CHAPTER 3

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

ABSTRACT DATA TYPES

- Defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations“

- Objects such as lists, sets and graphs, along with their operations can be viewed as ADTs.

ADTs

- The List ADT
- The Stack ADT
- The Queue ADT

The List ADT

- General List : $A_0, A_1, A_2, \ldots A_{i-1}, A_i, A_{i+1}, \ldots, A_{N-1}$
  Size = N

- List of size zero is called empty list

- The Position of element A_i in a list is i

Operations on List ADT

- printList
- makeEmpty
- find
- Insert
- Remove
- findKth

Example

- List : 34, 12, 52, 16, 12
- find(52) = 2

- insert(60,2) will make list
  - 34, 12, 60, 52, 16, 12

- remove (52) will make list
  - 34, 12, 60, 16, 12

Array Implementation of List

```
int [] arr = new int[10];
…

//Later on we decide arr needs to be large.
int [] newArr = new int[arr.length*2];
for(i=0; i<arr.length; i++)
    newArr[i] = arr[i];
arr = newArr
```

Cont..

- An array implementation allows printList to be carried out in linear time, and the findKth operation takes constant time.
- insertion and deletion takes O(N)
- If insertion and deletion occur throughout the list including at the front of the list, then the array is not a good option.

LINKED LIST

- The linked list consist of series of nodes which are not necessarily adjacent in the memory.

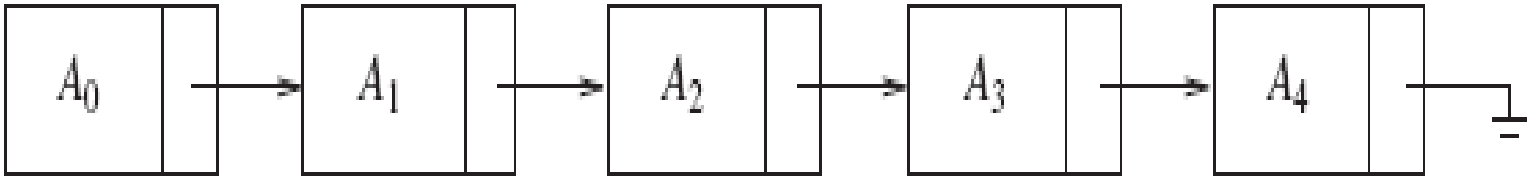


Figure 3.1 A linked list

Cont..

- printList and find(X) are O(N)
- findKth(i) takes O(i)

remove

- The remove method can be executed in one *next* reference change.

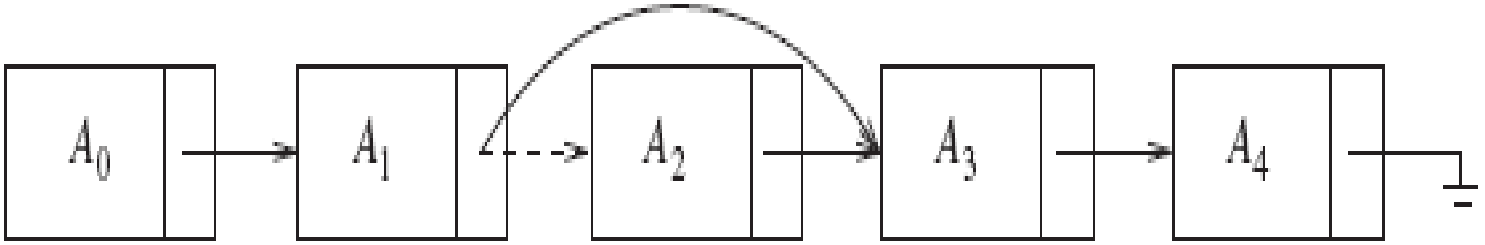


Figure 3.2 Deletion from a linked list

insert

- The insert method requires obtaining a new node from system by using a new call and then executing two reference maneuvers.

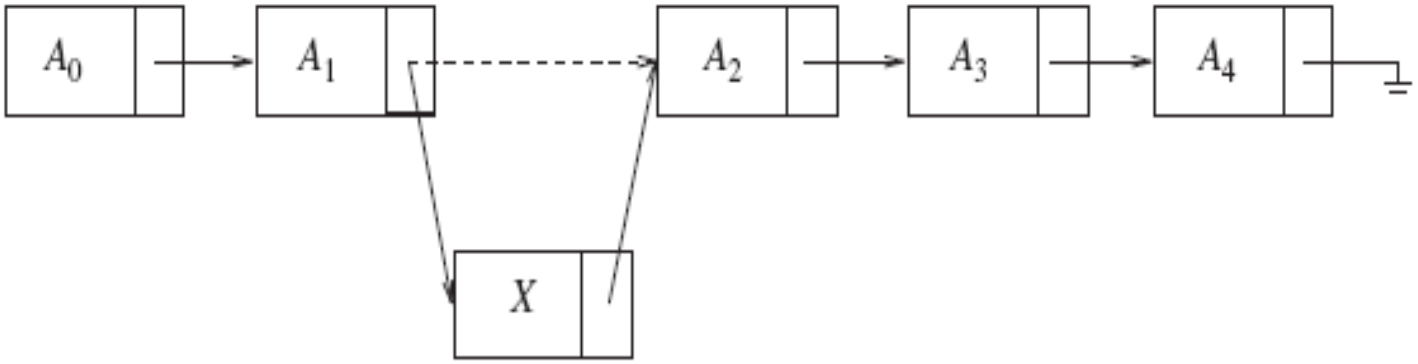


Figure 3.3 Insertion into a linked list

DOUBLY LINKED LIST

- We have every node maintain a link to its previous node in the list.

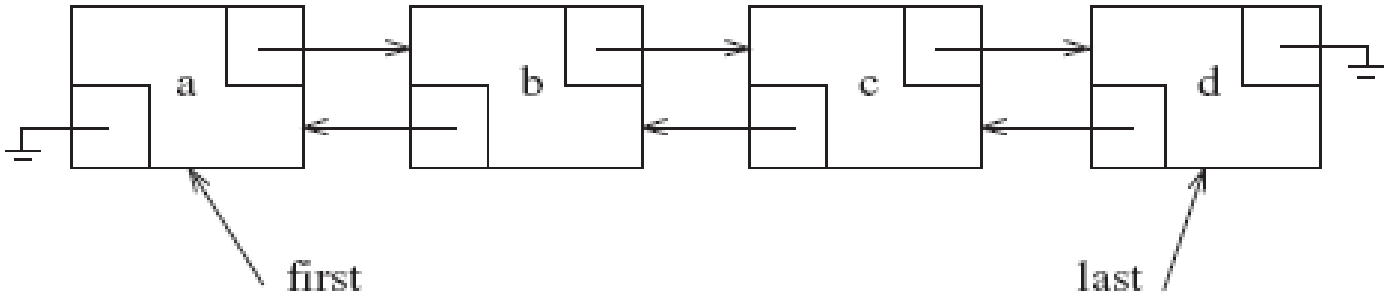


Figure 3.4 A doubly linked list

Java Collections API

- The Java language includes, in its library an implementation of common data structures. This part of the language is popularly known as the Collection API.

Collection Interface

- The Collection API resides in the package java.util.
- it has methods like
  - *isEmpty*
  - *size*
  - *contains*
  - *add*
  - *remove*

Iterable Interface

- The Collection interface extends Iterable interface.
- Iterable interface must provide a method named Iterator that returns an object of type Iterator.

Iterator

```
1   public interface Collection<AnyType> extends Iterable<AnyType>
2   {
3       int size( );
4       boolean isEmpty( );
5       void clear( );
6       boolean contains( AnyType x );
7       boolean add( AnyType x );
8       boolean remove( AnyType x );
9       java.util.Iterator<AnyType> iterator( );
10  }
```

Figure 3.5 Subset of the `Collection` interface in package `java.util`

Iterator Interface

```
1   public interface Iterator<AnyType>
2   {
3       boolean hasNext( );
4       AnyType next( );
5       void remove( );
6   }
```

Figure 3.7 The Iterator interface in package java.util

Enhanced for loop

```
1       public static <AnyType> void print( Collection<AnyType> coll )
2       {
3           for( AnyType item : coll )
4               System.out.println( item );
5       }
```

Figure 3.6 Using the enhanced `for` loop on an `Iterable` type

Enhanced for loop in Iterable type

```
1       public static <AnyType> void print( Collection<AnyType> coll )
2       {
3           Iterator<AnyType> itr = coll.iterator( );
4           while( itr.hasNext( ) )
5           {
6               AnyType item = itr.next( );
7               System.out.println( item );
8           }
9       }
```

Figure 3.8 The enhanced for loop on an `Iterable` type rewritten by the compiler to use an iterator

Iterator Interface

- Limited number of methods.
- *remove* : can remove the last item returned by *next* (after which you cannot call remove again until after another call to *next*)
- Fundamental rule : if a structural change to the collection is made the *iterator* is no longer valid – *ConcurrentModificationException* is thrown.

