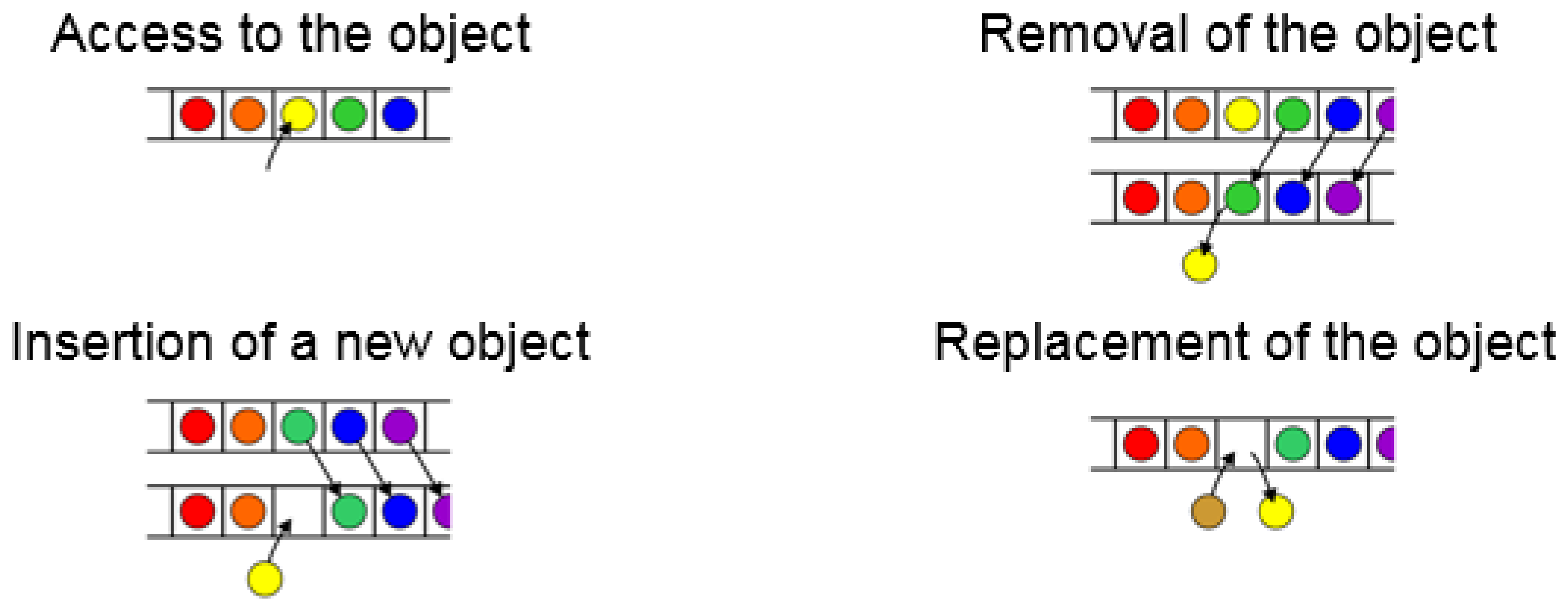
Data Structures:
LISTS

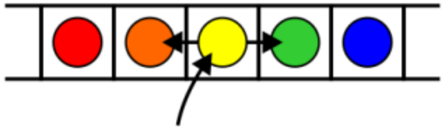
Abstract List

- An Abstract List (or List ADT) is a finite, ordered sequence of data items known as elements
- “Ordered” in this definition means that each element has a position in the list
- Operations at the i^{th} entry of the list include:



Abstract List (cont.)

