Data Recursion

Method vs. Data Recursion

Method Recursion

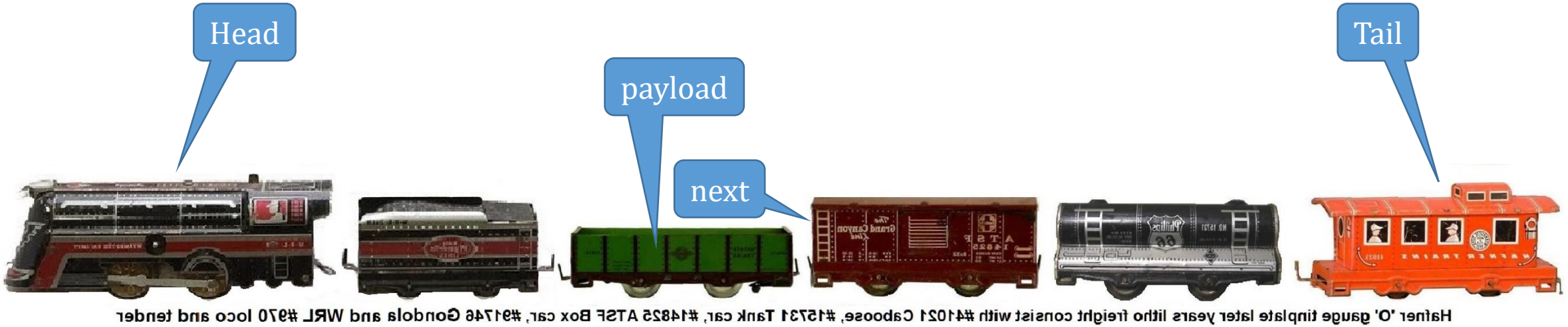
- A method invokes itself
- Simple case
  - no recusion required
- Avoid infinite recursion

Data Recursion

- An object references another object in the same class
- Reference may be null
  - no recursion required
- Avoid circular references

Start simple… singly linked list

- A list has multiple items… each item is a “node”
- Each node carries some data – a payload
  - To keep things simple, our payload will just be a single integer. A more sophisticated application would have a larger and more complex payload
- Each node (except tail) references its "next" node



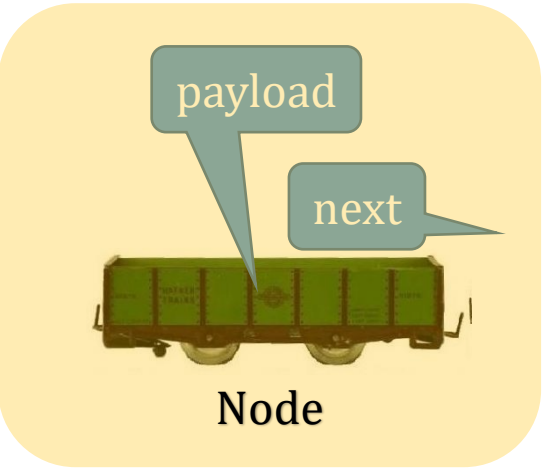
Singly Linked Node Class

```
class SllNode {
        private int payload;
        private SllNode next;

        public SllNode(int payload) {
            this.payload=payload;
            this.next=null;
        }

        public int getPayload() { return payload; }
        public SllNode getNext() { return next; }
        public void setNext(SllNode next) { this.next = next; }
}
```

Reference to another instance in the same class



Helper Class

- SllNode is a "helper" class for a singly linked list
  - Only used within singly linked lists
- No need to expose SllNode to the world
- We can define SllNode as a "helper" class
  - private ***inside*** a SingleLinkedList class

Singly Linked List Class

```
public class SingleLinkedList {
    private SllNode head;
    …
    private class SllNode {
        private int payload;
        …
    }
}
```

"Default" constructor initializes head to null which is an empty list

Helper class
Unavailable outside the SingleLinkedList class

SinglyLinkedList pushHead method

```
public void pushHead(int payload) {
        SllNode newNode = new SllNode(payload);
        newNode.setNext(head);
        head=newNode;
}
```

