Linked Lists

Systems Programming

Data Structures

- Abstraction that represents a collection of data in a program in order to ease its manipulation.
- The suitability of a data structure depends on the nature of the data to be stored and how that data will be manipulated

Linear Data Structures

- Organize data as a sequence, where each piece of data has a preceding datum (except the first one) and a succeeding datum (except the last one)
- Examples of linear data structures:
  - Arrays
  - Linked lists
  - Stacks
  - Queues
  - Doubly ended queues

Arrays

- Arrays have two main advantages for storing linear data collections:
  - Random access: any position in the array can be accessed in constant time.
  - Efficient use of memory when all the positions of the array are in use, because the array is stored in consecutive memory positions.

Arrays

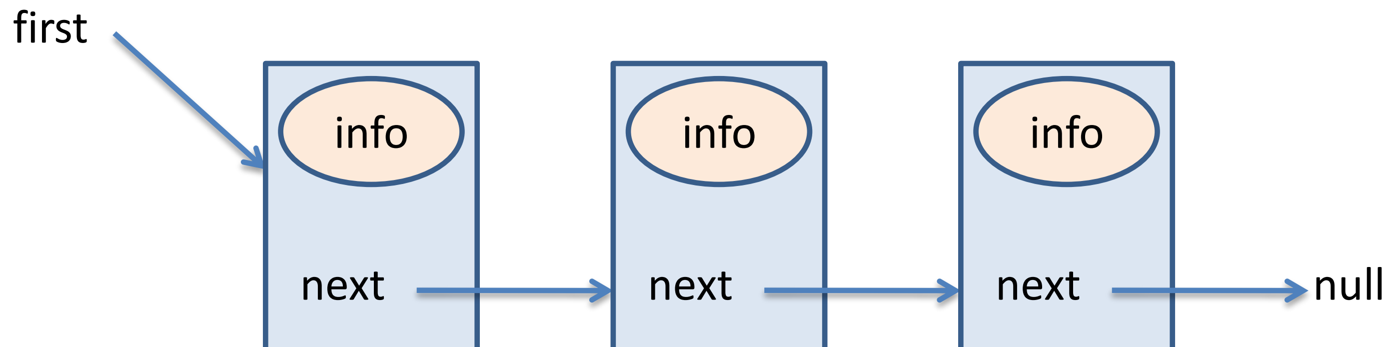
- Disadvantages (I):
  - Static size: a size must be established when the array is created, and cannot be changed later. The main problems it poses are:
    - Inefficient use of memory when more positions than needed are reserved, because of being the array sized for the worst case
    - It may happen at run-time that more positions than reserved are needed
  - Need of contiguous memory:
    - Even having the system enough free memory, it may happen that there is not enough contiguous space, due to memory fragmentation

Arrays

- Disadvantages (II):
  - Some operations on the array have a sub-optimum cost:
    - Insertions and removals of data in the first position or intermediate positions need data to be moved to consecutive memory positions
    - Concatenation of arrays: data has to be copied to a new array
    - Partition of an array in several pieces: data needs to be copied to new arrays

Linked Lists

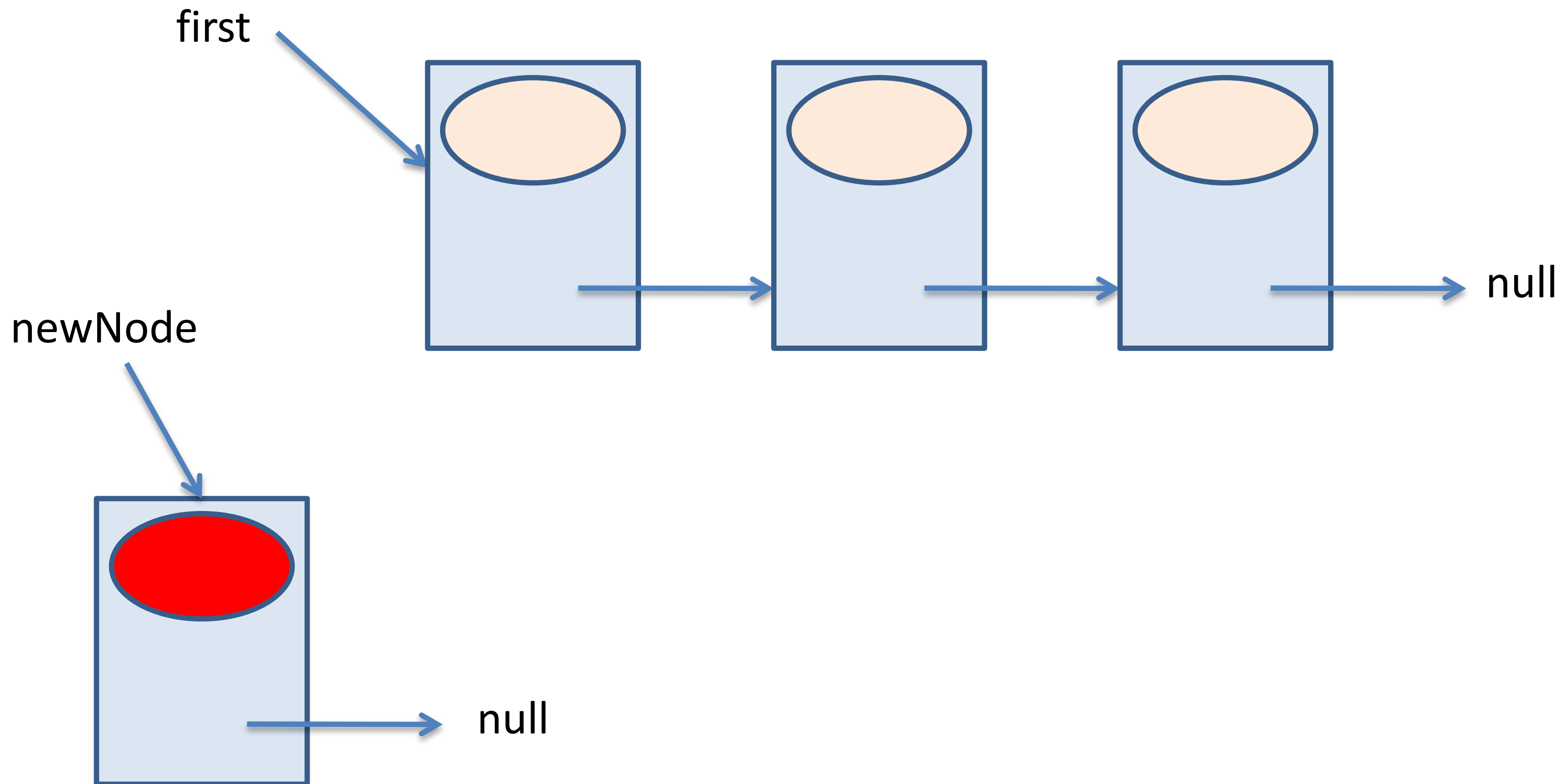
- Ordered sequence of nodes in which each node stores:
  - A piece of data
  - A reference pointing to the next node
- Nodes do not need to be in consecutive memory positions



