Linked List

Linked List Overview

- A **linked list** is a data structure consisting of a group of nodes which together represent a sequence.
- Under the simplest form, each node is composed of a datum and a reference(in other words, a *link*) to the next node in the sequence.
- This structure allows for efficient insertion or removal of elements from any position in the sequence.
- In a list the only way to access an item is to traverse the list.

- The class Link contains some data and a reference to the next link:

```
class Link
{
public int iData; // data
public double dData; // data
public Link next; // reference to next link
}
```

- This kind of class definition is sometimes called *self-referential because it contains a* field—called next in this case—of the same type as itself.

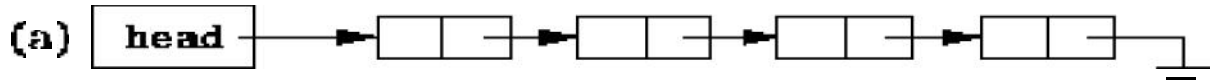
Types of Linked List

- **Singly-linked list**
  - Every element contains some data and a link to the next element, which allows to keep the structure.
- **Doubly-linked list**
  - Every node in a doubly-linked list also contains a link to the previous node.
- Linked list can be an underlying data structure to implement stack, queue or sorted list.

Singly Linked List

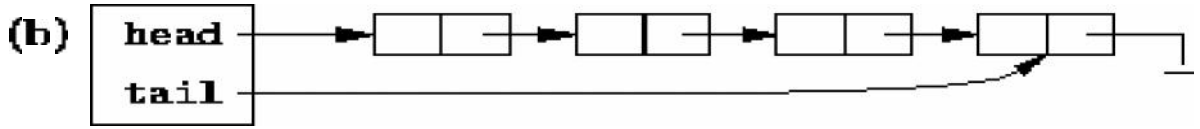
- Most basic of all the linked data structures.
- It is simply a sequence of dynamically allocated objects, each of which refers to its successor in the list.
- Each cell is called a **node** of a singly-linked list.
- First node is called **head** and it's a dedicated node.
- By knowing it, we can access every other node in the list.
- Sometimes, last node, called **tail**, is also stored in order to speed up add operation.

Singly Linked List



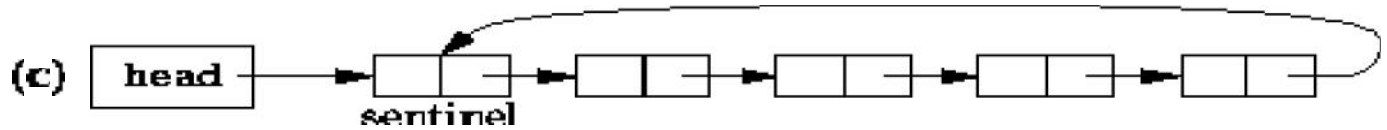
- The basic singly-linked list is inefficient in those cases when we wish to add elements to both ends of the list.
- While it is easy to add elements at the head of the list, to add elements at the other end (the *tail*) we need to locate the last element.
- If the basic singly-linked list is used, the entire list needs to be traversed in order to find its tail.

Singly Linked List



- The solution uses a second variable, tail , which refers to the last element of the list.
- Efficiency comes at the cost of the additional space used to store the variable tail.

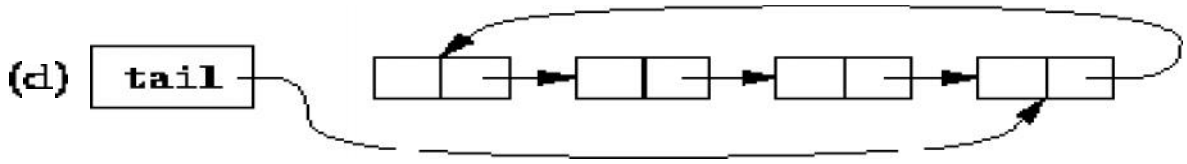
Singly Linked List (Circular)



- An extra element at the head of the list called a *sentinel which does not hold data* is used.
- Using a sentinel simplifies the programming of certain operations.
  - We need not modify the *head* variable.
  - Disadvantage is extra space required and creating sentinel when the list is initialized.

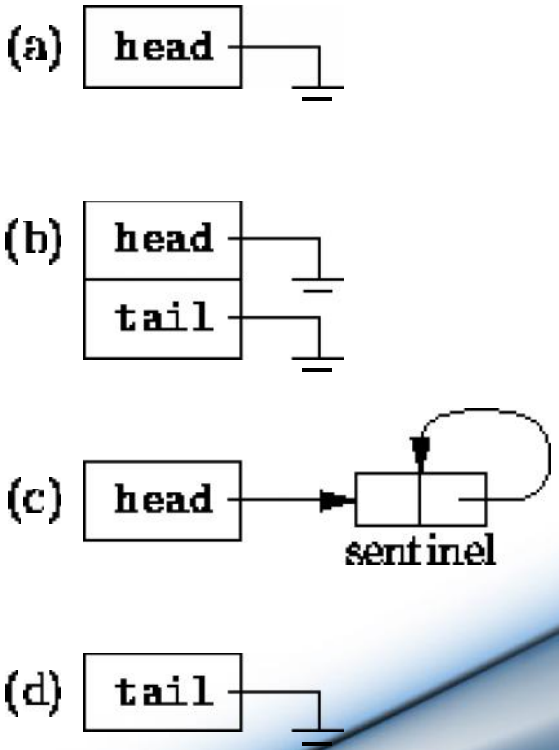
Singly Linked List

- Also called *circular list*.

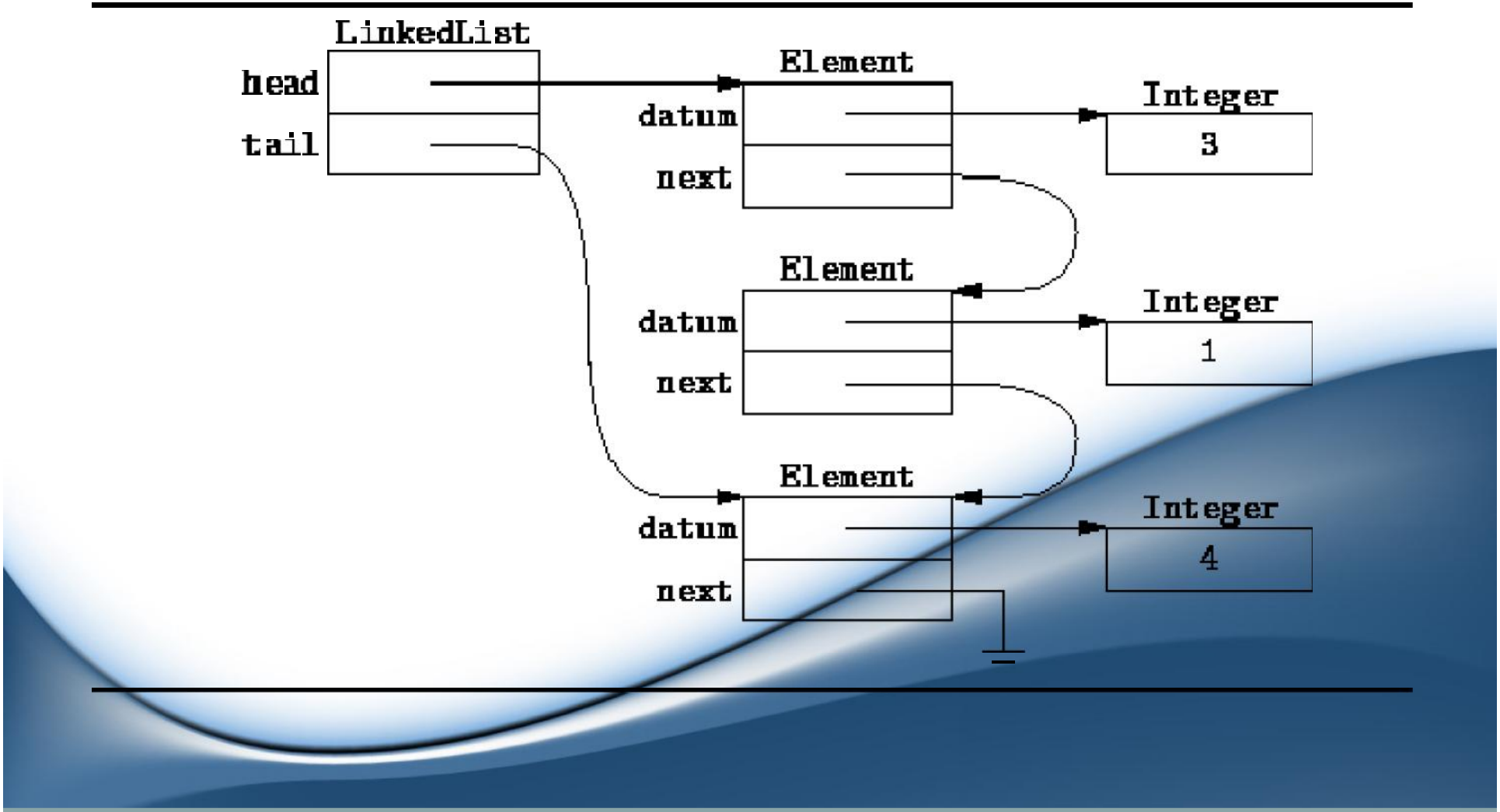


- tail refers to the last element of the list.
- The last element follows the first element of the list.
- Advantage is, insertion at the head of the list, insertion at the tail of the list, and insertion at an arbitrary position of the list are all identical operations.

