

# Lezione 9

## Alberi binari di ricerca

Rossano Venturini

[rossano.venturini@unipi.it](mailto:rossano.venturini@unipi.it)

Pagina web del corso

<http://didawiki.cli.di.unipi.it/doku.php/informatica/all-b/start>

# Esercizio 1

## Lista monodirezionale

Scrivere un programma che legga da tastiera una sequenza di  $n$  interi e li inserisca in una lista nell'ordine di immissione. La lista deve essere **monodirezionale**.

Il programma deve stampare il contenuto della lista percorrendola in ordine inverso.

# Esercizio 1

```
typedef struct _item {  
    struct _item* next;  
    int value;  
} item;
```

```
item* create_list(int value) {  
    item* l = (item*)malloc(sizeof(item));  
    l->value = value;  
    l->next = NULL;  
    return l;  
}
```

```
item* append(item* last_elem, int value) {  
    last_elem->next = (item*) malloc(sizeof(item));  
    last_elem->next->value = value;  
    last_elem->next->next = NULL;  
    return last_elem->next;  
}
```

# Esercizio 1

```
void print_list(item* l) {  
    if (l == NULL) return;  
    printf("%d\n", l->value);  
    print_list(l->next);  
    return;  
}
```

```
void print_reversed_list(item* l) {  
    if (l == NULL) return;  
    print_reversed_list(l->next);  
    printf("%d\n", l->value);  
    return;  
}
```

# Esercizio 1

```
int main(void) {  
    ...  
    l = create_list(val);  
    next = l; len--;  
    while (len > 0) {  
        scanf("%d", &val);  
        next = append(next, val);  
        len--;  
    }  
    print_reversed_list(l);  
  
    return 0;  
}
```

# Esercizio 2

## Lista bidirezionale

Scrivere un programma che legga da tastiera una sequenza di  $n$  interi e li inserisca in una lista nell'ordine di immissione. La lista deve essere **bidirezionale**.

Il programma deve stampare il contenuto della lista percorrendola in ordine inverso.

# Esercizio 2

```
typedef struct __node {
    struct __node* next;
    struct __node* prev;
    int value;
} item;

list create_list(int value) {
    item* l = (item*)malloc(sizeof(item));
    l->value = value;
    l->next = NULL;
    l->prev = NULL;
    return l;
}

item* append(item* last_elem, int value) {
    last_elem->next = (item*)malloc(sizeof(item));
    last_elem->next->value = value;
    last_elem->next->next = NULL;
    last_elem->next->prev = last_elem;
    return last_elem->next;
}
```

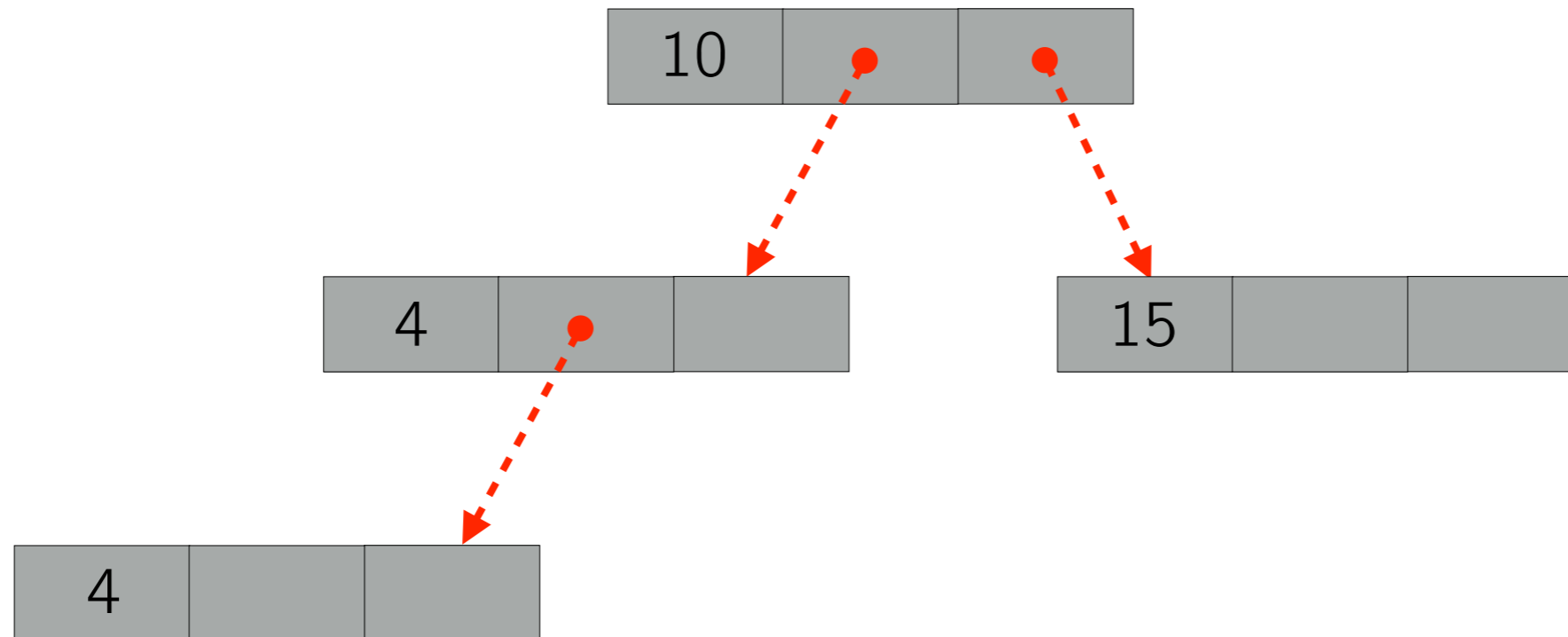
# Esercizio 2

```
void print_list(item* l) {  
    if (l == NULL) return;  
    printf("%d\n", l->value);  
    print_list(l->next);  
    return;  
}
```