The Node Class

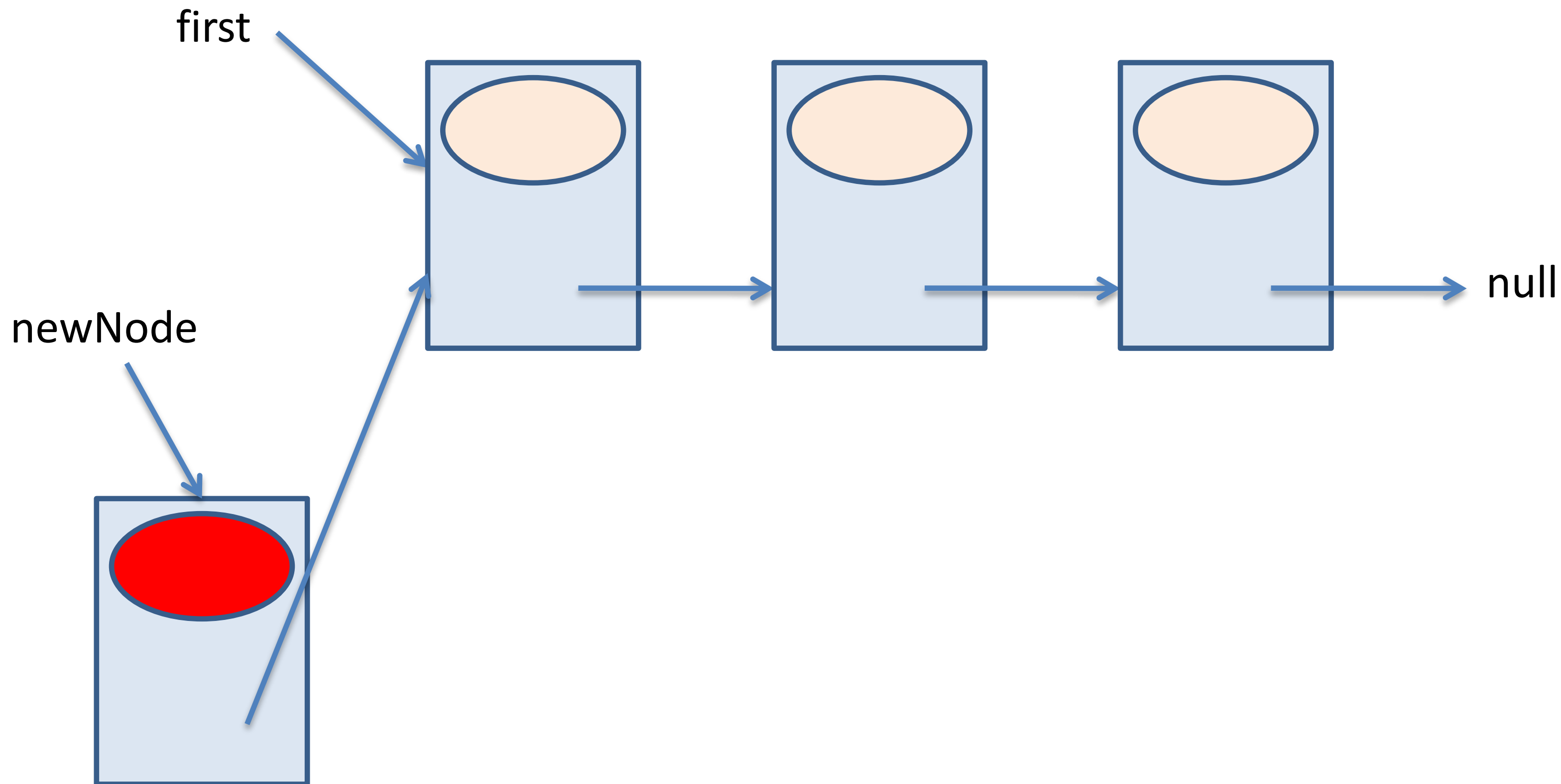
```
Public class Node {
    private Object info;
    private Node next;

    public Node(Object info) {…}

    public Node getNext() {…}
    public void setNext(Node next) {…}
    public Object getInfo() {…}
    public void setInfo(Object info) {…}
}
```