- Given access to the i^{th} object, gain access to either the previous or next object



- Given two abstract lists, we may want to
  - Concatenate the two lists
  - Determine if one is a sub-list of the other

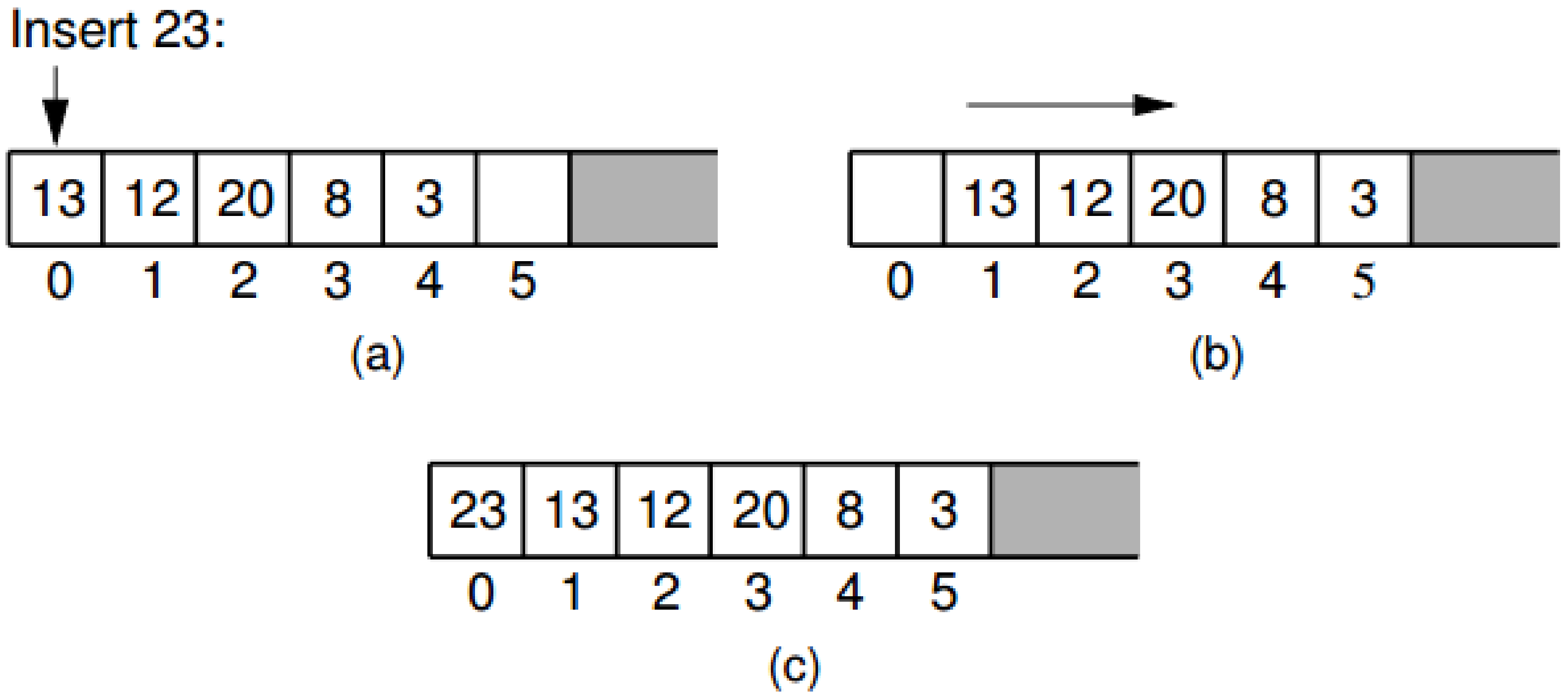
- The most obvious data structures for implementing an abstract list are ***arrays*** and ***linked lists***

Array-Based List

- `ArrayList.java` shows the array-based list implementation
- `ArrayList` implements all operations in List interface
- Class `ArrayList` stores the list elements in the first `listSize` contiguous array positions
- Array positions correspond to list positions (i.e, the element at position i in the list is stored at array cell i
- The head of the list is always at position 0
  - This makes random access to any element in the list easy
  - Given some position in the list, the value of the element in that position can be accessed directly

Array-Based List (cont.)

