

Java: pile e code (*stacks and queues*)

Liceo G.B. Brocchi

Classi quarte Scientifico - opzione scienze applicate

Bassano del Grappa, Novembre 2022

Prof. Giovanni Mazzocchin

Stack - implementazione *array-based*

- Realizziamo un tipo che rappresenti la struttura dati *stack* (pila) di interi tramite un array di interi (o di caratteri, o di qualche altro tipo)
- Uno stack è una struttura dati *LIFO* (*Last in, first out*), che mette a disposizione solo 2 operazioni:
 - **push(item)**: aggiunge item in cima alla pila e restituisce il nuovo *stack pointer*
 - **pop()**: rimuove l'elemento dalla cima della pila
 - **peek()**: restituisce l'elemento alla cima della pila senza rimuoverlo (operazione non indispensabile, aggiunta per comodità)
- Per realizzare uno stack è necessario tenere traccia dell'indice della cima della pila (*top*, spesso chiamato *stack pointer*). Nell'implementazione *array-based* **top** sarà semplicemente un indice intero.

Stack - implementazione *array-based*

```
class StackChar{  
    private char[] stack_array;  
    private int top;  
    private int size;  
  
    public StackInt(int size){  
        this.stack_array = new char[size];  
        this.top = -1;  
        this.size = size;  
    }  
}
```

Provate a immaginare cosa potrebbe succedere se i membri qui sopra non fossero privati

Stack - implementazione *array-based*

- L'interfaccia pubblica sarà costituita dai metodi:
 - *int push(int item)* → restituisce il nuovo *stack pointer*
 - *int pop()* → restituisce il nuovo *stack pointer*
 - *int peek()* → restituisce l'elemento alla cima della pila

```
public int push(char item){  
    if (top < size-1){  
        top++;  
        stack_array[top] = item;  
    }  
  
    return top;  
}
```

Stack - implementazione *array-based*

- L'interfaccia pubblica sarà costituita dai metodi:
 - *int push(int item)* → restituisce il nuovo *stack pointer*
 - *int pop()* → restituisce il nuovo *stack pointer*
 - *int peek()* → restituisce l'elemento alla cima della pila

```
public int pop(){  
    if (top >= 0){  
        top--;  
    }  
  
    return top;  
}
```

Stack - implementazione *array-based*

- L'interfaccia pubblica sarà costituita dai metodi:
 - *int push(int item)* → restituisce il nuovo *stack pointer*
 - *int pop()* → restituisce il nuovo *stack pointer*
 - *int peek()* → restituisce l'elemento alla cima della pila

```
public char peek() throws StackPeekException{  
    if (top >= 0){  
        return stack_array[top];  
    }  
    else {  
        throw new StackPeekException();  
    }  
}
```

Stack - implementazione *array-based*

- L'interfaccia pubblica sarà costituita dai metodi:
 - *boolean is_empty();*
 - *boolean is_full();*

```
public boolean is_empty(){  
    return top == -1;  
}
```

```
public boolean is_full(){  
    return top == size - 1;  
}
```

Stack - implementazione *array-based*

- Vogliamo verificare se una stringa che rappresenta un'espressione algebrica è bilanciata a livello di parentesi tonde (ad ogni parentesi aperta ne deve corrispondere una chiusa)
- I compilatori (gli analizzatori di sintassi in generale) devono effettuare controlli di questo tipo per verificare la correttezza della sintassi del codice sorgente scritto dal programmatore
- Esempi:
 - $((a) + (b - a) * (b + ((c - d))))$ è bilanciata
 - $()))(($ non è bilanciata
 - l'espressione vuota è bilanciata
 - $(2*3))$ non è bilanciata

Stack – implementazione *linked-list-based*

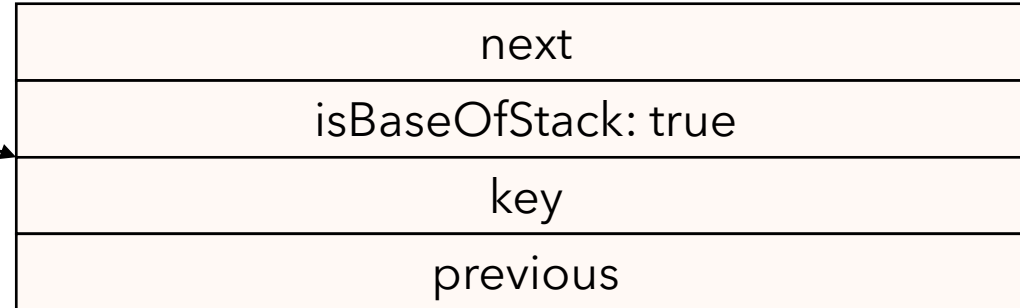
- Implementiamo uno stack **dinamico**, la cui dimensione varia in base alle necessità del programma a *runtime*
- Possiamo utilizzare una lista concatenata, sulla quale è possibile inserire nodi solo in coda (*push*) e rimuovere soltanto l'ultimo nodo (*pop*)
- Nelle liste concatenate viste nelle scorse lezioni era invece possibile aggiungere e rimuovere un nodo in qualsiasi posizione
- Creiamo uno stack con un nodo *base* fisso, che una volta creato non può essere *poppato*. Questo per poter invocare i metodi della classe senza incorrere in *NullPointerException*
- L'interfaccia pubblica restituirà sempre lo *stack pointer*, ossia un riferimento alla nodo in cima allo stack
- Le operazioni *push* e *pop* devono richiedere un tempo di esecuzione *costante*, ossia indipendente dal numero di elementi presenti nello stack
 - eg) una *push* su uno stack di 10^2 nodi deve costare come una *push* su uno stack contenente 10^4 nodi

