

# (Linked) Lists

**Last updated:** November 6<sup>th</sup> 2018, at 2.03pm

## Contents

<b>1</b>	<b>Structured Agglomerations</b>	<b>2</b>
1.1	Building Structured Agglomerations . . . . .	2
1.2	Binary Trees . . . . .	2
1.3	Rows (Lists, Arrays) . . . . .	3
1.4	Bags (Multisets, Unordered Lists) . . . . .	3
1.5	Sets . . . . .	3
<b>2</b>	<b>Rows</b>	<b>3</b>
2.1	Properties . . . . .	3
2.2	Properties of Primitive Row Types . . . . .	4
<b>3</b>	<b>Linked Lists</b>	<b>4</b>
3.1	Linked List Data Structures in Java . . . . .	4
3.2	An Aside: Atomic Types and Objects — Values and References .	5
3.3	References and <i>Call by Value</i> . . . . .	6
3.3.1	Call by Value . . . . .	6
3.3.2	References . . . . .	7
3.4	List Nodes: Code . . . . .	7
3.4.1	Interface . . . . .	7
3.4.2	Single Link Nodes . . . . .	8
3.4.3	Double Link Nodes . . . . .	9
<b>4</b>	<b>Linked Lists</b>	<b>11</b>
4.1	Interface . . . . .	11
4.2	Basic Implementation . . . . .	11
<b>5</b>	<b>Stacks</b>	<b>11</b>
5.1	Functionality . . . . .	11
5.2	Code . . . . .	12
5.2.1	Interface . . . . .	12
5.2.2	Implementation . . . . .	13

(Linked) Lists

<b>6</b>	<b>Queues</b>	<b>14</b>
6.1	Functionality . . . . .	14
6.2	Interface . . . . .	15
6.3	Implementation . . . . .	15
<b>7</b>	<b>Lists</b>	<b>16</b>
7.1	Functionality . . . . .	16
7.2	Code . . . . .	17
7.2.1	Interface . . . . .	17
7.2.2	Implementation . . . . .	17
<b>8</b>	<b>ArrayLists</b>	<b>17</b>

## 1 Structured Agglomerations

Depending on the properties of the  $\uplus$  operator, an agglomeration will be an instance of better known data types.

### 1.1 Building Structured Agglomerations

A data type containing a flexible number of objects can be called an “agglomeration”<sup>1</sup>. Here, “flexible” is used to mean that two agglomerations of the same type do not need to contain the same number of elements — in contrast to *records*, where the number of data fields *is* determined by the record type. Agglomerations can be constructed from:

- $\perp$  — the empty agglomeration
- $()$  — the *singleton* function  
If  $x$  is an object, then  $(x)$  is the agglomeration containing (only)  $x$ .
- $\uplus$  — the *join* operator  
If  $c_1$  and  $c_2$  are agglomerations then  $c_1 \uplus c_2$  is the agglomeration obtained by “joining”  $c_1$  and  $c_2$ .

### 1.2 Binary Trees

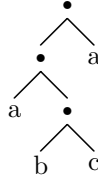
If  $\uplus$  has no additional properties, then this operator defines the (binary) tree type.

For example  $(a \uplus (b \uplus c)) \uplus a$  is the tree

---

<sup>1</sup>“Agglomeration” is *not* a technical term. I am simply using it since most synonyms (list, collection, set, . . .) have specific meanings.

## (Linked) Lists



Note: the expression should, formally, have been written as  $((a) \uplus ((b) \uplus (c)) \uplus (a))$ , using the singleton function.

### 1.3 Rows (Lists, Arrays)

If  $\uplus$  is associative (i.e.  $(a \uplus b) \uplus c = a \uplus (b \uplus c)$ ), so that both can be written  $a \uplus b \uplus c$ ) then agglomerations are rows (usually called “lists”, but we will be using “list” with a specific meaning in this lecture, so we will call this generic type of list a “row”).

The expression  $(a \uplus (b \uplus c)) \uplus a$  can be written as  $a \uplus b \uplus c \uplus a$ , and is the row  $[a, b, c, a]$ .

### 1.4 Bags (Multisets, Unordered Lists)

Bags (also known as multisets) are lists of values in which the order of the elements is not important. If  $\uplus$  is commutative (i.e.  $a \uplus b = b \uplus a$ ) and associative then an agglomeration is a bag.

The expression  $(a \uplus (b \uplus c)) \uplus a$  can be written as  $a \uplus a \uplus b \uplus c$ , and is the bag  $\{a, a, b, c\}$ . Note that the notation  $\{a, a, b, c\}$  is ad hoc — there is no general standard notation of multisets.

