22c:31 Algorithms
Ch3: Data Structures

Hantao Zhang
Computer Science Department
http://www.cs.uiowa.edu/~hzhang/c31/

Linear Data Structures

- Now we can now explore some convenient techniques for organizing and managing information

- Topics:
  - collections
  - Abstract Data Types (ADTs)
  - dynamic structures and linked lists
  - queues and stacks

Collections

- A *collection* is an object that serves as a repository for other objects

- A collection usually provides services such as adding, removing, and otherwise managing the elements it contains

- Sometimes the elements in a collection are ordered, sometimes they are not

- Sometimes collections are *homogeneous*, sometimes the are *heterogeneous*

Abstract Data Types

- Collections can be implemented in many different ways
- An *abstract data type* (ADT) is an organized collection of information and a set of operations used to manage that information
- The set of operations defines the *interface* to the ADT
- As long as the ADT fulfills the promises of the interface, it doesn't really matter how the ADT is implemented
- Objects are a perfect programming mechanism to create ADTs because their internal details are *encapsulated*

Abstraction

- Our data structures should be abstractions, so they can be reused to save cost

- That is, they should hide unneeded details

- We want to separate the interface of the structure from its underlying implementation

- This helps manage complexity and makes it possible to change the implementation without changing the interface

Collection Classes

- The Java standard library contains several classes that represent collections, often referred to as the *Java Collections API*
- Their underlying implementation is implied in the class names such as `ArrayList` and `LinkedList`
- `ArrayList` is an implementation of List based on arrays.
- `LinedList` is an implementation of List based on nodes.
- Many abstract data types can be implemented based on `List`, such as, `Set`, `Map`, `Stack`, and `Queue`

Generics Classes

- The Java Collection classes are implemented as *generic types*
- We give the type of object a collection will hold when we create the collection, like this:

```
ArrayList<String> myStringList =
             new ArrayList<String>();

List<Book> myBookList =
             new ArrayList<Book>();
```

➢ **If no type is given, the collection is defined as containing references to the `Object` class**

Example Uses of ArrayList

import java.util.*; // before define a class

```
ArrayList<String> myStringList = new
   ArrayList<String>();
```

```
myStringList.add("Item"); //add an item at the end of list
myStringList.add(new Integer(2));       //compilation error
```

```
int size = myStringList.size();      // return the size of the list
```

```
int index = stringList.indexOf("ABC");
  //location of  object  "ABC" in List
```

```
for (int i = 0; i < myStringList.size(); i++)
   String item = myStringList.get(i);
   System.out.println("Item " + i + " : " + item);}
System.out.println("All Items: " + myStringList);
```

Example Uses of ArrayList (cont.)

```
if (myStringList.contains("ABC") == true)
   System.out.println("the list has Item ABC");
```

```
myStringList.remove(0);  // remove the first item from the list
myStringList.remove("ABC");  // remove  the first copy of item "ABC" from the list
```

```
//  copy list into another list
ArrayList<String> copyOfStringList = new ArrayList<String>();
copyOfStringList.addAll(myStringList);
```

```
 myStringList.set(0,"Item2");  // replace the first item by item "Item2"
```

```
// converting from ArrayList to Array
String[] itemArray = new String[myStringList.size()];
myStringList.toArray(itemArray);
```

Example Uses of LinkedList

```
import java.util.*;
public class LinkedListDemo {
    public static void main(String args[]) {
        // create a linked list
        LinkedList<String> myList = new LinkedList<String>();
        // add elements to the linked list
        myList.add("F"); myList.add("B"); myList.add("D"); myList.add("E"); myList.add("C");
        myList.addLast("Z"); myList.addFirst("A"); myList.add(1, "A2");
        System.out.println("Original contents of myList: " + myList);
        // remove elements from the linked list
        myList.remove("F"); myList.remove(2);
        System.out.println("Contents of myList after deletion: " + myList);
        // remove first and last elements
        myList.removeFirst(); myList.removeLast();
        System.out.println("myList after deleting first and last: " + myList);
    }
}
```

Example Uses of LinkedList

Output of the previous program:

```
Original contents of myList: [A, A2, F, B, D, E, C, Z]
Contents of myList after deletion: [A, A2, D, E, C, Z]
myList after deleting first and last: [A2, D, E, C]
```

Static vs. Dynamic Structures

- A *static* data structure has a fixed size
- This meaning is different from the meaning of the `static` modifier in java
- Arrays are static; once you define the number of elements it can hold, the number doesn’t change
- A *dynamic data structure* grows and shrinks at execution time as required by its contents
- A dynamic data structure is implemented using *links*
- Is *ArrayList* dynamic or not?

