

Introduction, ADT, API, Bags, Lists, Stacks

Comp Sci 2C03

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton,
Ontario, Canada

Acknowledgments: Material mainly based on *Algorithms* by Robert Sedgewick and Kevin Wayne (Chapters 1.2-1.3)

Instructor: Dr. Ryszard Janicki, ITB 217, e-mail: janicki@mcmaster.ca, tel: 9(05) 525-9140 ext: 23919

Teaching Assistants:

- Morteza Alipour Langouri: alipoum@mcmaster.ca (**Main TA**)
- Duncan McKay: mckayd5@mcmaster.ca
- Yuxing Fang: fangy32@mcmaster.ca

Course website: <http://www.cas.mcmaster.ca/~cs2c03>,

Lectures: Monday: 10:30-11:20, Wednesday: 10:30-12:20, in T13/127

Tutorial: Friday: 1:30-3:20, in PGCLL M16 (both T01 nad T02), starts Jan 10, 2020

Office hours: Tuesday 12:00-13:00.

Midterm: To be announced

Calendar Description:

The mission of this course is to give students an understanding of the foundations of data structures and algorithms, to teach them basic implementation techniques and to show how they can be applied to solve practical problems.

Mission:

The mission of this course is to give students an understanding of the foundations of data structures and algorithms, to teach them basic implementation techniques and to show how they can be applied to solve practical problems.

Preconditions

Students should have basic knowledge of discrete mathematics and basic programming skills (especially in Java).

It is assumed that the students know Java sufficiently to understand all simple codes used in the textbook.

Postconditions:

Students should know and understand:

- 1 Worst case analysis of algorithms
- 2 Basic searching algorithms (elementary sorts, quicksort, mergesort, heapsort)
- 3 Basic sorting algorithms (binary search, search trees, hashing)
- 4 Elementary data structures (stacks, queues, priority queues, search trees, heaps, hash tables, tries, graph representations)
- 5 Graph algorithms (topological sort, breadth/depth-first-search, strongly connected components, minimum spanning trees, shortest paths)
- 6 Basic string algorithms
- 7 Finite State Automata and Regular Expressions

Postconditions:

Students should be able to:

- 1 Analyze the running time of algorithms
- 2 Identify the time/space trade-offs in designing data structures and algorithms
- 3 Given a problem such as searching, sorting, graph and string problems, select from a range of possible algorithms, provide justification for that selection
- 4 Understand implementation issues for the algorithms studied
- 5 Reduce a given application to (or decompose it into) problems already studied

Outline of Course Topics and Texts

Text: R. Sedgewick, K. Wayne, *Algorithms*, 4th Edition, Addison-Wesley 2011

Course Outline (Tentative, textbook chapters included, one topic per week or more)

- 1 Introduction (1.2, 1.3)
- 2 Analysis of algorithms - Union-Find (1.4, 1.5)
- 3 Elementary sorts - Mergesort - Quicksort (2.1, 2.2, 2.3)
- 4 Priority queues - Sorting applications (2.4, 2.5)
- 5 Symbol tables - Binary search trees (3.1, 3.2)
- 6 Balanced search trees - Hash tables - searching applications (3.3, 3.4, 3.5)
- 7 Graph representation - BFS/DFS - Connected components - Topological sort (4.1, 4.2)
- 8 Minimum Spanning Trees - Shortest paths (4.3, 4.4)
- 9 String sorts - Tries (5.1, 5.2)
- 10 Substring search - FSA's and regular expressions (5.3 except KMP, 5.4)

The course may not always follow the text-book closely.

Other texts that might be useful (each has its own virtues):

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd Ed., McGraw Hill, 2001 – this is an encyclopedia of algorithms, preferred reference book,
- J. Kleinberg, E. Tardos, Algorithm design, Addison-Wesley 2005 – more devoted to algorithms, assume knowledge of basic data structures,
- M. Soltys, An Introduction to the Analysis of Algorithms, 2nd Ed., World Scientific – excellent compact book that concentrates on the analysis of algorithms.

Lecture Notes and Tutorial Notes will be on the website a few days after or before a class.