### 1.5 Sets

If  $\uplus$  is idempotent (i.e.  $a \uplus a = a$ ), commutative and associative then an agglomeration is a set.

In this case  $(a \uplus (b \uplus c)) \uplus a$  is the same as  $a \uplus b \uplus c$ , and is the set  $\{a, b, c\}$ .

This classification of the tree, list, bag, set data types is known as the *Boom Hierarchy*, named after Hendrik Boom. This idea was credited to him by Lambert Meertens, in 1986. His name is very appropriate, since “boom” (pronounced “bome”, with a long ‘o’) is the Dutch for tree.

## 2 Rows

### 2.1 Properties

In this lecture we will be looking at data structures for implementing *rows*. These implementations can have different properties.

### (Linked) Lists

- Homogenous or heterogenous  
Must all elements of the row be of the same type, or can they be of different types?
- Static or dynamic  
Can the number of elements in the row change with time, or is it fixed (once the size of the list is initially specified)?
- If static, compile time or run time  
Must the size of the row be known at compile time, or can it be determined at run time?
- If dynamic, finite or infinite  
Are potentially infinite rows permitted or not?

## 2.2 Properties of Primitive Row Types

	Heterogenous	Static	Infinite
C	×	Compile time	×
Java	×	Run time	×
Gofer	×	Dynamic	✓

Note that very few languages offer heterogenous rows — because it makes type checking very difficult. It could be argued that Java does — you can, after all, declare an array of `Objects`. Also, Java *does* offer dynamic rows in the form of `java.util.List` implementations, such as `ArrayLists`. Here, however we are considering *primitive* objects and, in Java, the primitive row type is an array, which is static.

Generally, rows can be implemented in two fundamentally different ways:

**Arrays** Implemented as a (logically) contiguous block of memory. Access is by memory address calculation. Almost always static.

**Lists** Implemented as *linked lists*. Access by following pointers. Dynamic.

## 3 Linked Lists

### 3.1 Linked List Data Structures in Java

Linked Lists refer to *linear* data structures in which information is stored as a list made up of *nodes*.

Each *node* contains some data and a reference to the next node in the list.

A reference must be maintained to the *top* or *head* of the list.

This reference *points to* the first node, which in turn points to the second, which points to the third, etc., right to the end of the list.

Linked lists can be processed *only* in a *linear* manner

### (Linked) Lists

i.e. starting at the top of the list and working through.

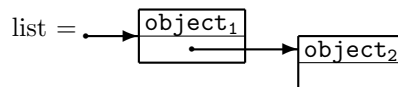
It is *not possible* to *jump into* a list part way through or to go straight to the end of the list.

Note also that if we should lose the reference to the top of the list we effectively lose the entire list.

Similarly, if we lose one of the *next* references part way through the list, we effectively break the chain and lose all data after that point.

Dynamic structures are used when the amount of data to be stored is variable or is not known in advance of writing the program.

It is possible to assemble linked lists according to some ordering mechanism (e.g. a prioritised print queue or a linked list of names sorted alphabetically) but it is more common to use tree structures when data needs to be sorted.



## 3.2 An Aside: Atomic Types and Objects — Values and References

Let's look at what happens in Java when you declare variables and objects.

For Example:

```
int x;  
int y = 32;  
Person p1;  
Person p2 = new Person("Fred", "Bloggs");
```

Ordinary (*atomic*) variables are created by setting aside an area of memory to contain a value.

Any attempt to access a variable, to get or set its value, accesses this memory directly.

Objects work differently.

**Object** (*non-atomic*) variables are created by setting aside an area of memory that contains a *reference* to *another area of memory* which will hold an *instance* of the class.

The instance *and* the reference are created by the **new** keyword.

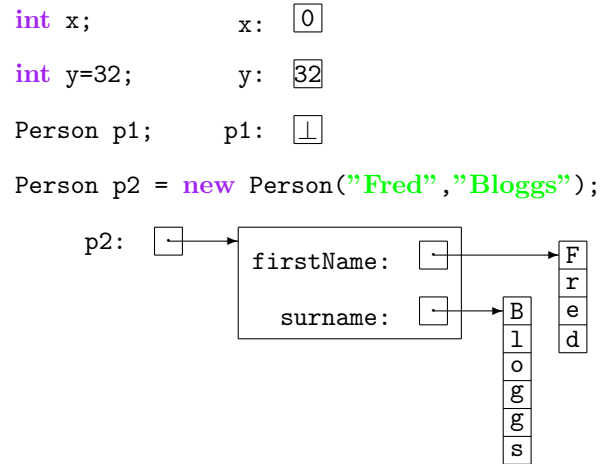
For example, the declaration of **p2** above sets up an area of memory which is a **reference** to another area of memory which will hold an instance of the person class.