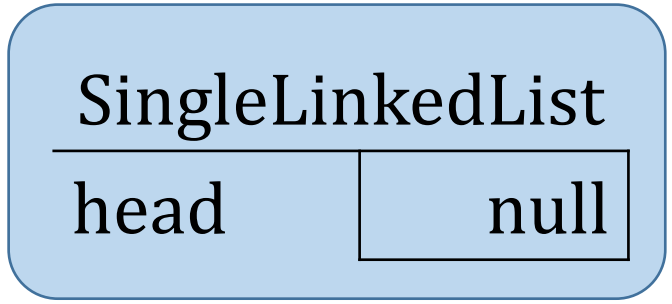
If head was null, newNode becomes tail

ListTester main method

```
public static void main(String[] args) {
        SingleLinkedList lst = new SingleLinkedList();
        lst.pushHead(14);
        lst.pushHead(23);
        lst.pushHead(9);
}
```

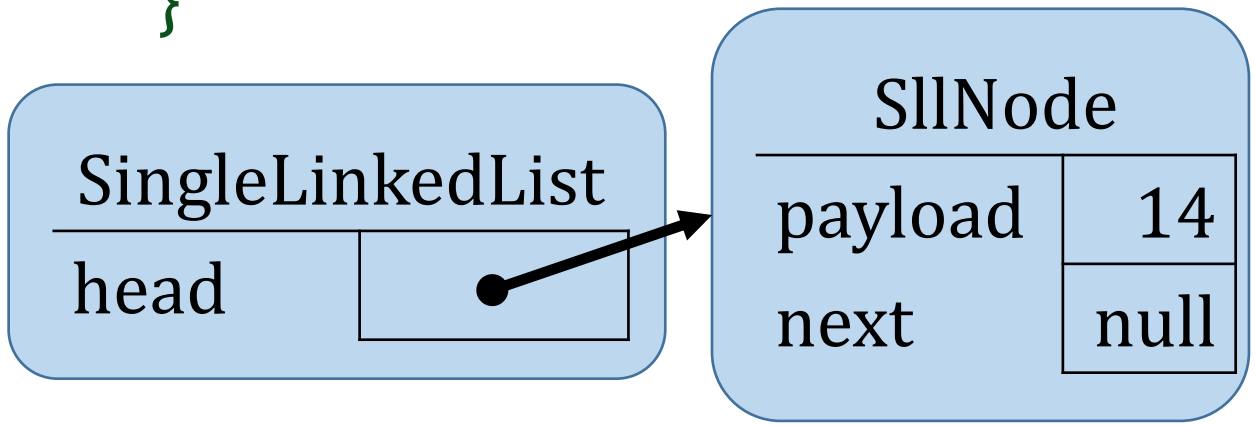
ListTester main method

```
public static void main(String[] args) {
        SingleLinkedList lst = new SingleLinkedList();
        lst.pushHead(14);
        lst.pushHead(23);
        lst.pushHead(9);
}
```



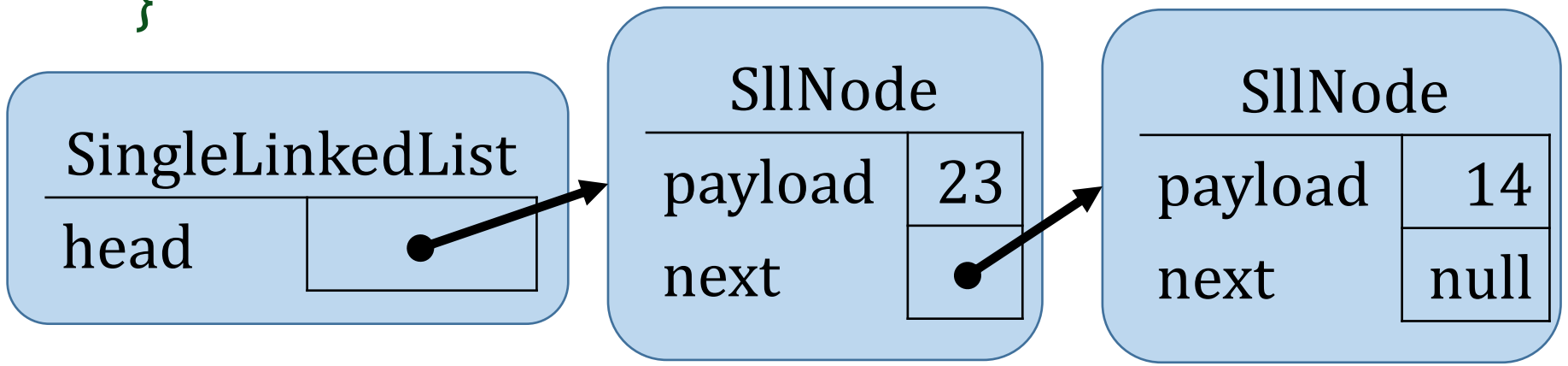
ListTester main method

```
public static void main(String[] args) {
        SingleLinkedList lst = new SingleLinkedList();
        lst.pushHead(14);
        lst.pushHead(23);
        lst.pushHead(9);
}
```