List Interface, ArrayList and LinkedList

- The *List* interface extends *Collection* interface.
- it contains all methods in *Collection* and few others.

  get

  set

  add

  remove

  listIterator()

List interface

```
1   public interface List<AnyType> extends Collection<AnyType>
2   {
3       AnyType get( int idx );
4       AnyType set( int idx, AnyType newVal );
5       void add( int idx, AnyType x );
6       void remove( int idx );
7
8       ListIterator<AnyType> listIterator( int pos );
9   }
```

Figure 3.9 Subset of the `List` interface in package `java.util`

Implementation of List ADT

- ArrayList
  - get() and set() are constant time
  - insertion and deletion are expensive
- LinkedList
  - doubly linked list implementation
  - insertion and removing is cheap
  - get() and set() are expensive O(N)

Comparing ArrayList and LinkedList

```
public static void makeList1(List<integer>  lst, int N)
{
    lst.clear();
    for( int i = 0; i< N; i++)
        lst.add( i ) ;
}
```

Comparing ArrayList and LinkedList

```
public static void makeList1(List<integer>  lst, int N)
{
    lst.clear();
    for( int i = 0; i< N; i++)
        lst.add( i ) ;
}
```

ArrayList and LinkedList will take O(N) time

Comparing ArrayList and LinkedList

```
public static void makeList2(List<integer>  lst, int N)
{
    lst.clear();
    for( int i = 0; i< N; i++)
        lst.add( 0, i );
}
```

Comparing ArrayList and LinkedList

```
public static void makeList2(List<integer>  lst, int N)
{
    lst.clear();
    for( int i = 0; i< N; i++)
        lst.add( 0, i );
}
```

Array List will take $O(N^2)$ time

Linked List will take O(N) time

Comparing ArrayList and LinkedList

```
public static int sum(List<integer>  lst)
{
    int total = 0 ;
    for( int i = 0; i < N; i++)
        total = total + lst.get( i ) ;
}
```

Comparing ArrayList and LinkedList

```
public static int sum(List<integer>  lst)
{
    int total = 0 ;
    for( int i = 0; i < N; i++)
        total = total + lst.get( i ) ;
}
```

ArrayList will take O(N) time

LinkedList will take $O(N^2)$ time

Example: Using *remove* on a LinkedList

- Remove all the even valued items in a list

```
1       public static void removeEvensVer1( List<Integer> lst )
2       {
3           int i = 0;
4           while( i < lst.size( ) )
5               if( lst.get( i ) % 2 == 0 )
6                   lst.remove( i );
7               else
8                   i++;
9       }
```

Example: Using *remove* on a LinkedList

- Remove all the even valued items in a list

```
1       public static void removeEvensVer1( List<Integer> lst )
2       {
3           int i = 0;
4           while( i < lst.size( ) )
5               if( lst.get( i ) % 2 == 0 )
6                   lst.remove( i );
7               else
8                   i++;
9       }
```

Figure 3.10 Removes the even numbers in a list; quadratic on all types of lists

Using Enhanced *for* loop

```
1       public static void removeEvensVer2( List<Integer> lst )
2       {
3           for( Integer x : lst )
4               if( x % 2 == 0 )
5                   lst.remove( x );
6       }
```

Using Enhanced *for* loop

```
1       public static void removeEvensVer2( List<Integer> lst )
2       {
3           for( Integer x : lst )
4               if( x % 2 == 0 )
5                   lst.remove( x );
6       }
```

Figure 3.11 Removes the even numbers in a list; doesn't work because of ConcurrentModificationException

Using Iterator

```
1       public static void removeEvensVer3( List<Integer> lst )
2       {
3           Iterator<Integer> itr = lst.iterator( );
4
5           while( itr.hasNext( ) )
6               if( itr.next( ) % 2 == 0 )
7                   itr.remove( );
8       }
```

Using Iterator

```
1       public static void removeEvensVer3( List<Integer> lst )
2       {
3           Iterator<Integer> itr = lst.iterator( );
4
5           while( itr.hasNext( ) )
6               if( itr.next( ) % 2 == 0 )
7                   itr.remove( );
8       }
```

Figure 3.12 Removes the even numbers in a list; quadratic on `ArrayList`, but linear time for `LinkedList`

ListIterator

```
1   public interface ListIterator<AnyType> extends Iterator<AnyType>
2   {
3       boolean hasPrevious( );
4       AnyType previous( );
5
6       void add( AnyType x );
7       void set( AnyType newVal );
8   }
```

Figure 3.13 Subset of the `ListIterator` interface in package `java.util`

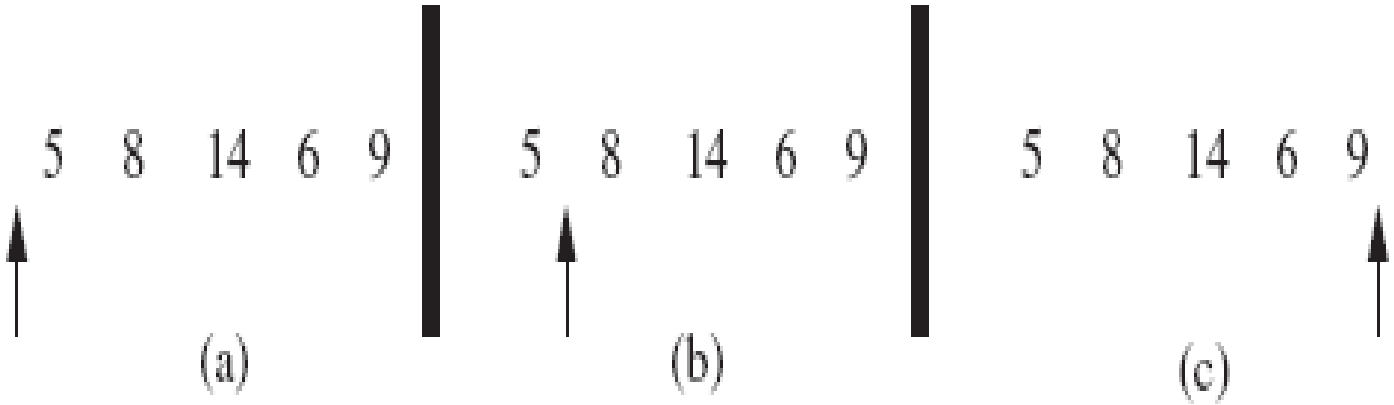


Figure 3.14 (a) Normal starting point: `next` returns 5, `previous` is illegal, `add` places item before 5; (b) `next` returns 8, `previous` returns 5, `add` places item between 5 and 8; (c) `next` is illegal, `previous` returns 9, `add` places item after 9

Implementation Of ArrayList

- To avoid ambiguities with the library class, we will name our class *MyArrayList*.
  - it maintains the underlying array, the array capacity and current number of items stored in the *MyArrayList*.
  - provides a mechanism to change the capacity of the underlying array.
  - implementation of get and set
  - routines such *as size, isEmpty, clear, add, remove* ..
  - provides a class that implements Iterator interface.

```
1    public class MyArrayList<AnyType> implements Iterable<AnyType>
2    {
3        private static final int DEFAULT_CAPACITY = 10;
4
5        private int theSize;
6        private AnyType [ ] theItems;
7
8        public MyArrayList( )
9          { doClear( ); }
10
11       public void clear( )
12         { doClear( ); }
13
14       private void doClear( )
15         { theSize = 0; ensureCapacity( DEFAULT_CAPACITY ); }
16
17       public int size( )
18         { return theSize; }
19       public boolean isEmpty( )
20         { return size( ) == 0; }
21       public void trimToSize( )
22         { ensureCapacity( size( ) ); }
23
24       public AnyType get( int idx )
25       {
26           if( idx < 0 || idx >= size( ) )
27               throw new ArrayIndexOutOfBoundsException( );
28           return theItems[ idx ];
29       }
30
31       public AnyType set( int idx, AnyType newVal )
32       {
33           if( idx < 0 || idx >= size( ) )
34               throw new ArrayIndexOutOfBoundsException( );
35           AnyType old = theItems[ idx ];
36           theItems[ idx ] = newVal;
37           return old;
38       }
39
40       public void ensureCapacity( int newCapacity )
41       {
42           if( newCapacity < theSize )
43               return;
44
45           AnyType [ ] old = theItems;
46           theItems = (AnyType []) new Object[ newCapacity ];
47           for( int i = 0; i < size( ); i++ )
48               theItems[ i ] = old[ i ];
49       }
```

Figure 3.15 MyArrayList class (Part 1 of 2)

```
50          public boolean add( AnyType x )
51          {
52              add( size( ), x );
53              return true;
54          }
55
56          public void add( int idx, AnyType x )
57          {
58              if( theItems.length == size( ) )
59                  ensureCapacity( size( ) * 2 + 1 );
60              for( int i = theSize; i > idx; i-- )
61                  theItems[ i ] = theItems[ i - 1 ];
62              theItems[ idx ] = x;
63
64              theSize++;
65          }
66
67          public AnyType remove( int idx )
68          {
69              AnyType removedItem = theItems[ idx ];
70              for( int i = idx; i < size( ) - 1; i++ )
71                  theItems[ i ] = theItems[ i + 1 ];
72
73              theSize--;
74              return removedItem;
75          }
76
77          public java.util.Iterator<AnyType> iterator( )
78            { return new ArrayListIterator( ); }
79
80          private class ArrayListIterator implements java.util.Iterator<AnyType>
81          {
82              private int current = 0;
83
84              public boolean hasNext( )
85                { return current < size( ); }
86
87              public AnyType next( )
88              {
89                  if( !hasNext( ) )
90                      throw new java.util.NoSuchElementException( );
91                  return theItems[ current++ ];
92              }
93
94              public void remove( )
95                { MyArrayList.this.remove( --current ); }
96          }
97      }
```

Figure 3.16 `MyArrayList` class (Part 2 of 2)

Routines

- *ensureCapacity*
  - Used to expand capacity as well as shrink the underlying array.
  - At line 46, an idiom that is required because generic array creation is illegal.
- *add*
  - computational expensive
  - requires increasing capacity of the array
- *remove*
  - similar to *add*

iterator

- iterator method returns an instance of ArrayListIterator. which is a class that implements iterator interface.
- ArrayListInterator is a inner class, this is allowed in java.
- It should be an inner class to use the data items from MyArrayList class like theItems variable.

