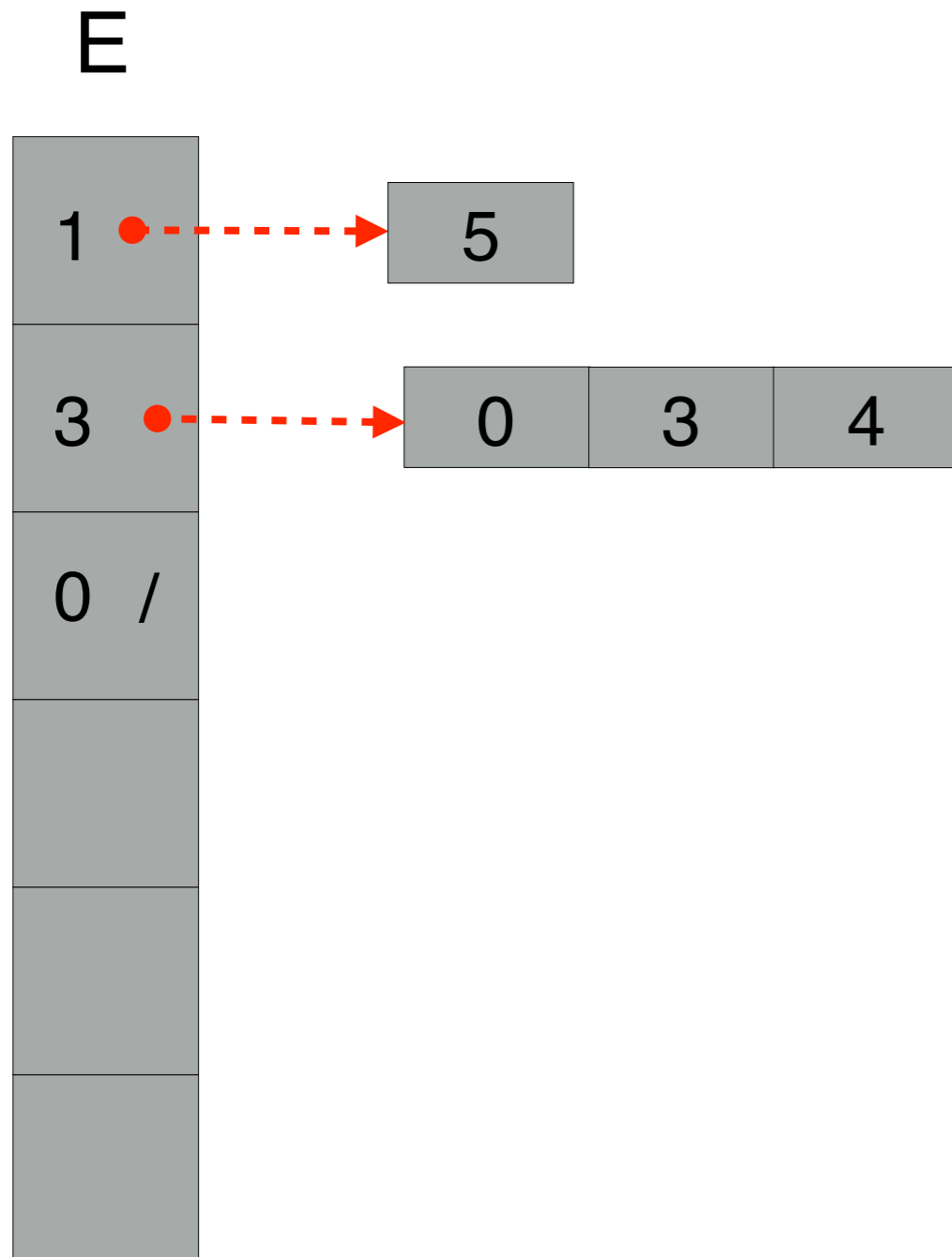
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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



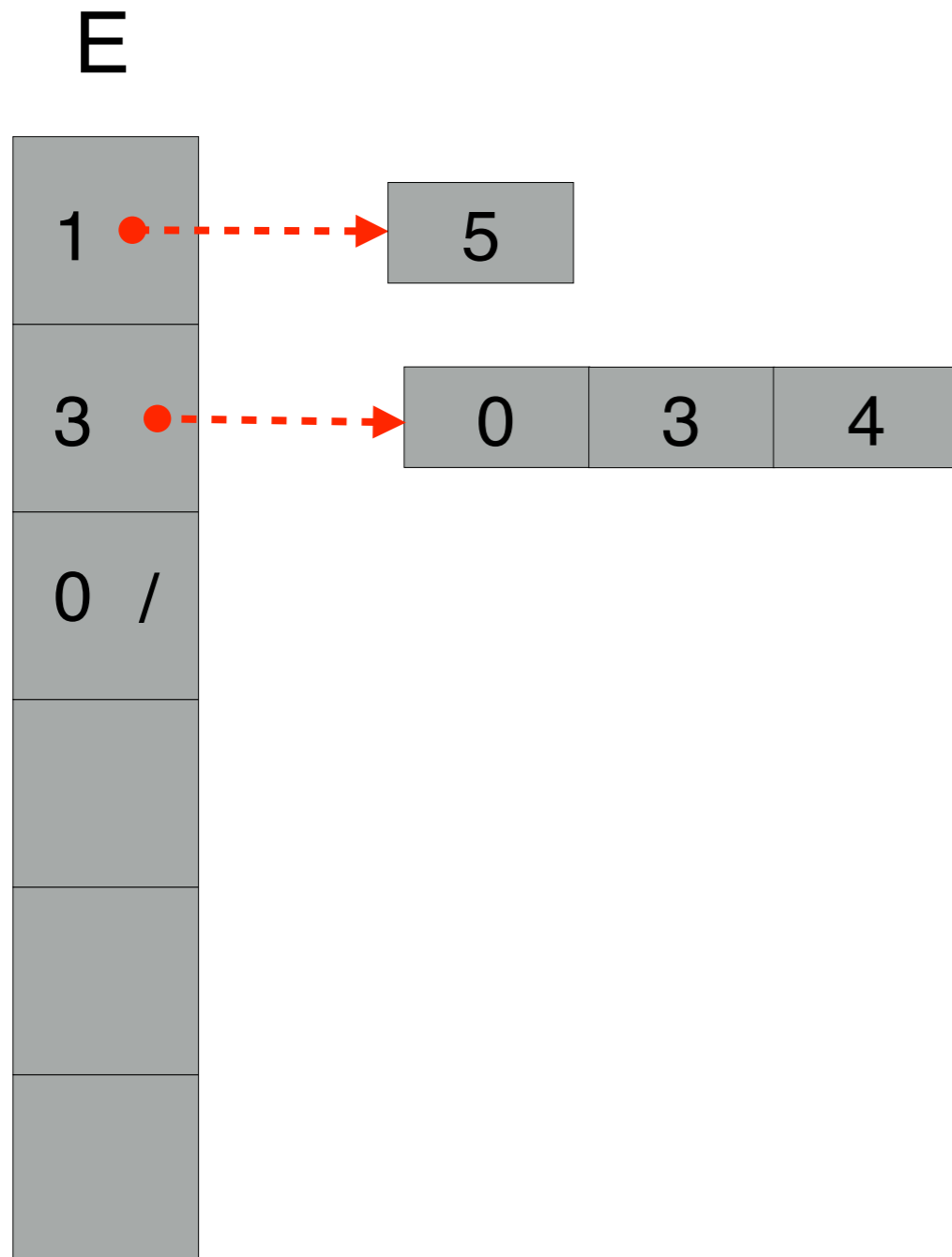
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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