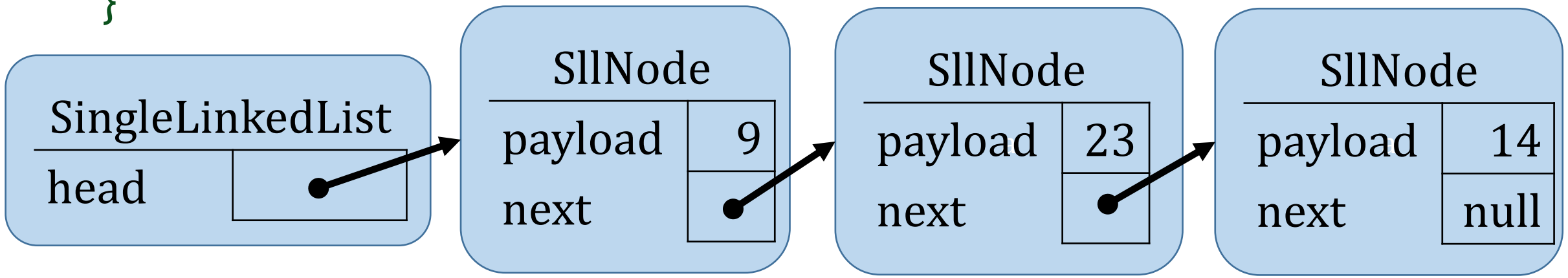
ListTester main method

```
public static void main(String[] args) {
        SingleLinkedList lst = new SingleLinkedList();
        lst.pushHead(14);
        lst.pushHead(23);
        lst.pushHead(9);
}
```



ListTester main method

```
public static void main(String[] args) {
        SingleLinkedList lst = new SingleLinkedList();
        lst.pushHead(14);
        lst.pushHead(23);
        lst.pushHead(9);
}
```



How do we push to the tail?

- We track head, and we can find the tail if we know the head
- The tail is the node with a "null" next reference.
- We could write a simple loop:

```
SllNode tail;
for(tail=head;
    (tail != null) && (tail.getNext()!=null);
    tail=tail.getNext()){ }
```

- But we just learned about recursion… let's try it…

Single Linked List getTail method

```
private SllNode getTail(SllNode from) {
    if (from==null) return null;
    SllNode next=from.getNext();
    if (next==null) return from;
    return getTail(next);
}
public SllNode getTail() {
    return getTail(head);
}
```

Special case for empty list

Simplest case… from IS the tail

Invoke recursively on a simpler problem -- a shorter list

If no node specified, start from head.

Single Linked List pushTail method

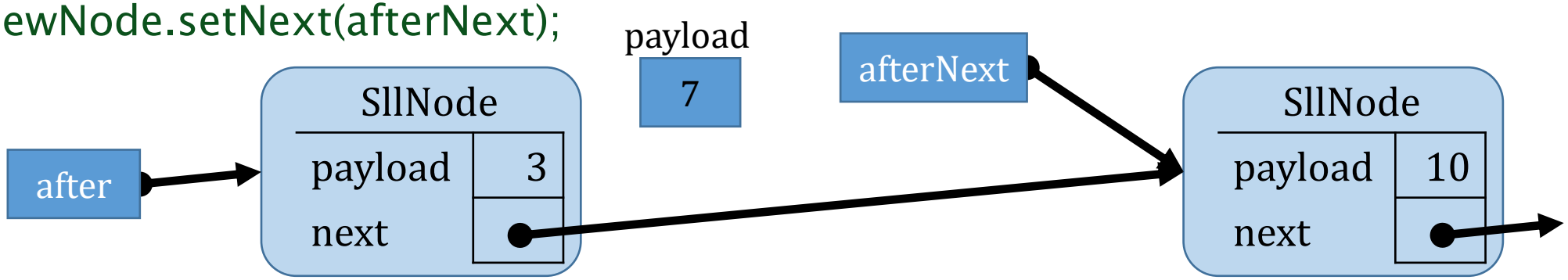
```
public void pushTail(int payload) {
    if (head==null) head=new SllNode(payload);
    else getTail().setNext(new SllNode(payload));
}
```