- Inserting an element at the head of an array-based list requires shifting all existing elements in the array by one position toward the tail



(a)

(b)

(c)

Linked List

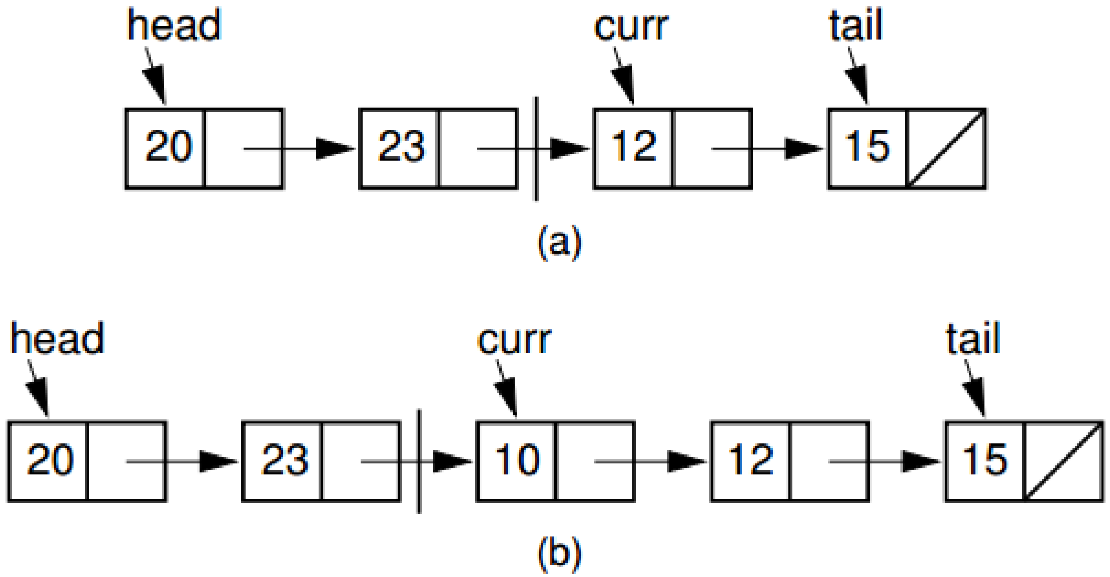
- The second traditional approach to implementing lists makes use of pointers and is usually called a **linked list**
- The linked list uses dynamic memory allocation, that is, it allocates memory for new list elements as needed
- A linked list is made up of a series of objects, called the **nodes**
- Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class

Singly Linked List

- `Link.java` shows the implementation of list nodes
- Objects in the `Link` class contain an element field to store the element value, and a next field to store a pointer to the next node on the list
- The list built from such nodes is called a **singly linked list**, or a **one-way list**
- The list's first node is accessed from a pointer named `head`
- To speed access to the end of the list a pointer named `tail` is also kept to the last link of the list
- The position of the current element is indicated by another pointer, named `curr`

Singly Linked List (cont.)

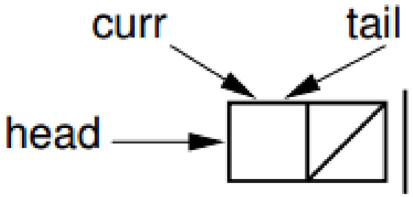
- A key design decision for the linked list implementation is how to represent the current position
  - a pointer `curr` pointing to the current element?