Empty List



List with Data



Operations in a simple linked list

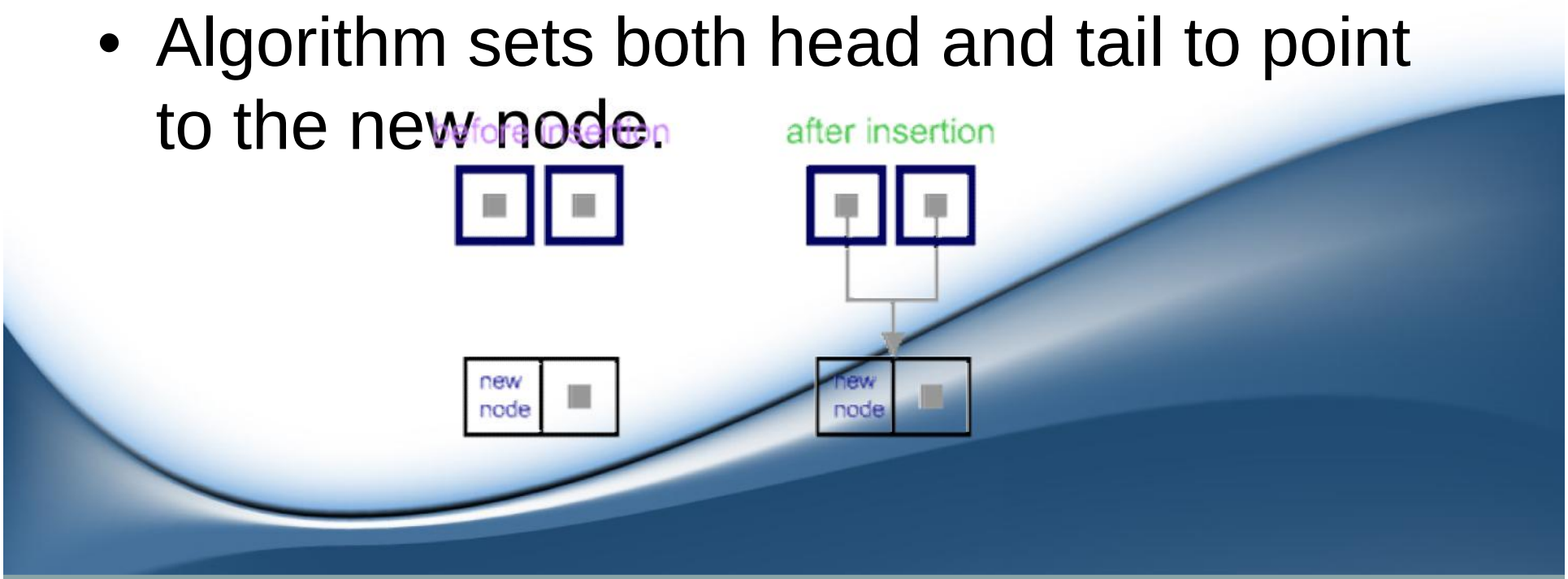
- Insertion
- Deletion
- Traversing
- Searching

Addition (insertion) operation

- Insertion into a singly-linked list has two special cases.
  - Inserting a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list).
  - In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list.

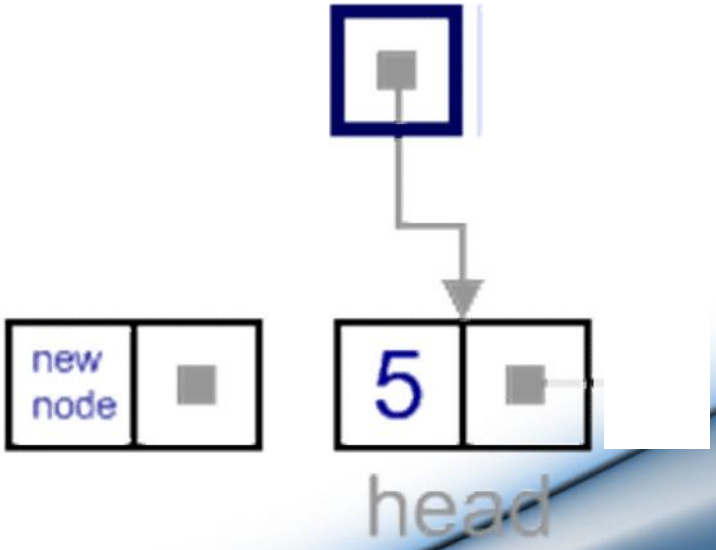
Empty list case

- When list is empty, which is indicated by (head == null)condition, the insertion is quite simple.
- Algorithm sets both head and tail to point to the new node.

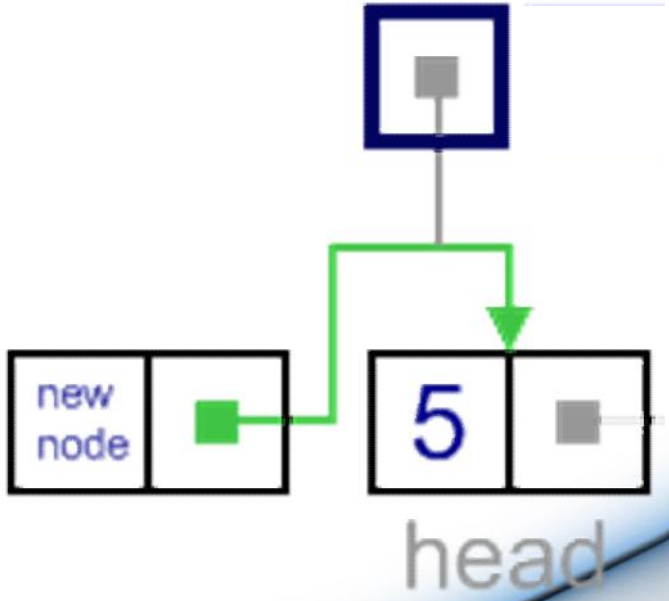


Add first

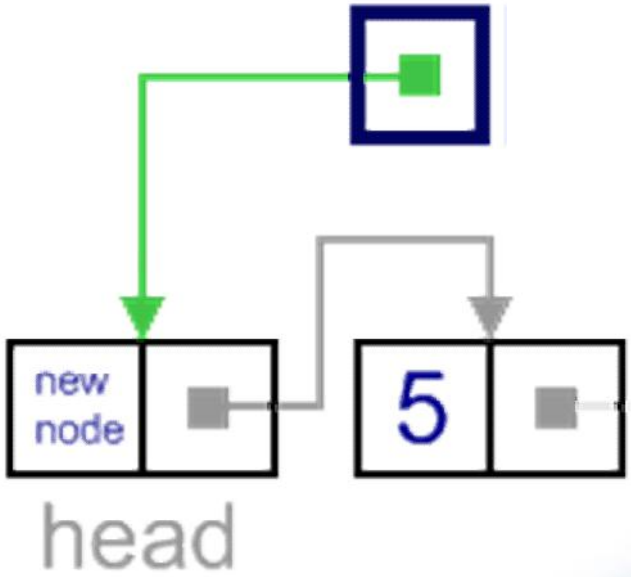
- In this case, new node is inserted right before the current head node.



- It can be done in two steps:
  1. Update the next link of a new node, to point to the current head node.