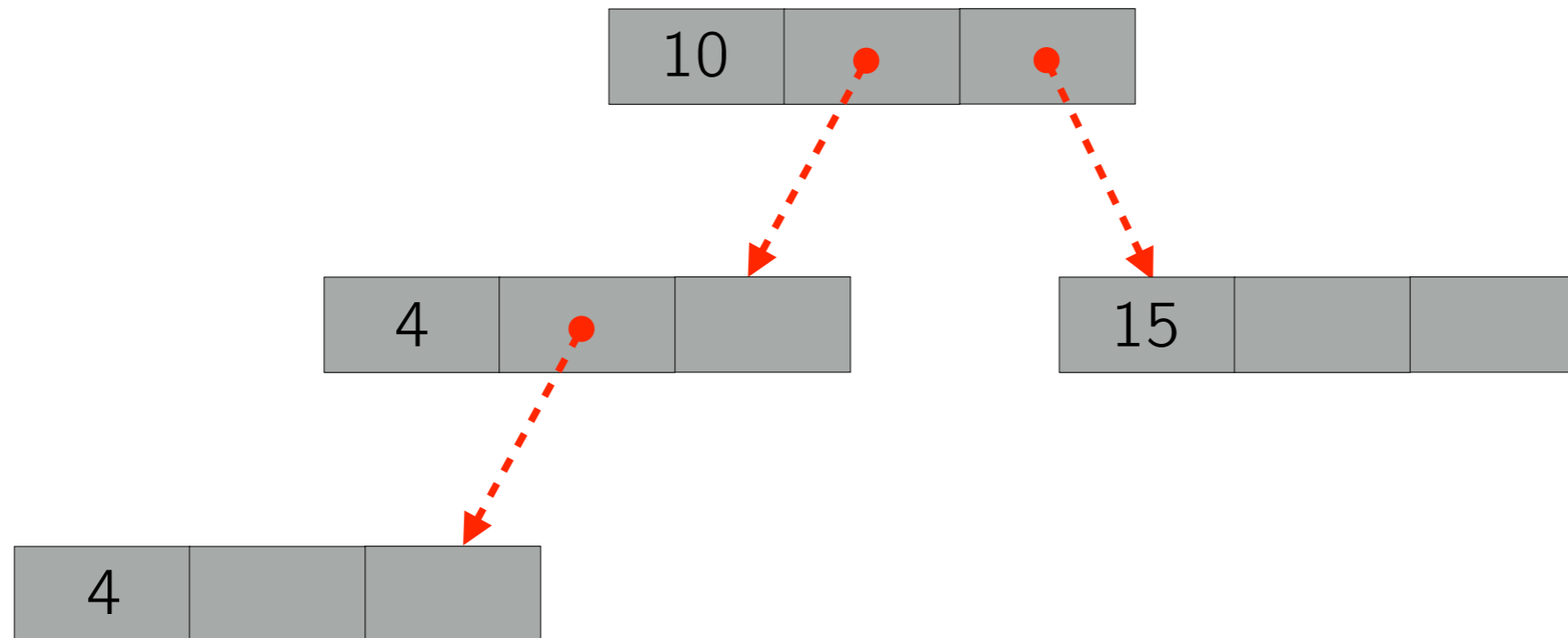
```
void print_reversed_list(item* last_elem) {  
    while (last_elem != NULL) {  
        printf("%d\n", last_elem->value);  
        last_elem = last_elem->prev;  
    }  
}
```