Erroneous Code

```
1   public class MyArrayList<AnyType> implements Iterable<AnyType>
2   {
3       private int theSize;
4       private AnyType [ ] theItems;
5           ...
6       public java.util.Iterator<AnyType> iterator( )
7         { return new ArrayListIterator<AnyType>( ); }
8   }
9   class ArrayListIterator<AnyType> implements java.util.Iterator<AnyType>
10  {
11      private int current = 0;
12          ...
13      public boolean hasNext( )
14        { return current < size( ); }
15      public AnyType next( )
16        { return theItems[ current++ ]; }
17  }
```

Figure 3.17 Iterator Version #1 (doesn't work): The iterator is a top-level class and stores the current position. It doesn't work because `theItems` and `size()` are not part of the `ArrayListIterator` class

```
1   public class MyArrayList<AnyType> implements Iterable<AnyType>
2   {
3       private int theSize;
4       private AnyType [ ] theItems;
5           ...
6       public java.util.Iterator<AnyType> iterator( )
7         { return new ArrayListIterator<AnyType>( this ); }
8   }
9   class ArrayListIterator<AnyType> implements java.util.Iterator<AnyType>
10  {
11      private int current = 0;
12      private MyArrayList<AnyType> theList;
13          ...
14      public ArrayListIterator( MyArrayList<AnyType> list )
15        { theList = list; }
16
17      public boolean hasNext( )
18        { return current < theList.size( ); }
19      public AnyType next( )
20        { return theList.theItems[ current++ ]; }
21  }
```

Figure 3.18 Iterator Version #2 (almost works): The iterator is a top-level class and stores the current position and a link to the `MyArrayList`. It doesn't work because `theItems` is private in the `MyArrayList` class

```
1   public class MyArrayList<AnyType> implements Iterable<AnyType>
2   {
3       private int theSize;
4       private AnyType [ ] theItems;
5           ...
6       public java.util.Iterator<AnyType> iterator( )
7         { return new ArrayListIterator<AnyType>( this ); }
8
9       private static class ArrayListIterator<AnyType>
10                                              implements java.util.Iterator<AnyType>
11      {
12          private int current = 0;
13          private MyArrayList<AnyType> theList;
14                ...
15          public ArrayListIterator( MyArrayList<AnyType> list )
16            { theList = list; }
17
18          public boolean hasNext( )
19            { return current < theList.size( ); }
20          public AnyType next( )
21            { return theList.theItems[ current++ ]; }
22      }
23  }
```

Figure 3.19 Iterator Version #3 (works): The iterator is a nested class and stores the current position and a link to the `MyArrayList`. It works because the nested class is considered part of the `MyArrayList` class

Nested Class and Inner Class

- Difference between nested class and inner class is nested class are static so they associated with class not with the instance of the class.
- inner classes are associated with outer class objects.
- inner class is useful in a situation in which each inner class object is associated with exactly one instance of an outer class object.

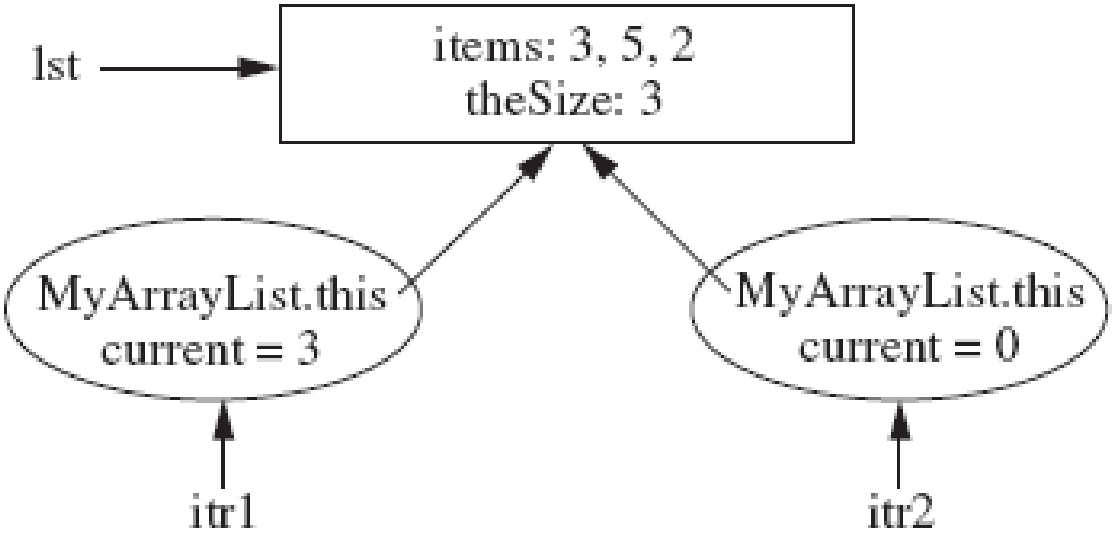


Figure 3.20 Iterator/container with inner classes

```
1   public class MyArrayList<AnyType> implements Iterable<AnyType>
2   {
3       private int theSize;
4       private AnyType [ ] theItems;
5           ...
6       public java.util.Iterator<AnyType> iterator( )
7         { return new ArrayListIterator( ); }
8
9       private class ArrayListIterator implements java.util.Iterator<AnyType>
10      {
11          private int current = 0;
12
13          public boolean hasNext( )
14            { return current < size( ); }
15          public AnyType next( )
16            { return theItems[ current++ ]; }
17          public void remove( )
18            { MyArrayList.this.remove( --current ); }
19      }
20  }
```

Figure 3.21 Iterator Version #4 (works): The iterator is an inner class and stores the current position and an implicit link to the `MyArrayList`

Implementation of *LinkedList*

- Named MyLinkedList to avoid ambiguity with library class.
- implemented as a doubly linked list.
- In design three classes are provided
  - MyLinkedList
  - Node
  - LinkedListIterator

Advantages of these extra nodes

- Greatly simplifies the code
  - For example without the header node then removing the first element becomes a special case

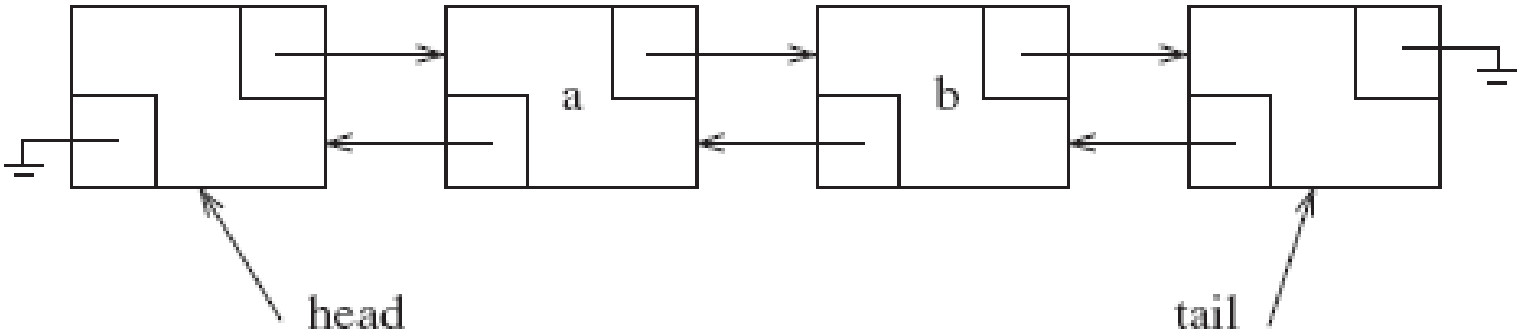


Figure 3.22 A doubly linked list with header and tail nodes

EMPTY LIST

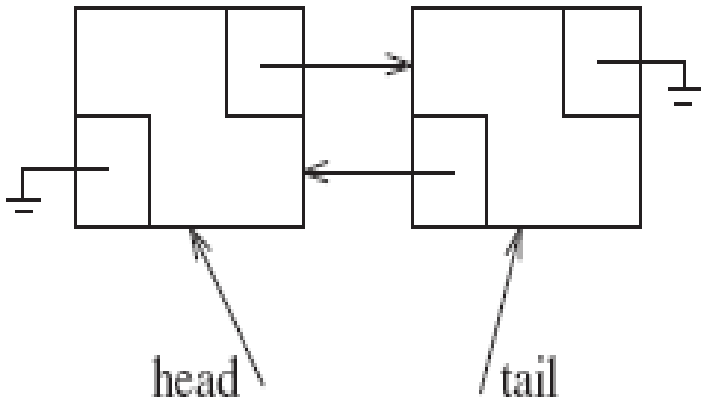


Figure 3.23 An empty doubly linked list with header and tail nodes

```
1    public class MyLinkedList<AnyType> implements Iterable<AnyType>
2    {
3        private static class Node<AnyType>
4          { /* Figure 3.25 */ }
5
6        public MyLinkedList( )
7          { doClear( ); }
8
9        public void clear( )
10         { /* Figure 3.26 */ }
11       public int size( )
12         { return theSize; }
13       public boolean isEmpty( )
14         { return size( ) == 0; }
15
16       public boolean add( AnyType x )
17         { add( size( ), x );  return true; }
18       public void add( int idx, AnyType x )
19         { addBefore( getNode( idx, 0, size( ) ), x ); }
20       public AnyType get( int idx )
21         { return getNode( idx ).data; }
22       public AnyType set( int idx, AnyType newVal )
23       {
24           Node<AnyType> p = getNode( idx );
25           AnyType oldVal = p.data;
26           p.data = newVal;
27           return oldVal;
28       }
29       public AnyType remove( int idx )
30         { return remove( getNode( idx ) ); }
31
32       private void addBefore( Node<AnyType> p, AnyType x )
33         { /* Figure 3.28 */ }
34       private AnyType remove( Node<AnyType> p )
35         { /* Figure 3.30 */ }
36       private Node<AnyType> getNode( int idx )
37         { /* Figure 3.31 */ }
38       private Node<AnyType> getNode( int idx, int lower, int upper )
39         { /* Figure 3.31 */ }
40
41       public java.util.Iterator<AnyType> iterator( )
42          { return new LinkedListIterator( ); }
43       private class LinkedListIterator implements java.util.Iterator<AnyType>
44         { /* Figure 3.32 */ }
45
46       private int theSize;
47       private int modCount = 0;
48       private Node<AnyType> beginMarker;
49       private Node<AnyType> endMarker;
50   }
```