Stack – implementazione *linked-list-based*

- **NB:** per eseguire *push* e *pop* non dobbiamo scorrere tutto lo stack partendo dalla base!
 - se fosse così, *push* e *pop* avrebbe un costo non costante, ma **lineare** sul numero di nodi presenti nello stack
- Uno stack è una struttura dati (**Abstract Data Type**) **LIFO**, quindi ci interessa avere sempre a disposizione un puntatore alla sua cima, detto **stack pointer** (nella versione *array-based* lo chiamavamo indice *top* perché era solo un intero e non un vero puntatore)

Stack – implementazione *linked-list-based*

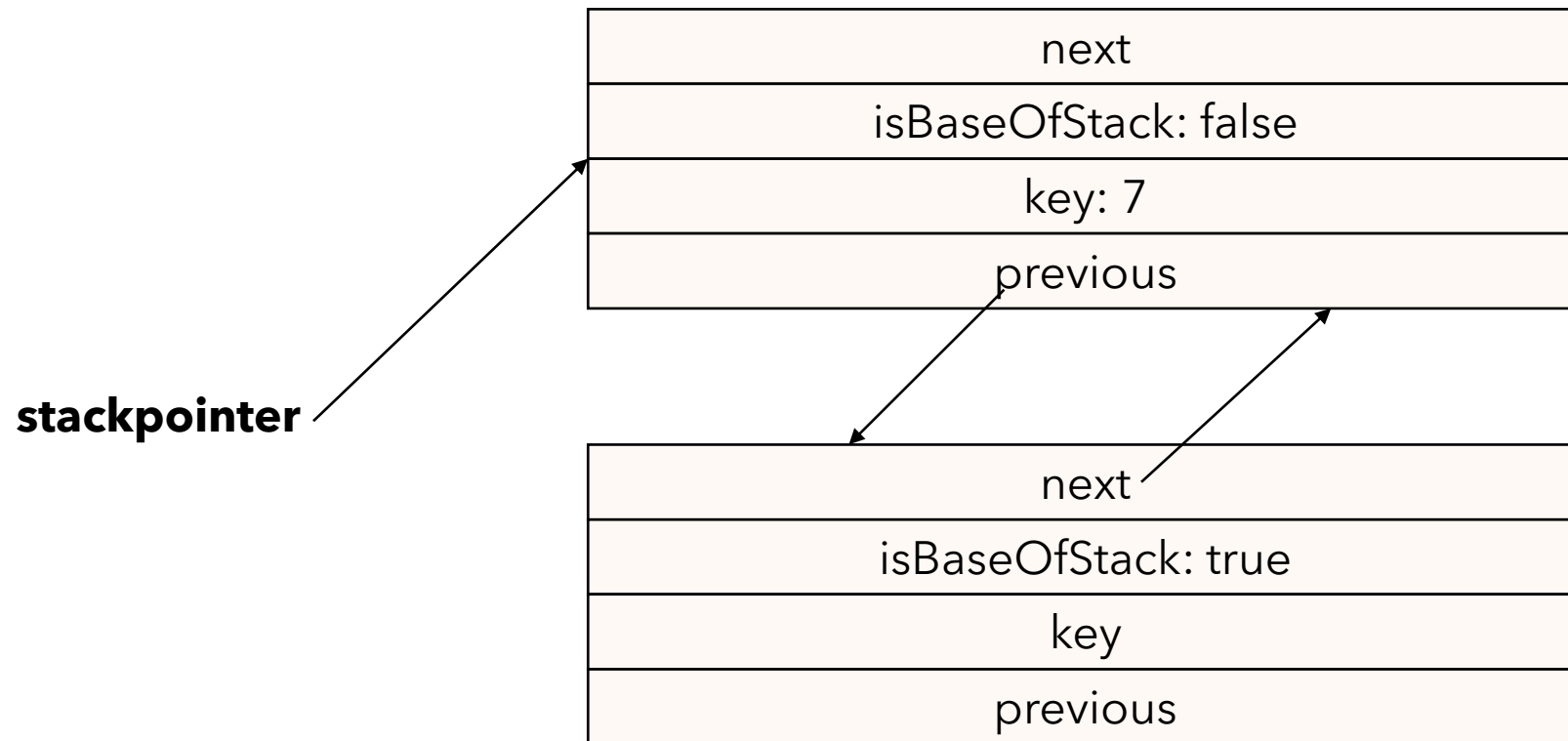
stackpointer



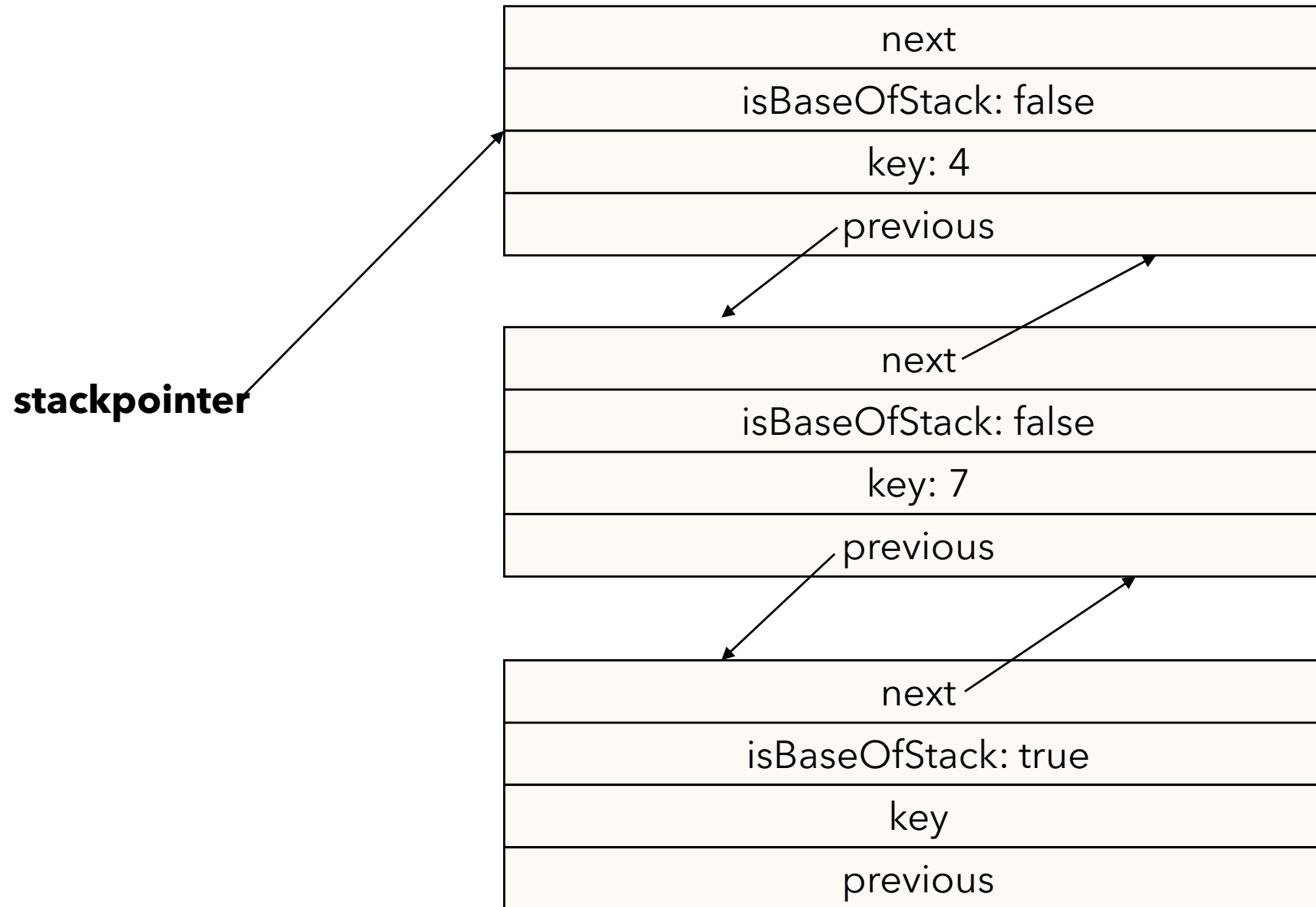
next
isBaseOfStack: true
key
previous

Stack – implementazione *linked-list-based*

push(7)



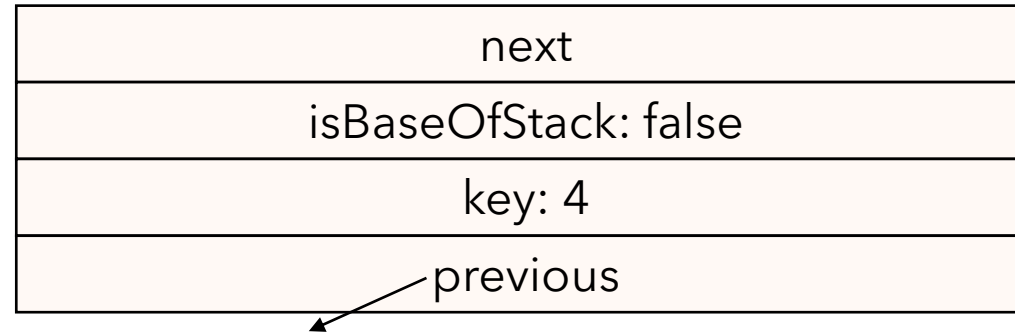
Stack – implementazione *linked-list-based*



push(4)