SLL recursive pushOrder method

```
private void pushOrder(int payload,SllNode after) {
    SllNode afterNext=after.getNext();
    if (afterNext==null) after.setNext(new SllNode(payload));
    else if (payload >= afterNext.getPayload()) pushOrder(payload,afterNext);
    else {
        SllNode newNode = new SllNode(payload);
        after.setNext(newNode);
        newNode.setNext(afterNext);
    }
}
```

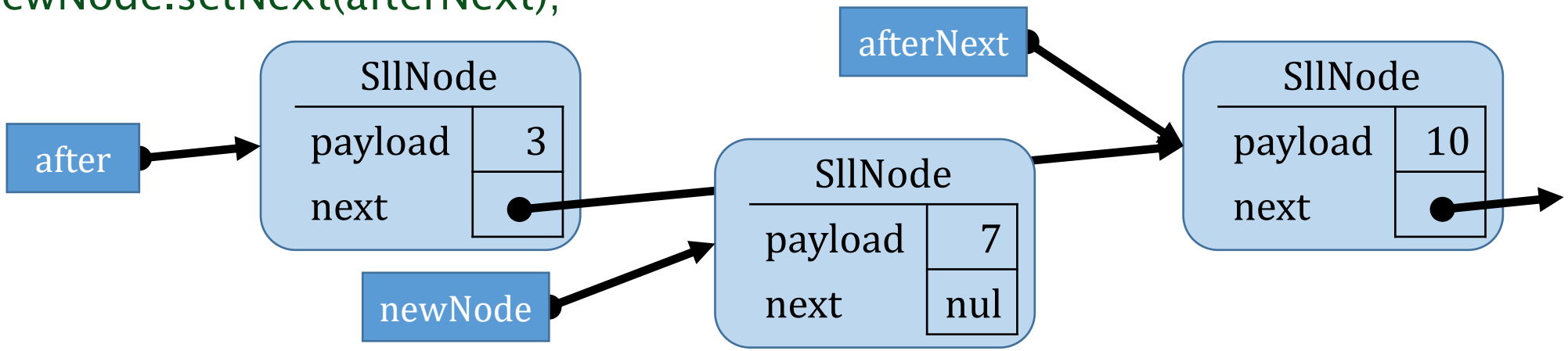
if payload>tail, make payload tail

if payload>next, recursive call



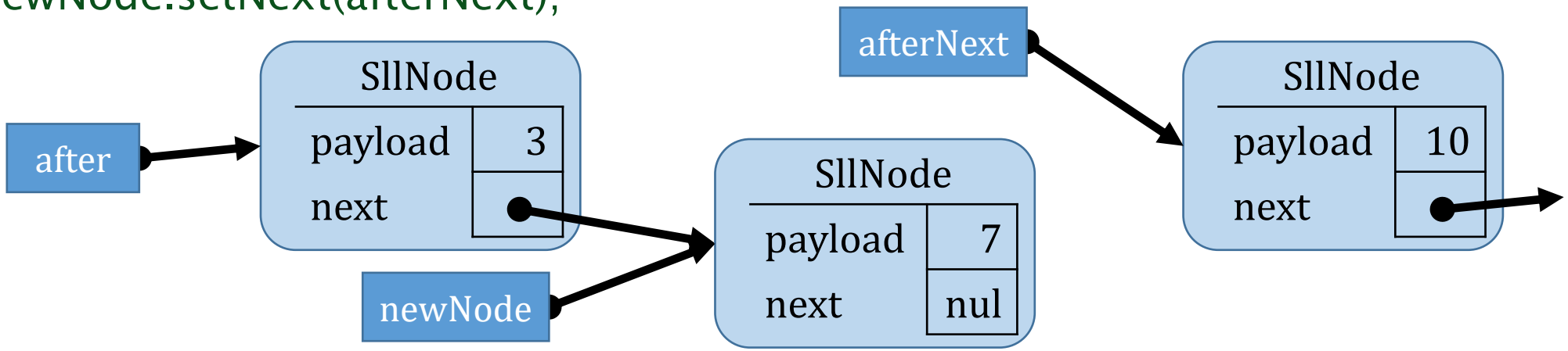
SLL recursive pushOrder method

```
private void pushOrder(int payload,SllNode after) {
    SllNode afterNext=after.getNext();
    if (afterNext==null) after.setNext(new SllNode(payload));
    else if (payload >= afterNext.getPayload()) pushOrder(payload,afterNext);
    else {
        SllNode newNode = new SllNode(payload);
        after.setNext(newNode);
        newNode.setNext(afterNext);
    }
}
```