# Alberi binari di ricerca (ABR)

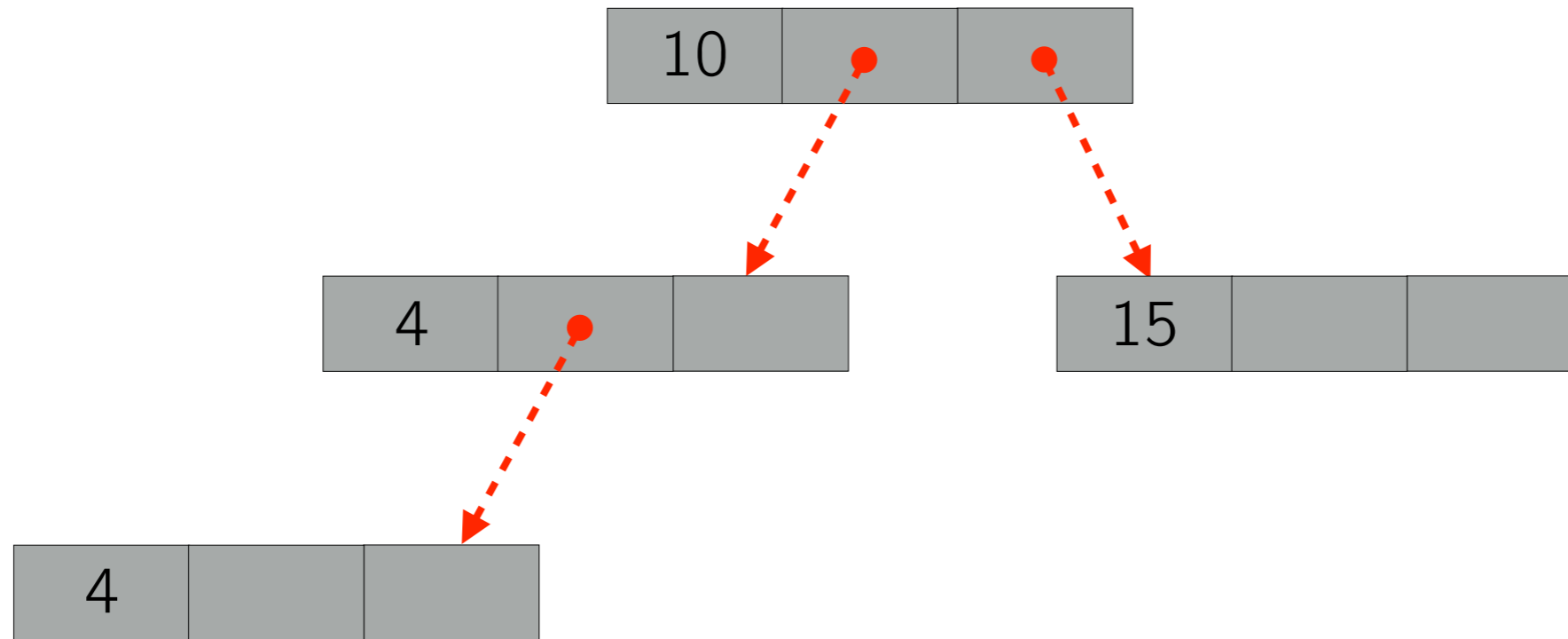


# Alberi binari di ricerca (ABR)

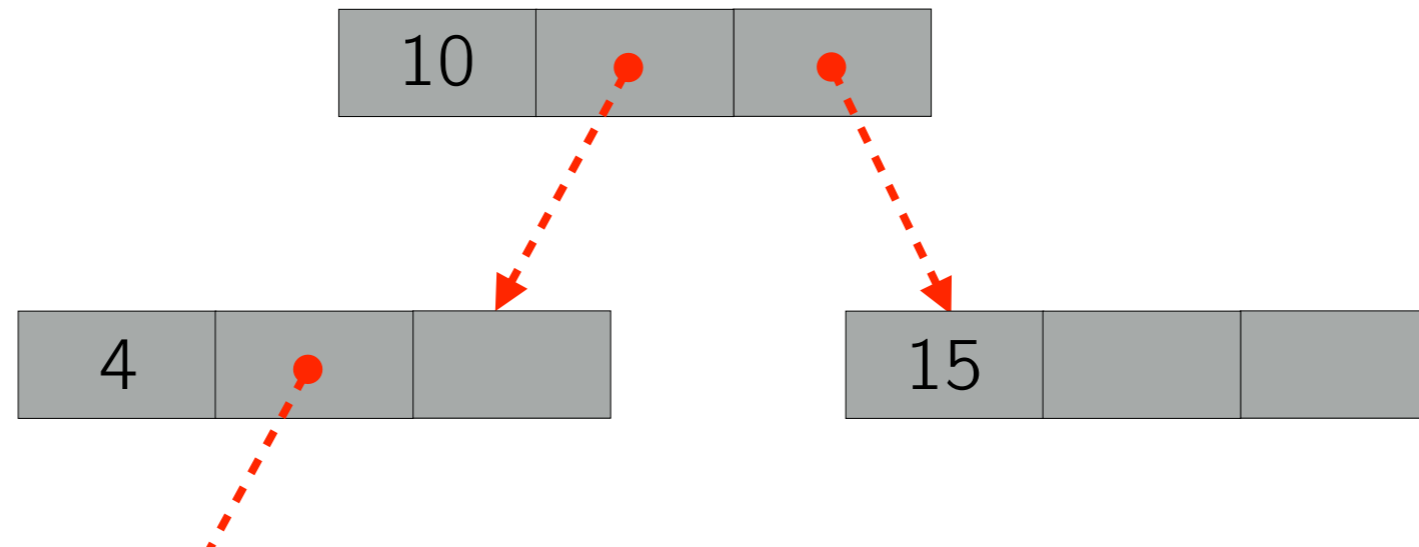


```
typedef struct _nodo {  
    int key;  
    struct _nodo *left;  
    struct _nodo *right;  
} nodo;
```