Figure 3.24 MyLinkedList class

Node Class

```
1       private static class Node<AnyType>
2       {
3           public Node( AnyType d, Node<AnyType> p, Node<AnyType> n )
4             { data = d; prev = p; next = n; }
5
6           public AnyType data;
7           public Node<AnyType>   prev;
8           public Node<AnyType>   next;
9       }
```

Figure 3.25 Nested `Node` class for `MyLinkedList` class

clear and *doclear* methods

```
1       public void clear( )
2         { doClear( ); }
3
4       private void doClear( )
5       {
6           beginMarker = new Node<AnyType>( null, null, null );
7           endMarker = new Node<AnyType>( null, beginMarker, null );
8           beginMarker.next = endMarker;
9
10          theSize = 0;
11          modCount++;
12      }
```

Figure 3.26 clear routine for MyLinkedList class, which invokes private doClear

addBefore Method

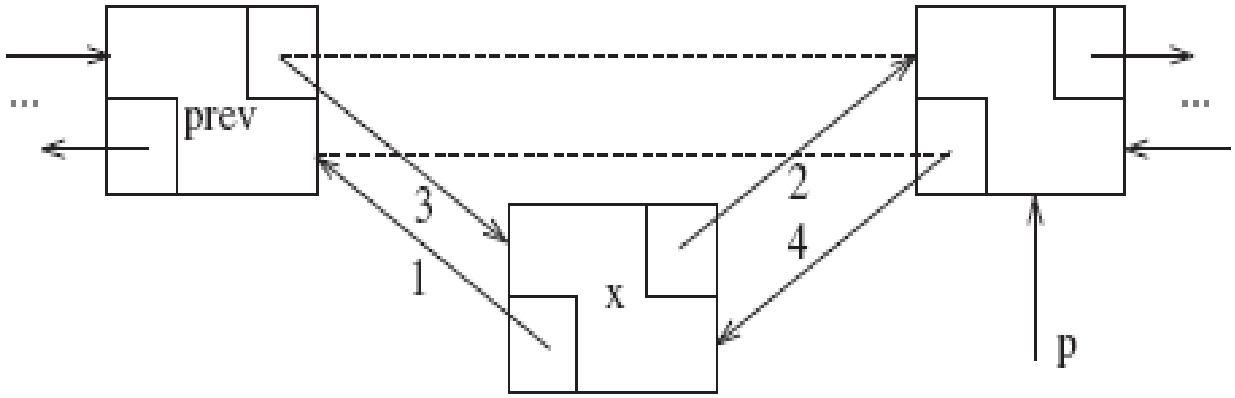


Figure 3.27 Insertion in a doubly linked list by getting new node and then changing pointers in the order indicated

addBefore Method

```
1        /**
2         * Adds an item to this collection, at specified position p.
3         * Items at or after that position are slid one position higher.
4         * @param p Node to add before.
5         * @param x any object.
6         * @throws IndexOutOfBoundsException if idx is not between 0 and size(),.
7         */
8        private void addBefore( Node<AnyType> p, AnyType x )
9        {
10           Node<AnyType> newNode = new Node<>( x, p.prev, p );
11           newNode.prev.next = newNode;
12           p.prev = newNode;
13           theSize++;
14           modCount++;
15       }
```

Figure 3.28 add routine for `MyLinkedList` class

Removing node p

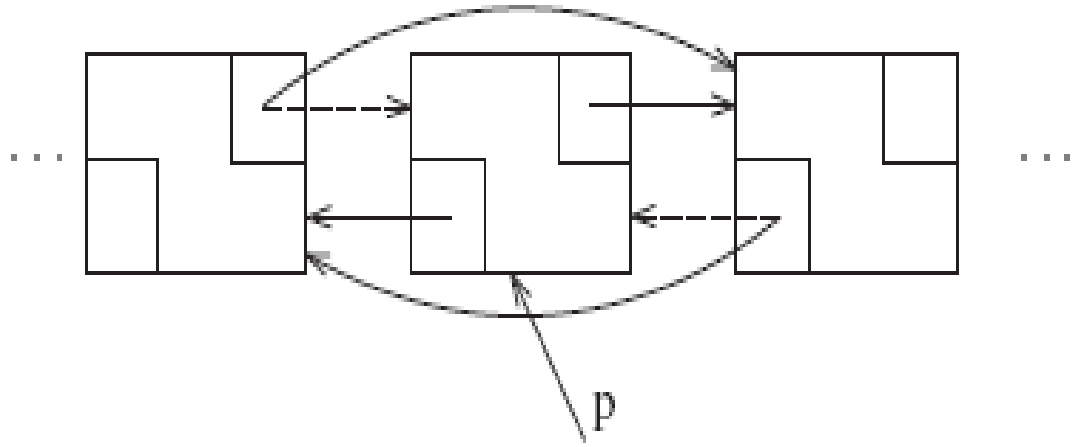


Figure 3.29 Removing node specified by p from a doubly linked list

Removing node p

```
1       /**
2        * Removes the object contained in Node p.
3        * @param p the Node containing the object.
4        * @return the item was removed from the collection.
5        */
6       private AnyType remove( Node<AnyType> p )
7       {
8           p.next.prev = p.prev;
9           p.prev.next = p.next;
10          theSize--;
11          modCount++;
12
13          return p.data;
14      }
```

Figure 3.30 remove routine for MyLinkedList class

getNode() Method

- if the index represents a node in the first half of the list then we step through the linked lit, in the forward direction otherwise we go backward starting at the end.

```
1         /**
2          * Gets the Node at position idx, which must range from 0 to size( ) - 1.
3          * @param idx index to search at.
4          * @return internal node corresponding to idx.
5          * @throws IndexOutOfBoundsException if idx is not
6          *           between 0 and size( ) - 1, inclusive.
7          */
8         private Node<AnyType> getNode( int idx )
9         {
10            return getNode( idx, 0, size( ) - 1 );
11        }
12
13        /**
14         * Gets the Node at position idx, which must range from lower to upper.
15         * @param idx index to search at.
16         * @param lower lowest valid index.
17         * @param upper highest valid index.
18         * @return internal node corresponding to idx.
19         * @throws IndexOutOfBoundsException if idx is not
20         *           between lower and upper, inclusive.
21         */
22        private Node<AnyType> getNode( int idx, int lower, int upper )
23        {
24            Node<AnyType> p;
25
26            if( idx < lower || idx > upper )
27                throw new IndexOutOfBoundsException( );
28
29            if( idx < size( ) / 2 )
30            {
31                p = beginMarker.next;
32                for( int i = 0; i < idx; i++ )
33                    p = p.next;
34            }
35            else
36            {
37                p = endMarker;
38                for( int i = size( ); i > idx; i-- )
39                    p = p.prev;
40            }
41
42            return p;
43        }
```

Figure 3.31 Private `getNode` routine for `MyLinkedList` class

Linked List Iterator

- Incorporates significant error checking.
- data field **expectedModCount** stores the modCount of the linked list at the time iterator is constructed.
- Boolean data field **okToRemove** is set to *false* initially and set to *true* in *next* and set to false in *remove* .

```
1       private class LinkedListIterator implements java.util.Iterator<AnyType>
2       {
3           private Node<AnyType> current = beginMarker.next;
4           private int expectedModCount = modCount;
5           private boolean okToRemove = false;
6
7           public boolean hasNext( )
8             { return current != endMarker; }
9
10          public AnyType next( )
11          {
12              if( modCount != expectedModCount )
13                  throw new java.util.ConcurrentModificationException( );
14              if( !hasNext( ) )
15                  throw new java.util.NoSuchElementException( );
16
17              AnyType nextItem = current.data;
18              current = current.next;
19              okToRemove = true;
20              return nextItem;
21          }
22
23          public void remove( )
24          {
25              if( modCount != expectedModCount )
26                  throw new java.util.ConcurrentModificationException( );
27              if( !okToRemove )
28                  throw new IllegalStateException( );
29
30              MyLinkedList.false.remove( current.prev );
31              expectedModCount++;
32              okToRemove = false;
33          }
34      }
```

Figure 3.32 Inner `Iterator` class for `MyList` class

The Stack ADT

- A stack is a list with restriction that insertions and deletions can be performed in only one position, namely the end of the list called top.
- Operations on Stack
  - push
  - pop