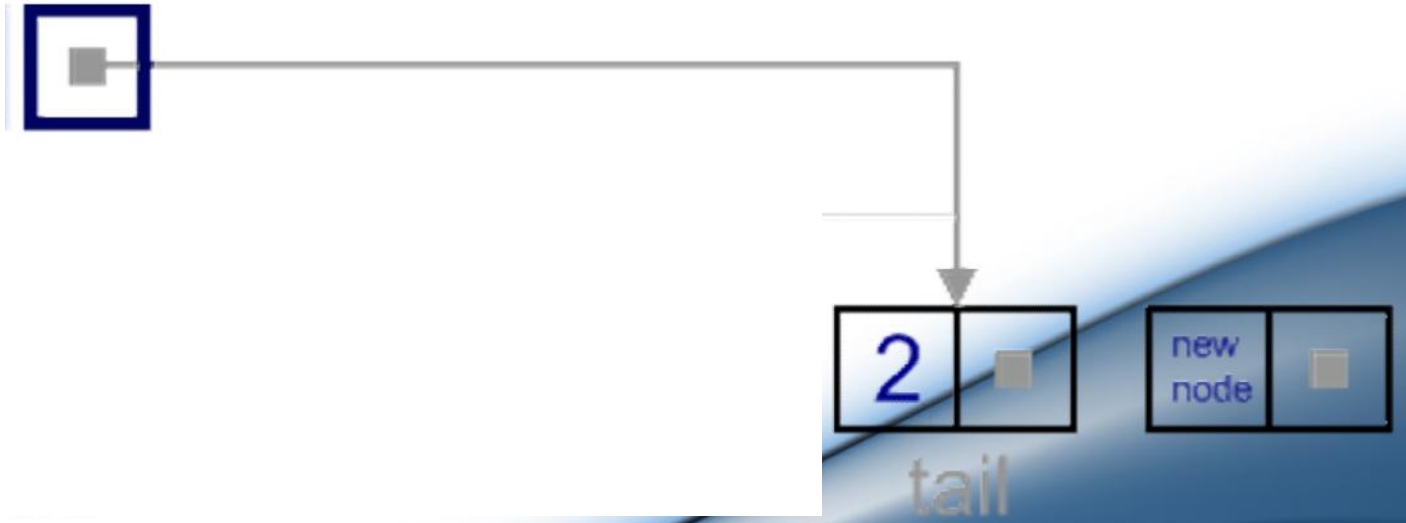


2. Update head link to point to the new node.

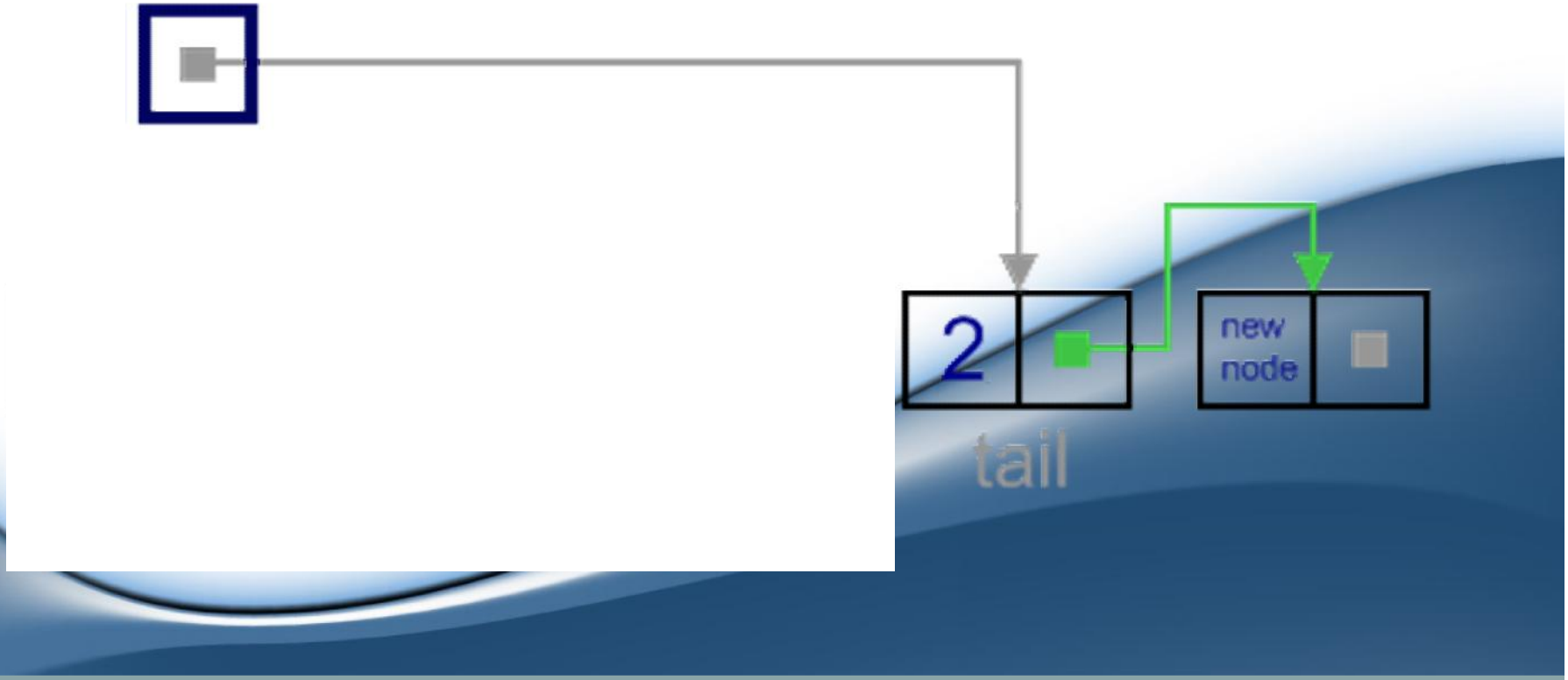


Add last

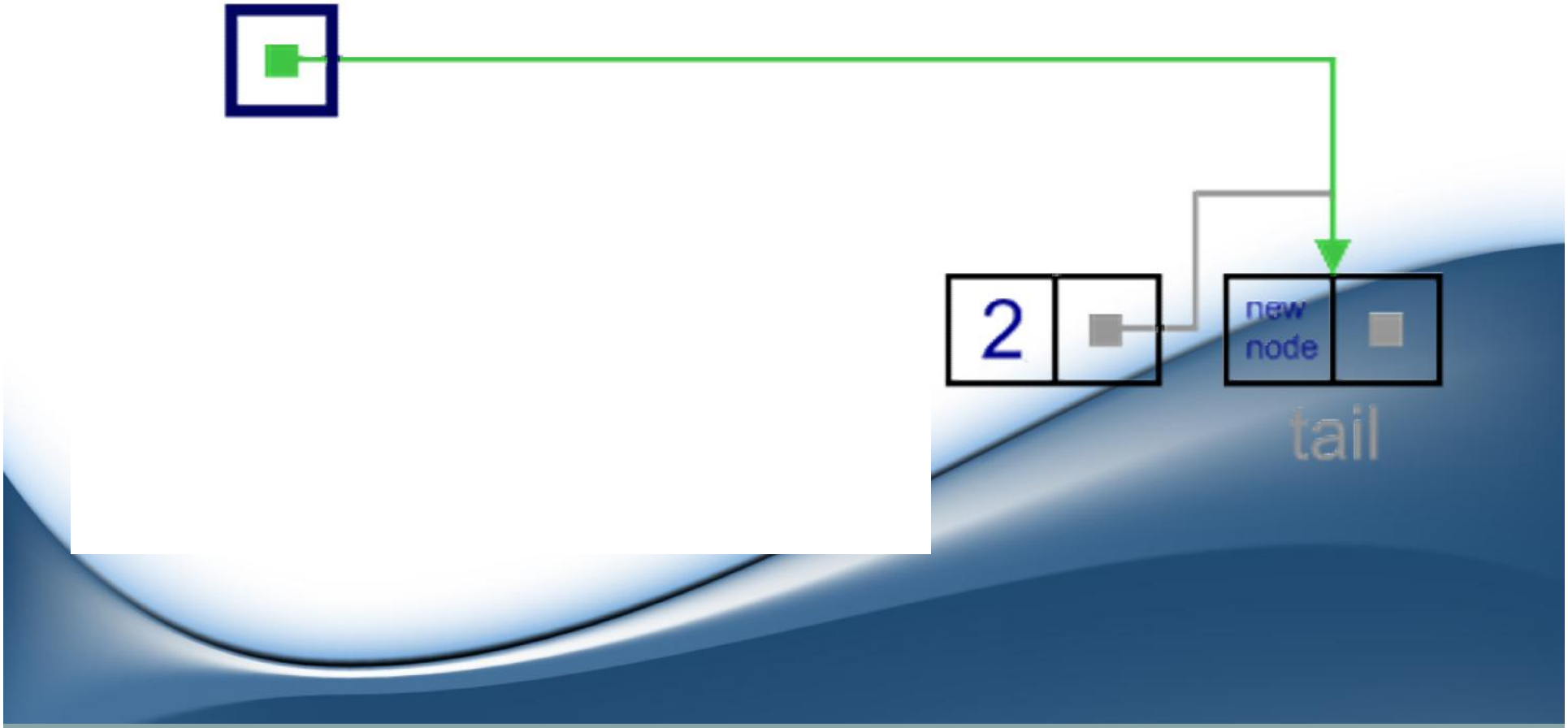
- In this case, new node is inserted right after the current tail node.



- It can be done in two steps:
  1. Update the next link of the current tail node, to point to the new node.