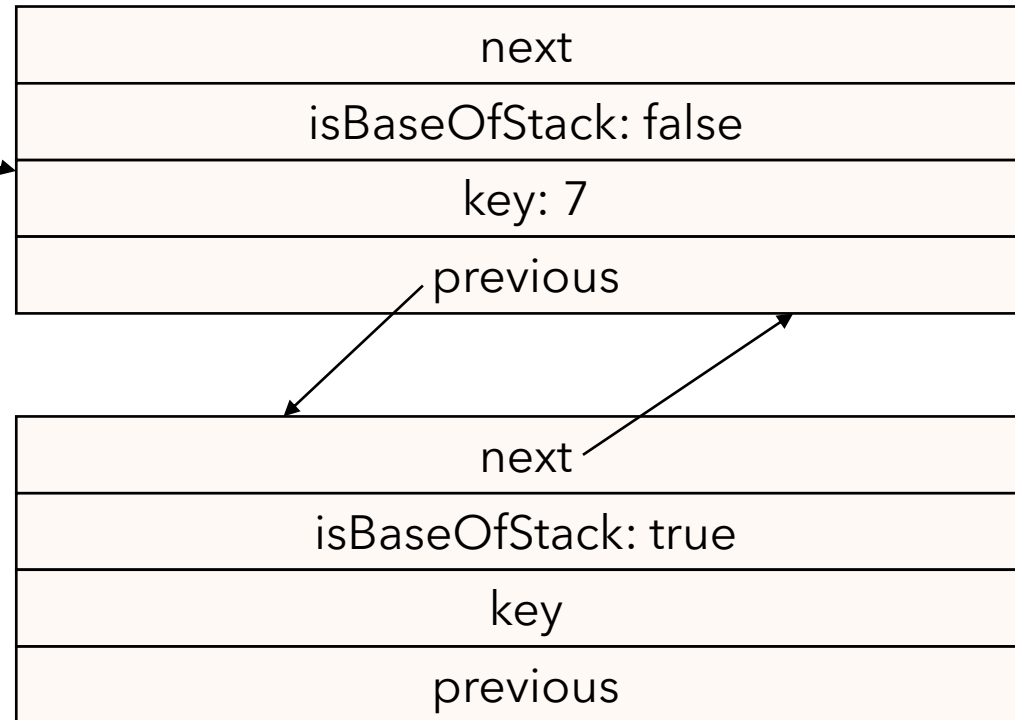
Stack – implementazione *linked-list-based*

eliminato dal garbage collector in quanto non è più puntato da nessun riferimento



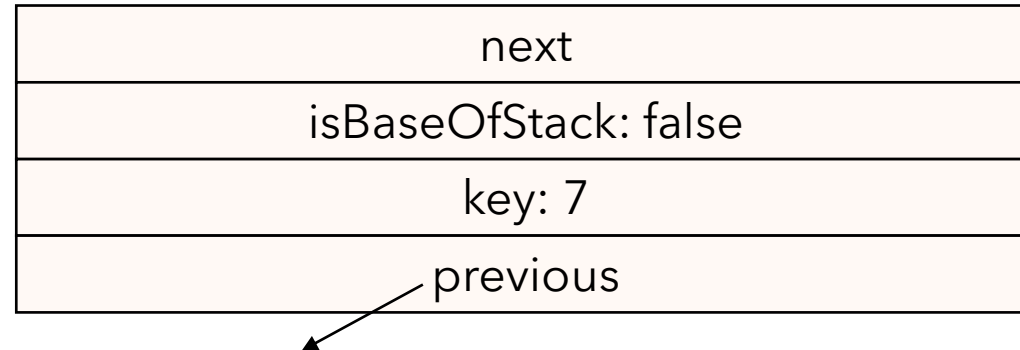
pop()

stackpointer

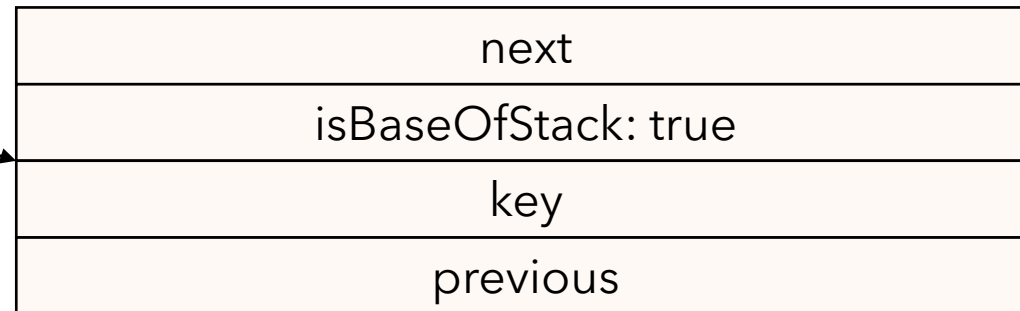


Stack – implementazione *linked-list-based*

eliminato dal garbage collector in quanto non è più puntato da nessun riferimento



stackpointer

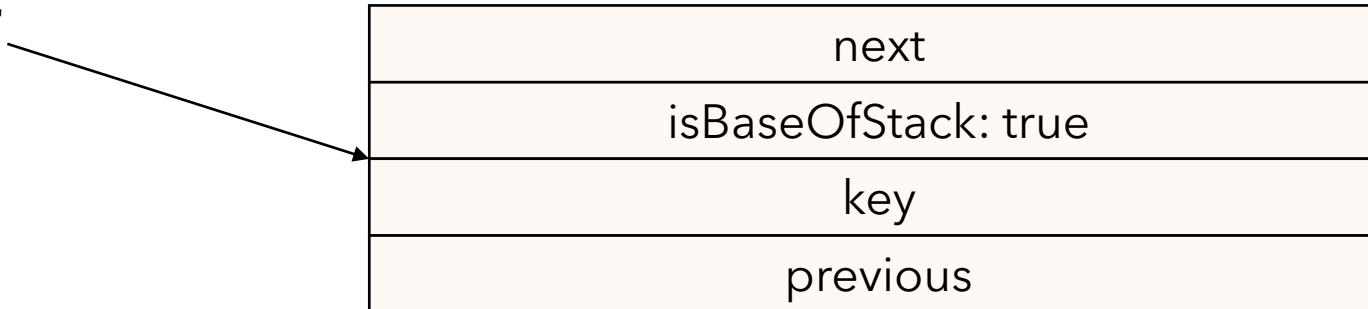


pop()