The declaration of **p1** allocates an area of memory for one of these references but that reference does not yet *point to* anything because the actual instance of the object has not yet been created (i.e. no **new** command yet).

The reference can be completed by:

(Linked) Lists

```
p1 = new Person("Jane", "Doe");
```



### 3.3 References and *Call by Value*

#### 3.3.1 Call by Value

What is the output of this code?

```
public class CallByValue {
    public static void main(String[] args) {
        int x=5, y=15;
        callByValue(x, y);
        System.out.println("x = " + x + ", y = " + y);
    } // end of method main

    public void callByValue(int x, int y) {
        x = 10; y = 100;
    } // end of method callByValue
} // end of class CallByValue
```

The output is:

`x = 5, y = 15`

Atomic variables are *passed by value*. This means that any changes made to variables inside a called method have *no effect* on variables outside that method.

(Linked) Lists

### 3.3.2 References

Similarly, if we pass a reference to an object, such as `p1` from above, as a parameter to a method then the called method *cannot* change the *reference*

i.e. the reference will still point to the same thing

but it *can* change values *inside* the object which the reference points to.

For example: What is the output of this code?

```
public class CallByName {
    public static void main(String[] args) {
        Person p1 = new Person("Joe", "Smith");
        callByName(p1);
        System.out.println("Person is " + p1.getName()
            + " " + p1.getSurname());
    } // end of method main

    public void callByName(Person person1) {
        person1.setName("Fred");
        person1.setSurname("Bloggs");
        person1 = null;
    } // end of method callByName
} // end of class CallByName
```

Non-atomic variables are *passed by reference*. The two lines inside `AMethod` which set values in the object `person1` will have the effect of setting attributes in the object `p1`.

The line which sets `person1` to `null` will not have any effect on the reference `p1` which will still be a pointer to an object, now set with the name *Fred Bloggs*, so the output is:

Person is Fred Bloggs

## 3.4 List Nodes: Code

### 3.4.1 Interface

```
public interface ListNode<T> {
    public T getValue();

    public ListNode<T> getNext();
}
```

This specifies a basic interface for nodes that contain values, but does not specify how nodes should be connected, other than that all nodes must have a next node (which may be `null`).

*(Linked) Lists*

### 3.4.2 Single Link Nodes

An implementation where the nodes have a single link, linking each node to the next one in the list.

```
public class SingleLinkNode<T> implements ListNode<T> {
    private T value; // the value held in this node
    private SingleLinkNode<T> next; // pointer to the next node

    ...// for methods see below
}
```

#### Constructors

##### Tail nodes

```
/**
 * Constructor for a “tail” node where there is no following node
 */

public SingleLinkNode(T value) {
    this.value = value;
    next = null;
}
```

##### Non-tail nodes

```
/**
 * Constructor for a “non-tail” node with nodes following
 */

public SingleLinkNode(T value, SingleLinkNode<T> next) {
    this.value = value;
    this.next = next;
}
```

#### Access methods



*(Linked) Lists*

```
    next

    /**
     * @return the next node in the list (and therefore the rest of the list)
     */

    public SingleLinkNode<T> getNext() {
        return next;
    }

    public void setNext(SingleLinkNode<T> next) {
        this.next = next;
    }

    value

    /**
     * @return the value held in this node
     */

    public T getValue() {
        return value;
    }
}
```

### 3.4.3 Double Link Nodes

Double link nodes extend the functionality of single link nodes by adding a second pointer to the node, which links to the previous node in the list.

```
public class DoubleLinkNode<T> extends SingleLinkNode<T> {

    private DoubleLinkNode<T> previous; // pointer to previous node

    ...methods follow
}
```

### Constructors

### *(Linked) Lists*

**Singleton nodes** Singleton nodes occur in list with exactly one entry, so that the single node has no predecessor and no successor.

```
public DoubleLinkNode(T value) {  
    super(value);  
    previous = null;  
}
```

**Head nodes** Head nodes are the initial nodes in lists with more than one entry, so the head node has a successor, but no predecessor.

```
public DoubleLinkNode(T value,  
    DoubleLinkNode<T> next) {  
    super(value,next);  
    previous = null;  
}
```