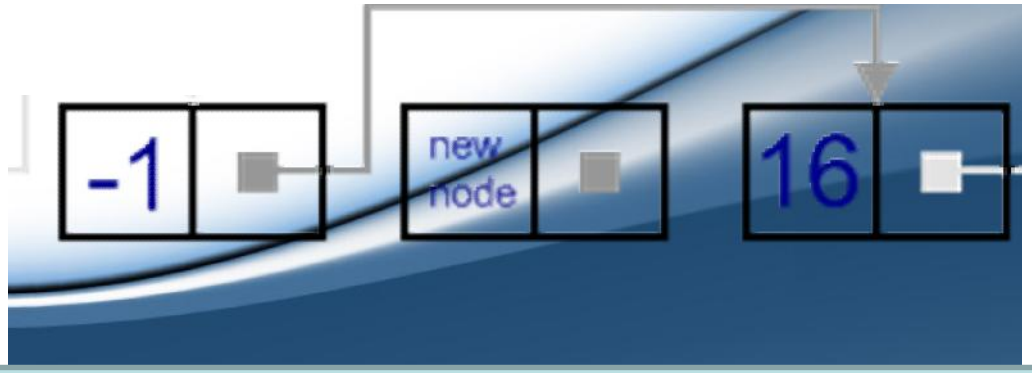


2. Update tail link to point to the new node.

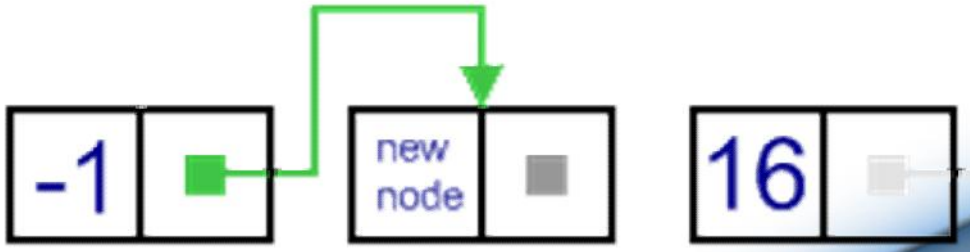


General case

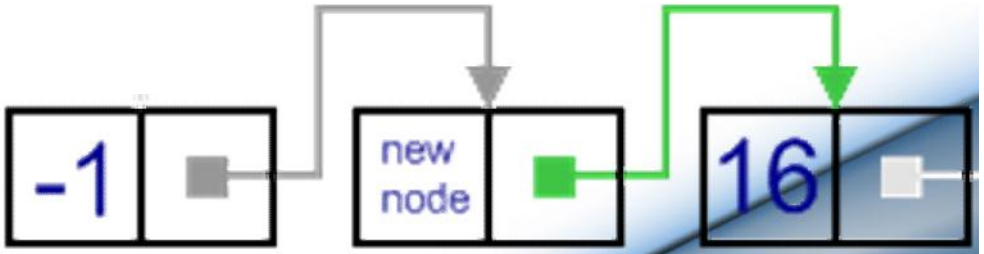
- In general case, new node is **always inserted between** two nodes, which are already in the list.
- Head and tail links are not updated in this case.



- Such an insert can be done in two steps:
  1. Update link of the "previous" node, to point to the new node.



2. Update link of the new node, to point to the "next" node.

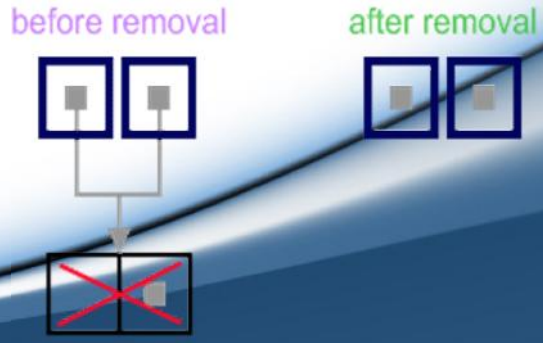


Removal (deletion) operation

- There are four cases, which can occur while removing the node.
- These cases are similar to the cases in add operation.
- We have the same four situations, but the order of algorithm actions is opposite.
- Notice, that removal algorithm includes the disposal of the deleted node, which may be unnecessary in languages with automatic garbage collection (i.e., Java).

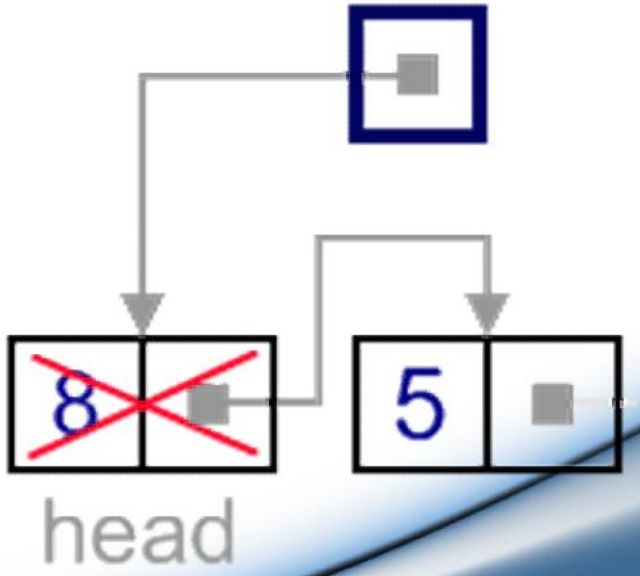
List has only one node

- When list has only one node, which is indicated by the condition, that the head points to the same node as the tail, the removal is quite simple.
- Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to *NULL*.

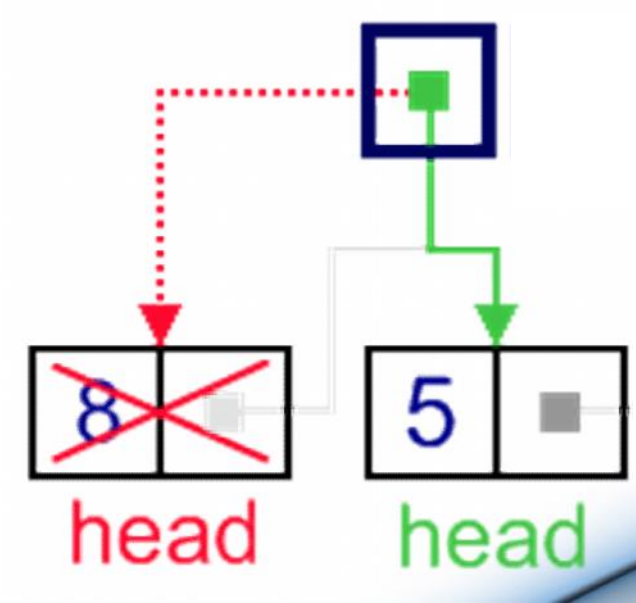


Remove first

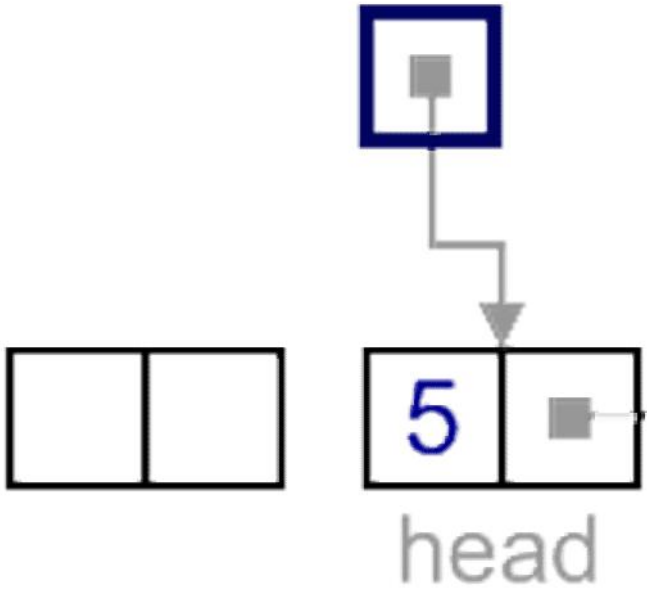
- In this case, first node (current head node) is removed from the list.



- It can be done in two steps:
  1. Update head link to point to the node, next to the head.