Stack – implementazione *linked-list-based*

Facciamo in modo che una chiamata a pop dal nodo *base* restituisca il nodo stesso, in modo da avere un riferimento non nullo sul quale si possano ancora invocare i metodi

stackpointer



pop()

Stack – implementazione *linked-list-based*

```
class StackIntNode{
    private int key;
    private StackIntNode next;
    private StackIntNode previous;
    private boolean isBaseOfStack;

    StackIntNode(int key, StackIntNode next, StackIntNode previous,
                  boolean isBaseOfStack){
        this.key = key;
        this.next = next;
        this.previous = previous;
        this.isBaseOfStack = isBaseOfStack;
    }
}
```

Stack – implementazione *linked-list-based*

```
StackIntNode push(int item){
    /*"this" is the current stack pointer, must point to the new node
    returns a reference to the new top
    the new node cannot be the base: "isBaseOfStack" must be false
    */
    this.next = new StackIntNode(item, null, this, false);
    return this.next;
}

StackIntNode pop(){
    StackIntNode ref = this;
    if (this.isBaseOfStack == false){
        //previous doesn't point to popped node anymore
        this.previous.next = null;
        ref = this.previous;
    }

    return ref;
}
```

Stack – implementazione *linked-list-based*

```
boolean isEmpty(){  
    return this.isBaseOfStack;  
}  
  
void print(){  
    if (this.isBaseOfStack == true){  
        return;  
    }  
    System.out.println(" " + this.key);  
    System.out.println(" =====");  
    this.previous.print();  
    return;  
}
```

Queue (code): implementazione *array-based*

- La **queue** (code) è un *Abstract Data Type* caratterizzato da due operazioni:
 - **enqueue(item)**: permette di aggiungere un elemento in fondo alla coda
 - **dequeue()**: permette di rimuovere l'elemento in testa alla coda
- Non sono ammesse altre operazioni: è una struttura dati **FIFO** (*first-in-first-out*)
- Facciamo alcuni esempi su una coda i cui elementi hanno chiavi intere

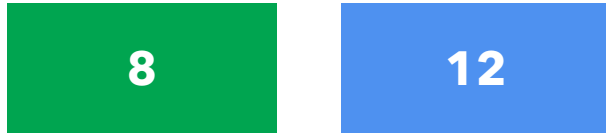
Queue (code): implementazione *array-based*

enqueue(8)



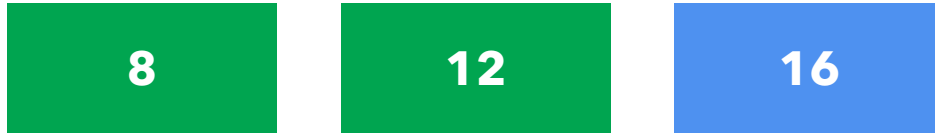
Queue (code): implementazione *array-based*

enqueue(12)



Queue (code): implementazione *array-based*

enqueue(16)



Queue (code): implementazione *array-based*

enqueue(11)



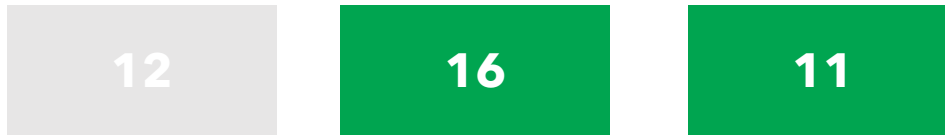
Queue (code): implementazione *array-based*

dequeue()



Queue (code): implementazione *array-based*

dequeue()



Queue (code): implementazione *array-based*

- Realizziamo una prima versione *array-based* di una coda
- Manteniamo la testa all'indice 0 dell'array:
 - vedremo che questa operazione ha un costo che rende questa implementazione non realistica

```
class QueueIntArrayBased{
    private int[] v;
    private int size;
    private final int headIndex = 0;
    private int tailIndex;

    QueueIntArrayBased(int size){
        this.v = new int[size];
        this.tailIndex = -1;
        this.size = size;
    }
}
```

Queue (code): implementazione *array-based*

```
int enqueue(int item){  
    if (this.tailIndex < this.size - 1){  
        this.tailIndex++;  
        this.v[this.tailIndex] = item;  
    }  
  
    return this.tailIndex;  
}
```

Queue (code): implementazione *array-based*

```
int dequeue() {  
    //has linear complexity, not a great idea!  
    if (this.tailIndex >= 0) {  
        if (this.tailIndex >= 1) {  
            for (int i = 0; i <= this.tailIndex - 1; i++) {  
                this.v[i] = this.v[i + 1];  
            }  
        }  
  
        this.tailIndex--;  
    }  
  
    return this.tailIndex;  
}
```