Inserting a node at the beginning



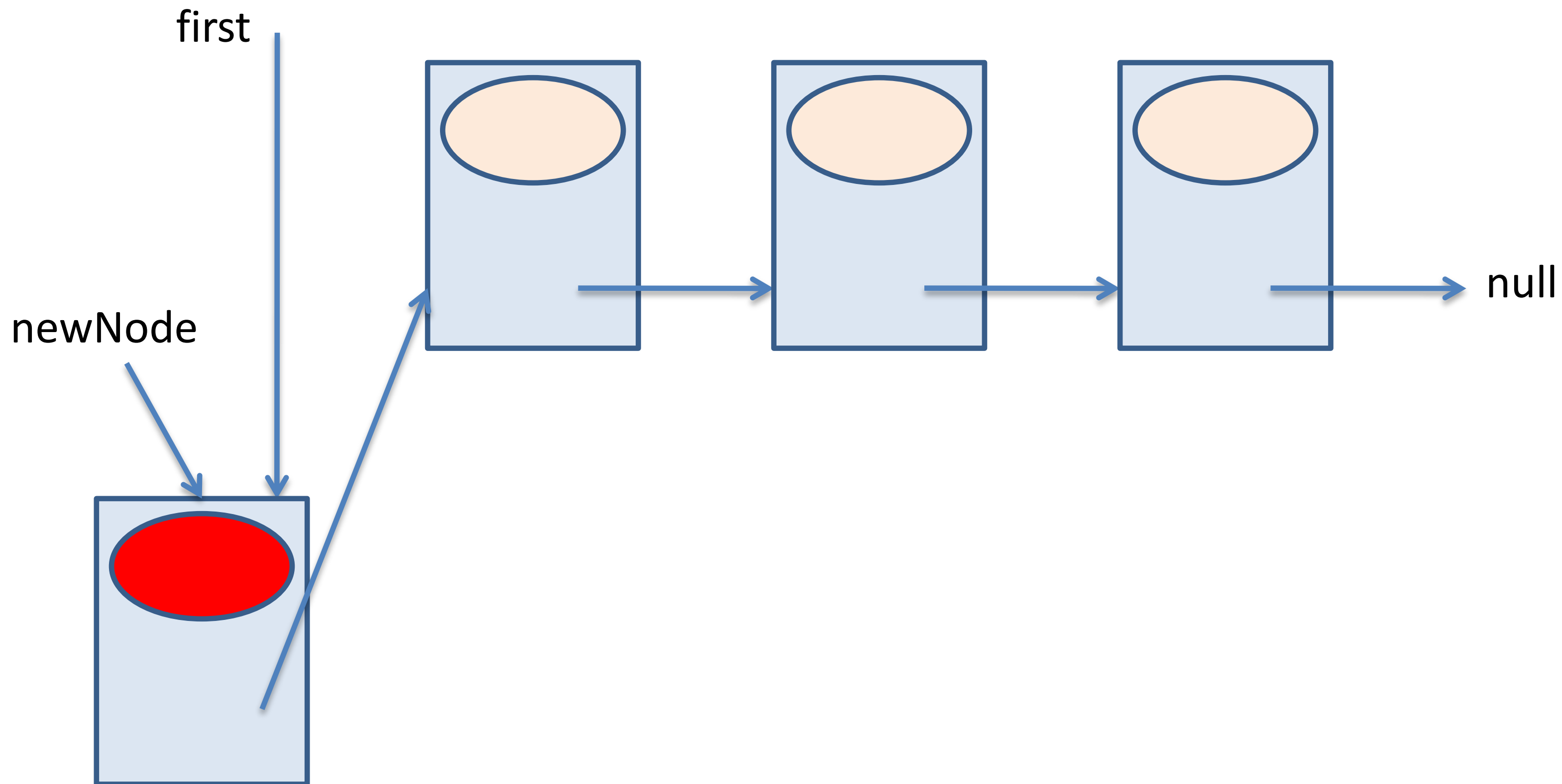
Node newNode = new Node(info);

Inserting a node at the beginning



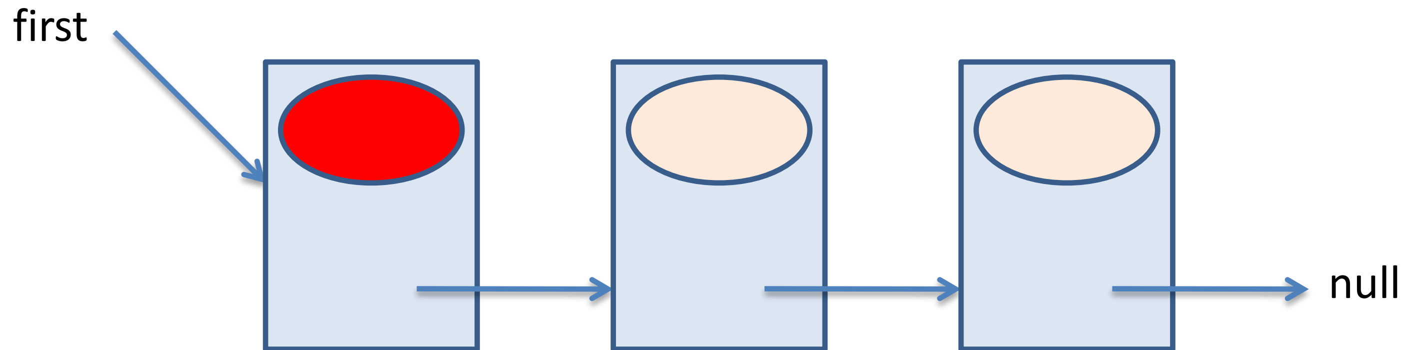
newNode .setNext(first);