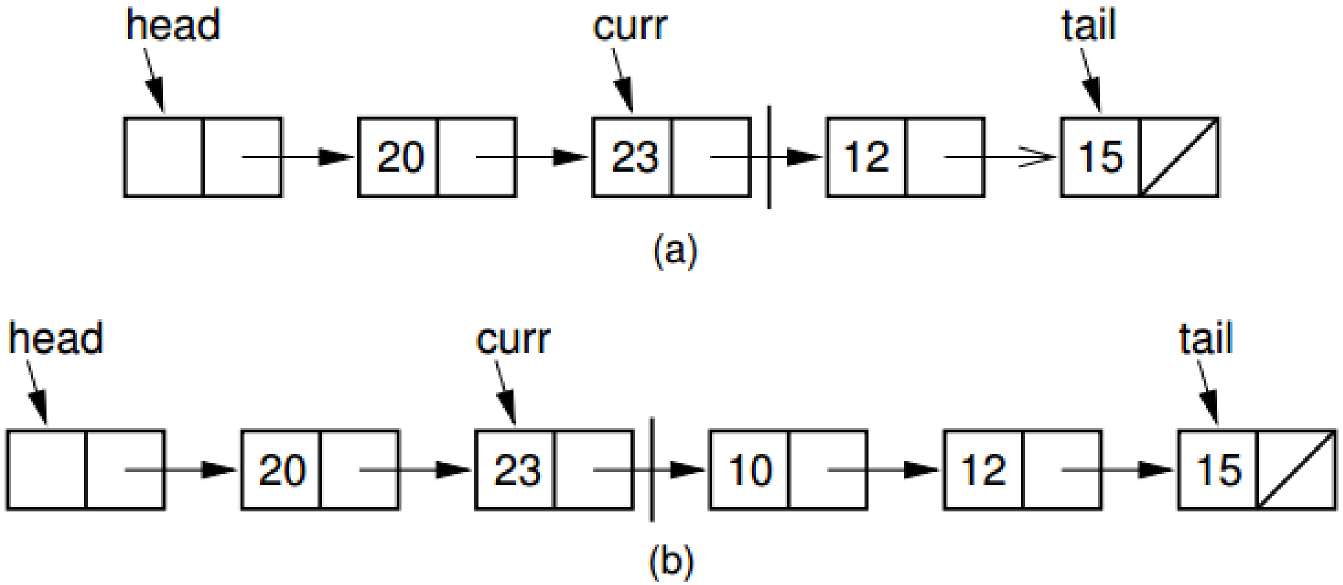
- Therefore, we set `curr` to point directly to the preceding element of the current element

Singly Linked List (cont.)

- When the list is empty we have no element for `head`, `tail`, and `curr` to point to
  - This problem can be solved by implementing linked lists with an additional header node as the first node of the list



Initial state of a linked list whenusing a header node

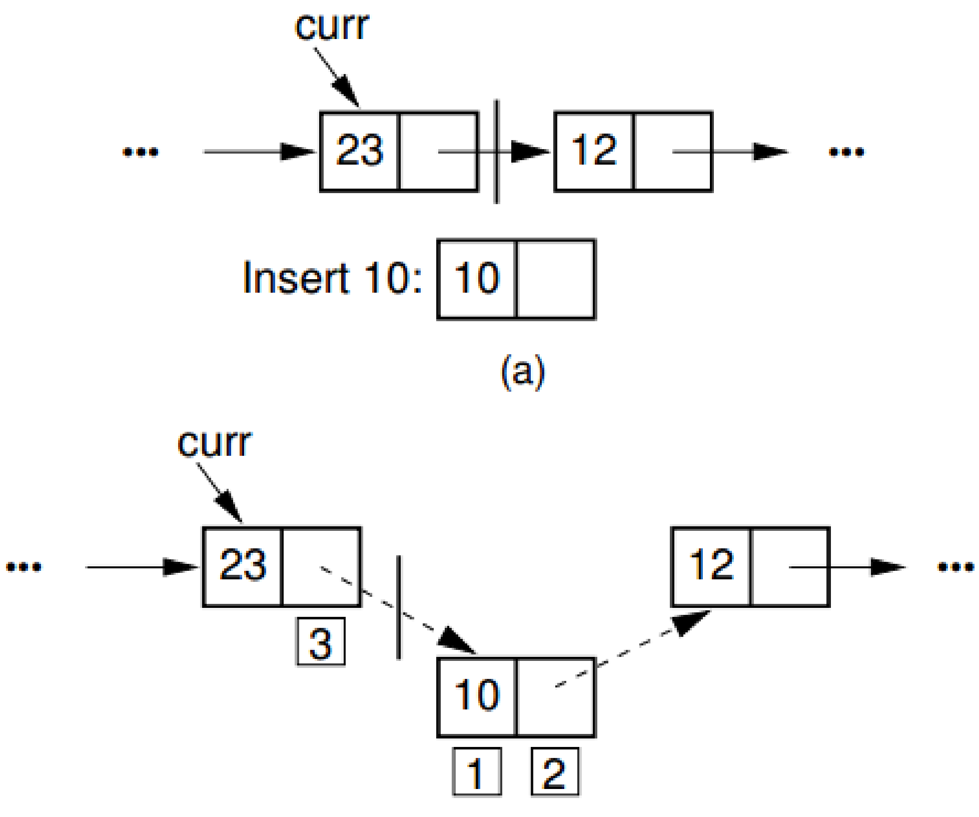


(a)

(b)

Singly Linked List (cont.)