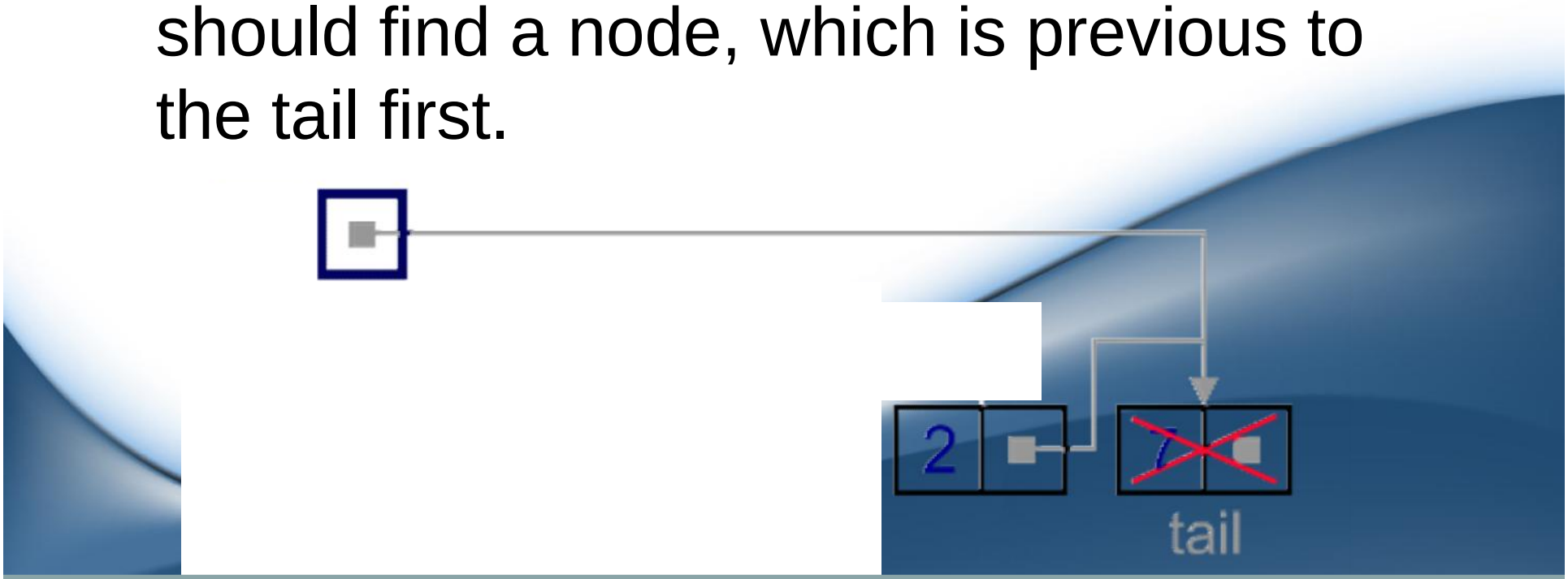


2. Dispose removed node.

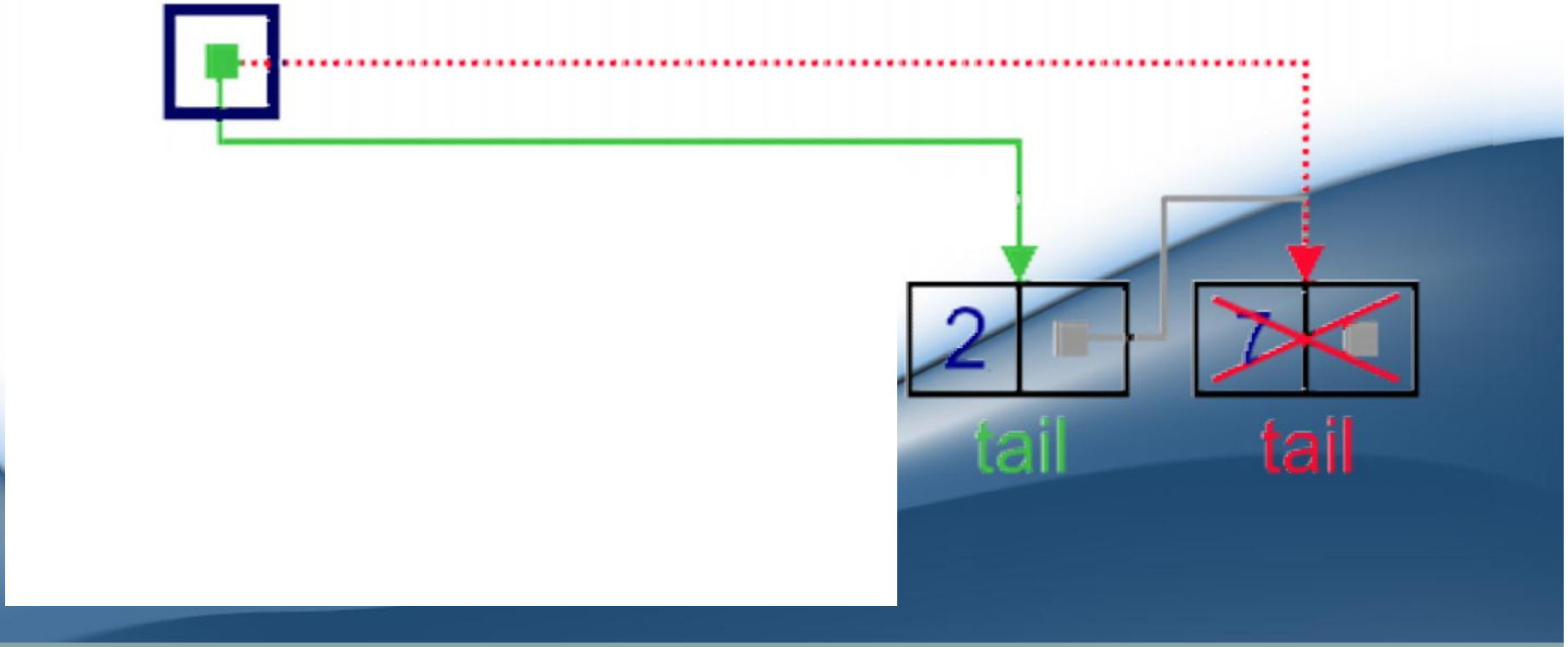


Remove last

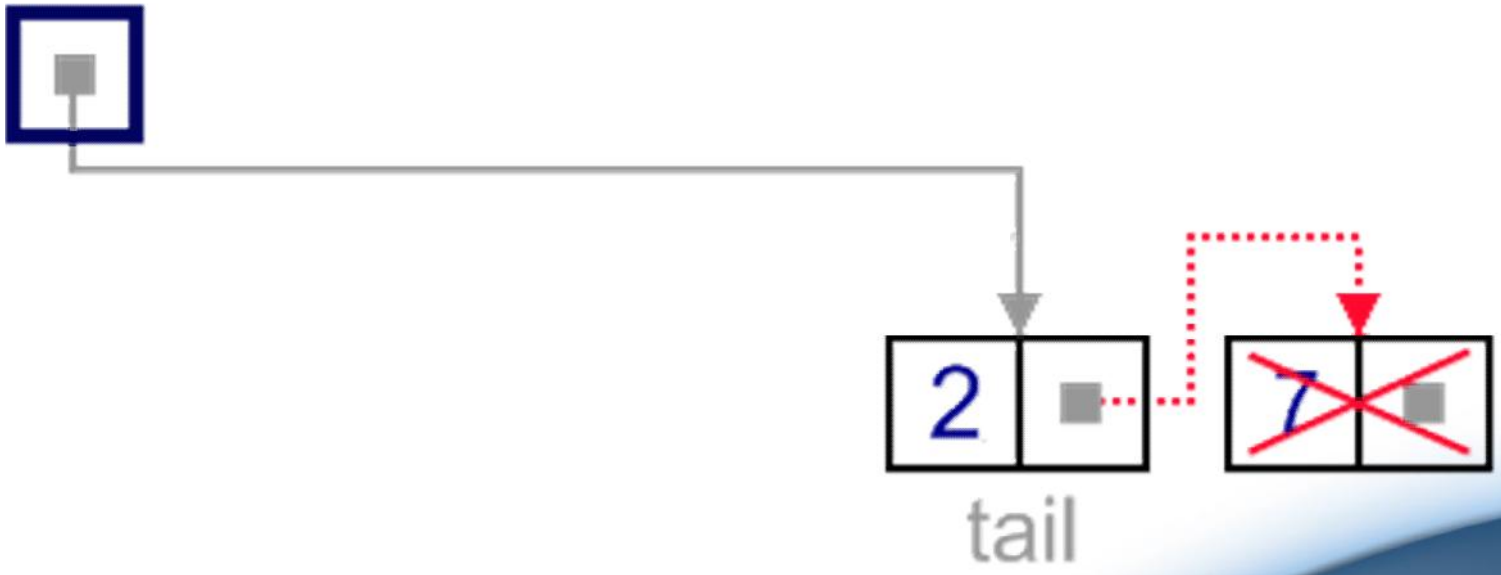
- In this case, last node (current tail node) is removed from the list.
- This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.