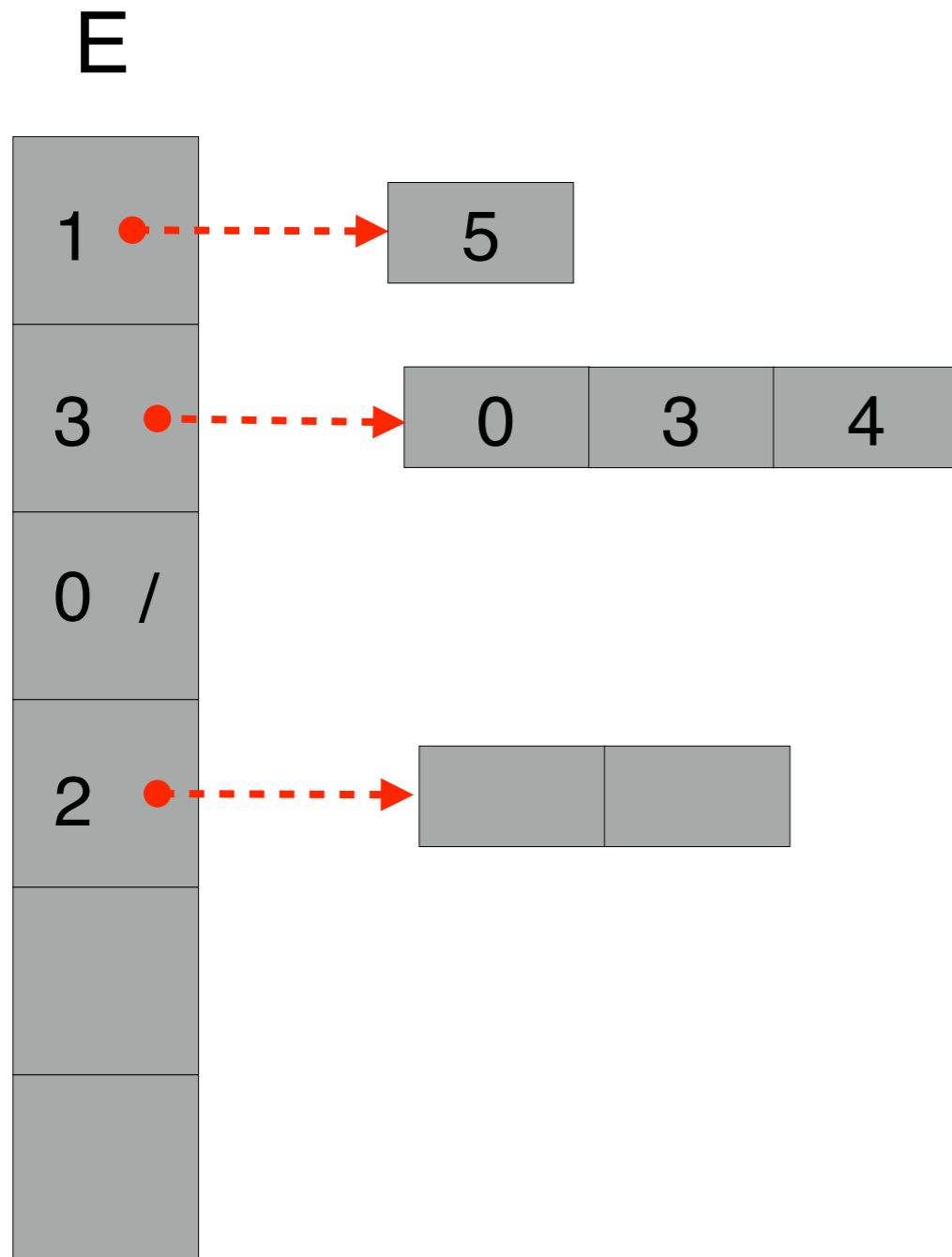
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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



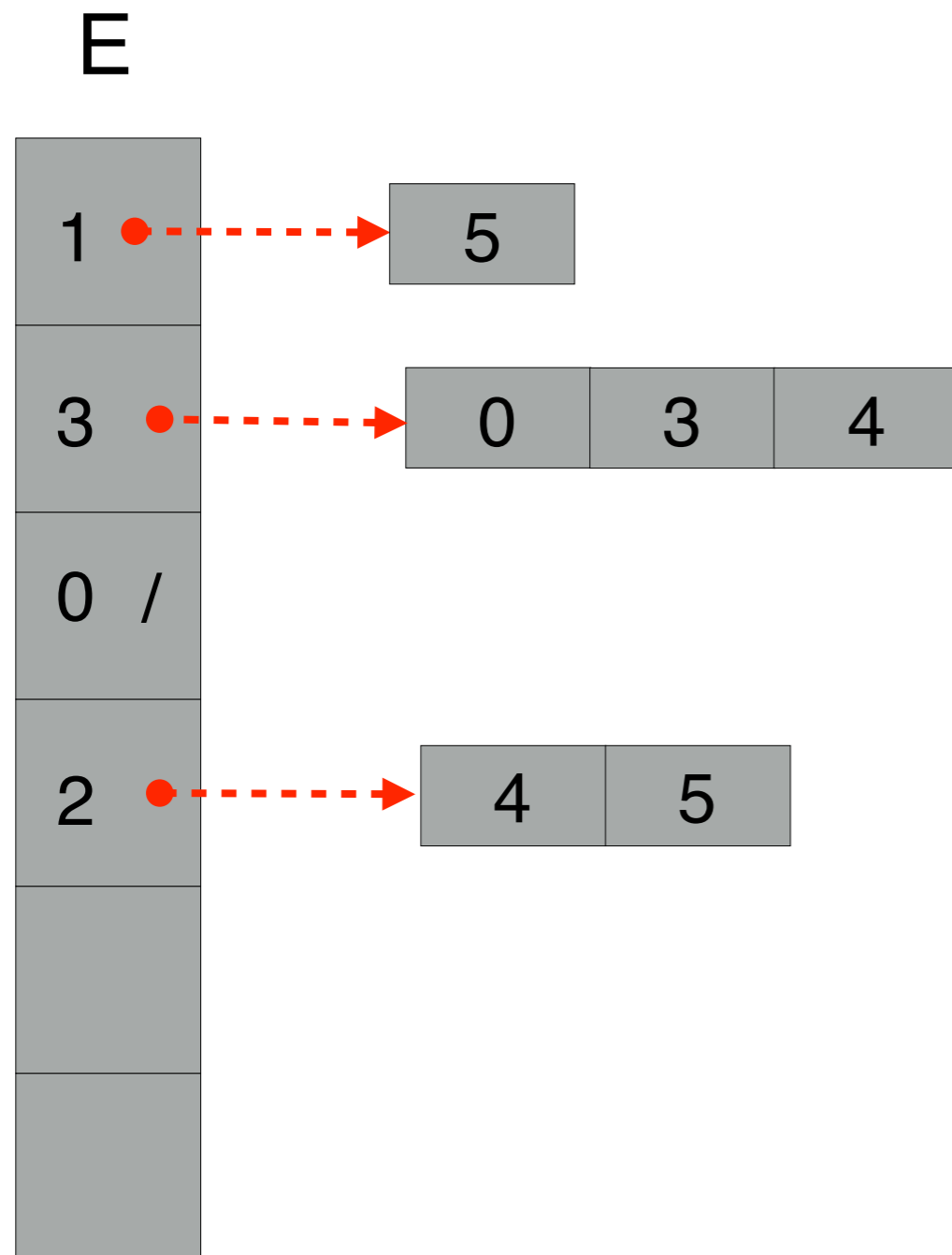
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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