- `LinkedList.java` shows the definition of the singly linked list class
- Inserting a new element is a three-step process:
  - The new list node is created and the new element is stored into it
  - The `next` field of the new list node is assigned to point to the current node (the one after the node that `curr` points to)
  - The `next` field of node pointed to by `curr` is assigned to point to the newly inserted node

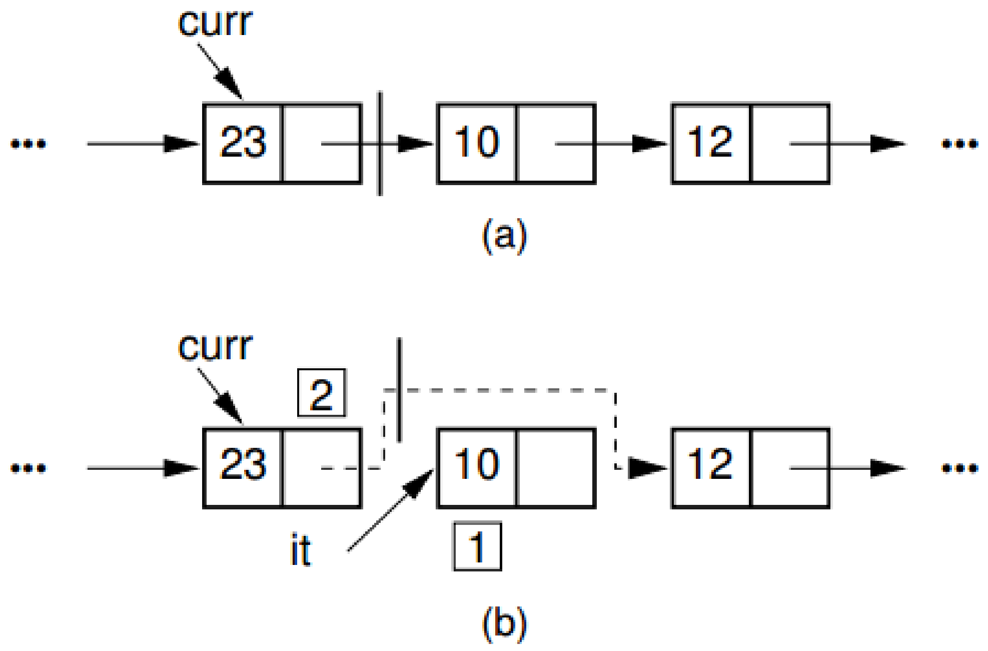


(a)

(b)

Singly Linked List (cont.)

- Removing a node from the linked list requires only that the appropriate pointer be redirected around the node to be deleted



Freelist

- The new operator is relatively expensive to use; garbage collection is also expensive
- List nodes are created and deleted in a linked list implementation in a way that allows the Link class programmer to provide simple but efficient memory management routines
- A **freelist** holds those list nodes that are not currently being used
  - When a node is deleted from a linked list, it is placed at the head of the freelist
  - When a new element is to be added to a linked list, the node is taken from the freelist if a list node is available (if the freelist is empty, the standard new operator must then be called)

Freelist (cont.)

- A new `Link.java` shows the link class with freelist support
- Below are `LinkedList` class members that are modified to use the freelist version of the `Link` class

