Dynamic Data Structures

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

Dynamic Data Structures

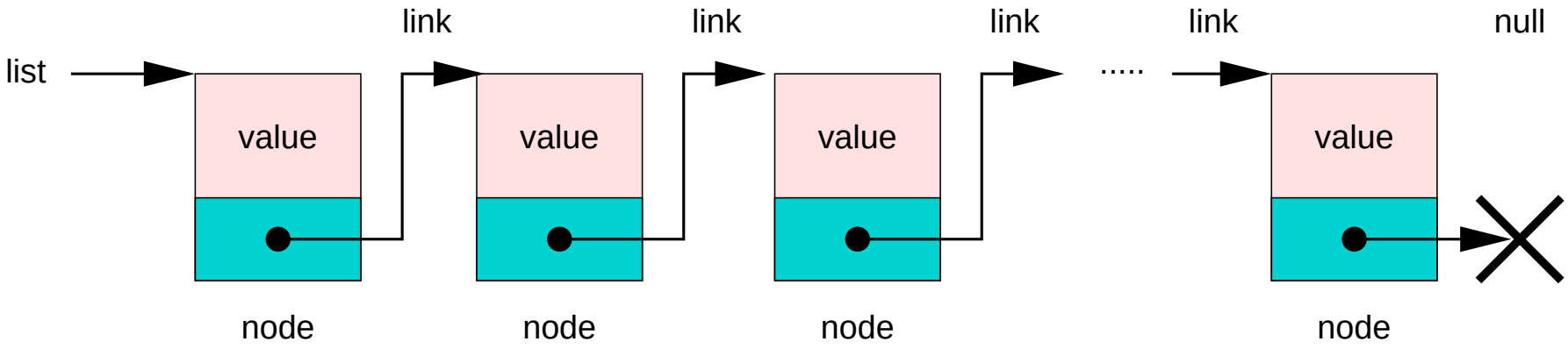
- Arrays have certain disadvantages as a data structure.
  - In an unordered array, searching is slow.
  - In an ordered array, inserting a new element is slow.
    - All elements after the inserted element have to be shifted one position to the right.
  - In an ordered array, deleting an element is slow.
    - All elements after the deleted element have to be shifted one position to the left.
  - The size of an array cannot be changed after creation.
- Linked lists solve some of these problems.
  - New elements can be easily inserted/deleted.
  - There is no a priori bound on number of elements that can be stored.
  - Linked lists are not used for fast searching.

Linked lists are another widely used data structure.

List Nodes

A linked list consists of nodes.

- A list node holds
  - a value of some type,
  - a link, i.e., a reference to another node of the list.



In Java, each list node is represented by an object.

Class Declaration

```
public class Node
{
  public int value; // value carried by node (any type possible)
  public Node next; // link to next node in list

  public Node(int value)
  {
    this.value = value;
  }

  public Node(int value, Node next)
  {
    this(value);
    this.next = next;
  }
}
```

Linking Nodes

- List $[1]$ (single node holding the value 1).

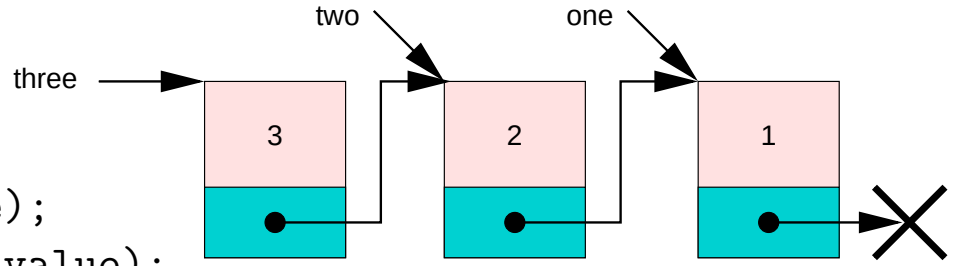
```
Node one = new Node(1);
System.out.println(one.value);
```

- List $[2, 1]$

```
Node two = new Node(2, one);
System.out.println(two.value);
System.out.println(two.next.value);
```

- List $[3, 2, 1]$

```
Node three = new Node(3, two);
System.out.println(three.value);
System.out.println(three.next.value);
System.out.println(three.next.next.value);
```



Updating Nodes

Different lists may have overlapping nodes.