Inserting a node at the beginning

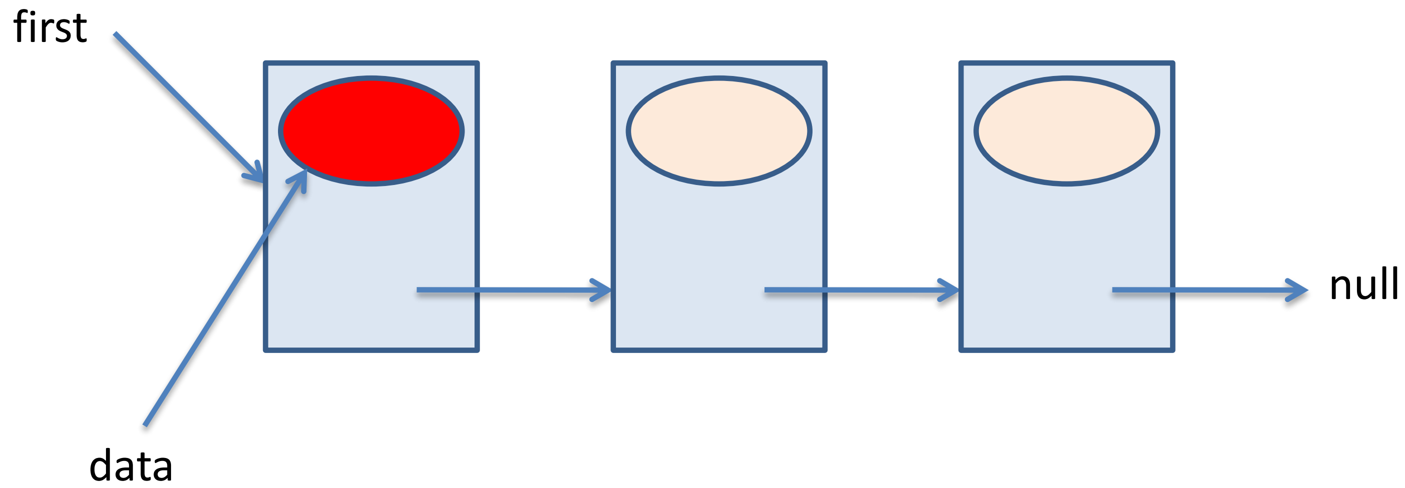


first = newNode;

Removing the first node

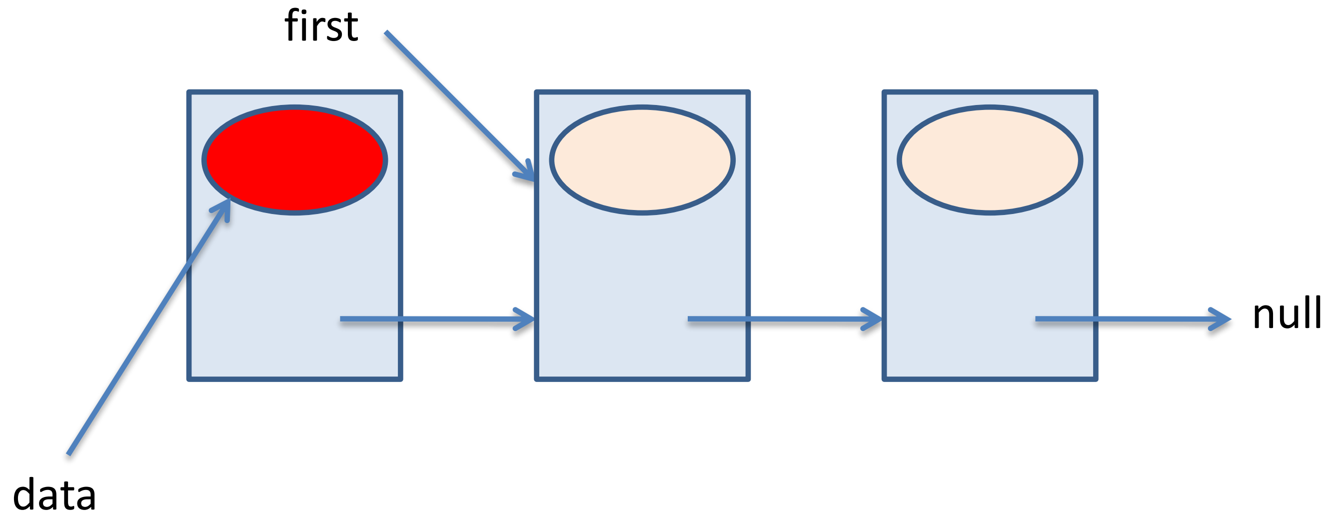


Removing the first node



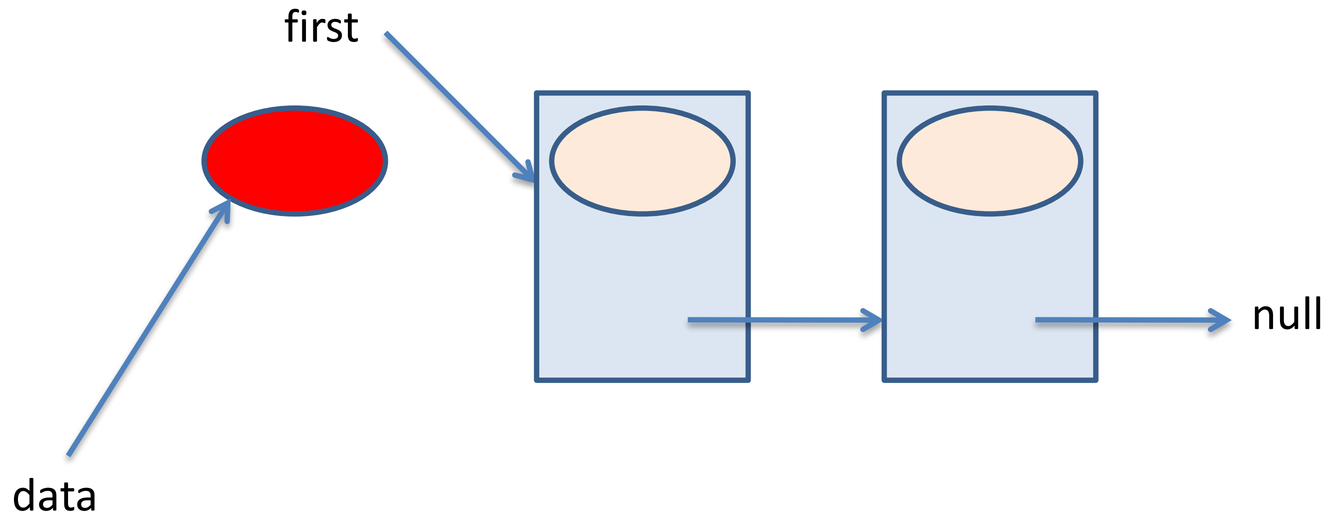
Object data = first.getInfo();