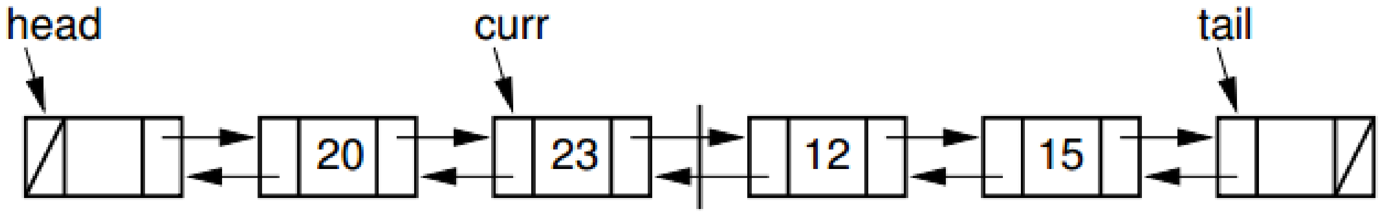
```
/** Insert "ele" at current position */
public void insert(E ele) {
       curr.setNext(Link.get(ele, curr.next())); // Get link
       if (tail == curr) tail = curr.next(); // New tail
       listSize++;
}

/** Append "it" to list */
public void append(E ele) {
       tail = tail.setNext(Link.get(ele, null));
       listSize++;
}
```

Freelist (cont.)

```
/** Remove and return current element */
public E remove() {
         // Nothing to remove
        if (curr.next() == null) return null;
         // Remember value
        E ele = curr.next().element();
         // Removed last
        if (tail == curr.next()) tail = curr;
        Link<E> tempptr = curr.next(); // Remember link
         // Remove from list
        curr.setNext(curr.next().next());
        tempptr.release(); // Release link
        listSize--; // Decrement listSize
        return ele; // Return removed
}
```

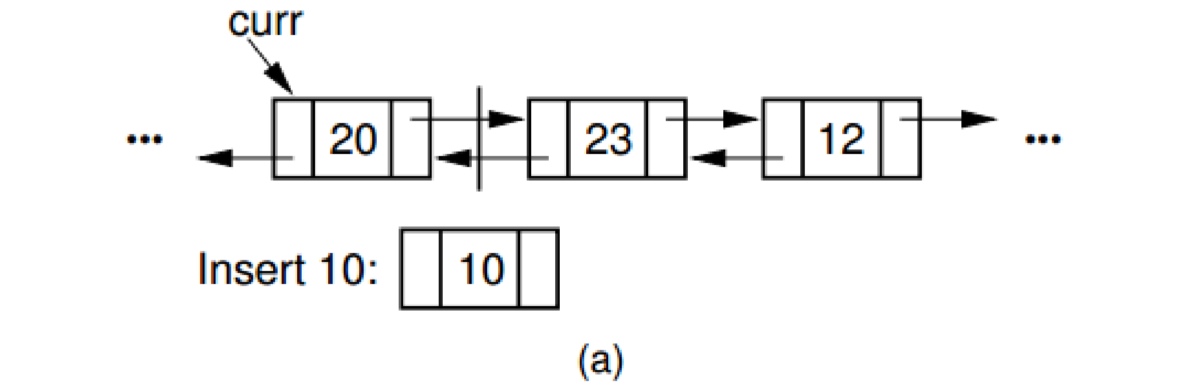
Doubly Linked List

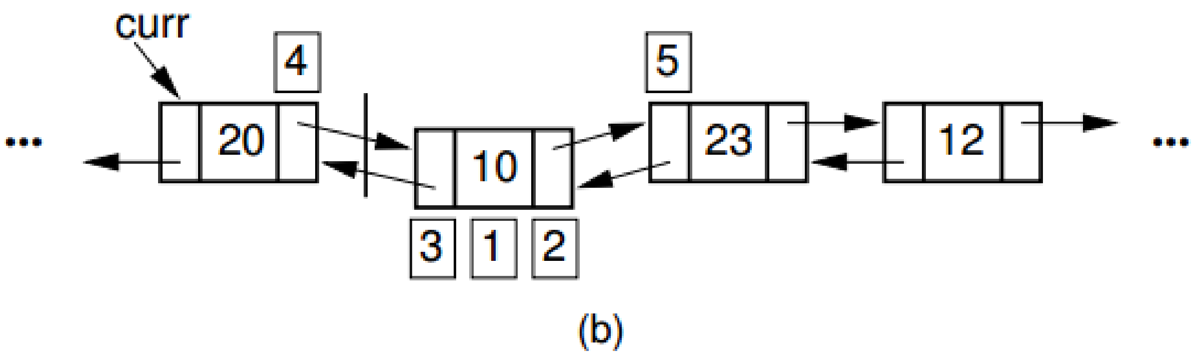
- The singly linked list presented above allows for direct access from a list node only to the next node in the list
- A doubly linked list allows convenient access from a list node to the next node and also to the preceding node on the list
- The doubly linked list node stores two pointers: one to the node following it, and a second pointer to the node preceding it



Doubly Linked List (cont.)