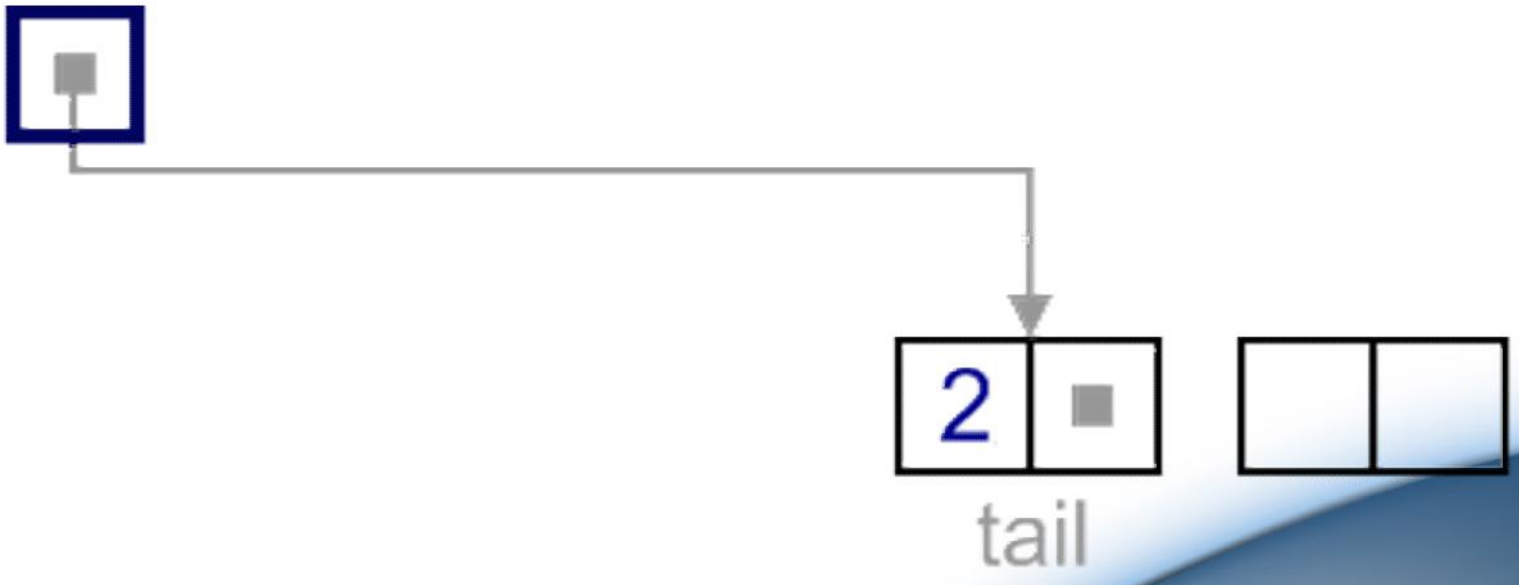
- It can be done in three steps:
  1. Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.



2. Set next link of the new tail to NULL.

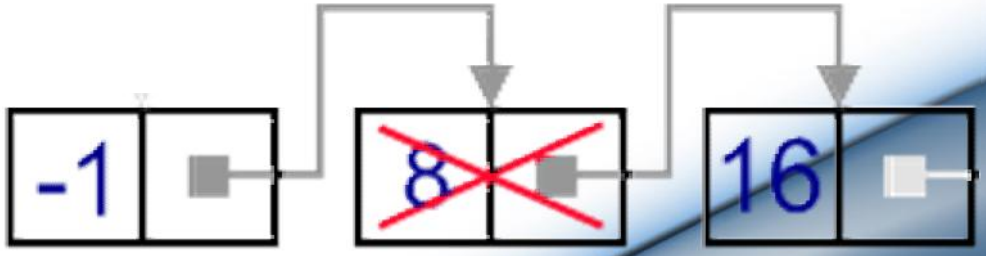


3. Dispose removed node.

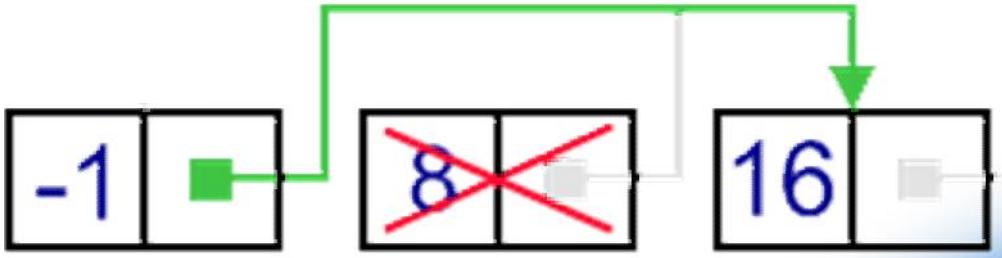


General case

- In general case, node to be removed is **always located between** two list nodes.
- Head and tail links are not updated in this case.



- Such a removal can be done in two steps:
  1. Update next link of the previous node, to point to the next node, relative to the removed node.



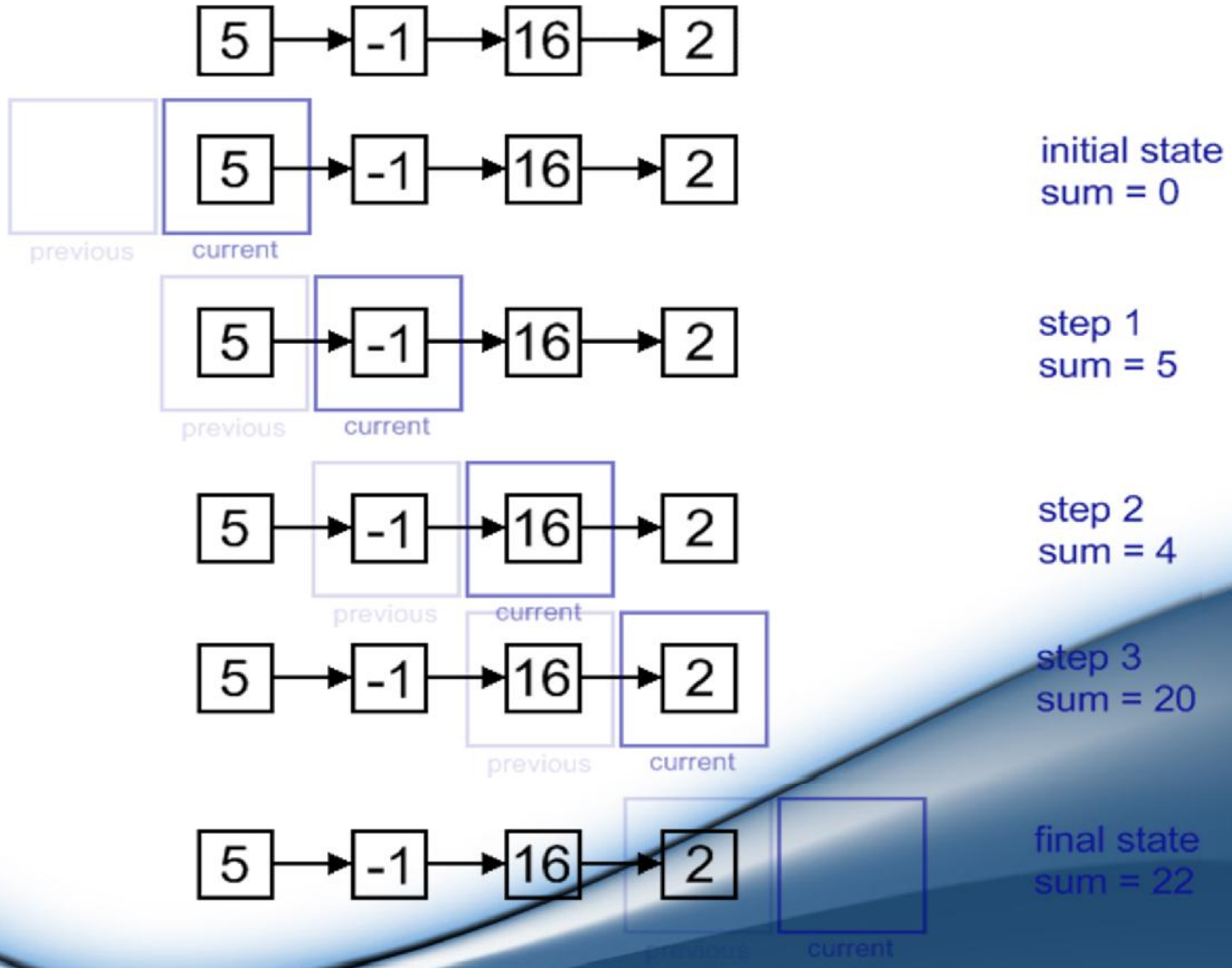
- Dispose removed node.

Singly-linked list-Traversal

- Assume, that we have a list with some nodes.
- Traversal is the very basic operation, which presents as a part in almost every operation on a singly-linked list.
- For instance, algorithm may traverse a singly-linked list to find a value, find a position for insertion, etc.
- For a singly-linked list, only forward direction traversal is possible.

Traversal algorithm

- Beginning from the head, check, if the end of a list hasn't been reached yet;
  1. Do some actions with the current node, which is specific for particular algorithm;
  2. Current node becomes previous and next node becomes current.
  3. Go to the step 1.
- As for example, let us see an example of summing up values in a singly-linked list.