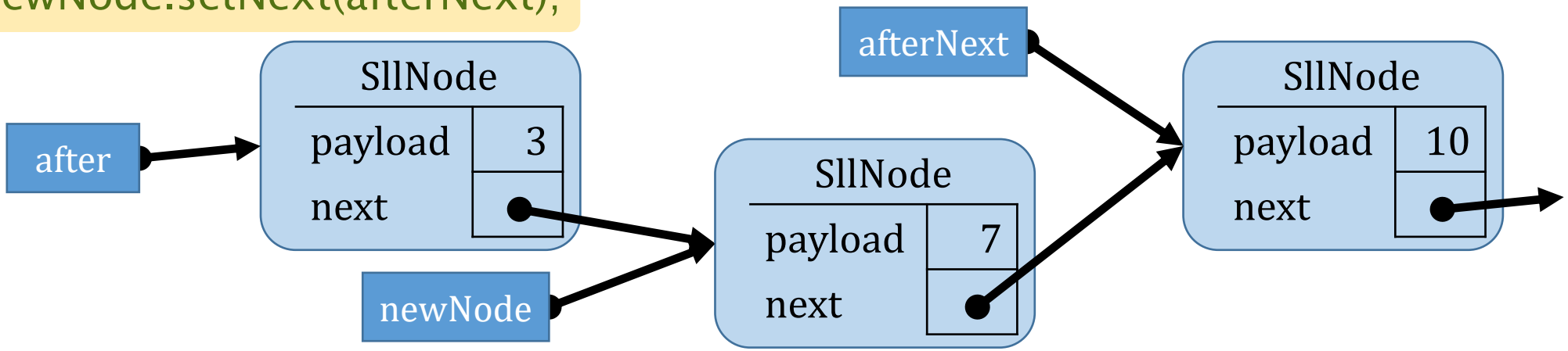
SLL recursive pushOrder method

```
private void pushOrder(int payload,SllNode after) {
    SllNode afterNext=after.getNext();
    if (afterNext==null) after.setNext(new SllNode(payload));
    else if (payload >= afterNext.getPayload()) pushOrder(payload,afterNext);
    else {
        SllNode newNode = new SllNode(payload);
        after.setNext(newNode);
        newNode.setNext(afterNext);
    }
}
```



SLL recursive pushOrder method

```
private void pushOrder(int payload,SllNode after) {
    SllNode afterNext=after.getNext();
    if (afterNext==null) after.setNext(new SllNode(payload));
    else if (payload >= afterNext.getPayload()) pushOrder(payload,afterNext);
    else {
        SllNode newNode = new SllNode(payload);
        after.setNext(newNode);
        newNode.setNext(afterNext);
    }
}
```



SLL recursive pushOrder method

```
private void pushOrder(int payload,SllNode after) {
    SllNode afterNext=after.getNext();
    if (afterNext==null) after.setNext(new SllNode(payload));
    else if (payload >= afterNext.getPayload()) pushOrder(payload,afterNext);
    else {
        SllNode newNode = new SllNode(payload);
        after.setNext(newNode);
        newNode.setNext(afterNext);
    }
}
```