# ABR: Ricerca

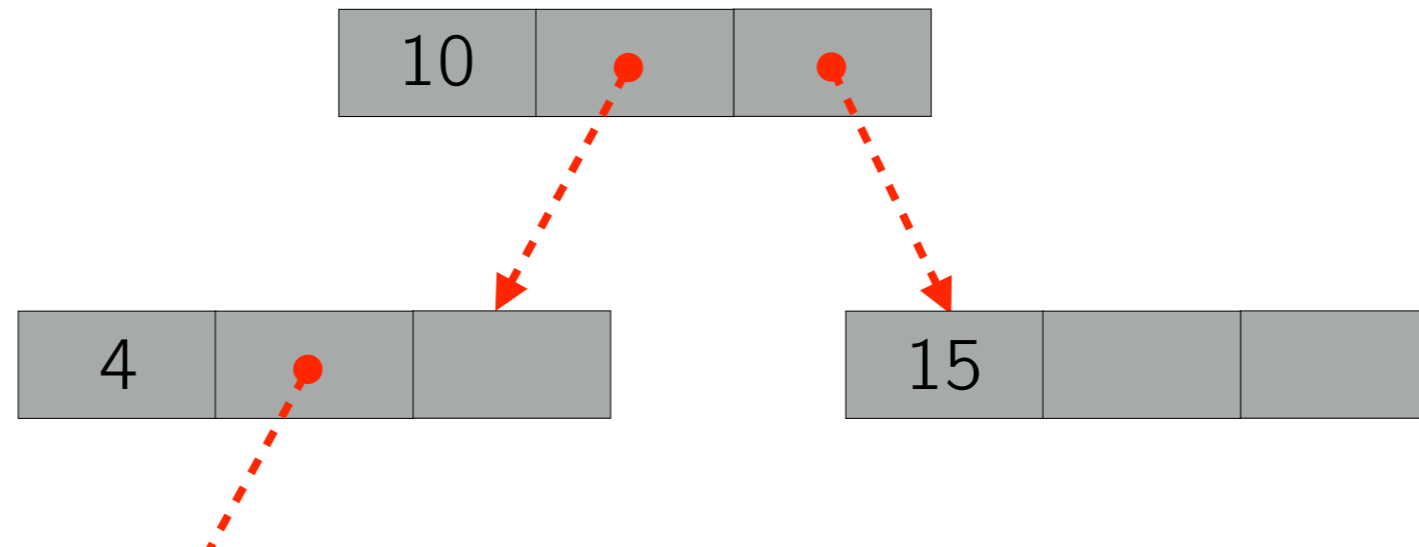


# ABR: Ricerca



```
int cerca(nodo* t, int key) {  
    int depth = 0;  
    nodo* current = t;  
    while (current != NULL) {  
        if (key == current->key) return depth;  
        if (key > current->key)  
            current = current->right;  
        else  
            current = current->left;  
        depth++;  
    }  
    return -1;  
}
```