Removing the first node

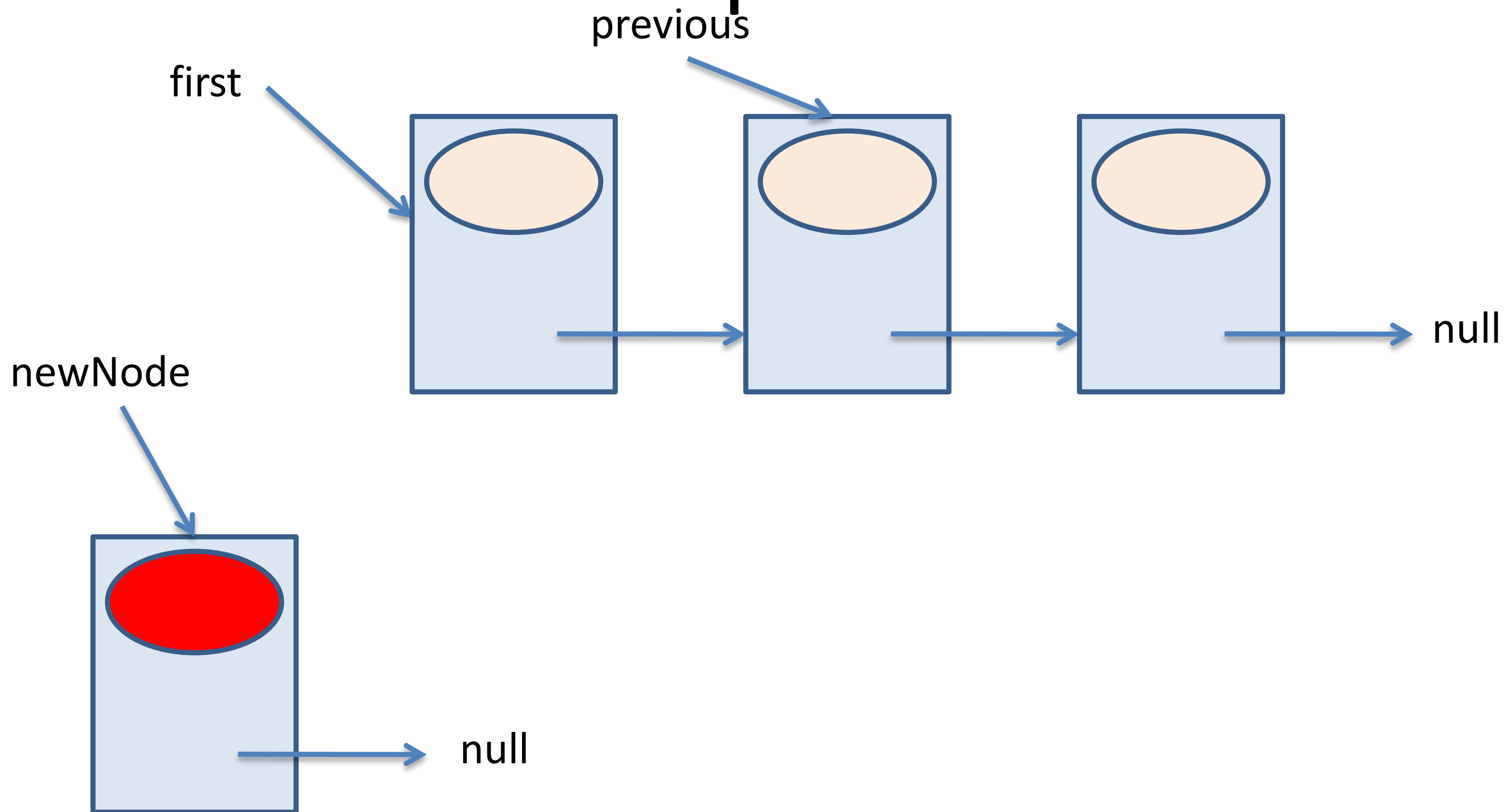


first = first.getNext();