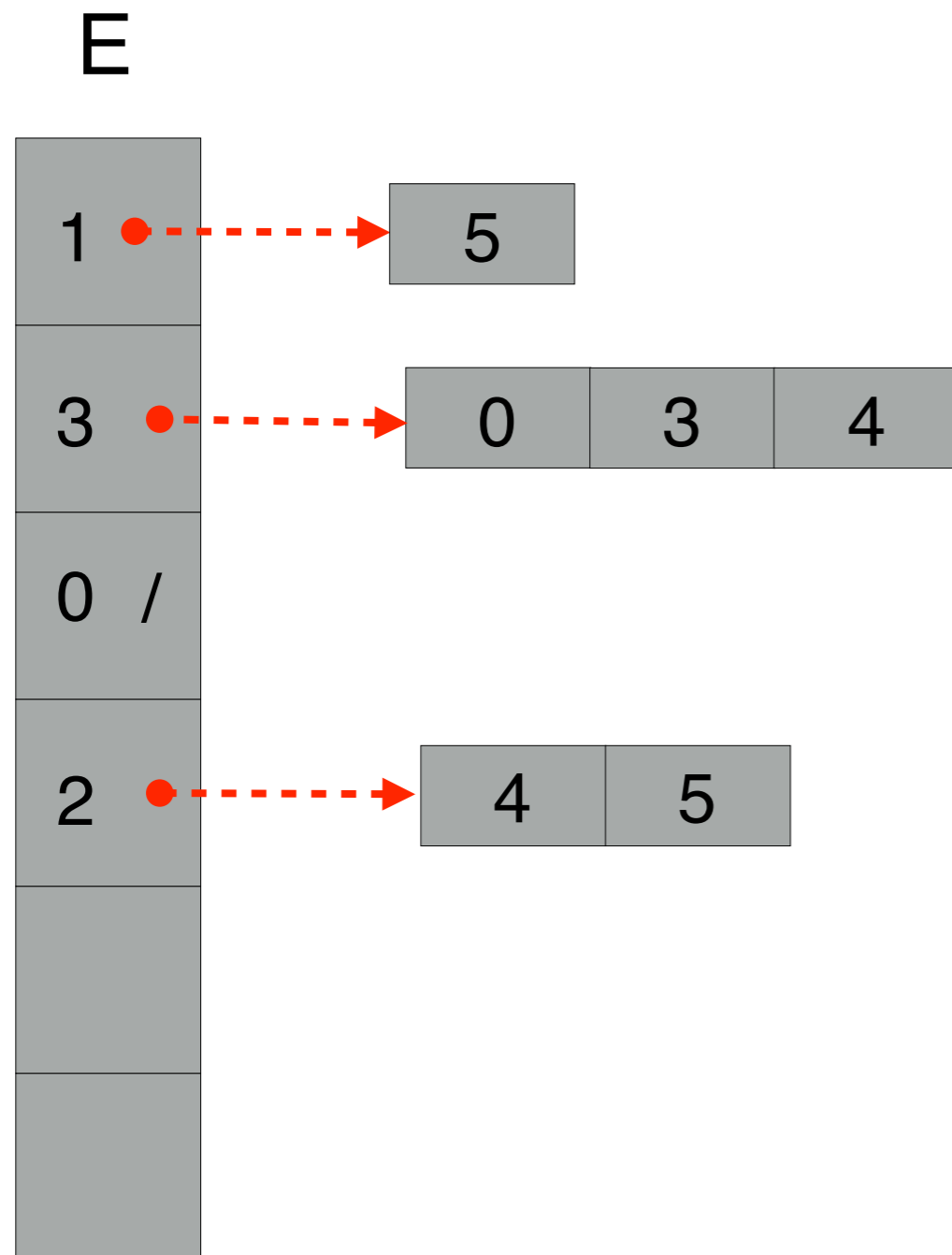
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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 sar  richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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 $[0, n)$.