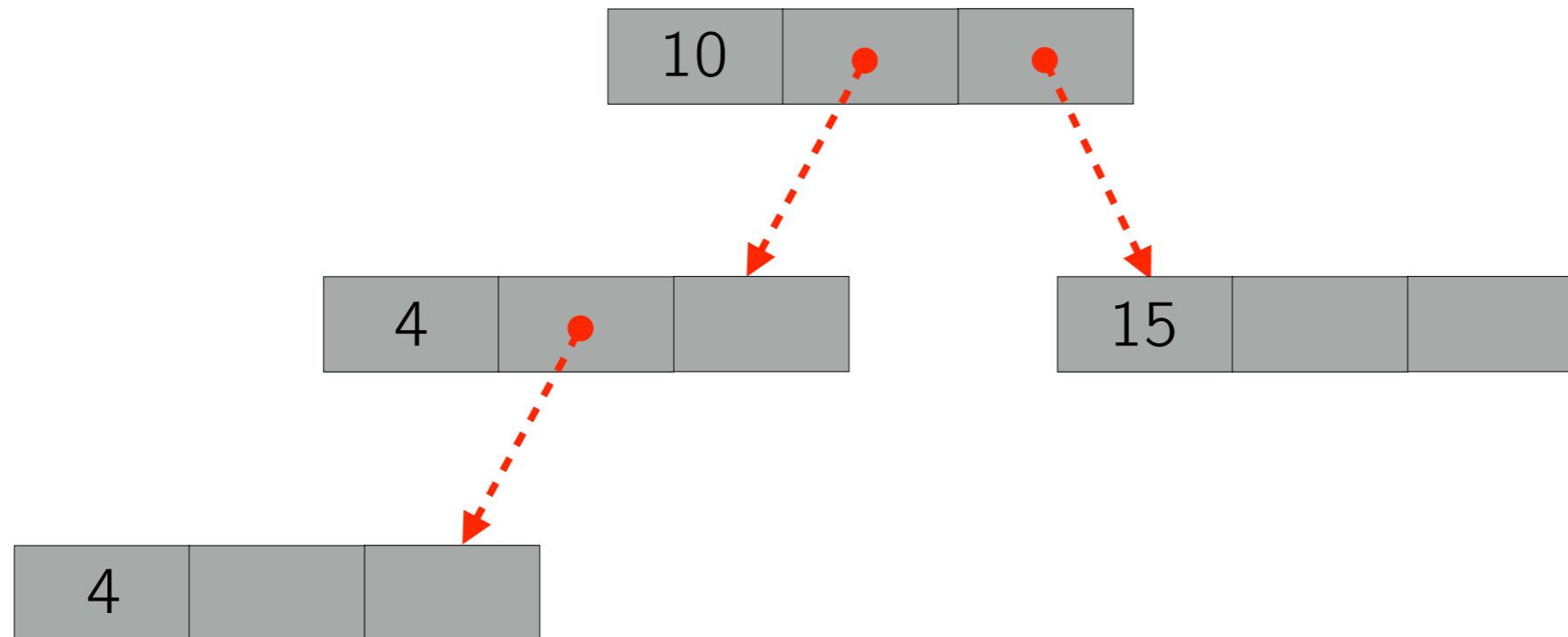
# ABR: Ricerca



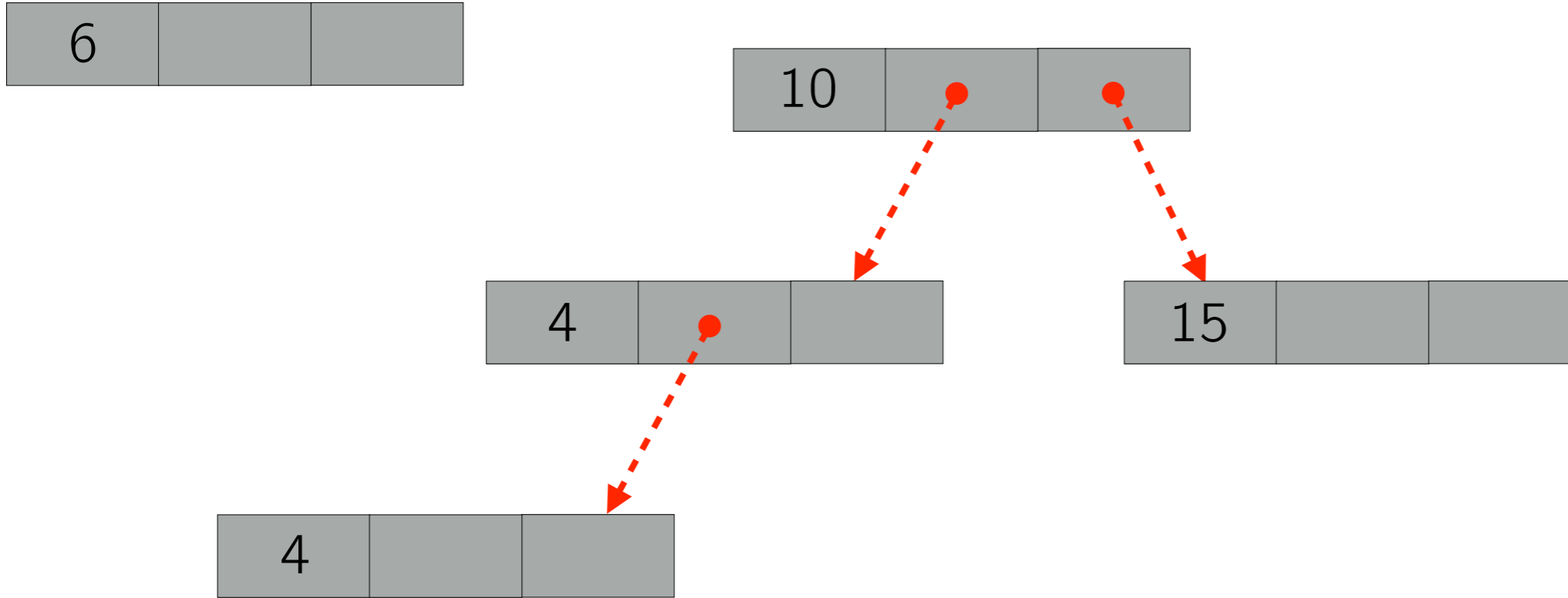
```
int cerca(nodo* t, int key) {  
    int depth = 0;  
    nodo* current = t;  
    while (current != NULL) {  
        if (key == current->key) return depth;  
        if (key > current->key)  
            current = current->right;  
        else  
            current = current->left;  
        depth++;  
    }  
    return -1;  
}
```

```
int cercaRic(nodo* t, int key) {  
    if (t == NULL) return -1;  
    if (key == t->key) return 0;  
    int found = -1;  
    if (key > t->key)  
        found = cercaRic(t->right, key);  
    else  
        found = cercaRic(t->left, key);  
  
    if (found >= 0) return 1 + found;  
    else return -1;  
}
```