- First node of two is second node of three.

```
two.value = -2;

System.out.println(three.next.value);

-2.
```

A list node may be referenced by multiple variables.

List Type

A list is an abstraction that is implemented by a sequence of nodes.

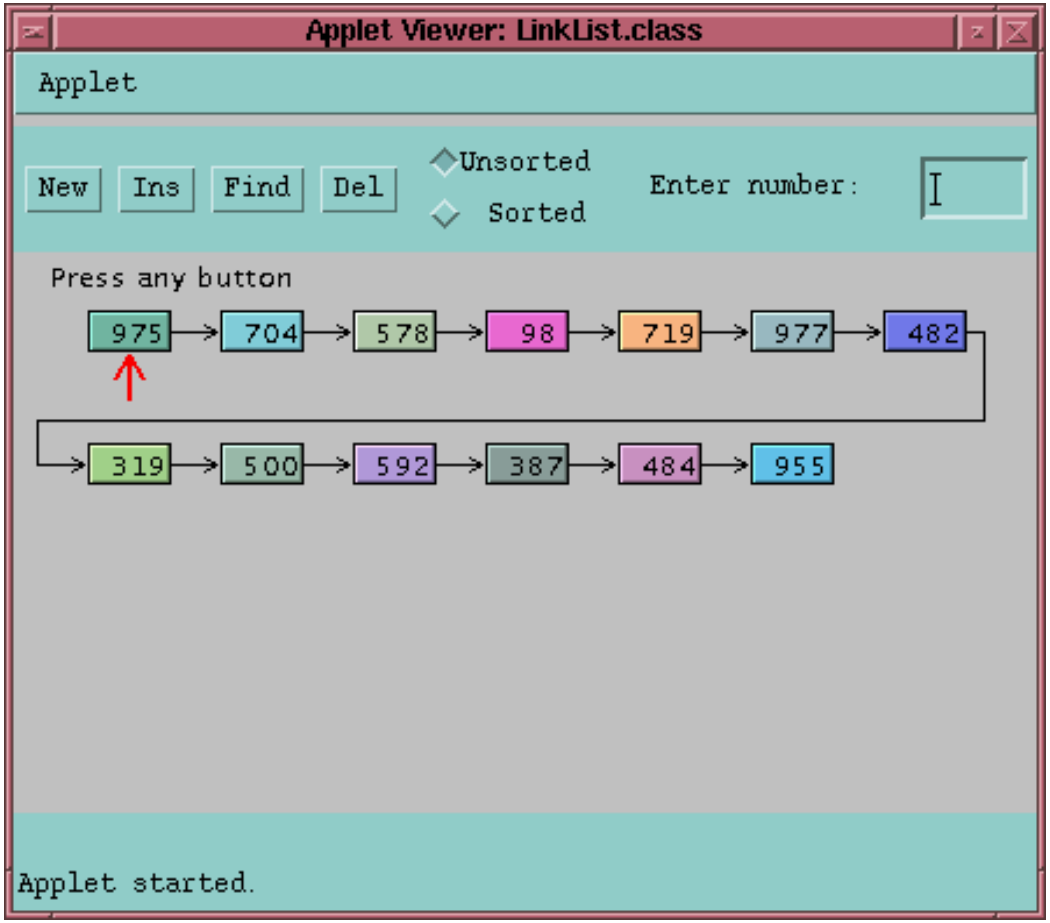
```
public class List
{
  public Node head; // reference to first node in list
  public List()
  {
  }
  ...
}
```

- Empty list:

```
List empty = new List()
```

We will now introduce various list operations.