Arrays

An array is a structure of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.

Advantages of contiguously-allocated arrays include:

- Constant-time access given the index.
- Arrays consist purely of data, so no space is wasted with links or other formatting information.
- Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

Dynamic Arrays

Unfortunately we cannot adjust the size of simple arrays in the middle of a program's execution.

Compensating by allocating extremely large arrays can waste a lot of space.

With *dynamic arrays* we start with an array of size 1, and double its size from m to $2m$ each time we run out of space.

How many times will we double for n elements? Only $\lceil \log_2 n \rceil$.

How Much Total Work?

The apparent waste in this procedure involves the recopying of the old contents on each expansion.

If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is given by

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

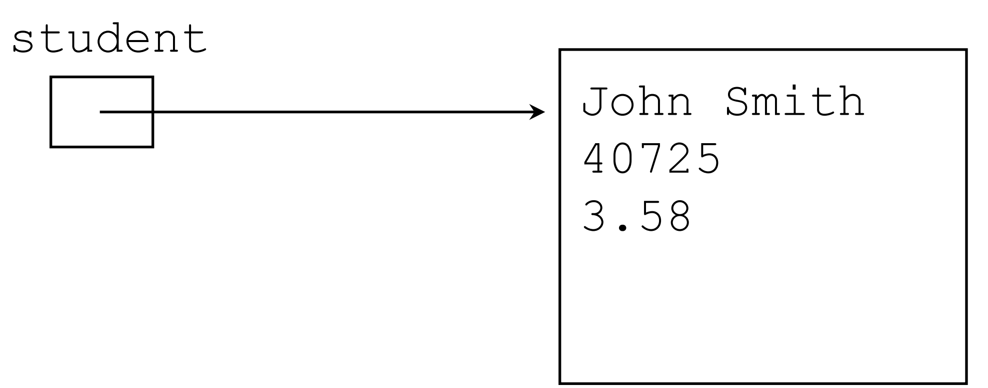
Thus each of the n elements move an average of only twice, and the total work of managing the dynamic array is the same $O(n)$ as a simple array.

Object References

- Recall that an *object reference* is a variable that stores the address of an object

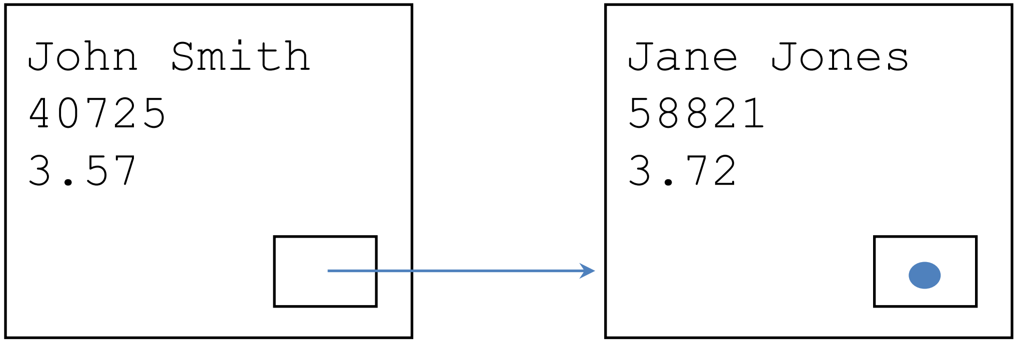
- A reference also can be called a *pointer*

- References often are depicted graphically:



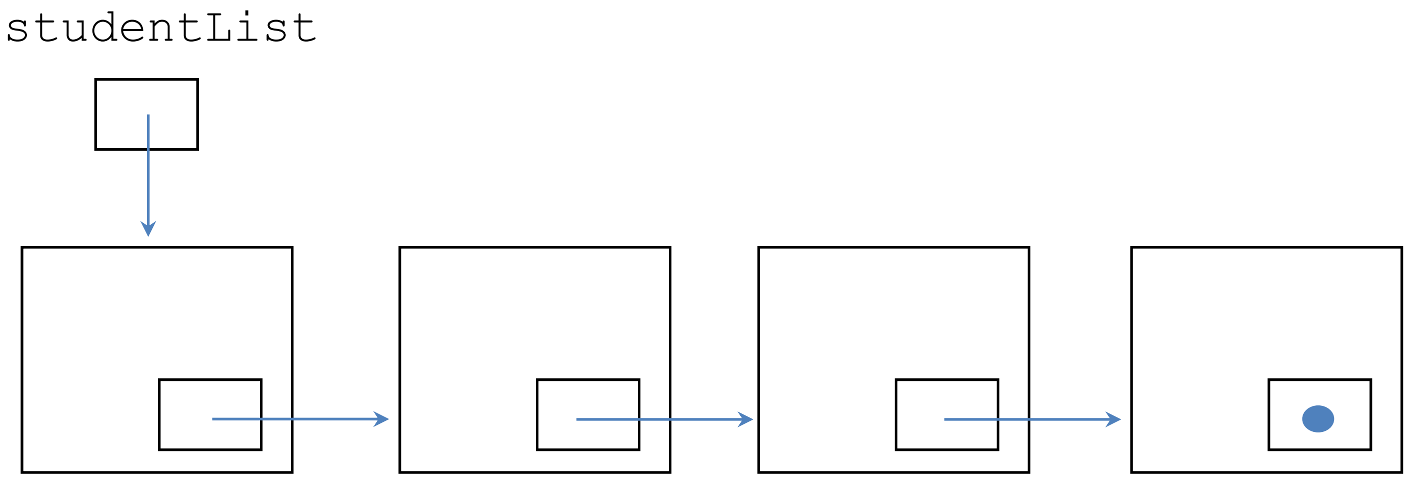
References as Links

- Object references can be used to create *links* between objects
- Suppose a `Student` class contains a reference to another `Student` object



References as Links

- References can be used to create a variety of linked structures, such as a *linked list*:



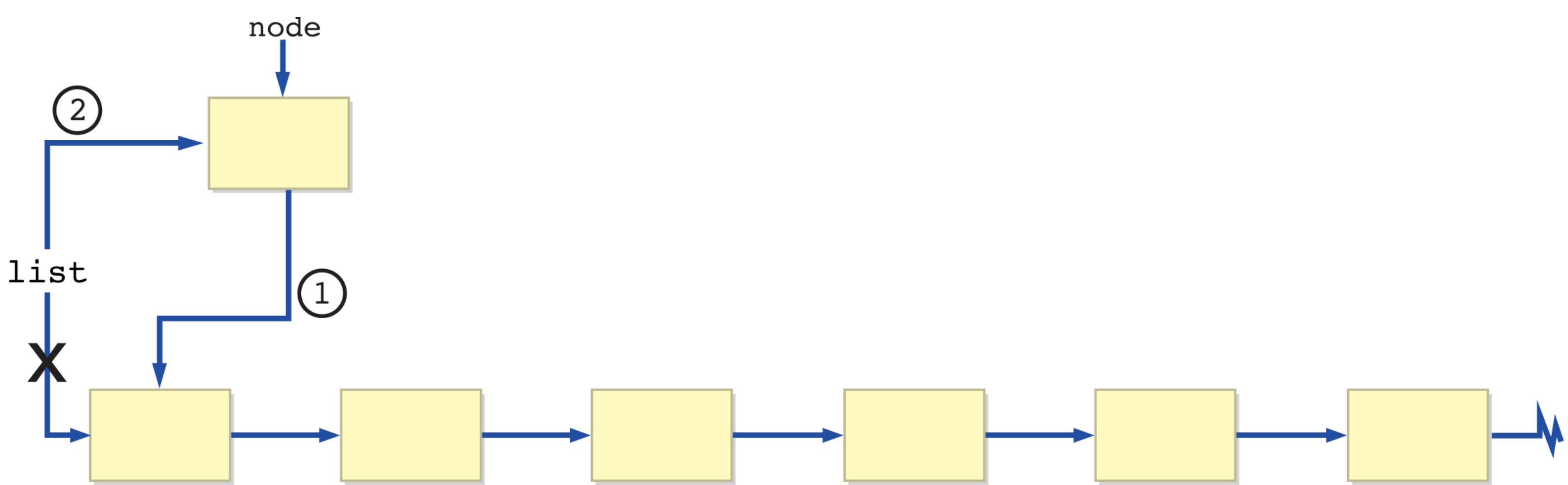
Intermediate Nodes

- The objects being stored should not be concerned with the details of the data structure in which they may be stored
- For example, the `Student` class should not have to store a link to the next `Student` object in the list
- Instead, we can use a separate node class with two parts: 1) a reference to an independent object and 2) a link to the next node in the list
- The internal representation becomes a linked list of nodes