Lecture Notes designed by the authors of the textbook can be found at <https://algs4.cs.princeton.edu/home/>

- **Evaluation:** There will be a 2.5 hours (one double sided cheat sheet will be allowed) final examination (50%), 60 minutes midterm test (17%, closed book) and three assignments ($3 \times 11 = 33\%$).
- **Detailed grading scheme:** $Grade = 0.50 \times exam + 0.17 \times midterm + 0.11 \times (assg1 + assg2 + assg3)$
- *Late assignments will not be accepted because solutions will be posted on the website a day after the due day.*
- Although you may discuss the general concept of the course material with your classmates, your assignment must be your individual effort.

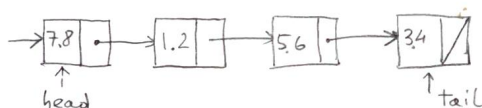
- A *data structure* is a data organization, management, and storage format that enables efficient access and modification.
- More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.
- Data structures serve as the basis for *abstract data types* (ADT).
- The ADT defines the *logical form* of the data type.
- The data structure implements the *physical form* of the data type.

Basic Data Structures: Arrays and Lists

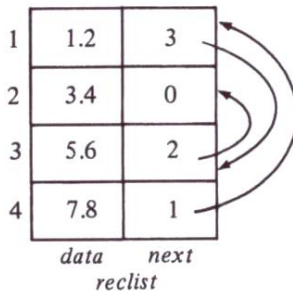
- An **array** is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type).
- Elements are accessed using an integer index to specify which element is required.
- Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity).
- Arrays may be fixed-length or resizable.
- A **linked list** (also just called *list*) is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list.
- The principal advantage of a linked list over an array, is that values can always be efficiently inserted and removed without relocating the rest of the list.
- Certain other operations, such as random access to a certain element, are however slower on lists than on arrays.

Arrays and Lists: An Example

- A list:



- The same list implemented as an array *reclist*:



head = reclist[4][1]

Abstract Data Types I

- An *abstract data type* (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) *from the point of view of a user of the data*, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.
- This contrasts with *data structures*, which are concrete representations of data, and are *the point of view of an implementer, not a user*.
- Formally, an ADT may be defined as a “*class of objects whose logical behavior is defined by a set of values and a set of operations*”.
- This is analogous to an *algebraic structure* in mathematics.

Abstract Data Types II

- An *abstract data type* (ADT) is a mathematical model for data types, where a data type is defined by its behavior (semantics) *from the point of view of a user of the data*, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.
- In practice many common data types are not ADTs, as the abstraction is *not perfect*, and users must be aware of issues like arithmetic overflow that are due to the representation.
- For example, integers are often stored as fixed width values (32-bit or 64-bit binary numbers), and thus experience integer *overflow* if the maximum value is exceeded.

Application Programming Interface (API)

- An *Application Programming Interface (API)* is an interface or communication protocol between different parts of a computer program intended to simplify the implementation and maintenance of software.
- An API may be for a web-based system, operating system, database system, computer hardware, or software library.
- To specify the behaviour of an *Abstract Data Type*, we use an *Application Programming Interface*.
- In the latter case, API is just a list of *constructors* and *instance methods* (operations), with an informal description of each effect of each, as in the API for Counter below:

```
public class Counter
```

Counter(String id)	<i>create a counter named id</i>
void increment()	<i>increment the counter by one</i>
int tally()	<i>number of increments since creation</i>
String toString()	<i>string representation</i>

An API for a counter



ADT vs API: *Counter*

- ADT *Counter* as abstract algebra:

$$\text{Counter} = (\mathbb{N}, \text{succ}, \text{value}, \text{init}),$$

where: \mathbb{N} are natural numbers,

$\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$ and $\text{succ}(x) = x + 1$,

$\text{value} : \mathbb{N} \rightarrow \mathbb{N}$ and $\text{value}(x) = x$,

$\text{init} \in \mathbb{N}$ is the initial value (often 0 by default).

- Its API representation:

```
public class Counter
```

```
    Counter(String id)
```

create a counter named id

```
    void increment()
```

increment the counter by one

```
    int tally()
```

number of increments since creation

```
    String toString()
```

string representation

- ADT vs API. *In this course we will often consider 'ADT \equiv API'.*

ADT	API
\mathbb{N}	int
$\text{succ}(x)$	void increment()
$\text{value}(x) - \text{init}$	int tally()

ADT Bags

A *bag* is a *multiset* (i.e. elements are not unique, two pink marbles below) with an operation of *adding elements*, but removing elements is not supported. In general, no order is assumed.

a bag of
marbles



add(●)



add(●)



for (Marble m : bag)



process each marble m
(in any order)