**Tail nodes** Tail nodes are the final nodes in lists with more than one entry, so the tail node has a predecessor, but no successor.

```
public DoubleLinkNode(DoubleLinkNode<T> previous,  
    T value) {  
    super(value); // next will be null  
    this.previous = previous;  
}
```

**Internal nodes** All other nodes will have both a predecessor and a successor.

```
public DoubleLinkNode(DoubleLinkNode<T> previous,  
    T value,  
    DoubleLinkNode<T> next) {  
    super(value,next);  
    this.previous = previous;  
}
```

### **Access methods**

**previous** We also need methods to set and get the previous pointer.

```
public DoubleLinkNode<T> getPrevious() {  
    return previous;  
}
```

*(Linked) Lists*

```
public void setPrevious(DoubleLinkNode<T> previous) {  
    this.previous = previous;  
}
```

## 4 Linked Lists

### 4.1 Interface

```
public interface LinkedList<N extends ListNode<T>,T> {  
    public boolean isEmpty();  
}
```

- Parameterised with value type T, and node type N.
- Must implement `isEmpty`.

### 4.2 Basic Implementation

```
public abstract class BasicList<N extends ListNode<T>,T>  
    implements LinkedList<N,T> {  
    N root; // the root node  
  
    ...  
}
```

plus:

- implementation of `isEmpty`
- methods to get and set the root
- method to “stringify” the list

## 5 Stacks

### 5.1 Functionality

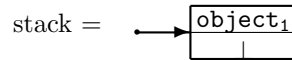
A stack is a simple linked list data structure in which data may only be added and removed from the front of the list — i.e. it has the following functionality. The methods for adding and removing values are usually called “push” and “pop”.

## (Linked) Lists

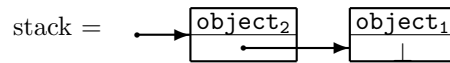
```
Stack<T> stack = new Stack<T>();
```

```
stack =  $\perp$ 
```

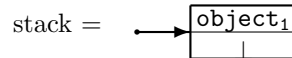
```
stack.push(object1);
```



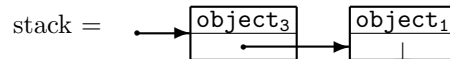
```
stack.push(object2);
```



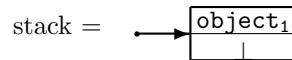
```
T value = stack.pop();
```



```
stack.push(object3);
```



```
value = stack.pop();
```



```
value = stack.pop();
```

```
stack =  $\perp$ 
```

```
value = stack.pop(); // again
```

**ERROR**

## 5.2 Code

### 5.2.1 Interface

The **Stack** class will be implemented using single link nodes. The root of each **Stack** object will be a reference to the top of the stack (which will be a **SingleLinkNode**) and the stack will have methods for adding new items (nodes) into the stack and removing the top item from the stack. In stacks adding a new node is usually called “pushing” a value, and removing a node is called “popping” a value.

*(Linked) Lists*

```
public interface Stack<T>
    extends LinkedList<SingleLinkNode<T>,T> {

    // Push a value to the top of the stack
    public void push(T value);

    // Pop and return a value from the top of the stack
    public T pop() throws ListAccessError;

}
```

### 5.2.2 Implementation

```
public class SingleLinkStack<T>
    extends BasicList<SingleLinkNode<T>,T>
    implements Stack<T> {

    // root inherited from BasicList

    // for code see below

    /**
     * Push a new element to the stack.
     * The root should now point to the new entry.
     * The pointer from the new root should point to the old root.
     */

    public void push(T value) {
        setRoot(value, getRoot());
    }

    /**
     * Pop a value from the stack.
     * @return the top value on the stack.
     * @throws ListAccessError if the stack is empty (there is nothing to pop).
     */

    public T pop() throws ListAccessError {
```

*(Linked) Lists*

```
    if (isEmpty()) throw new ListAccessError("Pop from an empty stack");
    T value = getRoot().getValue(); // get the top value
    setRoot(getRoot().getNext()); // update top to point to next node
    return value; // return the original top value
}
```

## 6 Queues

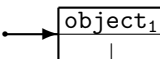
### 6.1 Functionality

Queues are FIFO (first in, first out) data structures. The methods for adding and removing values are usually called `enqueue` and `dequeue`.