Java Applet


Applet Viewer: LinkList.class
Applet
New
Ins
Find
Del
Unsorted
Sorted
Enter number:
Press any button
975
704
578
98
719
977
482
319
500
592
387
484
955
Applet started.

Inserting List Elements

Insert an element at the head of a list.

```
public class List
{
  ...
  public void insert(int value)
  {
    Node node = new Node(value, head);
    head = node;
  }
}

List l = new List();
l.insert(1);
l.insert(2);
l.insert(3);
```

Inserting List Elements

It is more convenient to have a function that returns the new list.

```
// ------------------------------------------------------
// list = insert(value)
// 'list' is the result of inserting 'value' at the head of
// 'this' list.
// ------------------------------------------------------
public List insert(int value)
{
  Node node = new Node(value, head);
  head = node;
  return this;
}

List l = new List();
l.insert(1).insert(2).insert(3);
```

Searching for List Elements

```
// ------------------------------------------------------
// node = search(value)
// 'node' is the result of searching 'value' in 'this' list
//
// Output:
//   all nodes referenced via 'this' remain unchanged.
//   if 'node == null', then 'value' does not occur in 'this'
//   if 'node != null', then 'node.value == value' and 'node'
//     is the first node in 'this' for which this holds.
// ------------------------------------------------------
public Node search(int value)
```

Searching for List Elements

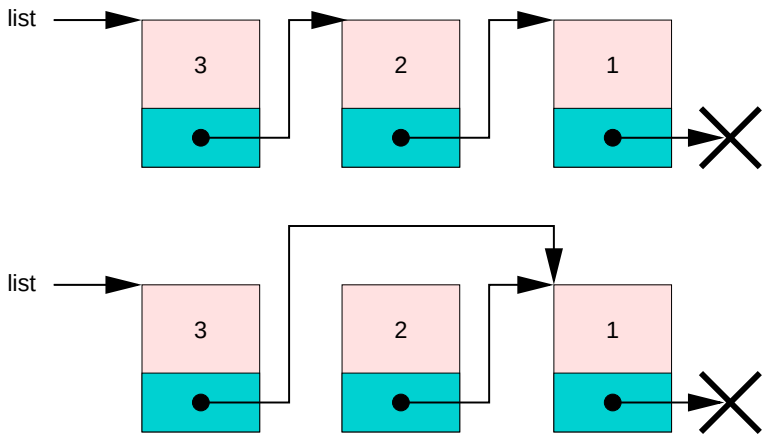
```
public Node search(int value)
{
  Node node = head;
  while (node != null && node.value != value)
    node = node.next;
  return node;
}

public Node search(int value)
{
  for (Node node = head; node != null; node = node.next)
    if (node.value == value) return node;
  return null;
}
```

Similar to searching in an unsorted array.

Deleting List Elements

Delete from list first node that holds a particular key value.

- Find appropriate list node and update any reference to this node:
  - If the deleted node was the first node, $head$ has to be updated.
  - If the deleted node was not the first node, the link of the predecessor node has to be updated.
  - Updated reference points to the successor of the deleted node.
- Example: delete value 2 from list $[3, 2, 1]$.



Specification of Method

```
// ------------------------------------------------------
// list = delete(value)
// 'list' is the result of deleting the first node that
/  holds 'value' from 'this' list.
//
// Output
//   'list == this'.
//   if 'value' was not in the old 'this', 'this' equals
//     the old 'this'.
//   if 'value' was in the old 'this', 'this' equals the
//     old 'this' except that the first node with
//     'node.value == value' is unlinked.
// ------------------------------------------------------
List delete(int value);
```

Implementation of Method

```
public List delete(int value)
{
  Node prev = null; // previous node
  Node node = head; // current node
  while (node != null && node.value != value)
  { prev = node;
    node = node.next;
  }
  if (node != null)
  { if (prev == null)
      head = node.next;
    else
      prev.next = node.next;
  }
  return this;
}
```

Usage of Method

Method `delete` has been implemented as a function.