5
-1
16
2
previous
current
initial state
sum = 0
step 1
sum = 5
step 2
sum = 4
step 3
sum = 20
final state
sum = 22

```
private class Node
{
// reference to the next node in the chain, or null if there isn't one.
Node next;
Object data;
public Node(Object  n_data)
{ next = null;
data = n_data;
}
 // another Node constructor if we want to specify the node to point to.
 public Node(Object  n_data, Node  n_next)
 { next = n_next;
    data =n_data;
} public Object getData()
 { return data; }
 public void setData(Object n_data)
{ data = n_data; }
public Node getNext()
 { return next; }
 public void setNext(Node n_next)
 { next = n_next; }
} }
```

```
public class LinkedList
{
private int listCount;
private Node head; // reference to the head node.
public LinkedList()
{
// this is an empty list, so the reference to the head node is set to a new node with no data
head = new Node(null);
listCount = 0;
}
 public void add(Object data) // appends the specified element to the end of this list.
 {
      Node temp = new Node(data);
      Node current = head;
      // starting at the head node, crawl to the end of the list
      while(current.getNext() != null)
      {
               current = current.getNext();
      }
current.setNext(temp); // the last node's "next" reference set to our new node
listCount++;    // increment the number of elements variable
}
```

```
public void add(Object data, int index)
// inserts the specified element at the specified position in this list.
{
Node temp = new Node(data);
Node current = head;
// crawl to the requested index or the last element in the list, whichever
    comes first
for(int i = 1; i < index && current.getNext() != null; i++)
{
    current = current.getNext();
}
 // set the new node's next-node reference to this node's next-node
    reference
temp.setNext(current.getNext());
// now set this node's next-node reference to the new node
current.setNext(temp);
listCount++;  // increment the number of elements variable
 }
```

```
public Object get(int index)
// returns the element at the specified position in this list.
{
// index must be 1 or higher
if(index <= 0)
    return null;
Node current = head;
for(int i = 1; i < index; i++)
{
if(current.getNext() == null)
    return null;
current = current.getNext();
}
 return current.getData();
}
```

```
public boolean remove(int index)
// removes the element at the specified position in this list.
{
// if the index is out of range, exit
if(index < 1 || index > size())
    return false;
Node current = head;
for(int i = 1; i < index; i++)
{
    if(current.getNext() == null)
        return false;
    current = current.getNext();
}
current.setNext(current.getNext().getNext());
listCount--; // decrement the number of elements variable
 return true;
 }
```

```
public int size()
// post: returns the number of elements in this list.
{
    return listCount;
}
public String toString()
{
    Node current = head;
    String output = "";
    while(current != null)
    {
        output += "[" + current.getData().toString() + "]";
        current = current.getNext();
    }
return output;
 }
```

LinkedList Class

- Provides a linked-list data structure.
- Constructors
  - LinkedList( )
    - builds an empty linked list
  - LinkedList(Collection c)
    - builds a linked list that is initialized with the elements of the collection c.

Methods

- **void add(int index, Object element)**
  - Inserts the specified element at the specified position index in this list.
  - Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
- **boolean add(Object o)**
  - Appends the specified element to the end of this list.

- **boolean addAll(Collection c)**
  - Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
  - Throws NullPointerException if the specified collection is null.
- **boolean addAll(int index, Collection c)**
  - Inserts all of the elements in the specified collection into this list, starting at the specified position.
  - Throws NullPointerException if the specified collection is null.

- **void addFirst(Object o)**
  - Inserts the given element at the beginning of this list.
- **void addLast(Object o)**
  - Appends the given element to the end of this list.
- **void clear()**
  - Removes all of the elements from this list.

- **boolean contains(Object o)**
  - Returns true if this list contains the specified element.
  - More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
- **Object get(int index)**
  - Returns the element at the specified position in this list.
  - Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()).
- **Object getFirst()**
  - Returns the first element in this list.
  - Throws NoSuchElementException if this list is empty.

- **Object getLast()**
  - Returns the last element in this list. Throws NoSuchElementException if this list is empty.
- **int indexOf(Object o)**
  - Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
- **int lastIndexOf(Object o)**
  - Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
- **Object remove(int index)**
  - Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.

- **boolean remove(Object o)**
  - Removes the first occurrence of the specified element in this list. Throws NoSuchElementException if this list is empty. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()).
- **Object removeFirst()**
  - Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty.
- **Object removeLast()**
  - Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.

- **Object set(int index, Object element)**
  - Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
- **int size()**
  - Returns the number of elements in this list.
- **Object[] toArray()**
  - Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.

```
import java.util.*;
 class LinkedListDemo
 {
public static void main(String args[]) { // create a linked list
LinkedList ll = new LinkedList(); // add elements to the linked list
ll.add("F");
ll.add("B");
 ll.add("D");
 ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
```

```
// remove elements from the linked list
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: " + ll);
// remove first and last elements
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: " + ll);
// get and set a value
Object val = ll.get(2);
ll.set(2, (String) val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

output

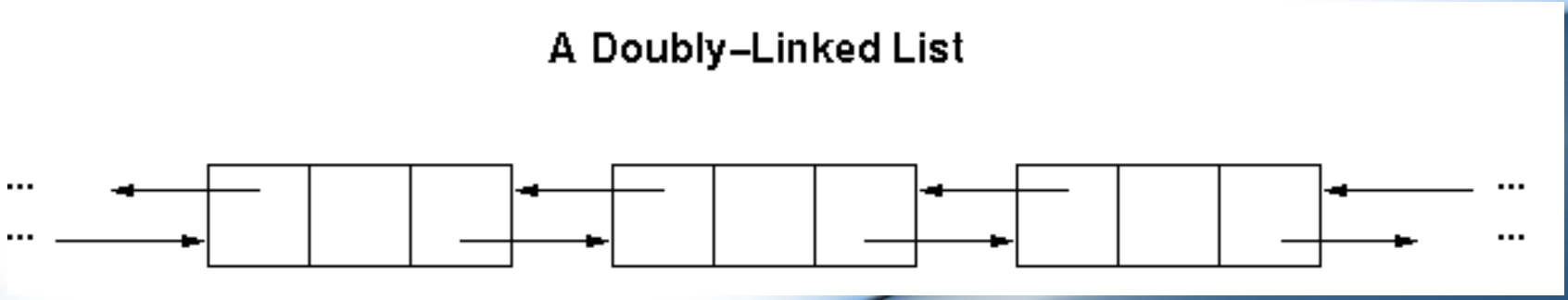
Original contents of ll: [A, A2, F, B, D, E, C, Z] Contents of ll after deletion: [A, A2, D, E, C, Z]

ll after deleting first and last: [A2, D, E, C]

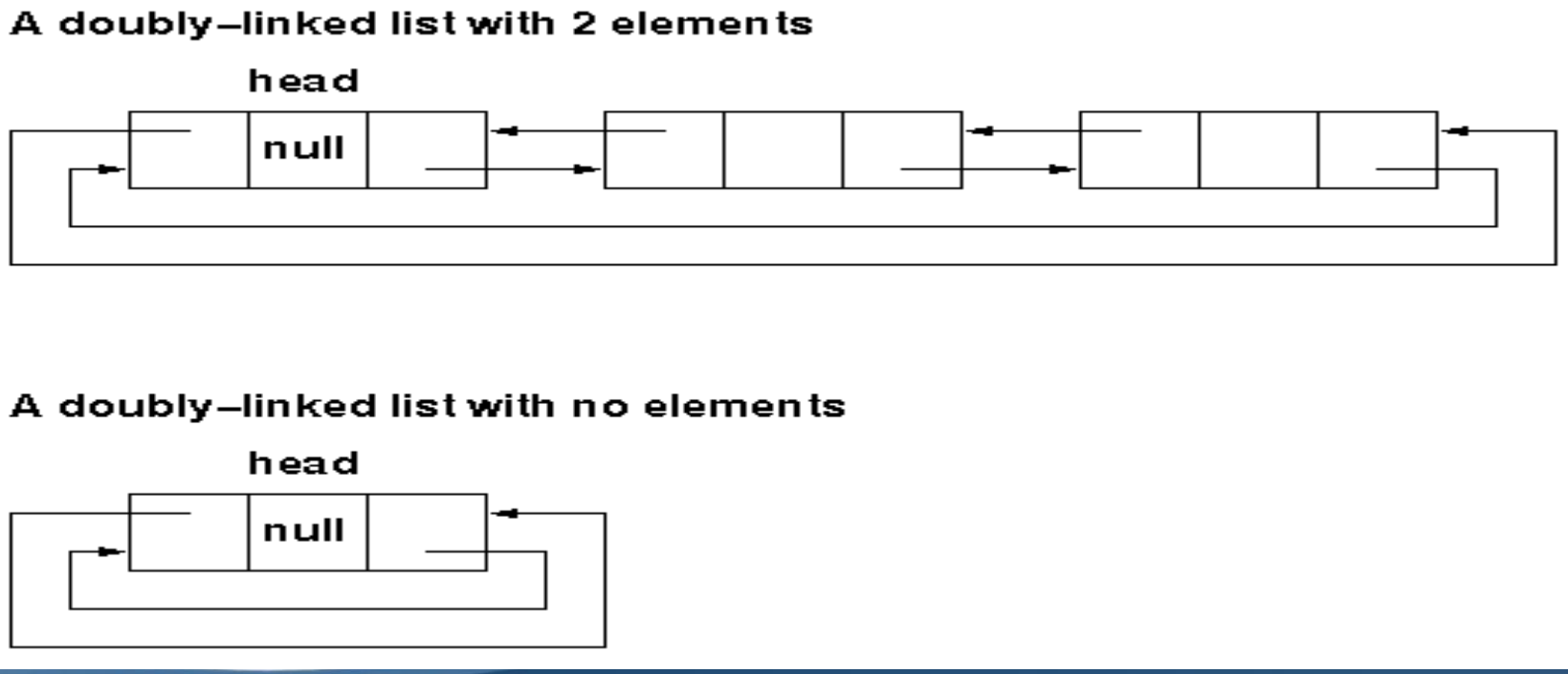
ll after change: [A2, D, E Changed, C]

Doubly-Linked Lists

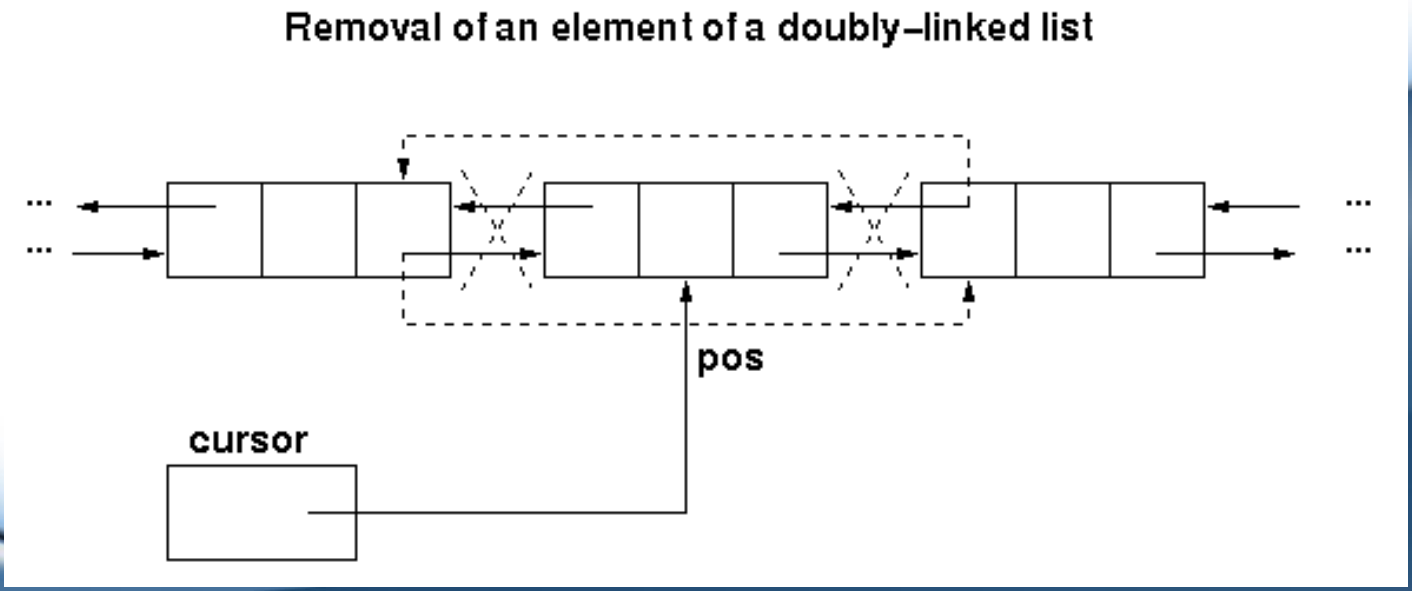
- Allows traversing a list both forward and backward efficiently.
- A list element contains the data plus pointers to the next and previous list items.



- Needs a pointer to some *link* in the doubly-linked list to access list elements.
- Convenient for doubly-linked lists to use a *list header*, or *head*, that has the same structure as any other list item.
- By following arrows it is easy to go from the list header to the first and last list element, respectively.



- Insertion and removal of an element in a doubly-linked list is easy.
- In the picture below the pointer changes for a removal of a list item (old pointers have been drawn solid and new pointers are dashed arrows).
- We first locate the previous list item using the *previous* field.
- We make the *next* field of this list item point to the item following the one in cursor position *pos*.
- Then we make the *previous* field of this following item point to the item preceding the one in the cursor position *pos*.
- The list item pointed to by the cursor becomes useless and should be automatically garbage collected.



Implementing the doubly linked List