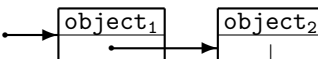
```
Queue<T> queue = new Queue<T>();
```

```
queue =  $\perp$ 
```

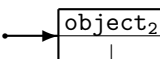
```
queue.enqueue(object1);
```

```
queue = 
```

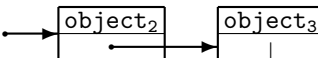
```
queue.enqueue(object2);
```

```
queue = 
```

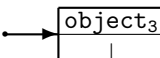
```
T value = queue.dequeue();
```

```
queue = 
```

```
queue.enqueue(object3);
```

```
queue = 
```

```
value = queue.dequeue();
```

```
queue = 
```

```
value = queue.dequeue();
```

```
queue =  $\perp$ 
```

(Linked) Lists

```
value = queue.dequeue(); // again
```

```
queue =  
ERROR
```

## 6.2 Interface

```
public interface Queue<T>  
    extends LinkedList<DoubleLinkNode<T>,T> {  
  
    // Add a value to the queue  
    public void enqueue(T value);  
  
    // Remove a value from the queue  
    public T dequeue() throws ListAccessError;  
  
}
```

## 6.3 Implementation

We will implement Queues using double linked nodes.

```
public class DoubleLinkQueue<T>  
    extends BasicList<DoubleLinkNode<T>,T>  
    implements Queue<T> {  
  
    // pointer to last entry in the queue  
    // first entry inherited from BasicList (root)  
    DoubleLinkNode<T> tail;  
  
    ...// see below for methods  
}  
  
/**  
 * Add a new value to the queue  
 */  
  
    public void enqueue(T value) {  
        if (isEmpty()) { // if the queue is empty  
            setRoot(new DoubleLinkNode<T>(value)); // create a new node, which will be the head (root) o  
            tail = getRoot(); // and which will also be the tail  
        } else {  
            DoubleLinkNode<T> oldHead = getRoot(); // make a note of the current head node
```

(Linked) Lists

```
        setRoot(new DoubleLinkNode<T>(value,oldHead)); // create a new head, holding the required value
        oldHead.setPrevious(getRoot()); // set previous pointer of the old head to point to the new head
    }
}

/**
 * Remove the oldest value from the queue
 */

public T dequeue() throws ListAccessError {
    if (isEmpty()) {
        throw new ListAccessError("Dequeue from an empty queue");
    }
    T result = tail.getValue(); // get the last value in the queue
    tail = tail.getPrevious(); // move tail to previous node
    if (tail != null) {
        tail.setNext(null); // and ensure that pointer to "removed" node is set to null
    } else {
        setRoot(null); // if tail is now null the queue is empty, so set head (root) to null as well
    }
    return result;
}
```

## 7 Lists

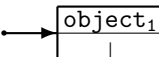
### 7.1 Functionality

In lists we want to be able to add, remove, and access values at any index in the list.

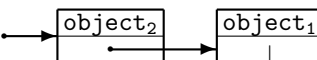
```
List<T> list = new List<T>();
```

list =  $\perp$

```
list.add(0,object1);
```

list = 

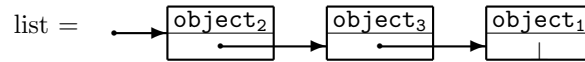
```
list.add(0,object2);
```

list = 



## (Linked) Lists

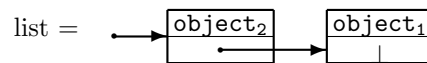
```
list.add(1,object3);
```



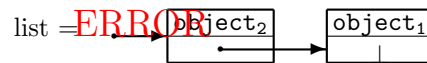
```
list.add(5,object4);
```



```
list.remove(1);
```



```
list.remove(6);
```



## 7.2 Code

### 7.2.1 Interface

```
public interface List<T>
    extends LinkedList<SingleLinkNode<T>,T> {

    // Add a new value to the list at position index
    public void add(int index,T value) throws ListAccessError;

    // Remove a value from the list
    public T remove(int index) throws ListAccessError;

    // Get a value from the list (do not remove it)
    public T get(int index) throws ListAccessError;
}
```

### 7.2.2 Implementation

Left as an exercise.

## 8 ArrayLists

Java's `ArrayLists` provide the functionality of lists — dynamic allocation, etc. — but provide an efficient internal implementation as arrays.

*(Linked) Lists*

- Implemented as an internal array
- `add(T value)` adds new value to first empty slot (at the end of the list)
- `add(int index, T value)` moves all entries above `index` up one, and then inserts the value
- With each insertion, if the internal array is not big enough a new, larger, array is created and the elements are copied across