Inserting a Node

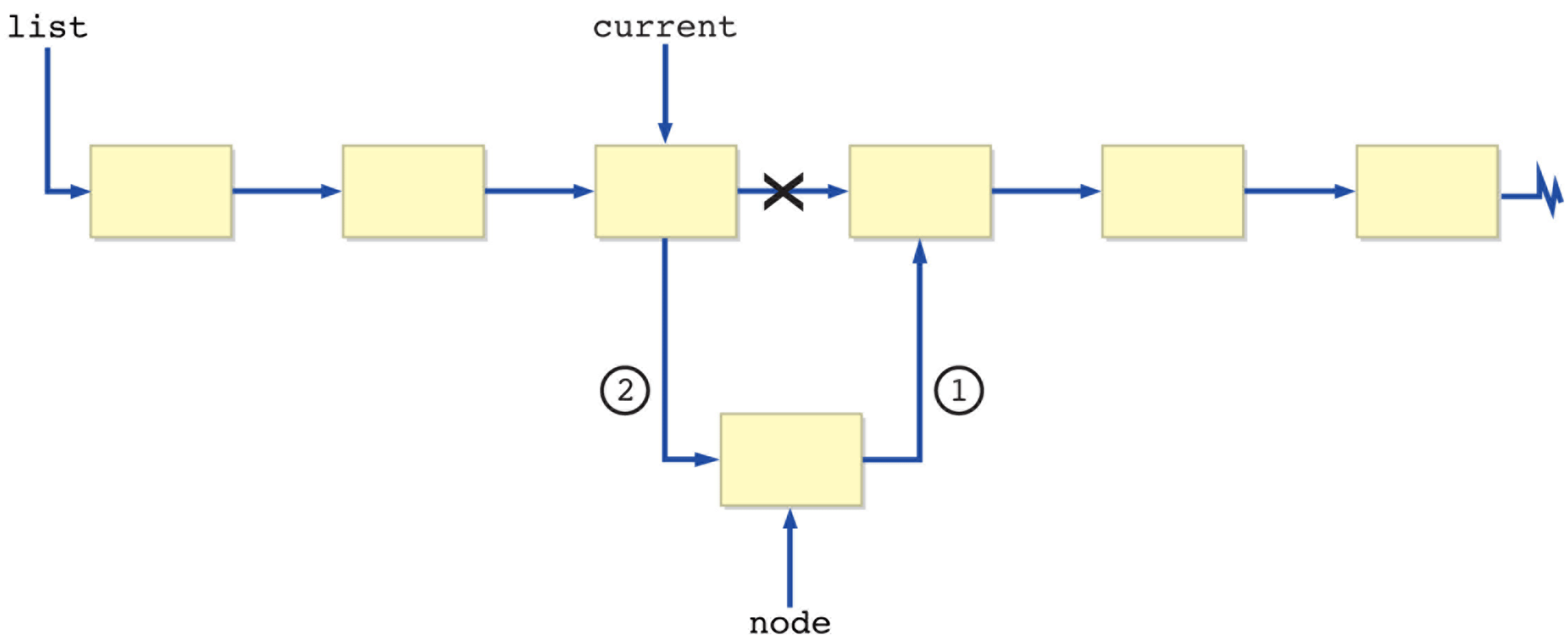
- A method called `insert` could be defined to add a node anywhere in the list, to keep it sorted, for example

- Inserting at the front of a linked list is a special case



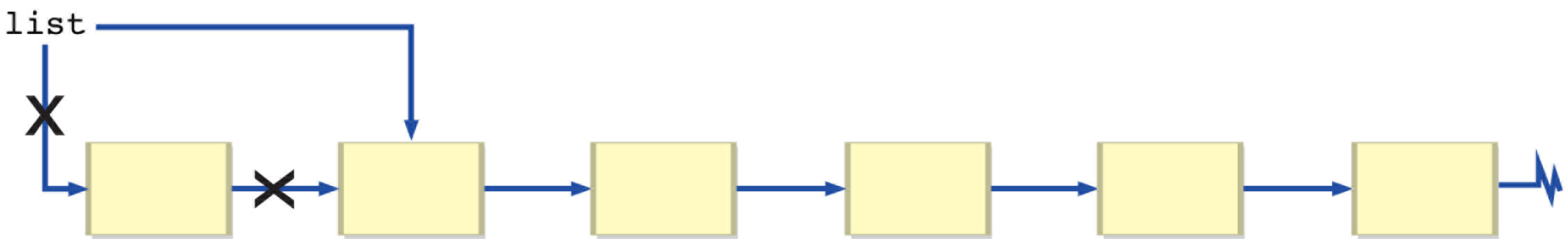
Inserting a Node

- When inserting a node in the middle of a linked list, we must first find the spot to insert it
- Let `current` refer to the node before the spot where the new node will be inserted