Stack Model

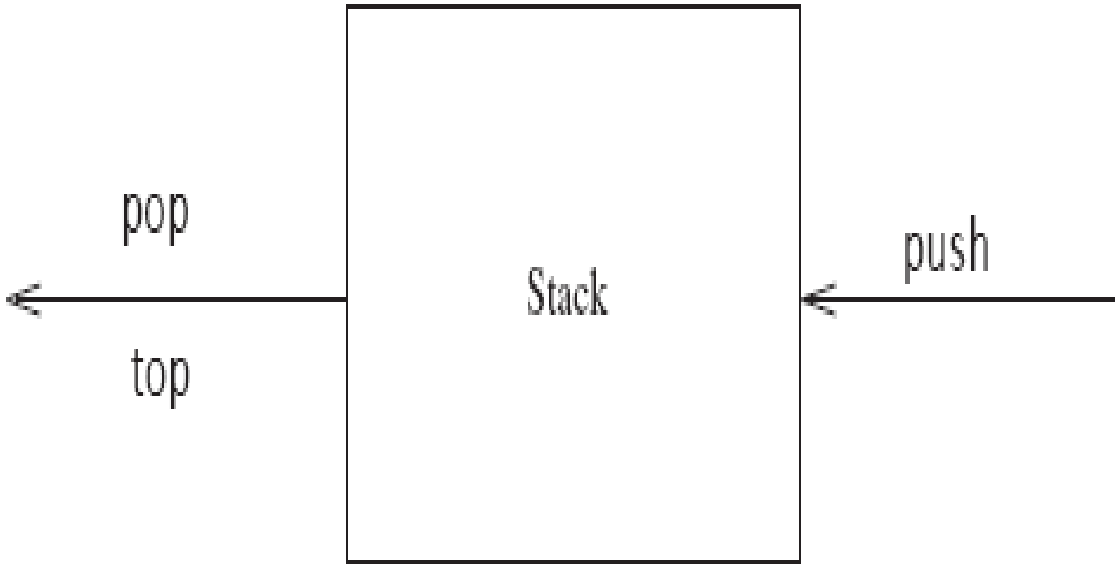


Figure 3.33 Stack model: input to a stack is by push, output is by pop and top

LIFO List

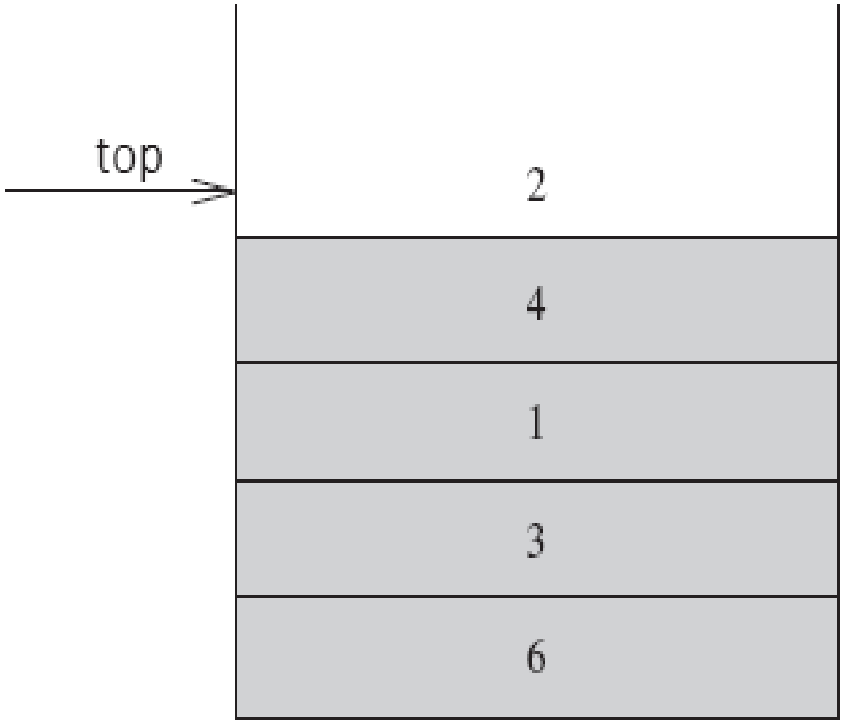


Figure 3.34 Stack model: Only the top element is accessible

Implementation of Stack

- Any list implementation will do.
- Linked List implementation of Stacks
  - Singly linked list is enough
  - push inserts at the front
  - pop deletes and returns the front list item
  - top just returns the front list item

Array List Implementation

- Associated with each stack is ***theArray*** and ***topOfStack*** which is -1 initially.
- To push some element x , increment ***topOfStack*** and then set

  theArray[topOfStack] = x
- To pop return ***theArray[topOfStack]*** and decrement ***topOfStack***

Applications Of Stack

- Balancing Symbols
- Postfix Expressions
- Infix to Postfix Conversion
- Method Calls

Balancing Symbols

- A program that checks whether everything is balanced. Every right brace, bracket and parenthesis must correspond to its left counterpart.
  - [()] – Legal
  - [(]) – Illegal

- It is easy to check these using stack data type

Algorithm

- Make an empty stack.
- Read characters until end of file.
  - If the character is an opening symbol, push it onto the stack
  - if it is a closing symbol,
    - if stack is empty report an error
    - else pop the stack, if symbol popped is not the corresponding opening symbol , report an error
  - At end of file, if the stack is not empty report an error.

Postfix Expressions

Postfix Expression	**Result**
4 5 +	9
9 3 /	3
9 8 -	1

This is also called reverse polished notation.

Stack can be used to evaluate postfix expressions.

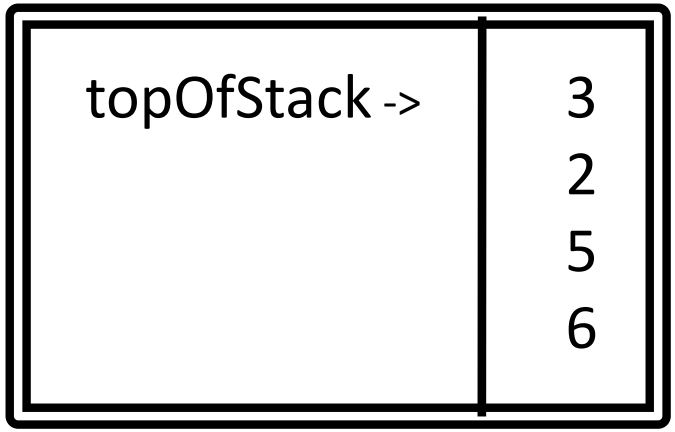
Algorithm

- When a number is seen push into stack
- When an operator is seen pop two numbers, apply the operator on them and push the result onto the stack

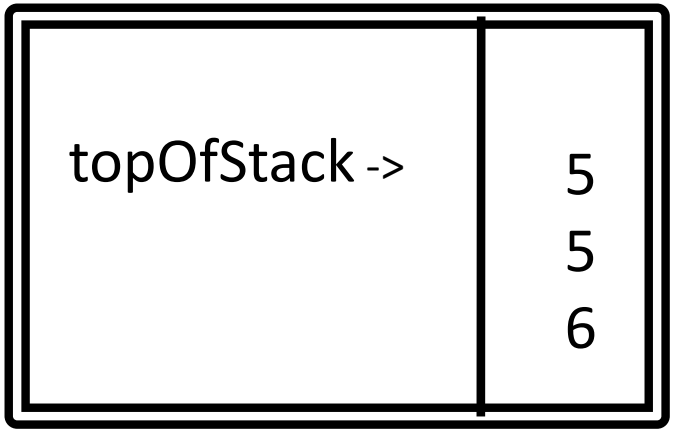
For example expression

6523+8*+3+*

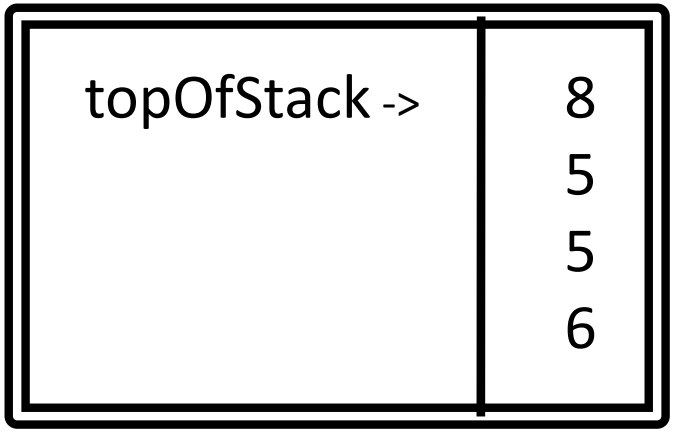
The First Four numbers are pushed



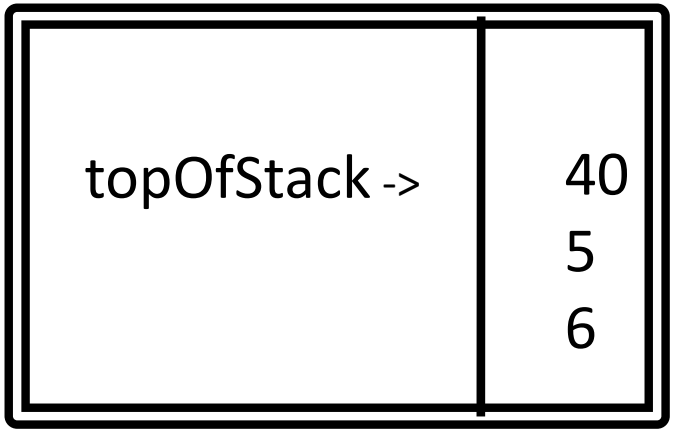
a ‘+’ is read 3 and 2 are popped and result 5 is pushed