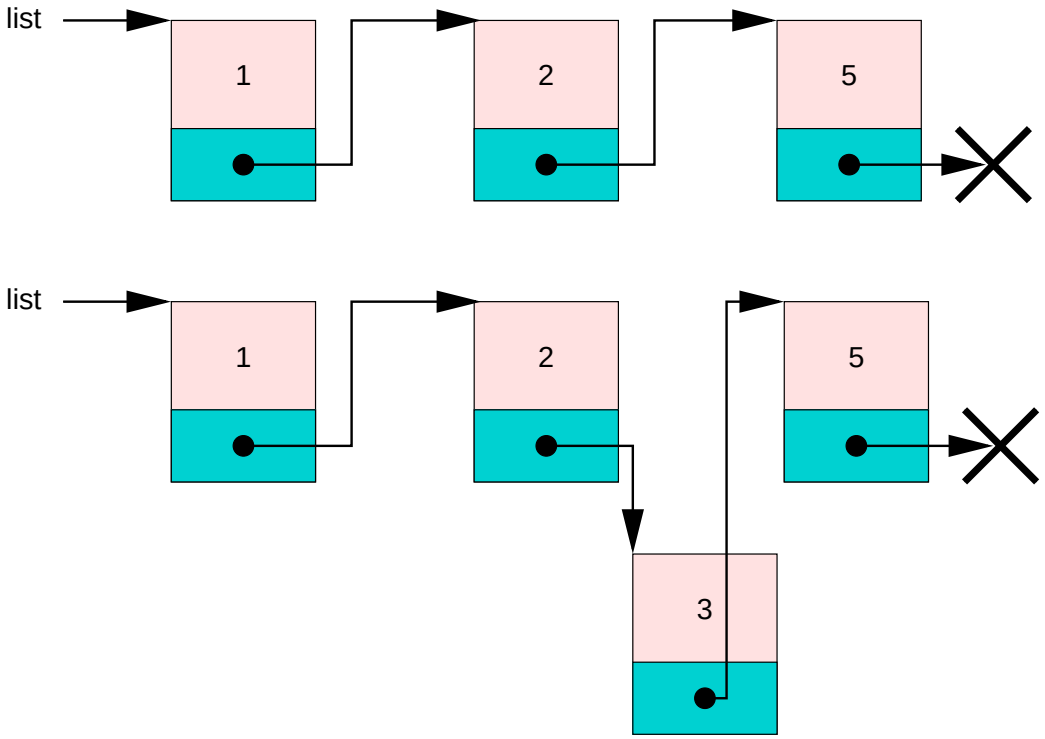
```
List l = new List();
l.insert(1).insert(2).insert(1).delete(1).delete(2).delete(2);
```

Convenient form of list manipulation.

Keeping a List Sorted

Insert new elements such that key values are ordered.

- Example: insert value 3 in list $[1, 2, 5]$ giving list $[1, 2, 3, 5]$.



Specification of Method

```
// ------------------------------------------------------
// list = insertSorted(value)
// 'list' is the result of inserting 'value' into
// the sorted 'this' list.
//
// Input
//   'this' is sorted.
// Output
//   'list == this' and 'this' is sorted.
//   'this' is the same as the old 'this' except that
//     a new node with 'value' has been inserted.
// ------------------------------------------------------
```

Implementation of Method

```
public List insertSorted(int value)
{
  Node prev = null; // previous node
  Node node = head; // current node
  while (node != null && node.value < value)
  {
    prev = node;
    node = node.next;
  }
  Node n = new Node(value, node);
  if (prev == null)
    head = n;
  else
    prev.next = n;
  return this;
}
```

Searching in Sorted Lists

```
// ------------------------------------------------------
// node = search(value)
// 'node' is the result of searching 'value' in 'this' list
//
// Input: 'this' list is sorted in ascending order.
// Output: ... (as before)
// ------------------------------------------------------
public Node search(int value)
{
  for (Node node = head; node != null; node = node.next)
  {
    if (node.value == value) return node;
    if (node.value > value) return null;
  }
  return null;
}
```