Removing the first node

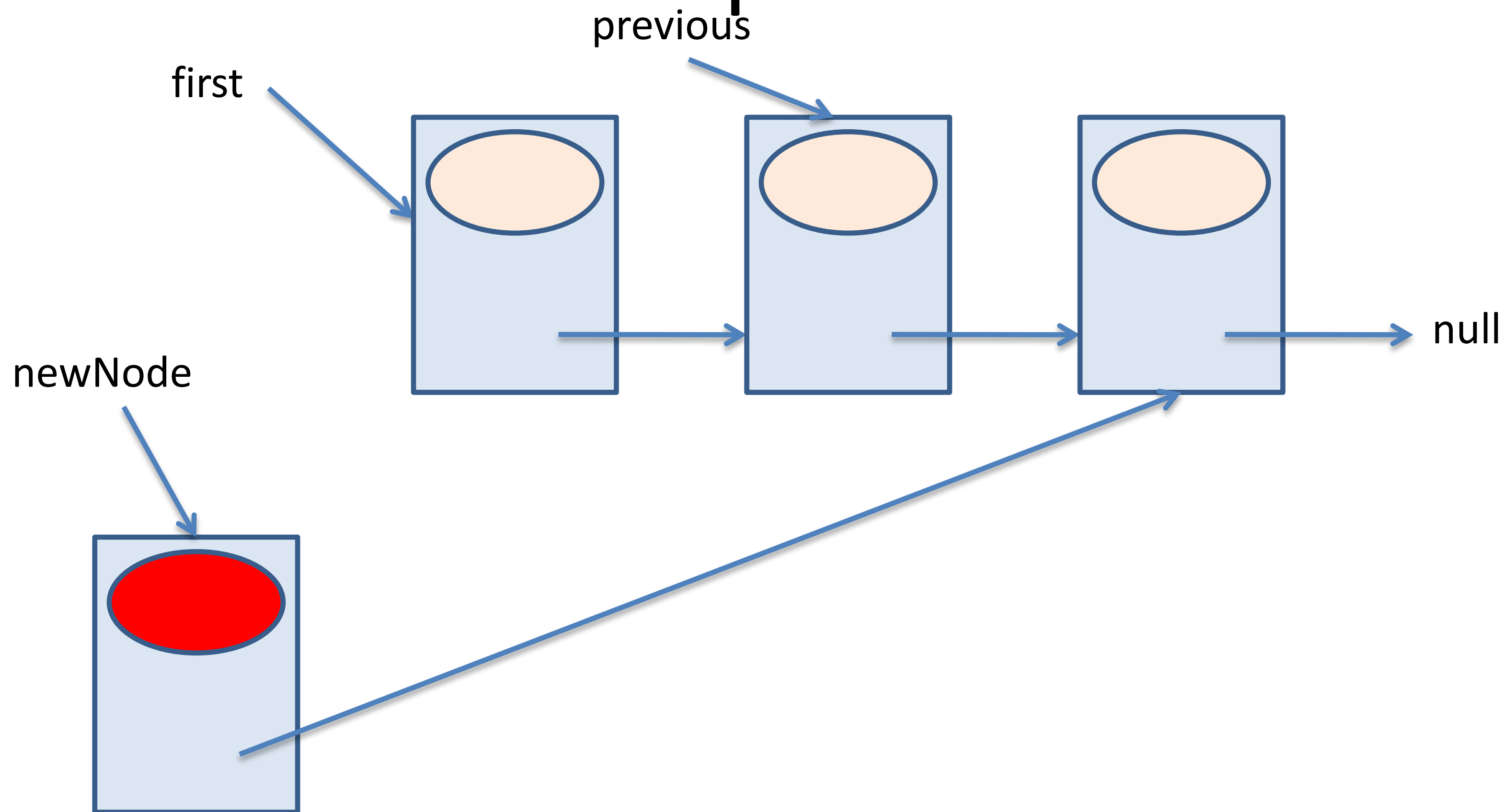


Inserting a node at an intermediate position



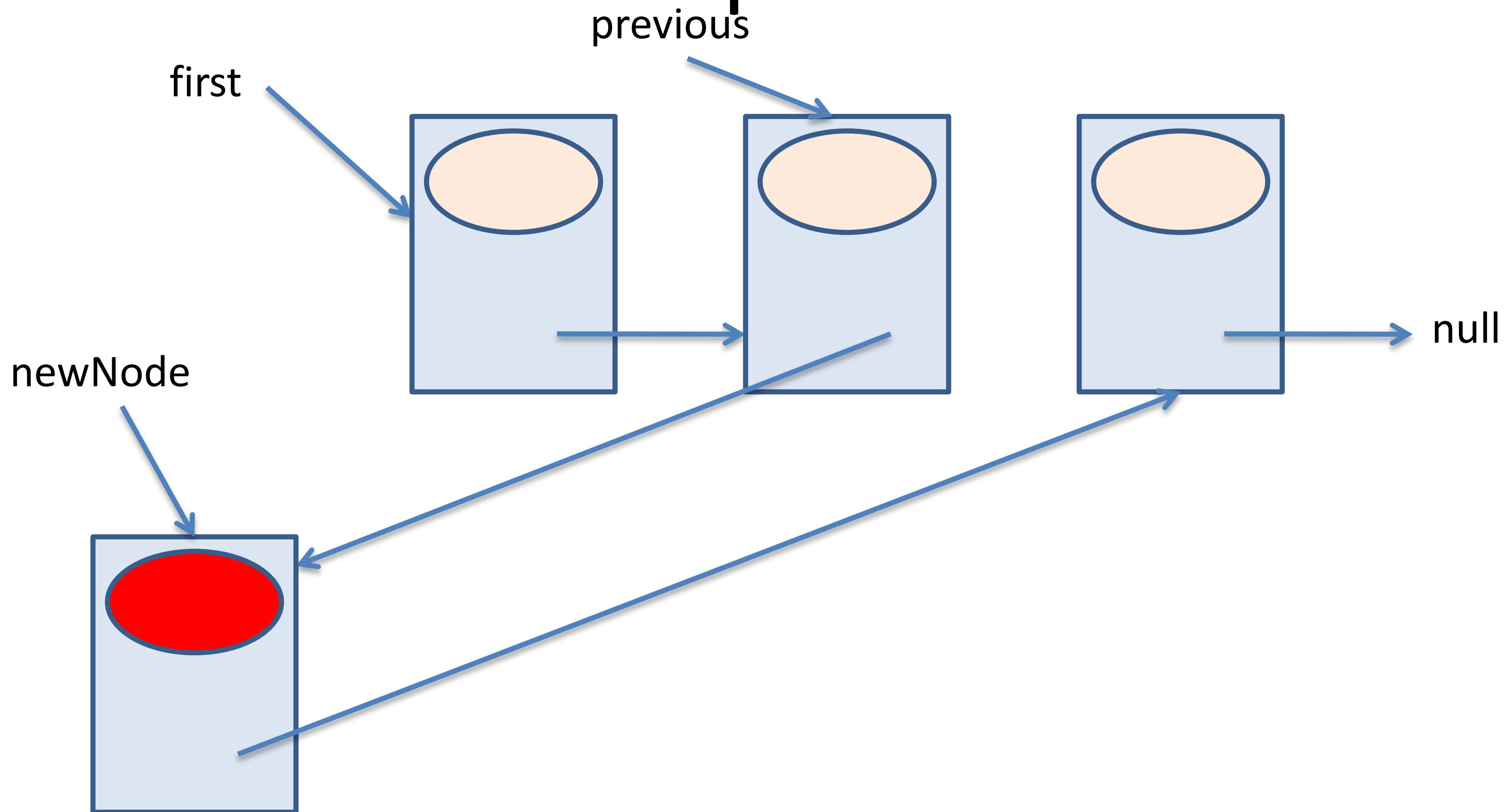
Node newNode = new Node(info);