Operations on a bag

Multisets (Bags, Weighted Sets)

- A multiset m , over a non-empty and finite set S is a function $m : S \rightarrow \mathbb{N} = \{0, 1, 2, \dots\}$
- $m(s)$ is the number of appearances of s in m .
- notation: M is usually represented by:

$$\sum_{s \in S} m(s)s$$

$$S = \{a, b, c, d, e\},$$
$$m(a) = 3, m(b) = 1, m(c) = 0, m(d) = 183, m(e) = 4$$

$$m = 3a + b + 183d + 4e$$

- $s \in m \iff m(s) \neq 0$
- $m(s)$ is a *coefficient*
- the *empty multiset* $m = \emptyset \iff m(s) = 0$ for each $s \in S$.

API Bags

- ADT: A *bag* is a *multiset* with an operation of *adding elements*, but removing elements is not supported. In general, no order is assumed.



- API:
Bag

```
public class Bag<Item> implements Iterable<Item>
    Bag()                                create an empty bag
    void add(Item item)                  add an item
    boolean isEmpty()                    is the bag empty?
    int size()                           number of items in the bag
```

- The above API is sufficient if a bag is a *set*, some additional parameters may be necessary if the bag is a strict *multiset*.

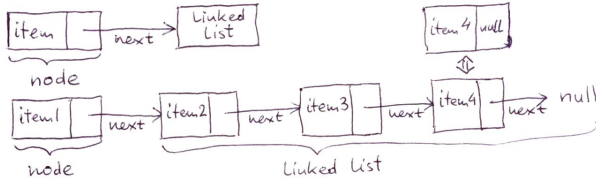
- **Iterable collections** For many applications, we just want to process each of the items in some way, or to *iterate* through the items in the collection.
- For example we maintain a collection of transactions in a Bag, as follows:

```
Bag<Transaction> collection = new Bag<Transaction>();
```
- Since Bag is *iterable*, we can print a transaction list as:

```
for (Transaction transaction : collection)  
    { StdOut.println(transaction); }
```
- *WE do not need to know anything about the representation or the implementation of the collection (which is often nontrivial).*

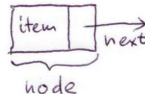
Linked Lists

- A **linked list** is a recursive data structure that is either empty (*null*) or a *node* having a generic *item* and a *reference* (*next* below) to a link list.



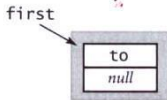
- The **node** is an abstract entity that might hold any kind of data, in addition to the node reference that characterizes its role in building link lists.
- **Node record:**

```
private class Node
{
    Item item
    Node next
}
```

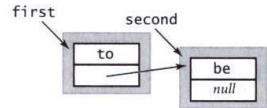


Creating Lists

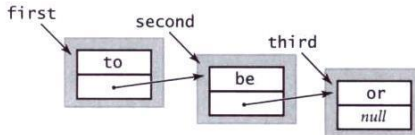
```
Node first = new Node();  
first.item = "to";
```



```
Node second = new Node();  
second.item = "be";  
first.next = second;
```



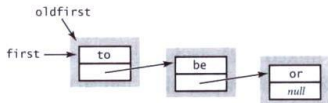
```
Node third = new Node();  
third.item = "or";  
second.next = third;
```



Insertion at the beginning

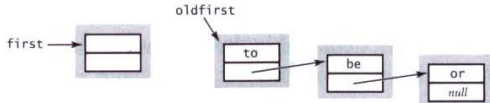
save a link to the list

```
Node oldfirst = first;
```



create a new node for the beginning

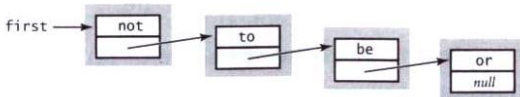
```
first = new Node();
```



set the instance variables in the new node



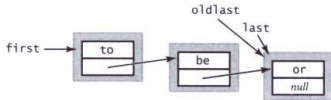
```
first.item = "not";  
first.next = oldfirst;
```



Insertion at the end

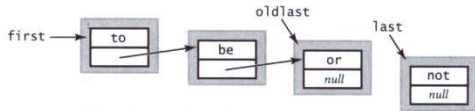
save a link to the last node

```
Node oldlast = last;
```