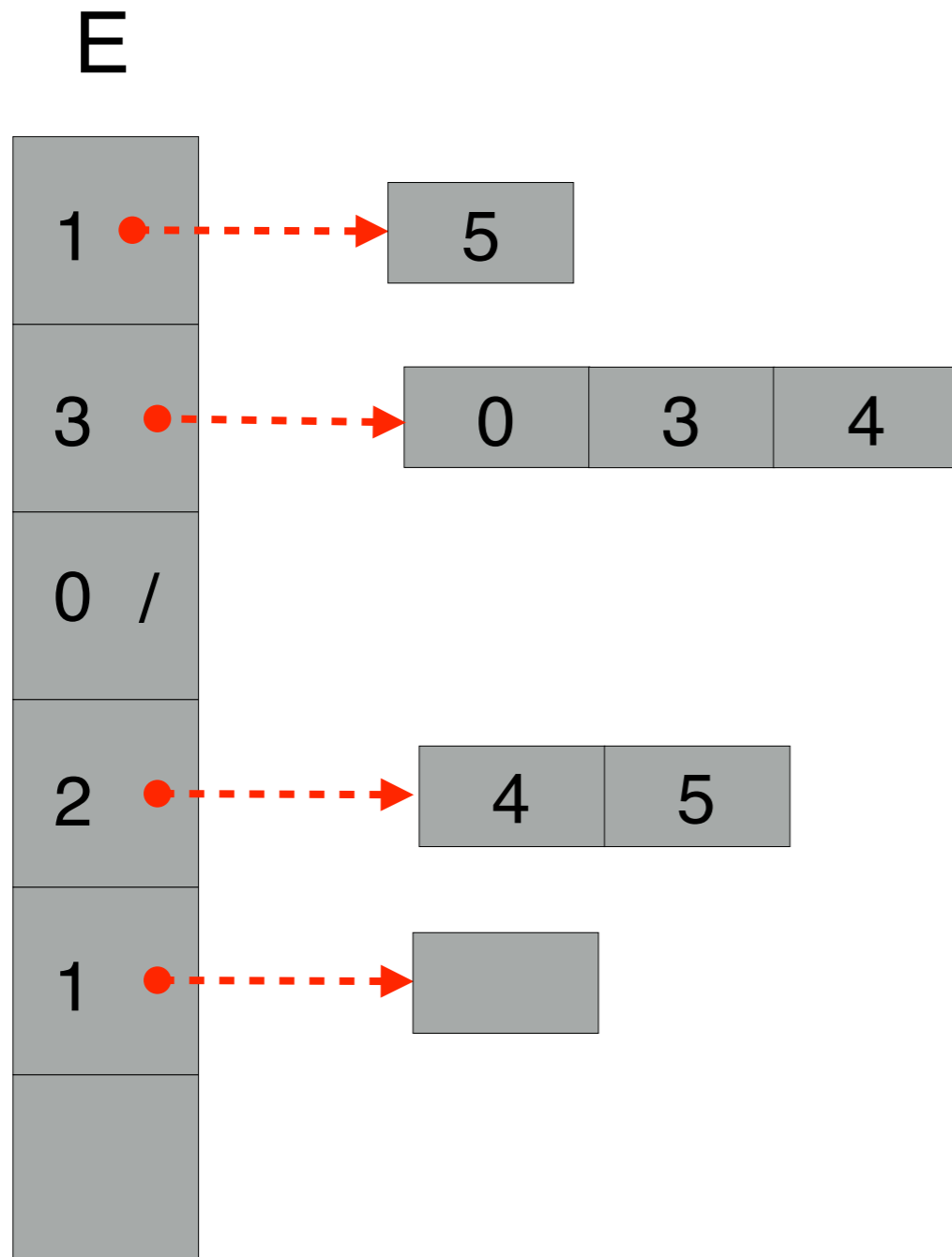
Esempio:

```
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:

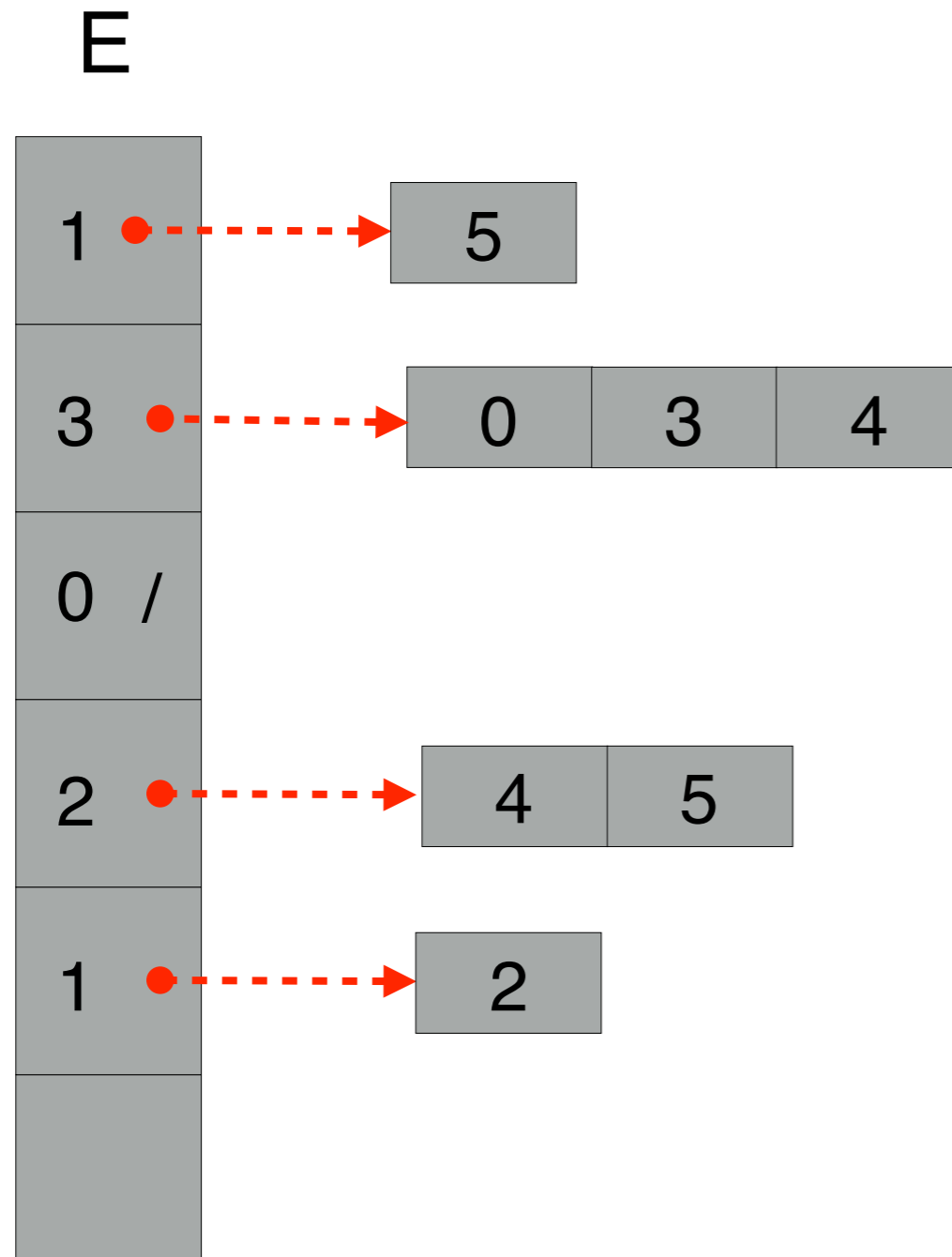
- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun nodo i , con $i \in [0, n)$, nel seguente formato:
 - numero n_i di archi uscenti da i ;
 - lista di n_i nodi destinazione, rappresentati con i numeri $[0, n)$.



Esempio:

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

Lettura grafo da input



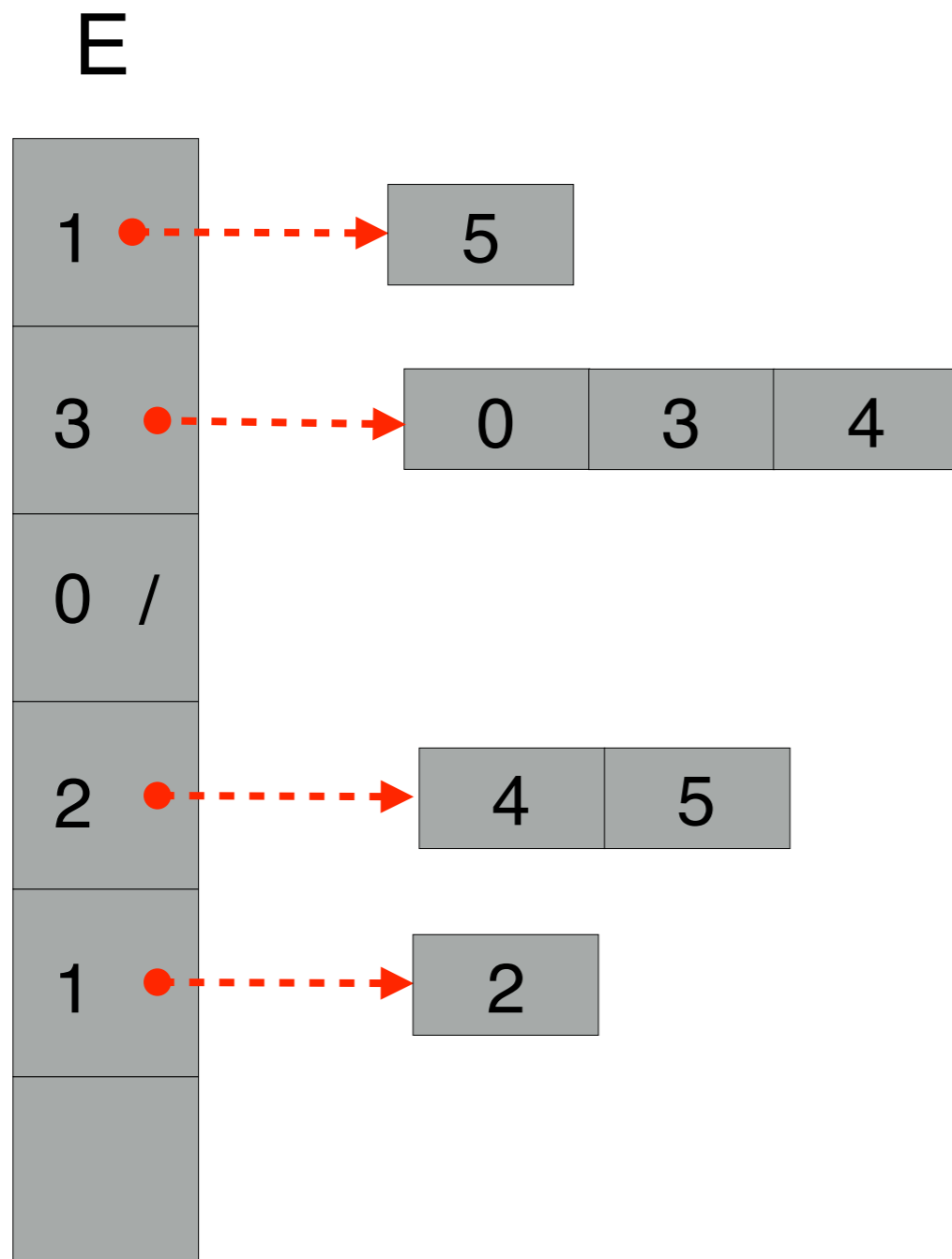
Negli esercizi sarà richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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 ciascun 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 $[0, n)$.

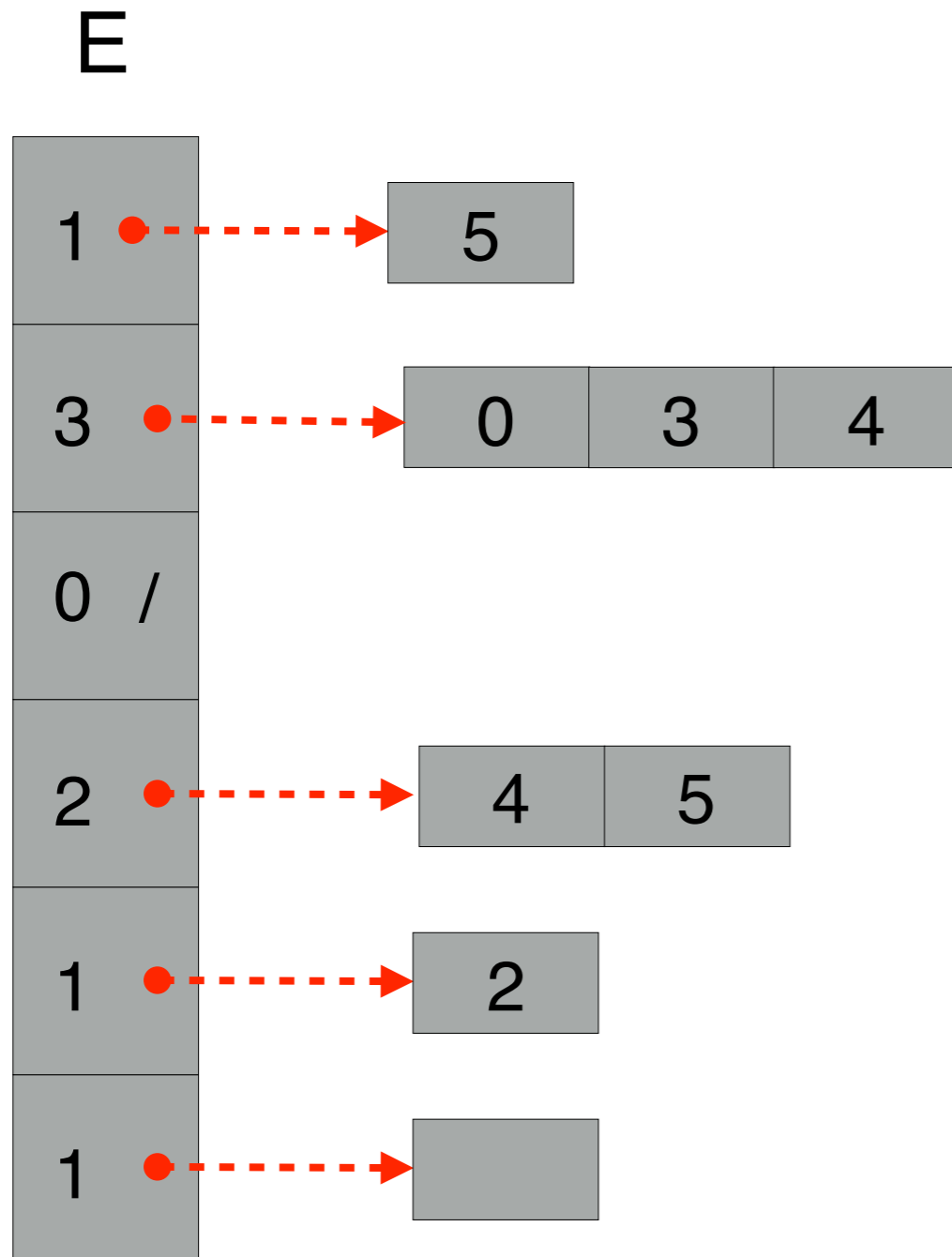
Esempio:

```
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:

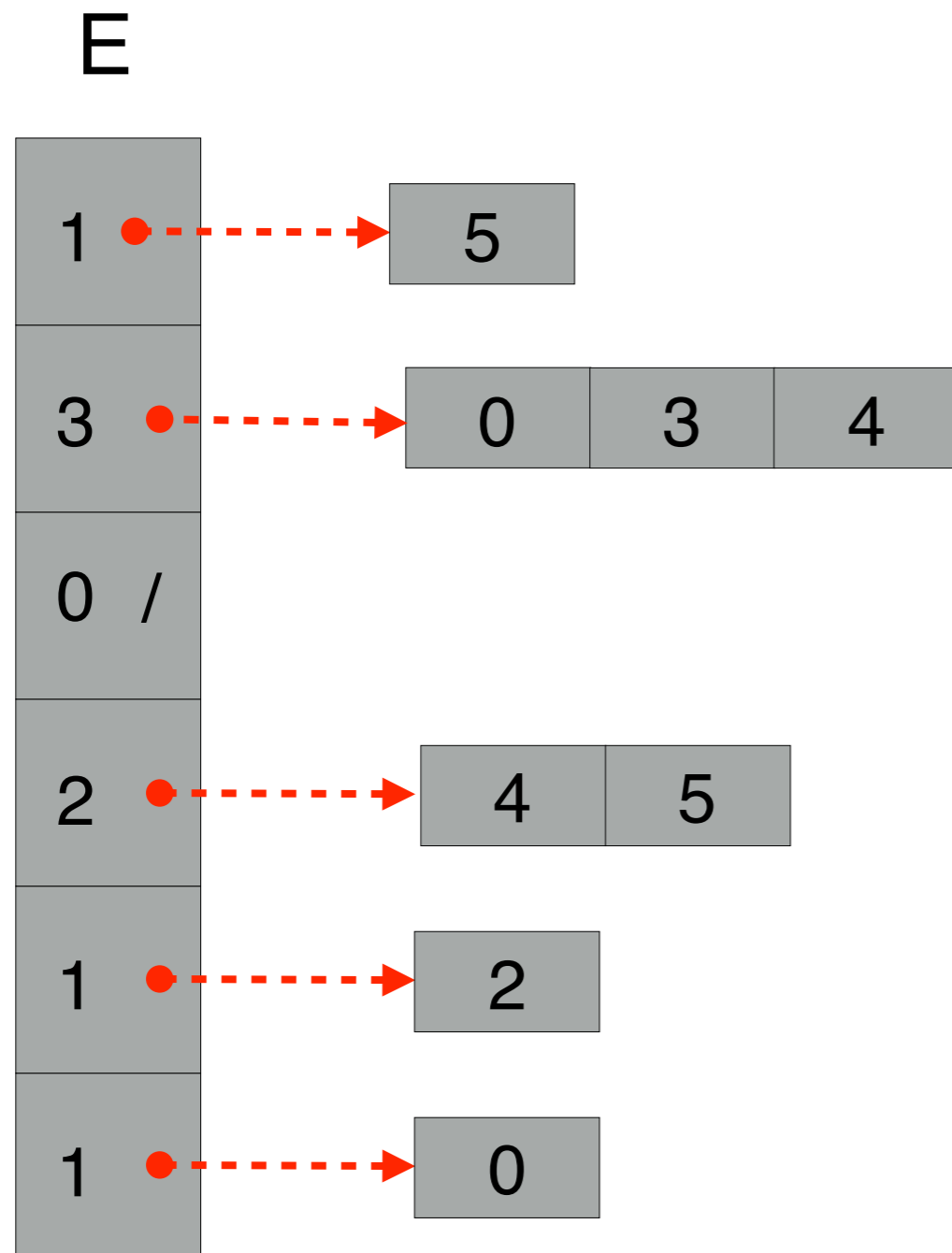
- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun nodo i , con $i \in [0, n)$, nel seguente formato:
 - numero n_i di archi uscenti da i ;
 - lista di n_i nodi destinazione, rappresentati con i numeri $[0, n)$.