Queue (code): implementazione *linked-list-based*

```
class QueueIntNode{  
    private int key;  
    private QueueIntNode next;  
  
    QueueIntNode(int key, QueueIntNode next){  
        this.key = key;  
        this.next = next;  
    }  
}
```

Queue (code): implementazione *linked-list-based*

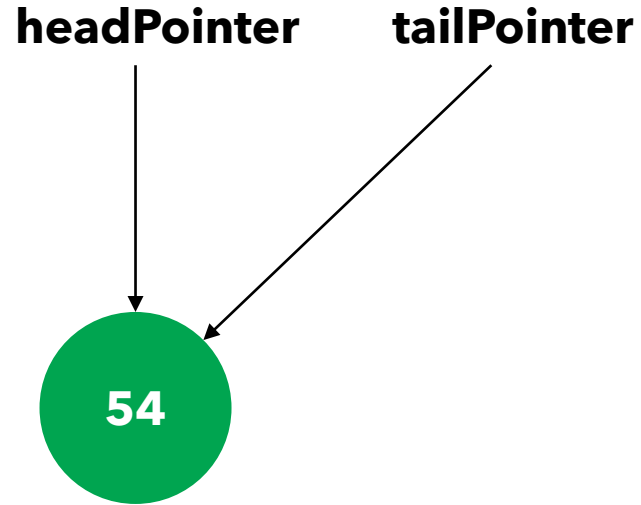
//called on tail pointer

```
QueueIntNode enqueue(int item){  
    /*"this" is the current tail pointer, its next member must point to the new node  
    returns a reference to the new tail  
    */  
    this.next = new QueueIntNode(item, null);  
    return this.next;  
}
```

//called on head pointer

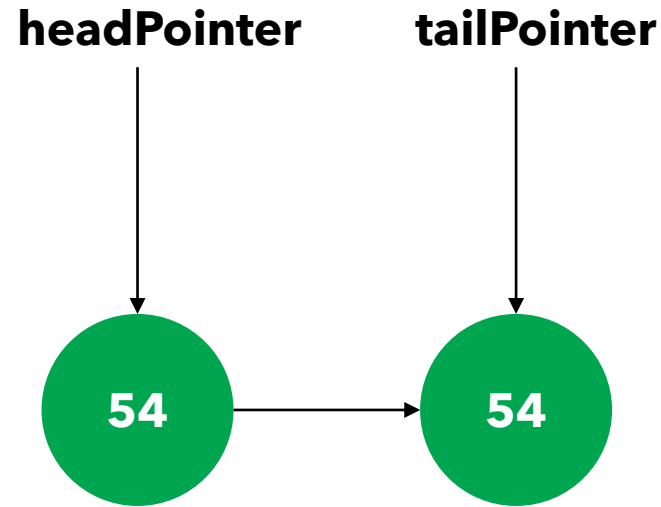
```
QueueIntNode dequeue(){  
    return this.next;  
}
```

Queue (code): implementazione *linked-list-based*



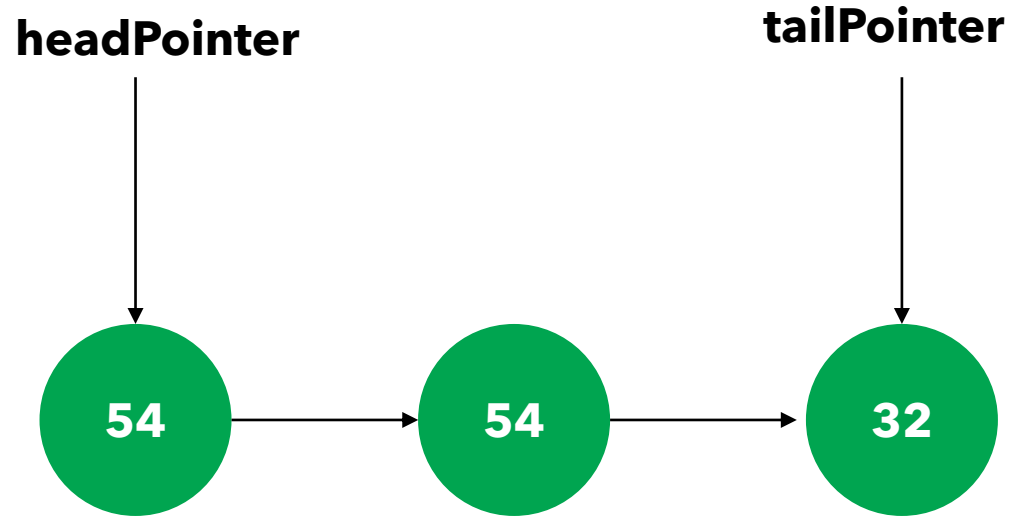
```
QueueIntNode headPointer = new QueueIntNode(54, null);  
QueueIntNode tailPointer = headPointer;
```


Queue (code): implementazione *linked-list-based*



```
tailPointer = tailPointer.enqueue(43);
```