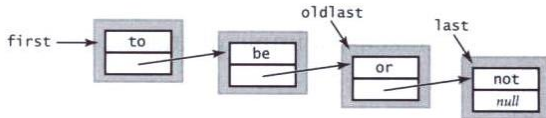
create a new node for the end

```
last = new Node();  
last.item = "not";
```



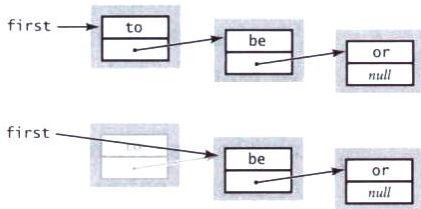
link the new node to the end of the list

```
oldlast.next = last;
```

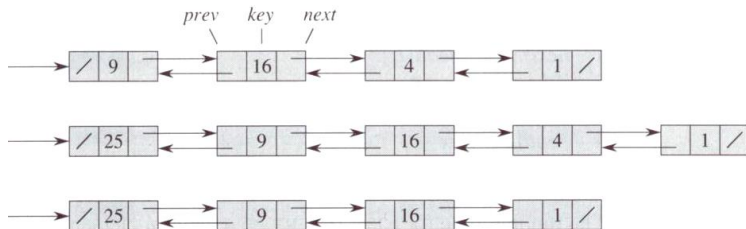


Deletion the first node

```
first = first.next;
```



Double Linked Lists



- **Node record** for double linked lists:

```
private class NodeDouble
```

```
{
```

```
    Item item
```

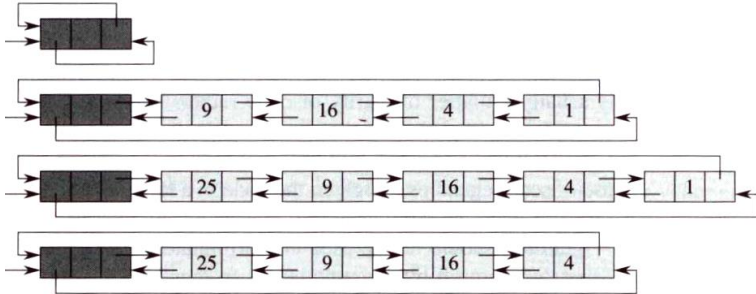
or 'key' as above

```
    NodeDouble prev
```

```
    NodeDouble next
```

```
}
```

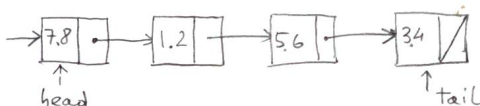
Circular Double Linked Lists with Sentinel



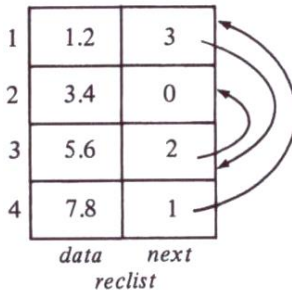
- The first node with 'external', often 'empty', item is called 'sentinel'

Implementing Linked Lists by Arrays

- A list:



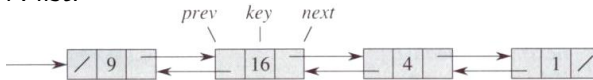
- The same list implemented as an array *reclist*:



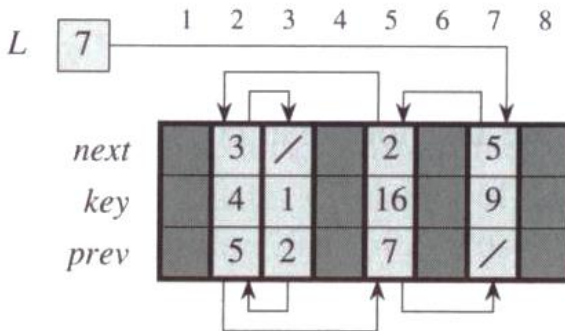
head = reclist[4][1]

Implementing Double Linked Lists by Arrays

- A list:



- The same list implemented as an array. Variable L keeps the index of the head.

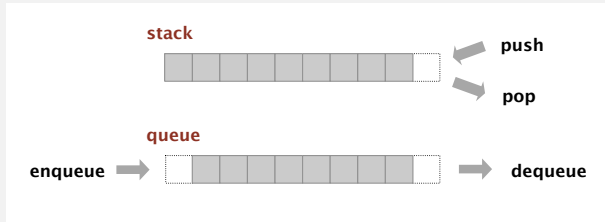


While all kinds of linked lists can be regarded as Abstract Data Types, they are usually interpreted as Data Structures used to implement other ADT's as *stack (LIFO)*, *queue (FIFO)*, *priority queue*, etc.