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 ciascun 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, ne, i, j;

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

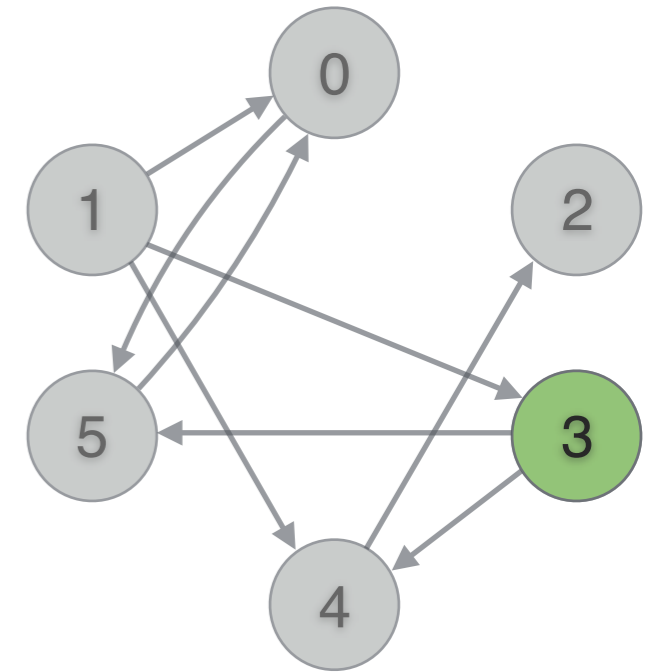
Negli esercizi sar  richiesto di leggere un file avente il seguente formato:

- una riga contenente il numero n di nodi del grafo;
- n righe, una per ciascun 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 $[0, n)$.

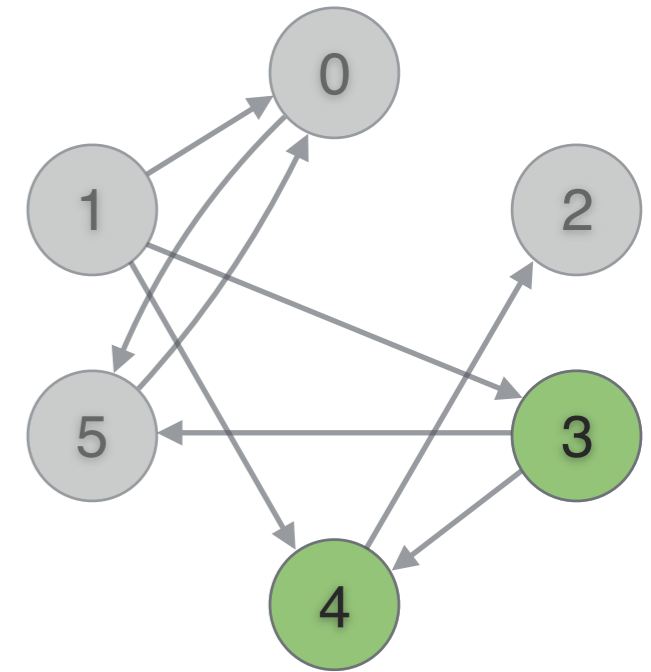
Esempio:

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

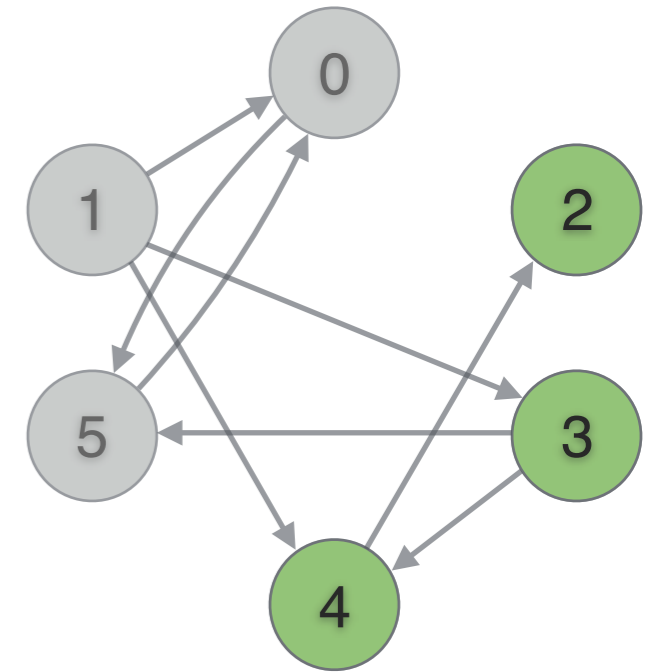
Visita in profondità



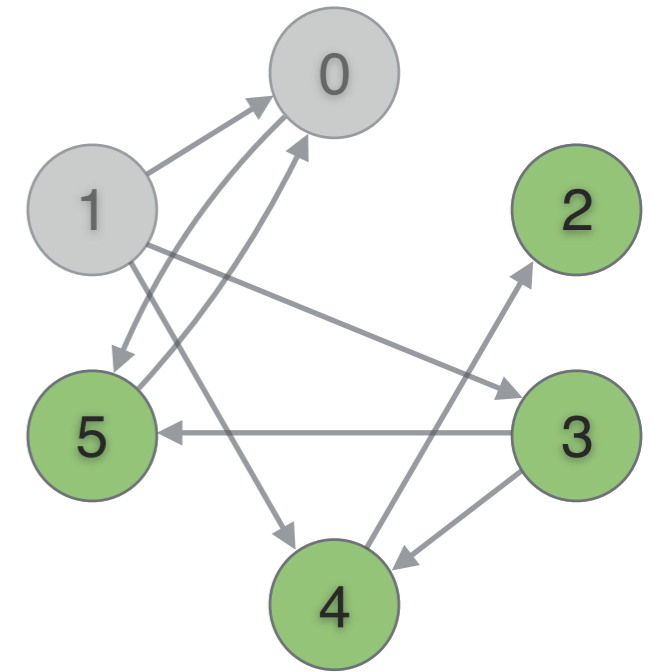
Visita in profondità



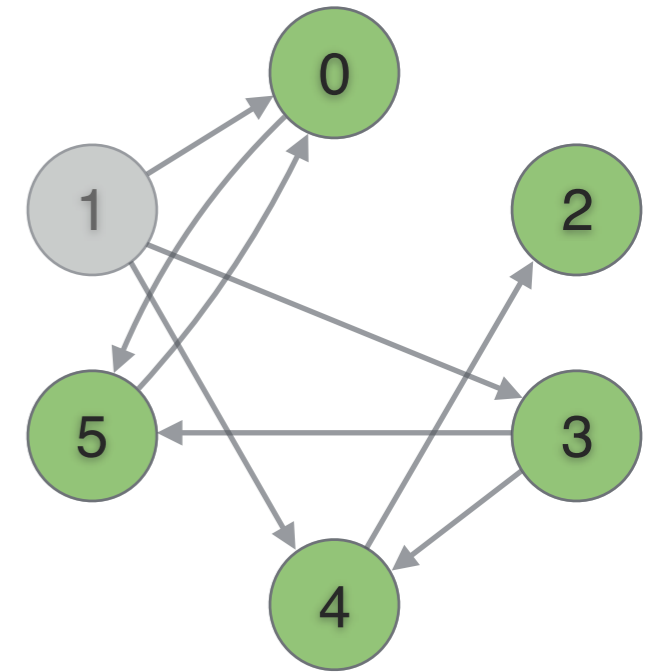
Visita in profondità



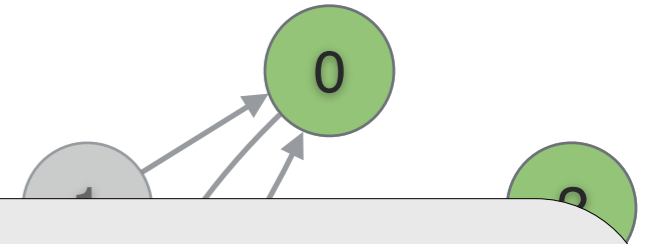
Visita in profondità



Visita in profondità



Visita in profondità

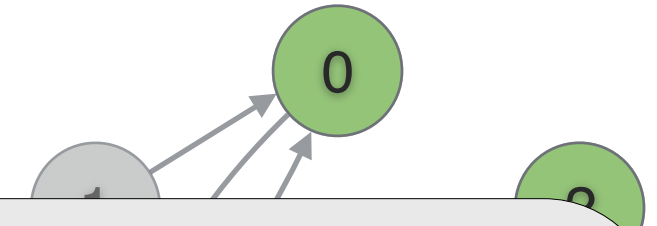


```
void recursive_dfs(
    edges *E, int src, int *colors
){
    int dest;
    for (int i=0; i < E[src].num_edges; ++i) {
        dest = E[src].edges[i];
        if (!colors[dest]) {
            colors[dest] = 1;
            recursive_dfs(dest, E, colors);
        }
    }
}

int * dfs(edges *E, int n, int from) {
    int * colors = (int *) malloc(sizeof(int)*n);
    // inizializzo i colori
    for (int i=0; i < n; ++i) colors[i] = 0;
    colors[from] = 1;
    // chiamata ricorsiva
    recursive_dfs(E, from, colors);
    return colors;
}
```