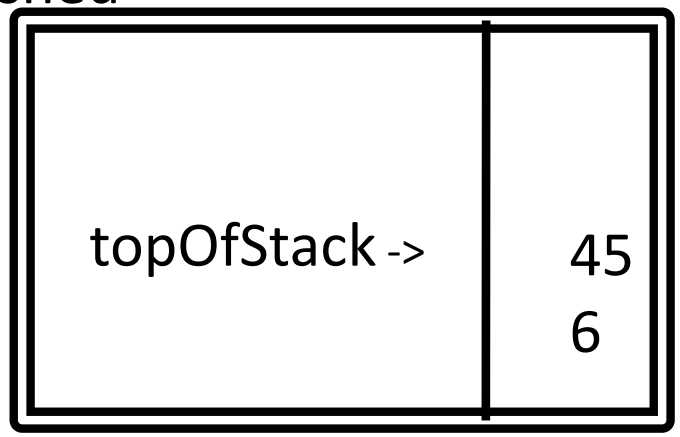
Next 8 is pushed



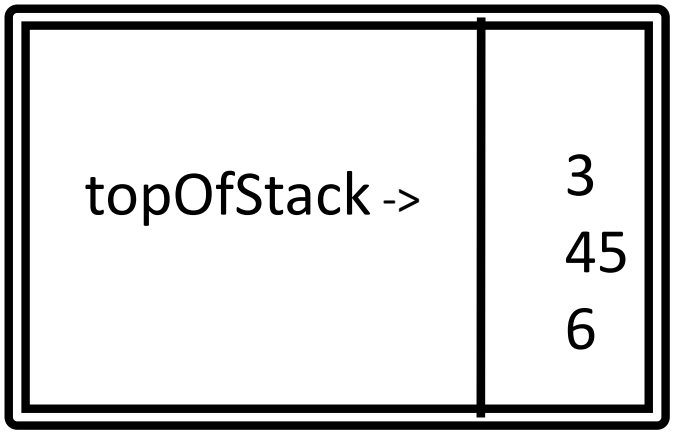
Next a ‘*’ is seen so 8 and 5 are popped and the result 40 is pushed



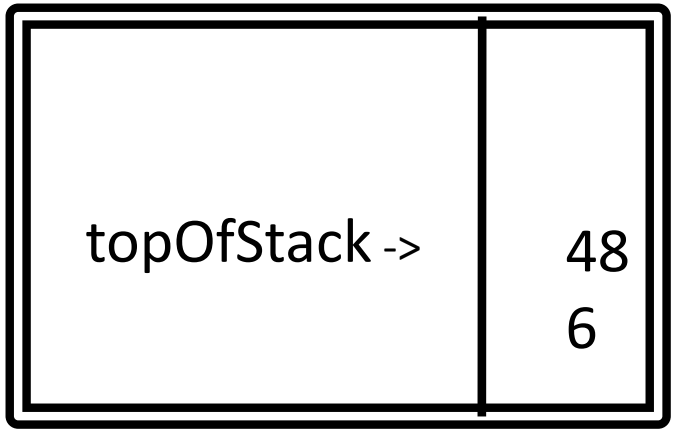
Next a ‘+’ is seen so 40 and 5 are popped and the result 45 is pushed



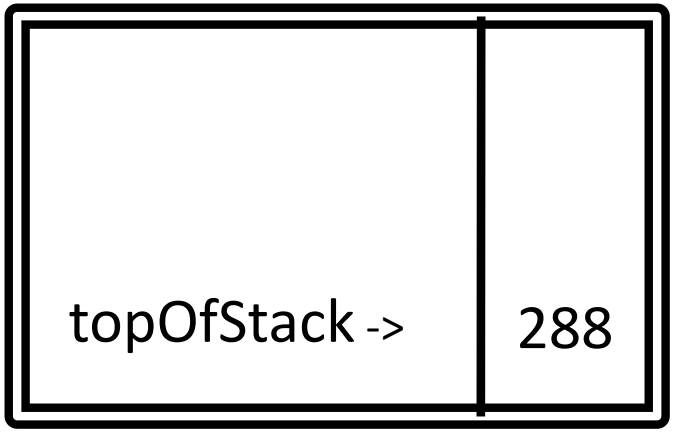
Now 3 is pushed



Nest ‘+’ pops 3 and 45 and pushes 48



a ‘*’ is seen 48 and 6 is popped and 288 is pushed



Infix to Postfix Conversion

- Stack can be used to convert from infix to postfix

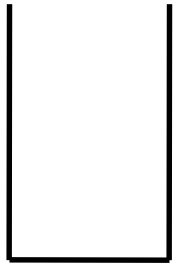
infix expression : a + b * c + (d * e + f) * g

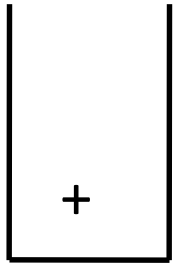
postfix expression : acb*+de*f+g*+

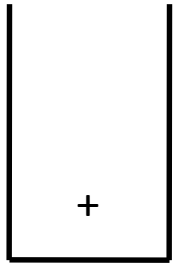
Algorithm

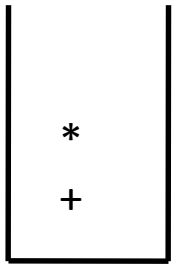
- Stack is initially empty
- When an operand is read it is immediately placed onto the output.
- If we see a right parenthesis, then we pop stack writing symbols until we see left parenthesis which is popped but not output.
- if we see any other symbol (, +, *, then we pop the entire stack until we see lower priority one exception is we never remove (unless we encounter)