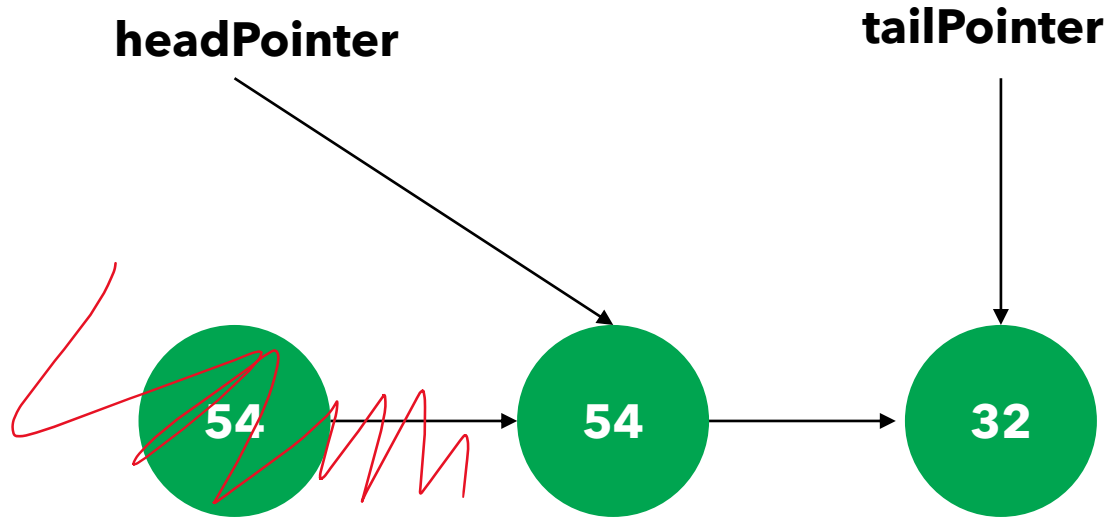
Queue (code): implementazione *linked-list-based*



```
tailPointer = tailPointer.enqueue(32);
```

Quindi, *enqueue* va invocata sul puntatore *tail* mentre *dequeue* sul puntatore...?

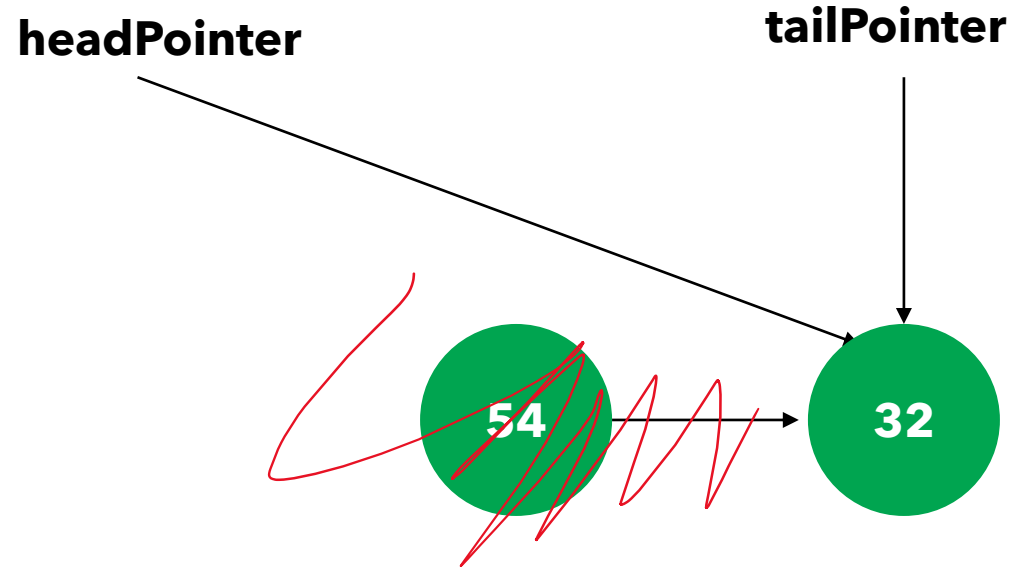
Queue (code): implementazione *linked-list-based*



```
headPointer = headPointer.dequeue();
```

Quindi, *enqueue* va invocata sul puntatore *tail* mentre *dequeue* sul puntatore...?

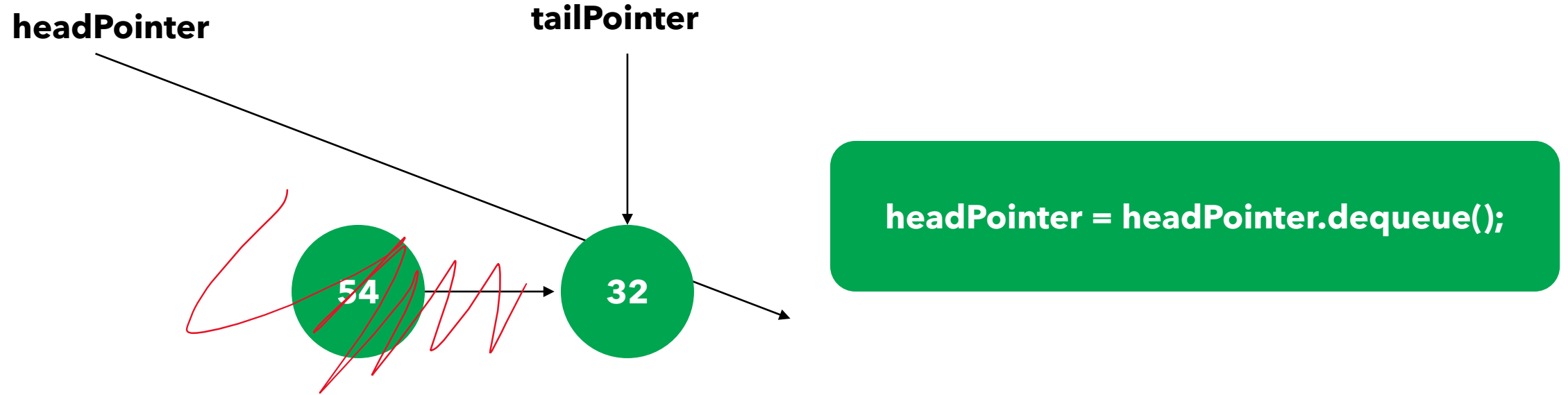
Queue (code): implementazione *linked-list-based*



```
headPointer = headPointer.dequeue();
```