Visita in profondità

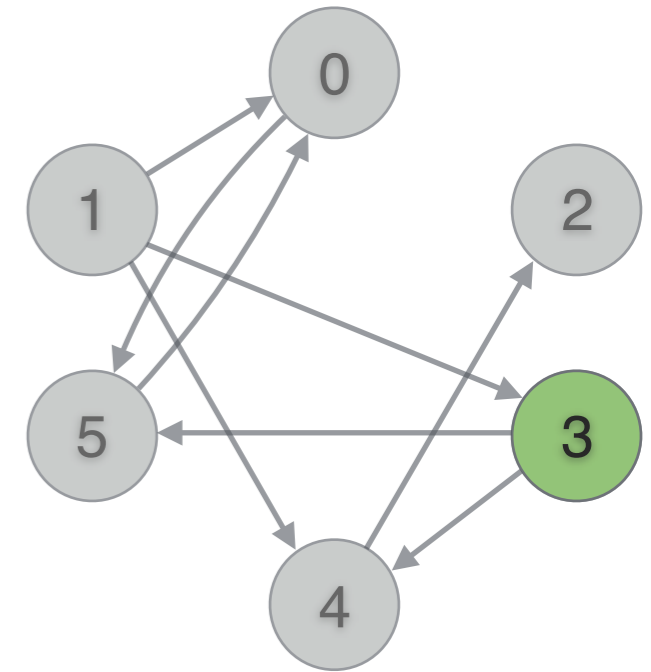
```
int * dfs(edges *E, int n, int from) {
    int * colors = (int *) malloc(sizeof(int) * n);
    int * stack = (int *) malloc(sizeof(int) * n);
    int stack_size, src, dest, i;
    // inizializzo i colori
    for (i=0; i < n; ++i) colors[i] = 0;
    colors[from] = 1;
    // inizializzo lo stack
 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



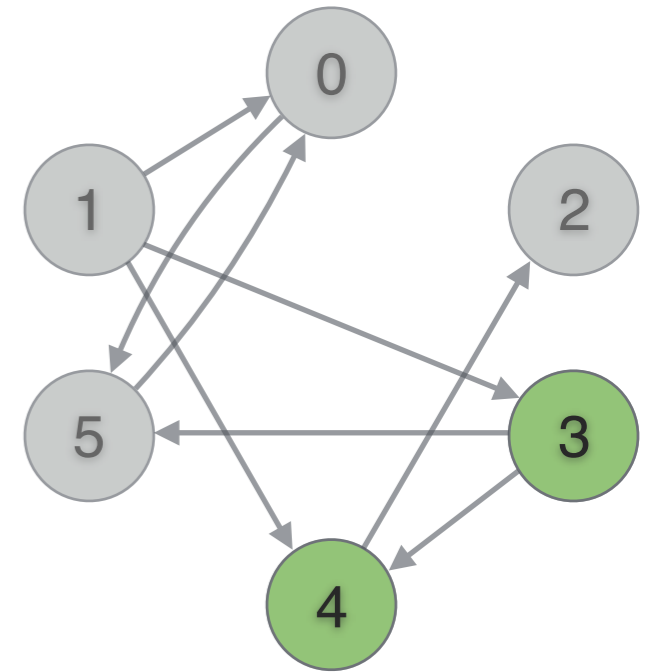
```
void recursive_dfs(
    edges *E, int src, int *colors
) {
    int dest;
    for (int i=0; i < E[src].num_edges; ++i) {
        dest = E[src].edges[i];
        if (!colors[dest]) {
            colors[dest] = 1;
            recursive_dfs(dest, E, colors);
        }
    }
}