a + b * c + (d * e + f) * g

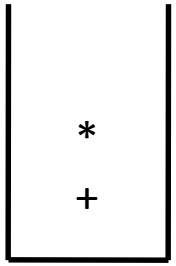




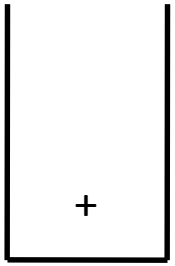




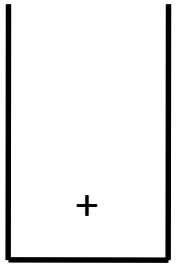
a + b * c + (d * e + f) * g



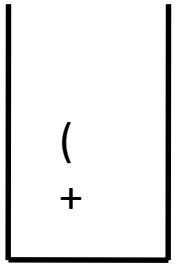
abc



abc*+

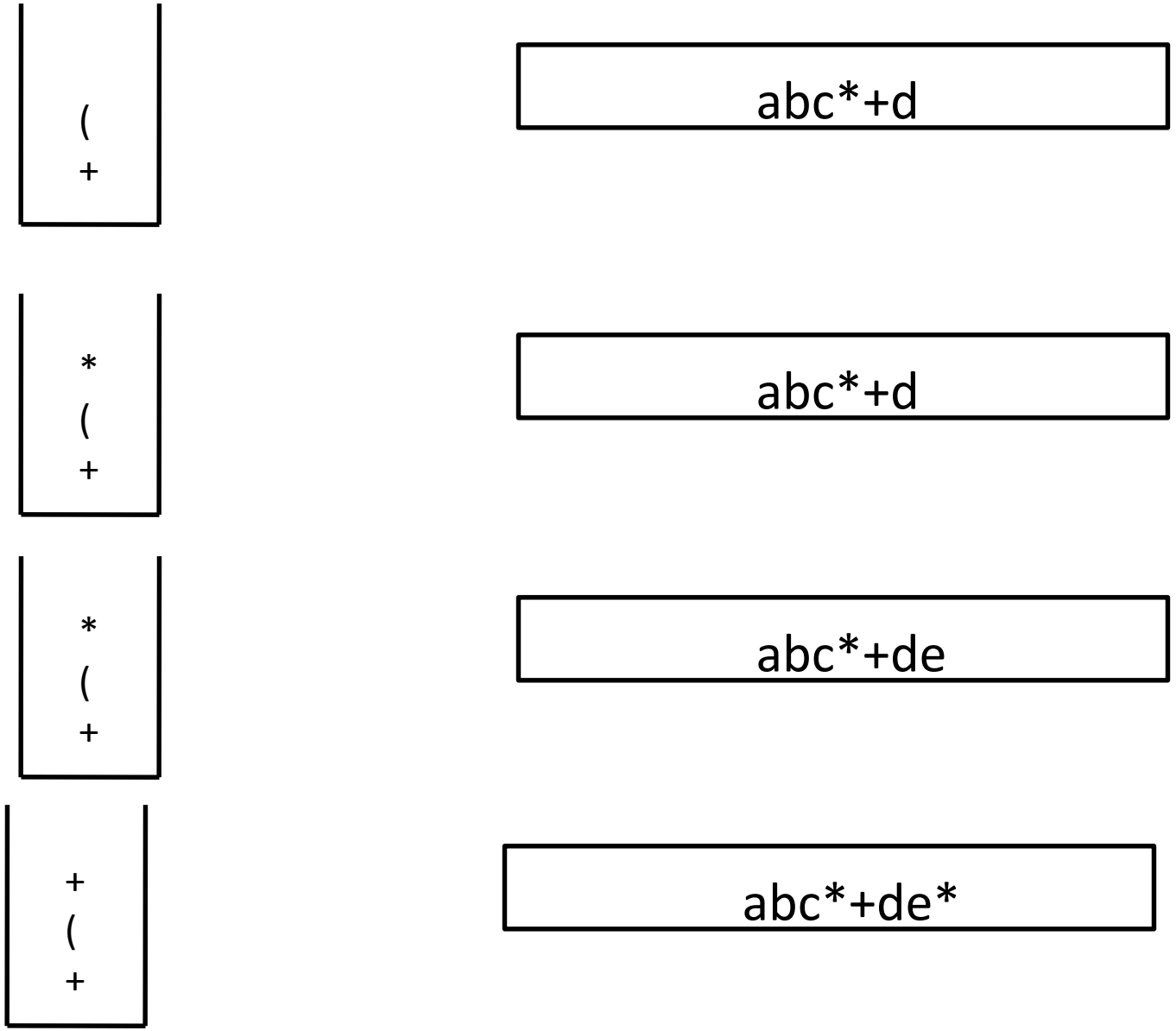


abc*+

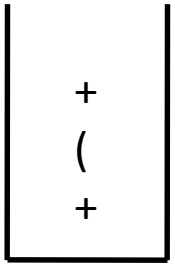


abc*+

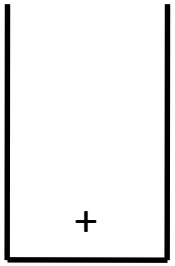
a + b * c + (d * e + f) * g



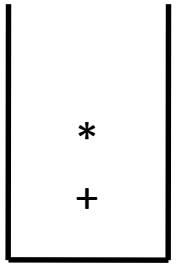
a + b * c + (d * e + f) * g



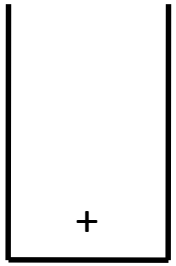
abc*+de*f



abc*+de*f+



abc*+de*f+



abc*+de*f+g

abc*+de*f+g*+

Method Calls

- Problem: When a call is made to a new method all the local variables and current location need to be saved.
- Method call and method return are same as an open and closed parenthesis.
- Stack is used by the programming language in implementing recursion

Stack Overflow

- There is always the possibility that you will run out of stack space by having too many simultaneously active methods.
- This usually an indication of runway recursion
- Tail recursion refers to recursive call at the last line and it can be avoided by using an while loop.

Tail Recursion

```
1       /**
2        * Print container from itr.
3        */
4       public static <AnyType> void printList( Iterator<AnyType> itr )
5       {
6           if( !itr.hasNext( ) )
7               return;
8
9           System.out.println( itr.next( ) );
10          printList( itr );
11      }
```

Figure 3.35 A bad use of recursion: printing a linked list

While Loop

```
1       /**
2        * Print container from itr.
3        */
4       public static <AnyType> void printList( Iterator<AnyType> itr )
5       {
6           while( true )
7           {
8               if( !itr.hasNext( ) )
9                   return;
10
11              System.out.println( itr.next( ) );
12          }
13      }
```

Figure 3.36 Printing a list without recursion; a compiler might do this

The Queue ADT

- Insertion are done at one end and deletion at the other end. (LILO)
- Basic Operations
  - *enqueue* - inserts an element at the end of the list
  - *dequeue* – deleted the element at the start of the list

The Queue ADT

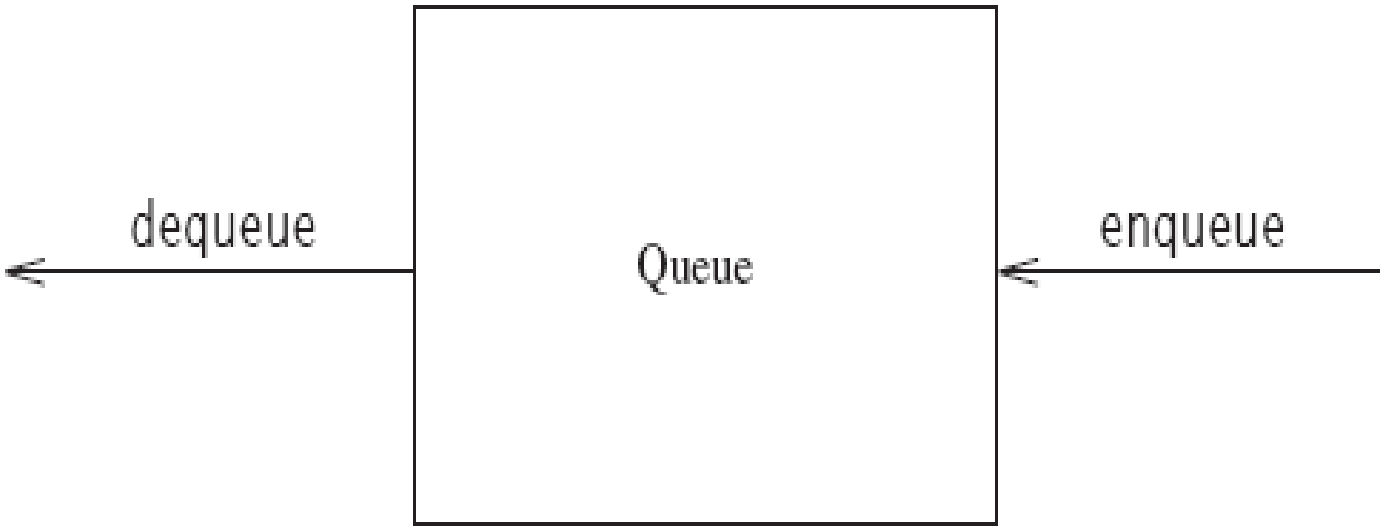
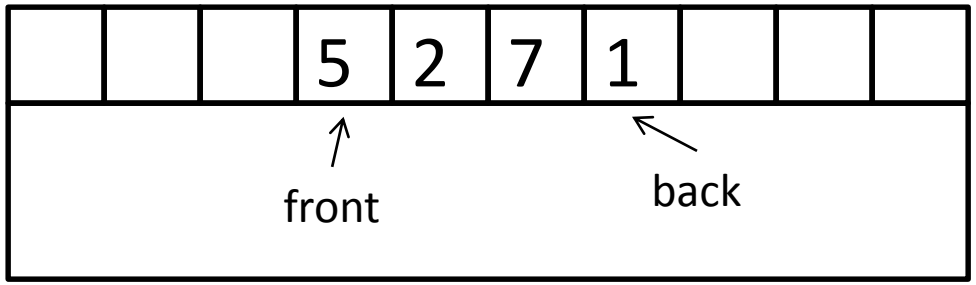


Figure 3.37 Model of a queue

Array Implementation

- O(1) running time for both operations
- For each queue data structure we keep an array *theArray* and positions *front* and *back.*
- *currentSize is used to keep track of number of elements.*

Example

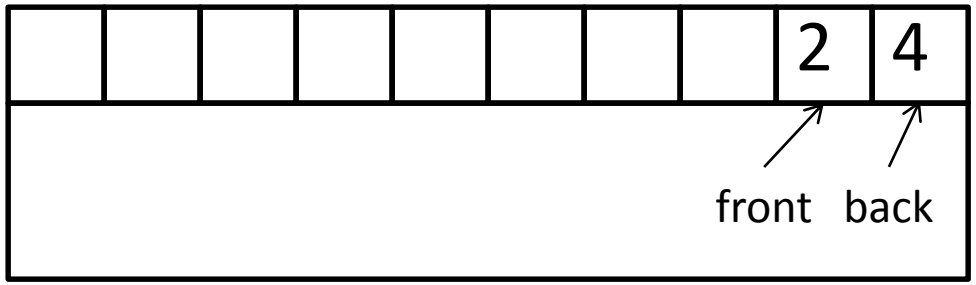


- Shows a queue in some intermediate state.

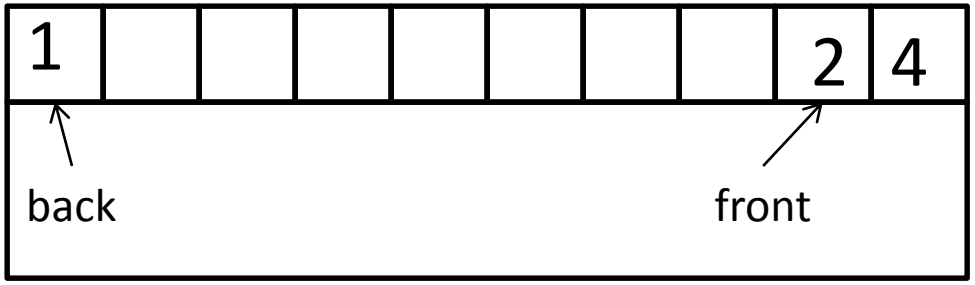
Operations

- enqueue
  - increment currentSize and back
  - set theArray[back] = x
- dequeue
  - return theArray[front]
  - decrement currentSize and then increment front

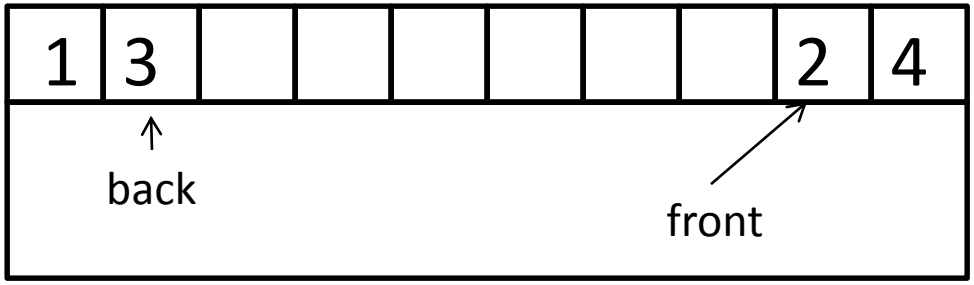
Circular Array Implementation



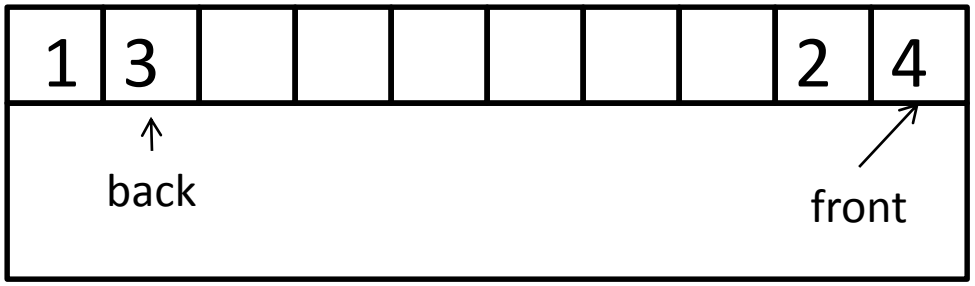
- After enqueue(1)