Inserting a node at an intermediate position



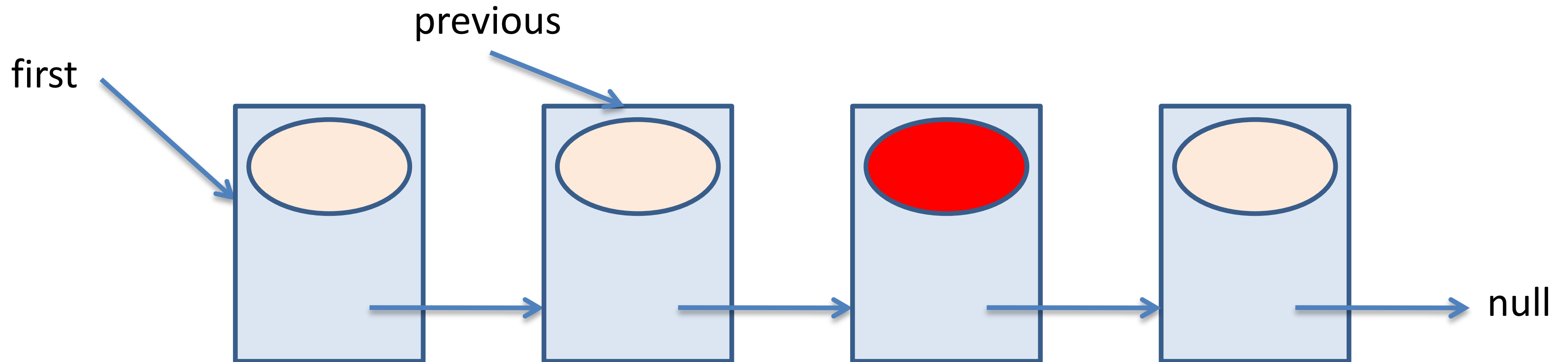
newNode.setNext(previous.getNext())

Inserting a node at an intermediate position

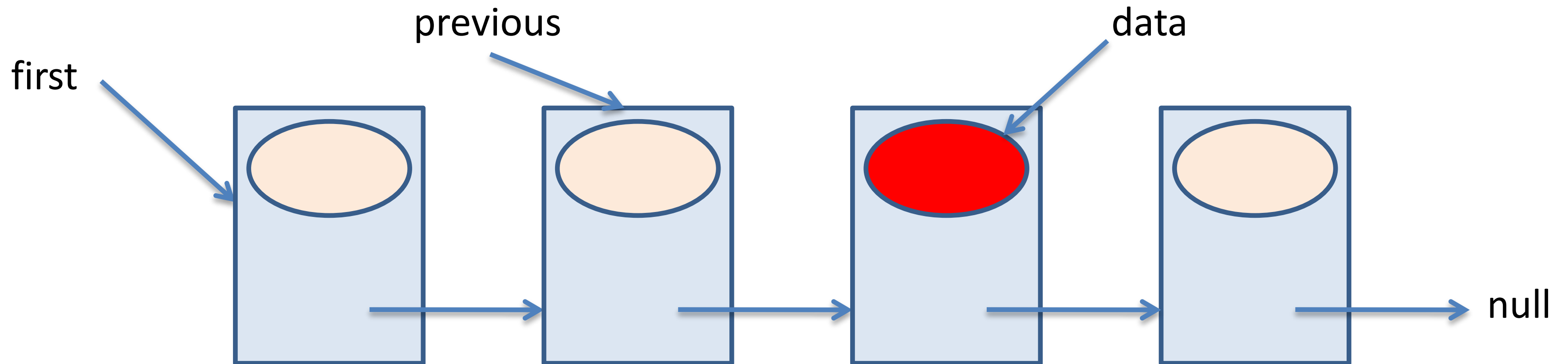


previous.setNext(newNode)