int * dfs(edges *E, int n, int from) {
    int * colors = (int *) malloc(sizeof(int)*n);
    // inizializzo i colori
    for (int i=0; i < n; ++i) colors[i] = 0;
    colors[from] = 1;
    // chiamata ricorsiva
    recursive_dfs(E, from, colors);
    return colors;
}
```

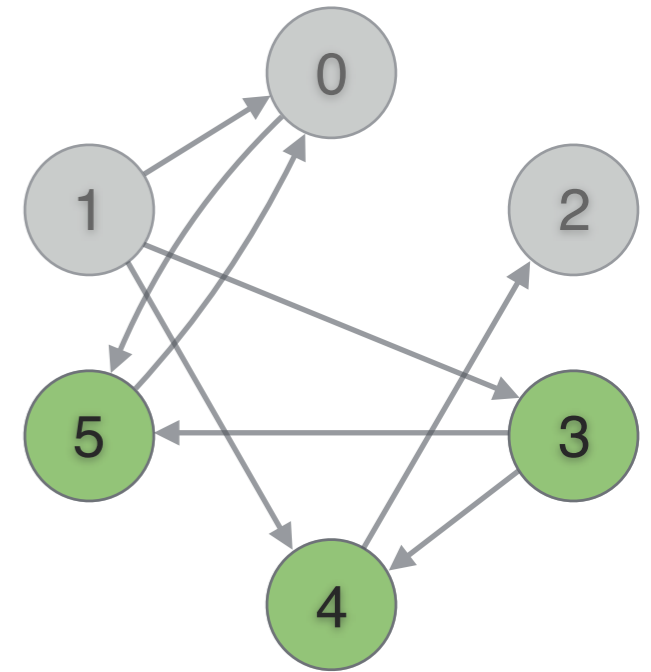
Visita in ampiezza



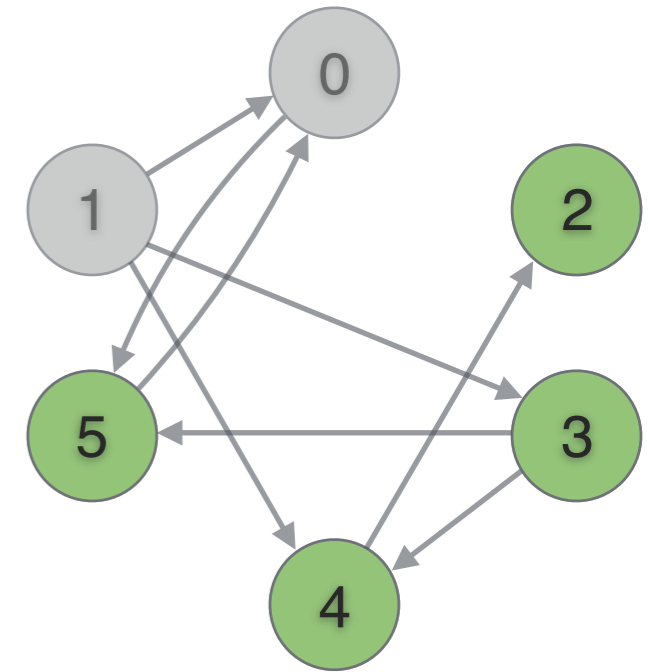
Visita in ampiezza



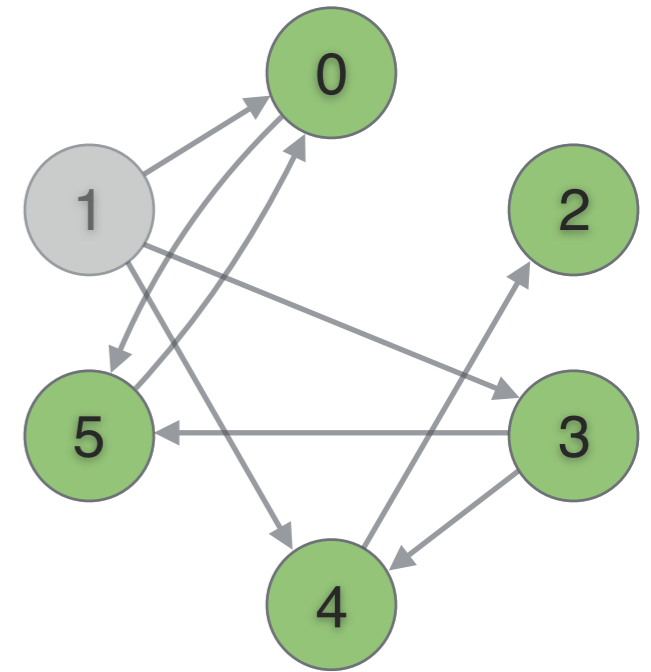
Visita in ampiezza



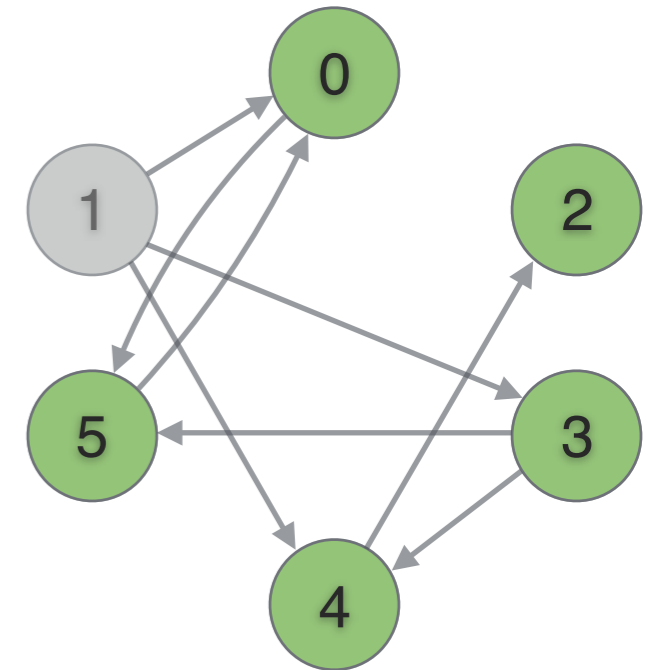
Visita in ampiezza



Visita in ampiezza



Visita in ampiezza

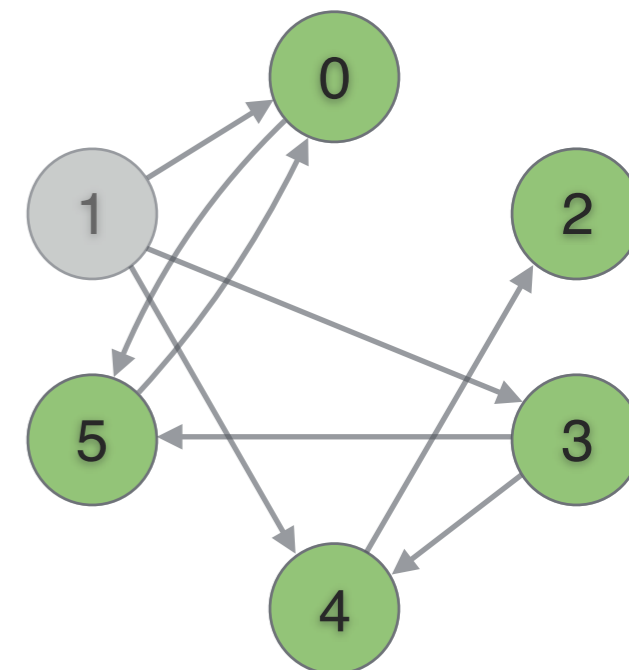


```
typedef struct _queue {
    int * values;
    int capacity;
    int head;
    int tail;
} queue;

void queue_init(queue * q, int capacity);
void queue_deinit(queue * q);
void queue_pushBack(queue * q, int value);
int queue_popFront(queue * q);
int queue_isEmpty(queue * q);
```

Visita in ampiezza

```
int * bfs(edges *E, int n, int from) {
    int * colors = (int *) malloc(sizeof(int) * n);
    queue q;
    int src, dest, i;
    // inizializzo i colori
    for (i=0; i < n; ++i) colors[i] = 0;
    colors[from] = 1;
    // inizializzo la coda
    queue_init(&q, n);
    queue_pushBack(&q, from);
    // loop fino a terminazione della coda
    while (!queue_isEmpty(&q)) {
        src = queue_popFront(&q);
        for (i=0; i < E[src].num_edges; ++i) {
            dest = E[src].edges[i];
            if (!colors[dest]) {
                colors[dest] = 1;
                queue_pushBack(&q, dest);
            }
        }
    }
    // libero la memoria
    queue_deinit(&q);
    return colors;
}
```

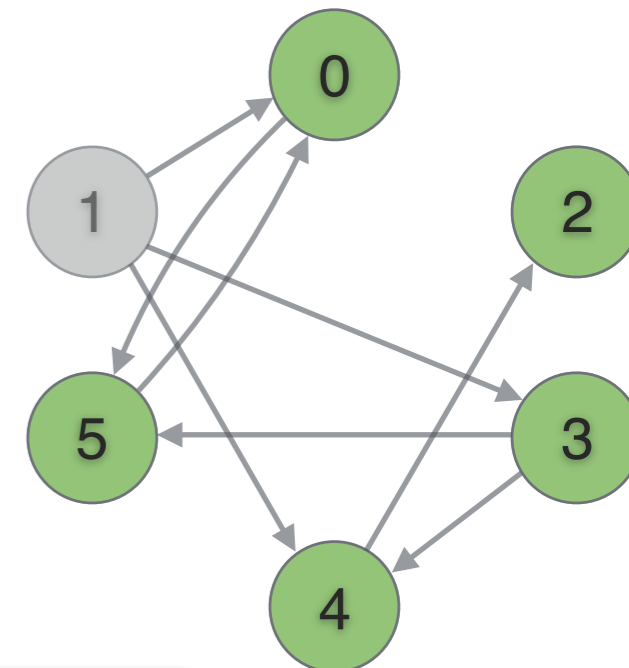


```
typedef struct _queue {
    int * values;
    int capacity;
    int head;
    int tail;
} queue;

void queue_init(queue * q, int capacity);
void queue_deinit(queue * q);
void queue_pushBack(queue * q, int value);
int queue_popFront(queue * q);
int queue_isEmpty(queue * q);
```


Visita in ampiezza

```
int * bfs(edges *E, int n, int from) {
    int * colors = (int *) malloc(sizeof(int) * n);
    queue q;
    int src, dest, i;
    // inizializzo i colori
    for (i=0; i < n; ++i) colors[i] = 0;
    colors[from] = 1;
    // inizializzo la coda
    queue_init(&q, n);
    queue_pushBack(&q, from);
    // loop fino a terminazione della coda
    while (!queue_isEmpty(&q)) {
        src = queue_popFront(&q);
        for (i=0; i < E[src].num_edges; ++i) {
            dest = E[src].edges[i];
            if (!colors[dest]) {
                colors[dest] = 1;
                queue_pushBack(&q, dest);
            }
        }
    }
    // libero la memoria
    queue_deinit(&q);
    return colors;
}
```



Da implementare...

```
typedef struct _queue {
    int * values;
    int capacity;
    int head;
    int tail;
} queue;

void queue_init(queue * q, int capacity);
void queue_deinit(queue * q);
void queue_pushBack(queue * q, int value);
int queue_popFront(queue * q);
int queue_isEmpty(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.