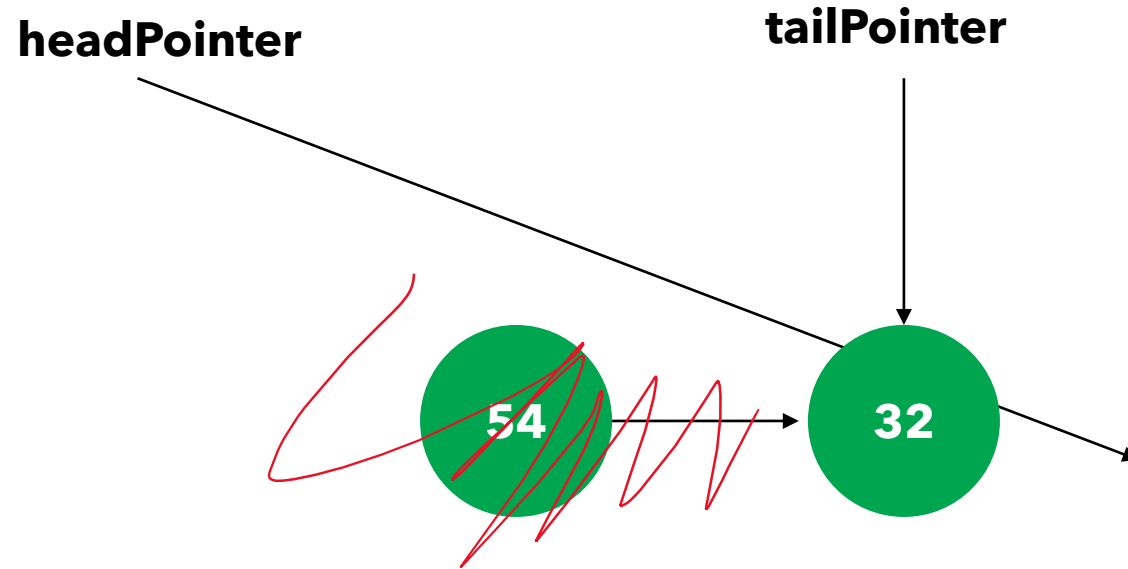
Quindi, *enqueue* va invocata sul puntatore *tail* mentre *dequeue* sul puntatore...?

Queue (code): implementazione *linked-list-based*



Quindi, *enqueue* va invocata sul puntatore *tail* mentre *dequeue* sul puntatore...?

Queue (code): implementazione *linked-list-based*

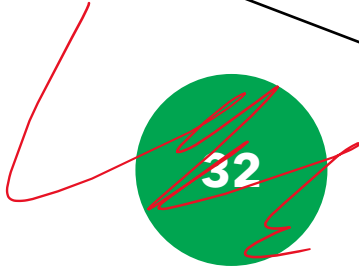


Affinché anche l'ultimo nodo rimasto venga *garbage-collected*, tailPointer deve diventare null

Queue (code): implementazione *linked-list-based*

headPointer: null

tailPointer: null



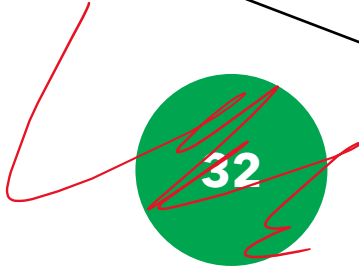
tailPointer = null;

Affinché anche l'ultimo nodo rimasto venga *garbage-collected*, tailPointer deve diventare null

Queue (code): implementazione *linked-list-based*

headPointer: null

tailPointer: null



tailPointer = null;

Affinché anche l'ultimo nodo rimasto venga *garbage-collected*, tailPointer deve diventare null

Queue (code): implementazione *linked-list-based*

```
class QueueLecture{  
    public static void main(String[] args){  
        QueueIntNode head = new QueueIntNode(80, null);  
        QueueIntNode tail = head;  
  
        tail = tail.enqueue(5);  
        tail = tail.enqueue(8);  
        tail = tail.enqueue(7);  
        tail = tail.enqueue(73);  
        tail = tail.enqueue(77);  
  
        head.print();  
        System.out.println("head's key is: " + head.key);  
        System.out.println("tail's key is: " + tail.key);  
        if (head != null){  
            head = head.dequeue();  
            head.print();  
        }  
    }  
}
```