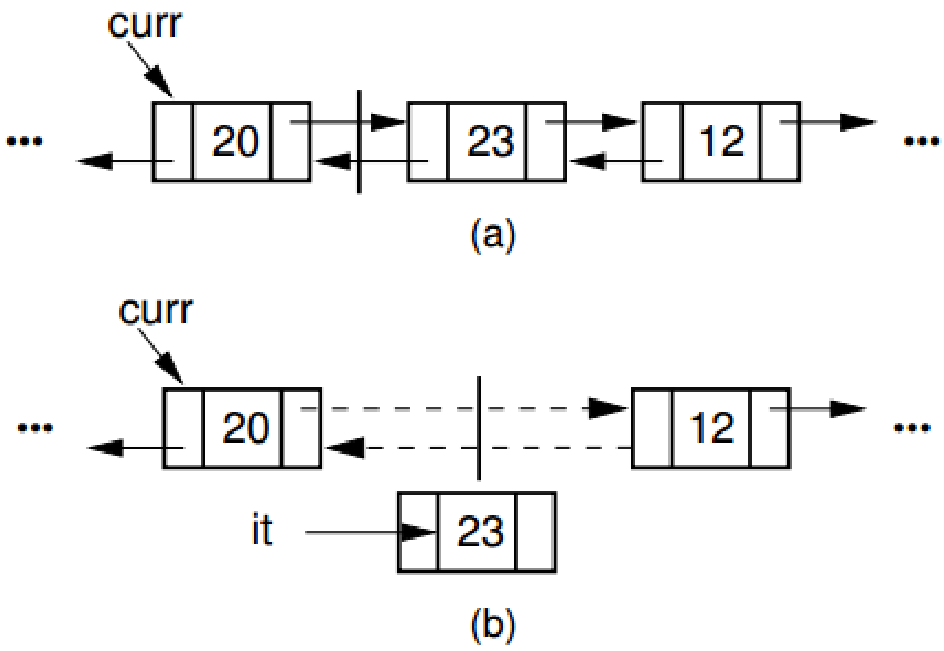
- Insertion of a doubly linked list





Doubly Linked List (cont.)

- Doubly linked list removal



Dictionaries

- The most common objective of computer programs is to store and retrieve data
- It is important to find efficient ways to organize collections of data records so that they can be quickly
  - Stored
  - Retrieved
- Solution: a simple interface for such a collection, called a **dictionary**

Definition

- A dictionary is a collection of elements each of which has a unique search **key**
- The dictionary ADT provides operations for
  - storing records
  - finding records
  - removing records from the collection
- The dictionary ADT gives us a standard basis for comparing various data structures (see `Dictionary.java`)

A Key and Comparable Objects

- If we want to search for a given record in a database, how should we describe what we are looking for?
- A database record could simply be
  - a single data
  - quite complicated with many fields of varying types

    ⇒ We need to define the record in terms of a **key** value
- To implement a search function, we require that the keys be comparable
  - Equal
  - Less/Greater than (order)

Implementation

- Dictionary ADT can be implemented using
  - Unsorted List (see `UALdictionary.java`)
  - Sorted List

References

- Clifford A. Shaffer. (2012). *Data Structures and Algorithm Analysis.* Department of Computer Science. Virginia Tech.
- Douglas Wilhelm Harder. (2013). Queues. University of Waterloo, Waterloo, Ontario.
- Kurt Mehlhorn and Peter Sanders. (2007). *Algorithms and Data Structures.* Springer.
- Naveen Garg. (2008). *Data Structures and Algorithms*. Department of Computer Science and Engineering. Indian Institute of Technology, Delhi.
- Robert Lafore. (2003). *Data Structures & Algorithms in Java*. Sams Publishing.