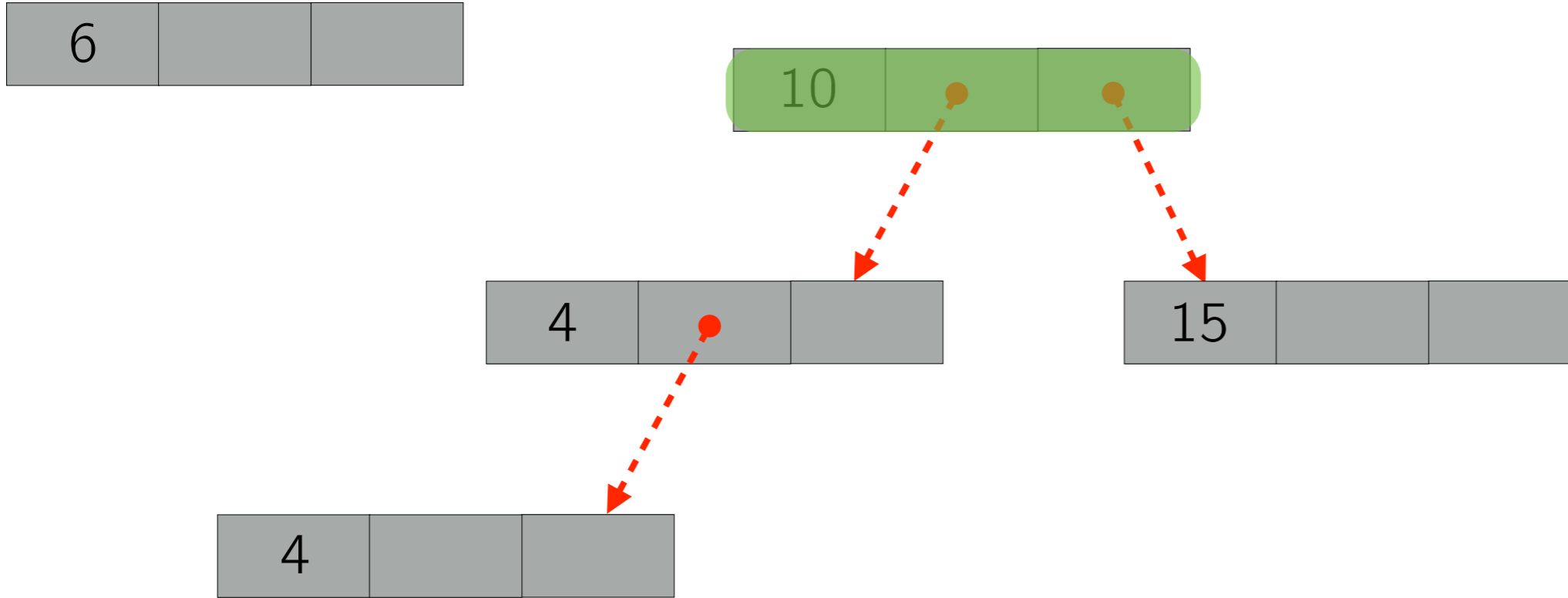
# ABR: Inserimento



# ABR: Inserimento

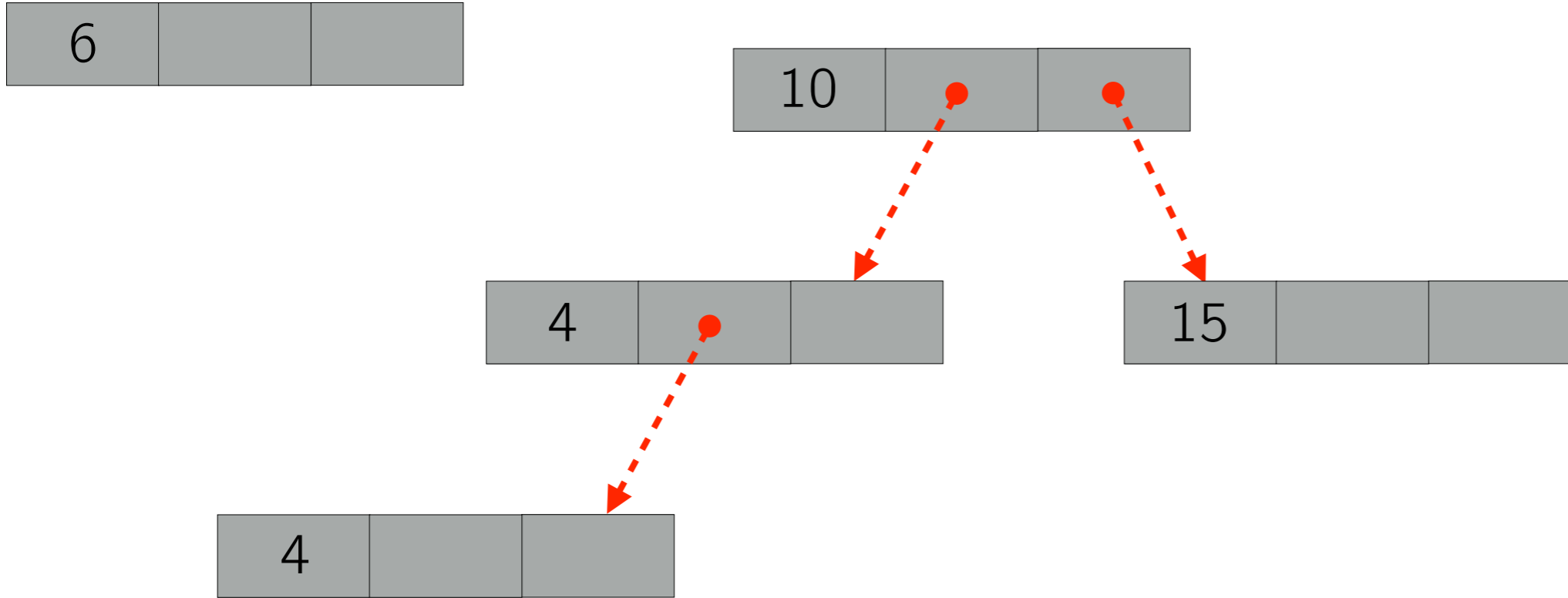


# ABR: Inserimento

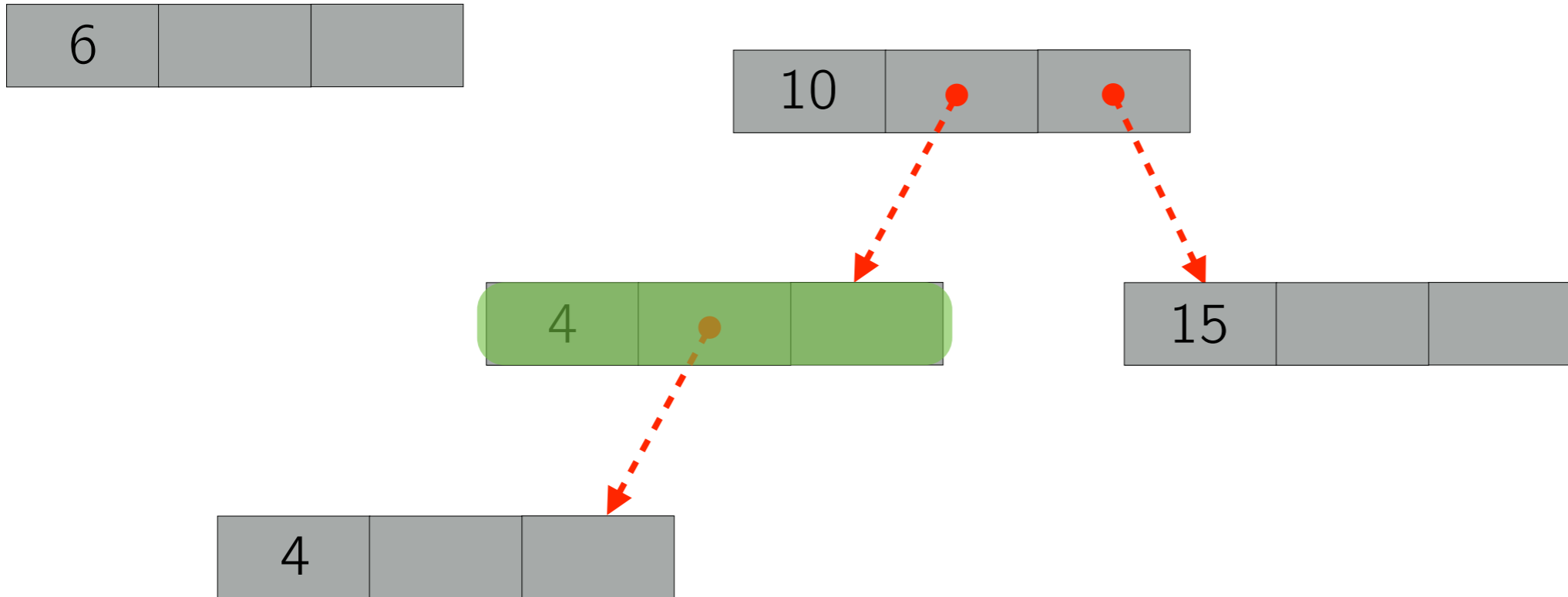




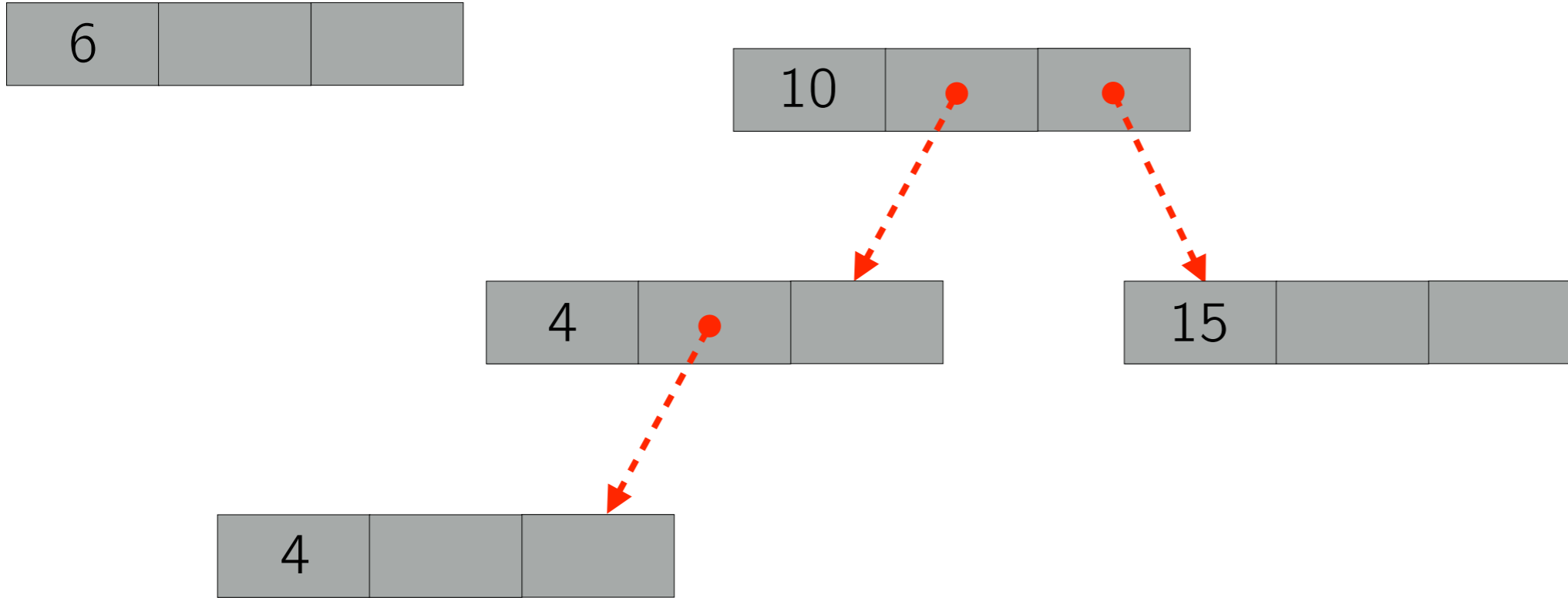
# ABR: Inserimento



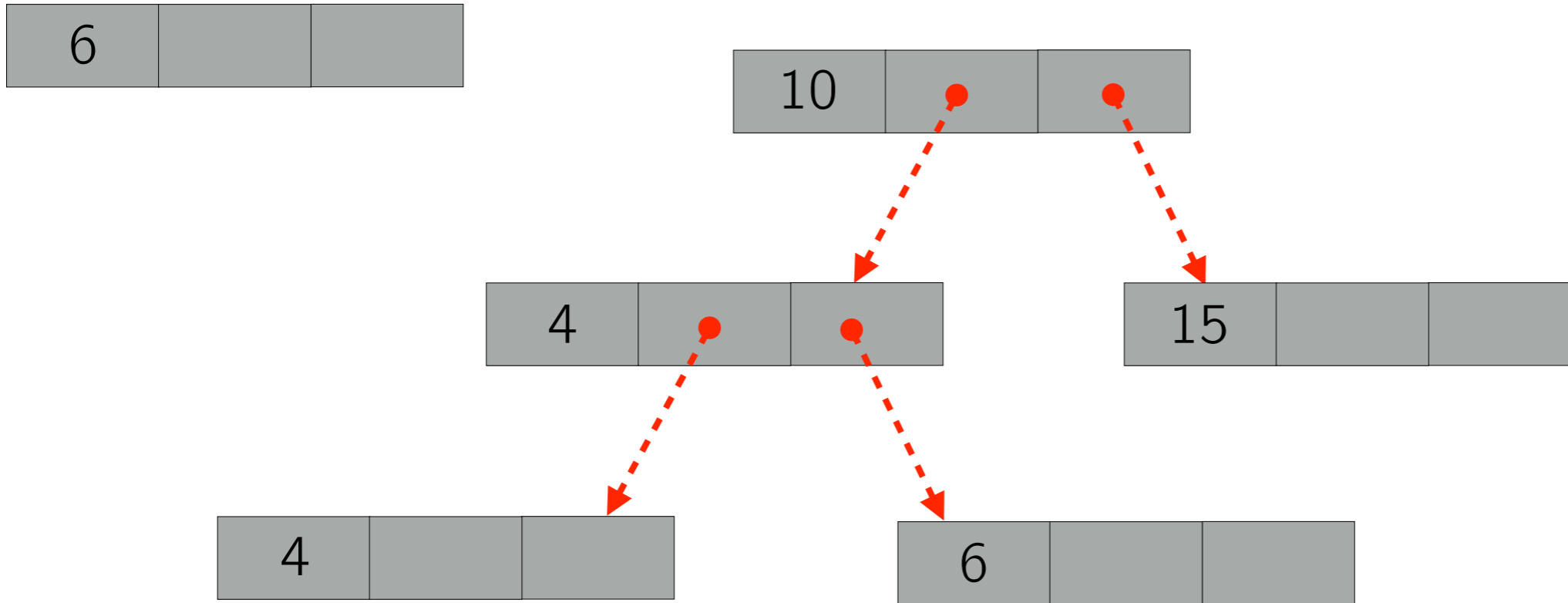
# ABR: Inserimento



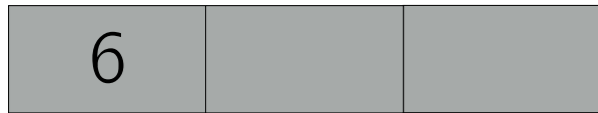
# ABR: Inserimento



# ABR: Inserimento

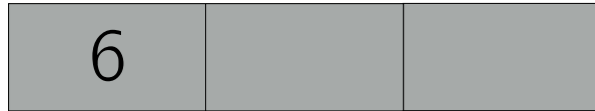


# ABR: Inserimento



```
nodo* inserisci(nodo *t, int key) {  
    nodo* new = (nodo *) malloc(sizeof(nodo));  
    new->key = key;  
    new->right = NULL; new->left = NULL;  
    if (t == NULL) { return new; }  
    nodo* parent;  
    nodo* current = t;  
    while (current != NULL) {  
        parent = current;  
        if (current->key < key)  
            current = current->right;  
        else current = current->left;  
    }  
    if (parent->key < key) parent->right = new;  
    else parent->left=new;  
    return t;  
}
```

# ABR: Inserimento



```
nodo* inserisci(nodo *t, int key) {
    nodo* new = (nodo *) malloc(sizeof(nodo));
    new->key = key;
    new->right = NULL; new->left = NULL;
    if (t == NULL) { return new; }
    nodo* parent;
    nodo* current = t;
    while (current != NULL) {
        parent = current;
        if (current->key < key)
            current = current->right;
        else current = current->left;
    }
    if (parent->key < key) parent->right = new;
    else parent->left=new;
    return t;
}
```

```
nodo* inserisciRic(nodo *t, int key) {
    if (t == NULL) {
        nodo* new = (nodo *) malloc(sizeof(nodo));
        new->key=key;
        new->left=NULL;
        new->right=NULL;
        return new;
    }

    if(t->key < key)