Stacks and Queues

Fundamental data types.

- Value: collection of objects.
- Operations: **insert**, **remove**, **iterate**, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

Stack (of strings) API

```
public class StackOfStrings
```

```
    StackOfStrings()           create an empty stack
```

```
    void push(String item)     insert a new string onto stack
```

```
    String pop()               remove and return the string  
                               most recently added
```

```
    boolean isEmpty()          is the stack empty?
```

```
    int size()                 number of strings on the stack
```

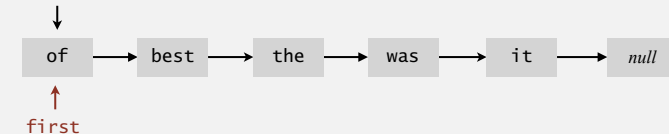
push pop



Linked List implementation of Stack

- Maintain pointer **first** to first node in a singly-linked list.
- Push new item before first.
- Pop item from first.

top of stack



Stack Pop

inner class

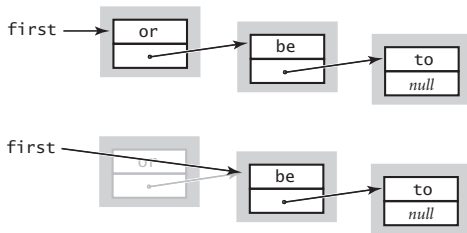
```
private class Node
{
    String item;
    Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```

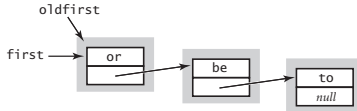


return saved item

Stack Push

save a link to the list

```
Node oldfirst = first;
```

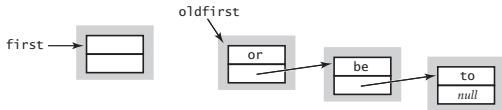


inner class

```
private class Node
{
    String item;
    Node next;
}
```

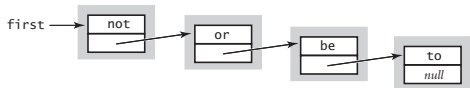
create a new node for the beginning

```
first = new Node();
```



set the instance variables in the new node

```
first.item = "not";
first.next = oldfirst;
```



Java implementation of Stack

```
public class LinkedStackOfStrings
{
    private Node first = null;
```

```
    private class Node
    {
        String item;
        Node next;
    }
```

```
    public boolean isEmpty()
    { return first == null; }
```

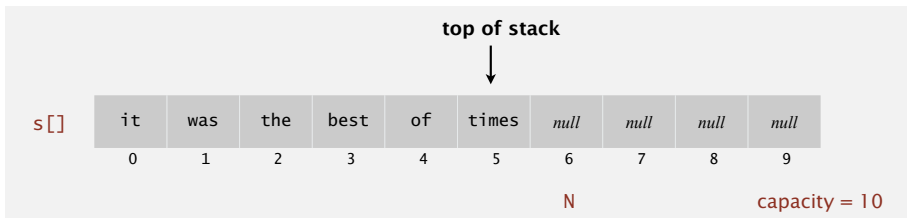
```
    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }
```

```
    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

← private inner class
(access modifiers for instance
variables don't matter)

Fixed-capacity stack: array implementation

- Use array $s[]$ to store N items on stack.
- $\text{push}()$: add new item at $s[N]$.
- $\text{pop}()$: remove item from $s[N-1]$.



- Stack overflows when N exceeds capacity.

Stack Applications

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

Arithmetic Expression Evaluation

Goal. Evaluate infix expressions.

$(1 + ((2 + 3) * (4 * 5)))$

operand

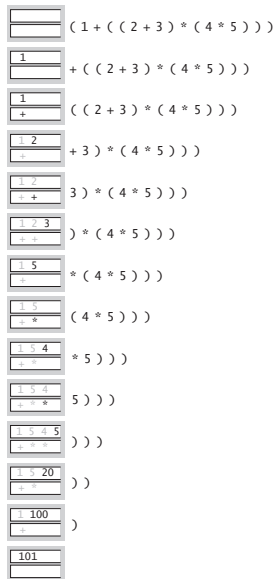
operator

value stack
operator stack

Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!