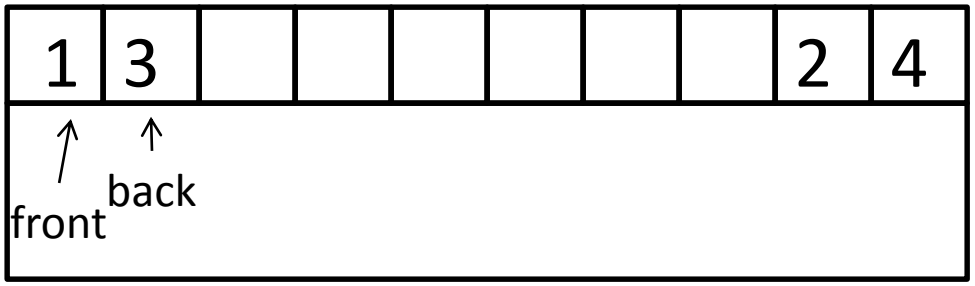
After enqueue(3)



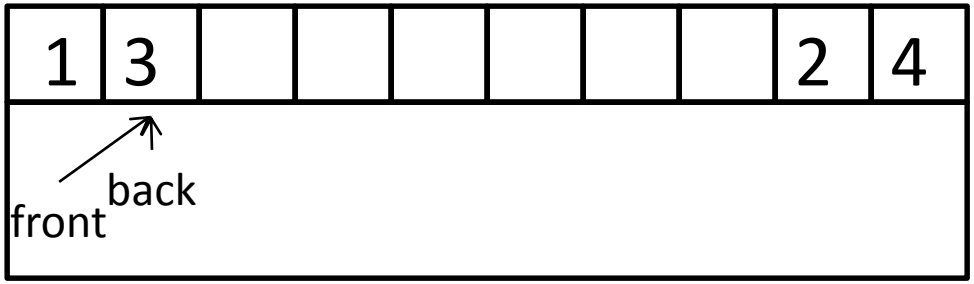
After dequeue which returns 2



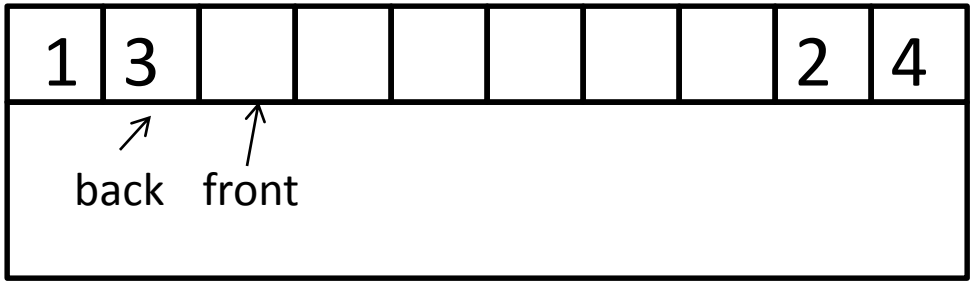
After dequeue which returns 4



After dequeue which returns 1



After dequeue which returns 3 and makes the Queue Empty



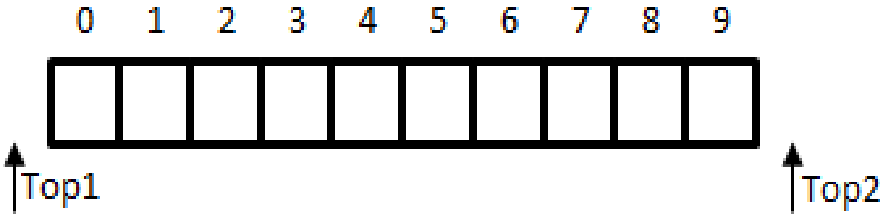
Applications of Queues

- When jobs are submitted to a printer, they are arranged in order of arrival.
- Computer networks
- Calls to large companies
- Wherever resources are limited queues are used

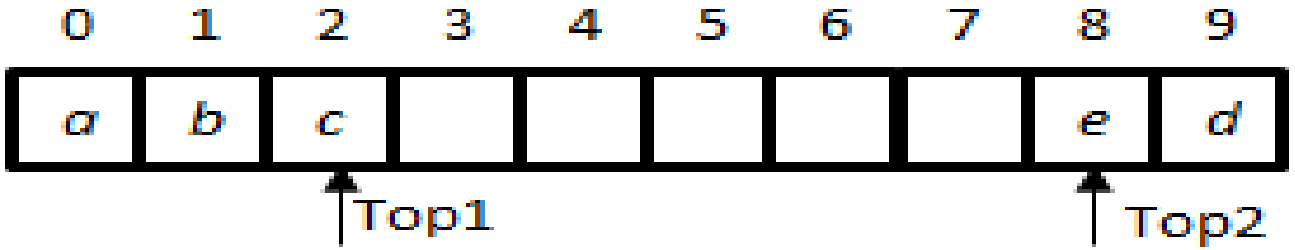
TWO STACKS IN ONE ARRAY

How can you implement two stacks in a single array, where no stack overflows until no space left in the entire array space?

Analysis: An array has two ends, so each of the two stacks may grow from an end in the array. Since two stacks are empty at first, so indexes of top items are initialized as -1 and 10 at first. When items are pushed into first stack, it grows from left to right. Similarly, the second stack grows from right to left when items are items are pushed into it.



No more items can be pushed into stacks when two top items are adjacent to each other, because all space in the array has been occupied.



No more items can be pushed into stacks when two top items are adjacent to each other, because all space in the array has been occupied.

3 Stacks in One Array

- Three stacks can be implemented by having one grow from the bottom up, another from the top down and a third somewhere in the middle growing in some (arbitrary) direction. If the third stack collides with either of the other two, it needs to be moved. A reasonable strategy is to move it so that its center (at the time of the move) is halfway between the tops of the other two stacks.

Implementation of Queue Using Circular Array

```
import java.util.*;

public class SingleQueueArray<AnyType>
{

  SingleQueueArray()
  { this(101); }  // note:  actually holds one less than given size

  SingleQueueArray(int s)
  {
    maxSize = s;
    front = 0;
    rear = 0;
    elements = new ArrayList<AnyType>(maxSize);
  }

  void enqueue(AnyType x)
  {
    if ( !full() )
    {
      if (elements.size() < maxSize)  // add elements until size is reached
        elements.add(x);
      else
        elements.set(rear, x);  // after size is reached, use set

      rear = (rear + 1) % max Size;
    }
  }
```

```
 AnyType dequeue()
  {
    AnyType temp=null;
    if ( !empty() )
    {
      temp = elements.get(front);
      front = (front+1) % maxSize;
    }

    return temp;
  }

  boolean empty()
  { return front == rear; }

  boolean full()
  { return (rear + 1) % maxSize == front; )


  private int front, rear;
  private int maxSize;
  private ArrayList<AnyType> elements;
}
```

Implementation of Queue Using Linked List

```
public class SingleQueue<AnyType>
{
  SingleQueue()
  {
    front = null;
    rear = null;
  }

  void enqueue(AnyType x)
  {
    Node<AnyType> p = new Node<AnyType>(x, null);
    if (rear != null)
       rear = rear.next = p;
    else
       front = rear = p;
  }
AnyType dequeue()
  {
    AnyType temp = front.data;
    if (front.next == null)  // only 1 node
       front = rear = null;
    else
       front = front.next;

    return temp;
  }
```

Implementation of Queue Using Linked List

```
private class Node<AnyType>
  {
    Node()
    { this(null, null); }
    Node(AnyType x)
    { this(x, null); }
    Node(AnyType x, Node p)
    {
      data = x;
      next = p;
    }
    AnyType data;
    Node next;
  }

  private Node<AnyType> front, rear;
}
```

Implementation of Stack Using Linked List

```
public class SingleStack<AnyType>{
    SingleStack()
    {   head = null;  }
    void push(AnyType x)  {
         Node<AnyType> p = new Node<AnyType>(x, head);
         head = p;
    }
    AnyType top()
    {    return head.data;  }
    void pop()
    {    head = head.next;  }

    private class Node<AnyType>  {
         Node()
         {        this(null, null); }
         Node(AnyType x)
         {        this(x, null); }
         Node(AnyType x, Node p)    {
                  data = x;
                  next = p;
         }
         AnyType data;
         Node next;
  }
  private Node<AnyType> head;
}
```

CYCLES IN LINKED LIST

- A linked list contains cycle, if starting from node p, following a sufficient number of next links brings us back to node p.

- Use two iterators *itr1* and itr2, both initially at the start of the list. Advance *itr1* one step at a time, and *itr2* two steps at a time. If *itr2* reaches the end there is no cycle; otherwise, *itr1* and *itr2* will eventually catch up to each other in the middle of the cycle.