Removing an intermediate node

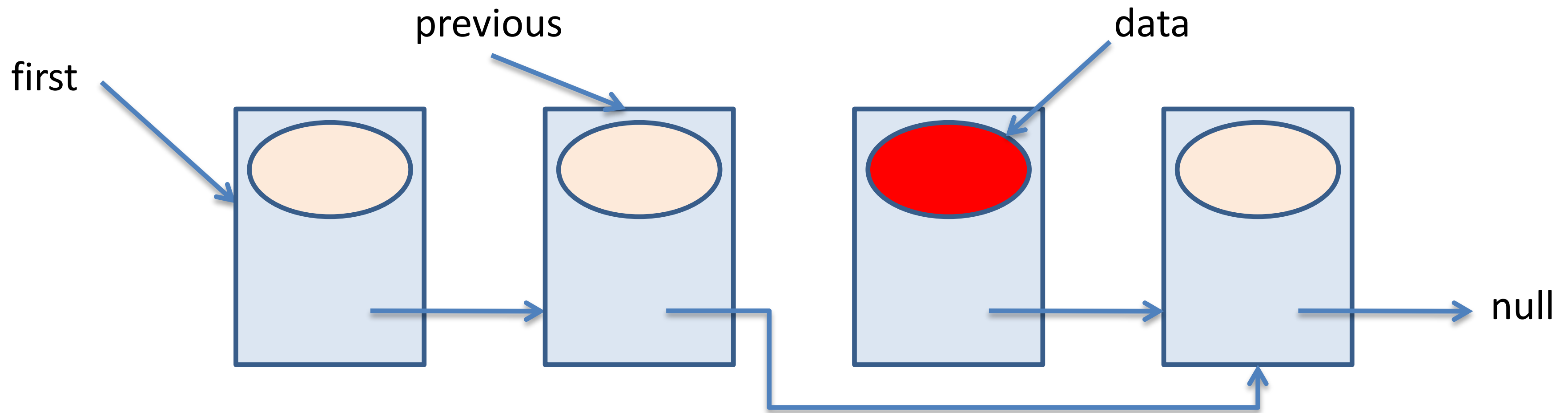


Removing an intermediate node



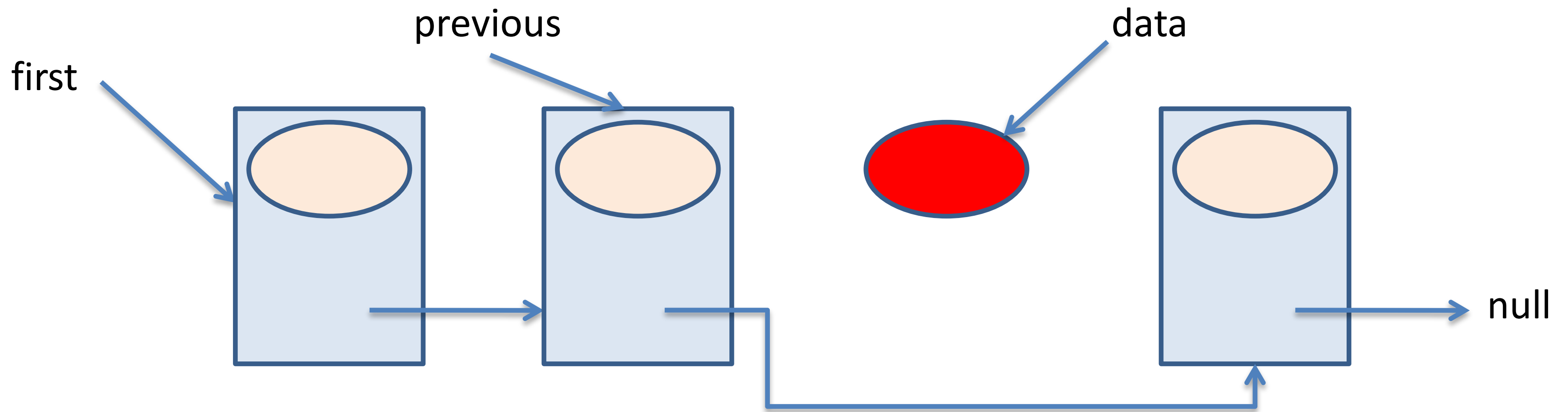
Object data = previous.getNext().getInfo();

Removing an intermediate node

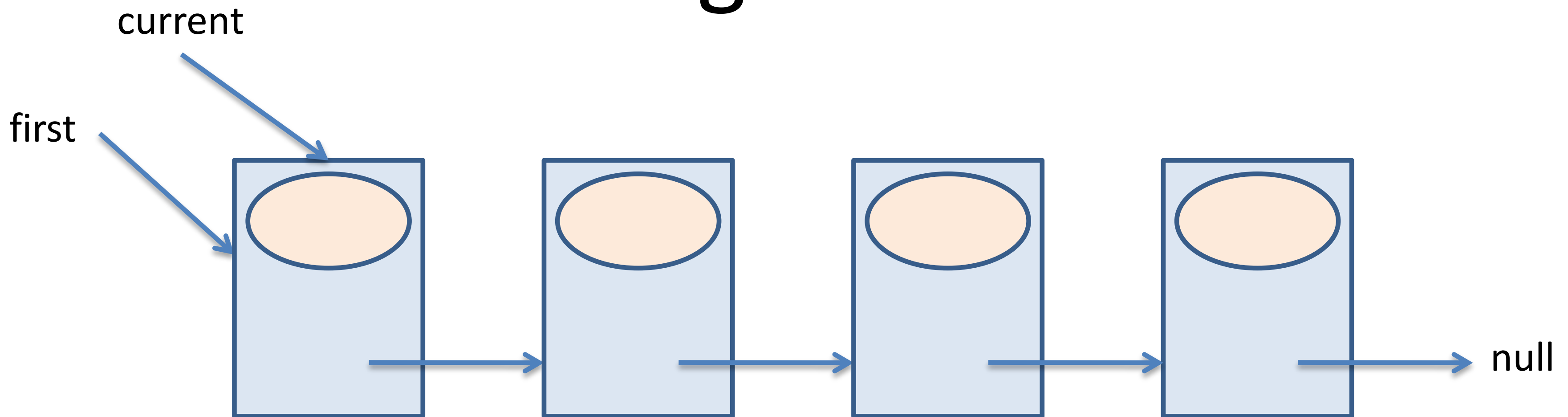


previous.setNext(previous.getNext().getNext())

Removing an intermediate node



Traversing the linked list




```
Node current = first;
while (current != null) {
    current = current.getNext();
}
```

Traversing the list: looking for the last node

- A reference steps the list until a node is reached whose reference to the next node is null:

```
public Node searchLastNode() {
    Node last = null;
    Node current = first;
    if (current != null) {
        while (current.getNext() != null) {
            current = current.getNext();
        }
        last = current;
    }
    return last;
}
```

Traversing the list: looking for a piece of data

- A reference steps the list until the piece of information is reached. A counter is used in order to return its position in the list:

```
public int search(Object info) {
    int pos = 1;
    Node current = first;
    while (current != null
           && !current.getInfo().equals(info)) {
        pos += 1;
        current = current.getNext();
    }
    if (current != null)
        return pos;
    else
        return -1;
}
```

Advantages of Linked Lists

- Inserting and extracting nodes have a cost that do not depend on the size of the list
- Concatenation and partition of lists have a cost that do not depend on the size of the list
- There is no need for contiguous memory
- Memory actually in use at a given instant depends only on the number of data items stored in the list at that instant

Disadvantages of Linked Lists

- Accessing to arbitrary intermediate positions has a cost that depends on the size of the list
- Each node represents an overhead in memory usage