Queue API

```
public class QueueOfStrings
```

```
    QueueOfStrings()           create an empty queue
```

```
    void enqueue(String item)  insert a new string onto queue
```

```
    String dequeue()           remove and return the string  
                                least recently added
```

```
    boolean isEmpty()          is the queue empty?
```

```
    int size()                 number of strings on the queue
```

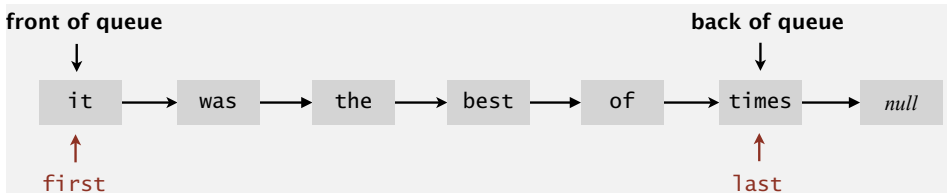
enqueue



dequeue

Queue: linked-list implementation

- Maintain one pointer **first** to first node in a singly-linked list.
- Maintain another pointer **last** to last node.
- Dequeue from **first**.
- Enqueue after **last**.



- Often: **first** \equiv **head** and **last** \equiv **tail**.

Queue *dequeue*

inner class

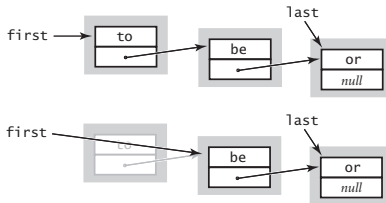
```
private class Node
{
    String item;
    Node next;
}
```

save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

- The procedure and code are identical to stack '*pop*'.

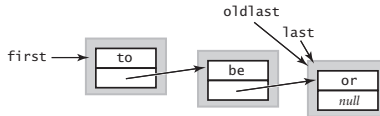
Queue enqueue

inner class

```
private class Node
{
    String item;
    Node next;
}
```

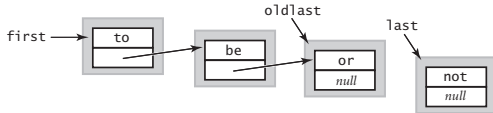
save a link to the last node

```
Node oldlast = last;
```



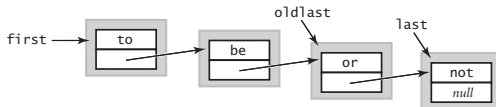
create a new node for the end

```
last = new Node();
last.item = "not";
```



link the new node to the end of the list

```
oldlast.next = last;
```



Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings
{
    private Node first, last;

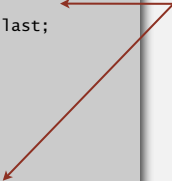
    private class Node
    { /* same as in LinkedStackOfStrings */ }

    public boolean isEmpty()
    { return first == null; }

    public void enqueue(String item)
    {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else
            oldlast.next = last;
    }

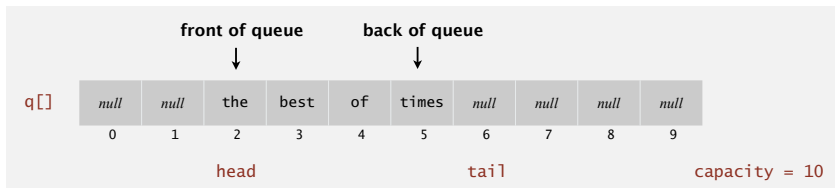
    public String dequeue()
    {
        String item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
```

special cases for
empty queue

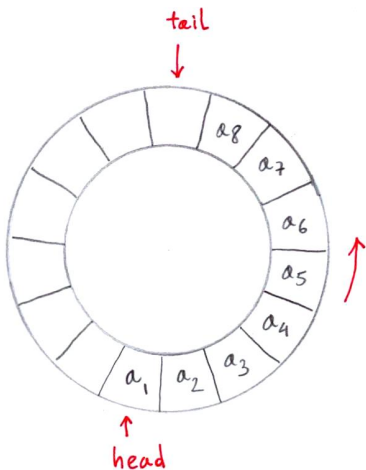


Queue: array implementation

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update head and tail modulo the capacity.



Queue: circular buffer



- The list is full is **tail** precedes **head**, i.e. $|\text{tail} - \text{head}| = 1$.
- $\text{List_capacity} = \text{Array_size} - 1$, as **tail** always contains *null*.