```
class Link
{
   public long dData;              // data item
   public Link next;               // next link in list
   public Link previous;           // previous link in list

   public Link(long d)             // constructor
   {
         dData = d;
         next=previous=null;
   }

   public void displayLink()       // display this link
  {
    System.out.print(dData + " ");
   }
 }  // end class Link
```

Implementing the Doubly-Linked List

```
class DoublyLinkedList
{
    Link first;              // ref to first item
    Link last;               // ref to last item

   public DoublyLinkedList()        // constructor
  {
      first = null;                 // no items on list yet
      last = null;
   }

   public boolean isEmpty()         // true if no links
  { return first==null; }
```

Implementing the Doubly-Linked List

```
public void insertFirst(long dd)  // insert at front of list
{
     Link newLink = new Link(dd);   // make new link

     if( isEmpty() )              // if empty list
     {
        first=last = newLink;            // newLink <-- last
    System.out.println("first next:"+first.next);
     }
     else
    {
        first.previous = newLink;
        newLink.next = first;
        newLink.previous=null;
        first = newLink;             // first --> newLink
    }
}
```

```
 public void insertLast(long dd)    // insert at end of list
{
      Link newLink = new Link(dd);    // make new link
      if( isEmpty() )                          // if empty list
         first =last= newLink;               // first --> newLink
      else
      {
         last.next = newLink;         // old last --> newLink
         newLink.previous = last;    //old last <-- newLink
       }
      last = newLink;                    // newLink <-- last
}
```

```
public Link deleteFirst()         // delete first link
    {                             // (assumes non-empty list)
    Link temp = first;
    if(first.next == null)          // if only one item
       last = null;                 // null <-- last
    else
       first.next.previous = null; // null <-- old next
    first = first.next;             // first --> old next
    return temp;
    }
```

```
public Link deleteLast()         // delete last link
   {                             // (assumes non-empty
  list)
    Link temp = last;
    if(first.next == null)       // if only one item
      first = null;              // first --> null
    else
      last.previous.next = null; //old previous -->
  null
    last = last.previous;        // old previous <--
  last
    return temp;
    }
```

```
public boolean insertAfter(long key, long dd)
    {                           // (assumes non-empty list)
    Link current = first;          // start at beginning
    while(current.dData != key)    // until match is found,
      {
      current = current.next;     // move to next link
      if(current == null)
        return false;            // didn't find it
      }
    Link newLink = new Link(dd);   // make new link

    if(current==last)              // if last link,
      {
      newLink.next = null;        // newLink --> null
      last = newLink;             // newLink <-- last
      }
    else                        // not last link,
      {
      newLink.next = current.next; // newLink --> old next
                                // newLink <-- old next
      current.next.previous = newLink;
      }
    newLink.previous = current;    // old current <-- newLink
    current.next = newLink;        // old current --> newLink
    return true;                // found it, did insertion
    }
```

```
public Link deleteKey(long key)   // delete item w/ given key
    {                             // (assumes non-empty list)
    Link current = first;         // start at beginning
    while(current.dData != key)   // until match is found,
      {
      current = current.next;     // move to next link
      if(current == null)
        return null;              // didn't find it
      }
    if(current==first)            // found it; first item?
      first = current.next;       // first --> old next
    else                          // not first
                                  // old previous --> old next
      current.previous.next = current.next;

    if(current==last)             // last item?
      last = current.previous;    // old previous <-- last
    else                          // not last
                                  // old previous <-- old next
      current.next.previous = current.previous;
    return current;               // return value
    }
```

```
public void displayForward()
    {
    System.out.print("List (first-->last): ");
    Link current = first;          // start at beginning
    while(current != null)         // until end of list
       {
       current.displayLink();      // display data
       current = current.next;     // move to next link
       }
    System.out.println("");
    }
```

```
public void displayBackward()
    {
    System.out.print("List (last-->first): ");
    Link current = last;           // start at end
    while(current != null)         // until start of list,
       {
       current.displayLink();      // display data
       current = current.previous; // move to
  previous link
       }
    System.out.println("");
    }
} // end class DoublyLinkedList
```

```
public class DoublyLinkedApp
   {
   public static void main(String[] args)
      {                              // make a new list
      DoublyLinkedList theList = new
    DoublyLinkedList();

      theList.insertFirst(22);      // insert at front
      theList.insertFirst(44);
      theList.insertFirst(66);

      theList.insertLast(11);       // insert at rear
      theList.insertLast(33);
      theList.insertLast(55);
```

```
theList.displayForward();     // display list forward
    theList.displayBackward();    // display list backward

    theList.deleteFirst();        // delete first item
    theList.deleteLast();         // delete last item
    theList.deleteKey(11);        // delete item with key 11

    theList.displayForward();     // display list forward

    theList.insertAfter(22, 77);  // insert 77 after 22
    theList.insertAfter(33, 88);  // insert 88 after 33

    theList.displayForward();     // display list forward
    }  // end main()
 }  // end class DoublyLinkedApp
```

Thank You