        t->right = inserisciRic(t->right, key);
    else
        t->left = inserisciRic(t->left, key);
    return t;
}
```

# Esercizio 1

## ABR: Ricerca

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 entra poi in un ciclo infinito nel quale legge un intero  $i$  da tastiera e lo cerca nell'albero. Il ciclo termina solo se l'intero  $i$  inserito è strettamente minore di 0. Se  $i$  si trova nell'albero stampa la profondità alla quale l'elemento si trova (contando da 0), altrimenti stampa NO.

# 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 3

## ABR: Ordinamento

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 quindi utilizzare un'opportuna visita dell'albero per stampare gli interi della sequenza in ordine non decrescente.

# Esercizio 4

## ABR: Confronto (prova del 11/06/2011)

Scrivere un programma che riceve in input due sequenze di  $N$  interi positivi, dalle quali devono essere costruiti due alberi binari di ricerca NON bilanciati (un albero per sequenza).

Al programma viene data una chiave intera  $K$ . Si può assumere che l'intero  $K$  sia presente in entrambe le sequenze. Il programma deve verificare che le sequenze di chiavi incontrate nel cammino che dalla radice porta al nodo con chiave  $K$  nei due alberi coincidano.

# 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 6 (Facoltativo)

La ditta Serling Enterprises compra lunghe barre d'acciaio e le taglia in barrette più piccole, che poi rivende. L'operazione di taglio non comporta costi aggiuntivi. La direzione della Serling Enterprises vuole scoprire la maniera migliore per tagliare le barre, in maniera da massimizzare il guadagno.

Detta  $n$  la lunghezza in pollici di una barra da tagliare, supponiamo di conoscere, per  $i = 1, 2, \dots, n$ , il prezzo  $p_i$  in dollari a cui l'azienda vende una barra di lunghezza  $i$  pollici. Le lunghezze delle barre sono sempre uguali ad un numero intero di pollici. Un esempio è dato dalla tabella seguente:

length $i$	1	2	3	4	5	6	7	8	9	10
price $p_i$	1	5	8	9	10	17	17	20	24	30

Data una barra di lunghezza  $n$  pollici e una tabella dei prezzi  $p_i$  per  $i = 1, 2, \dots, n$ , si vuole determinare una maniera di tagliare la barra in modo da raggiungere il profitto massimo  $r_n$  ottenibile tagliando la barra e vendendone i pezzi. Si noti che se il prezzo  $p_n$  di una barra di lunghezza  $n$  è sufficientemente grande, allora una soluzione ottima potrebbe non richiedere alcun taglio. Si noti inoltre che in generale esistono più soluzioni ottime, tutte ovviamente caratterizzate dallo stesso valore  $r_n$ .

Consideriamo il caso in cui  $n = 4$ . La figura 1 mostra tutti i modi possibili di tagliare una barra di 4 pollici, incluso il caso che non prevede tagli. Facendo riferimento alla tabella dei prezzi precedente, sopra ciascuna immagine sono mostrati i prezzi delle barrette risultanti. Si vede facilmente che tagliare la barra in due barrette lunghe 2 pollici produce il guadagno massimo pari a  $p_2 + p_2 = 5 + 5 = 10$  dollari.

# Puzzle

## Griglia infetta

Abbiamo una griglia  $8 \times 8$  e la possibilità di infettare 7 delle sue celle. Ad ogni intervallo di tempo, vengono infettate tutte le celle che hanno almeno due lati infetti. Il processo termina non appena non sono più possibili nuove infezioni. Mostrare come scegliere le prime 7 celle in modo che alla fine del processo l'intera griglia sia infetta, o dimostrare che una tale configurazione non può esistere.