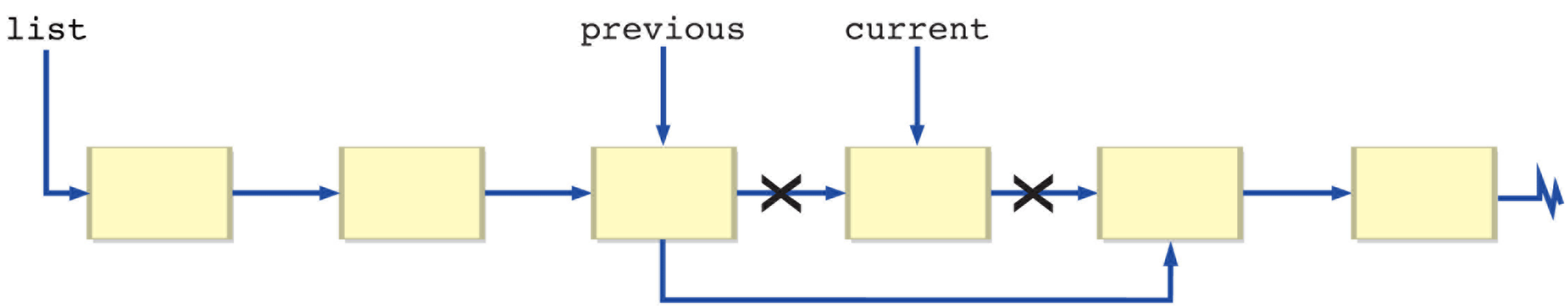


Deleting a Node

- A method called `delete` could be defined to remove a node from the list
- Again the front of the list is a special case:

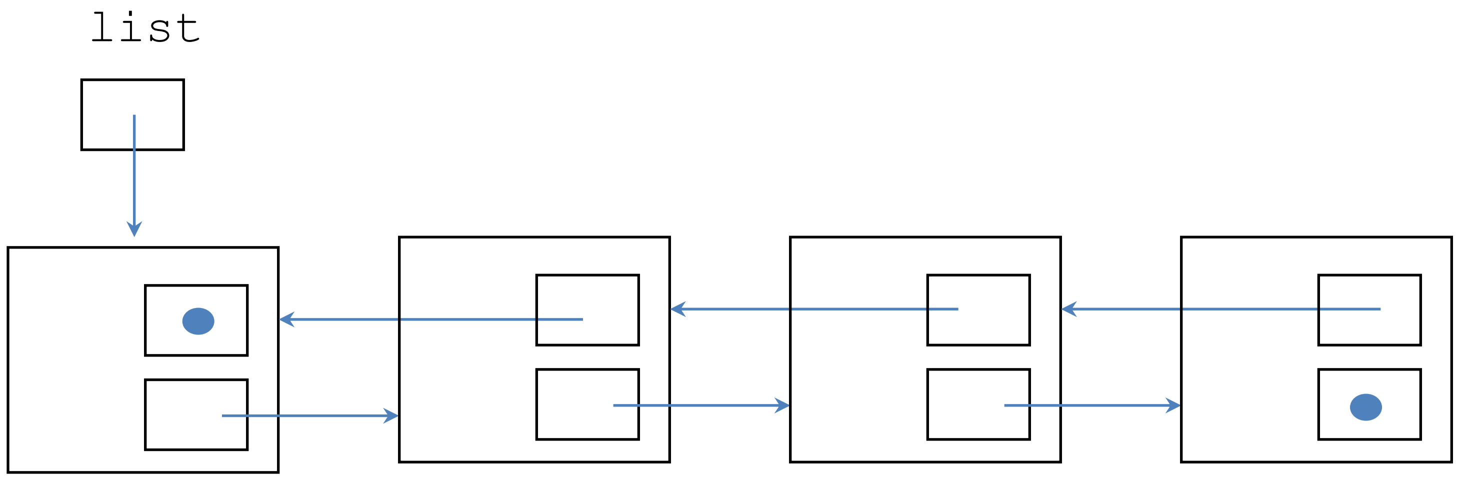


➢ **Deleting from the middle of the list:**



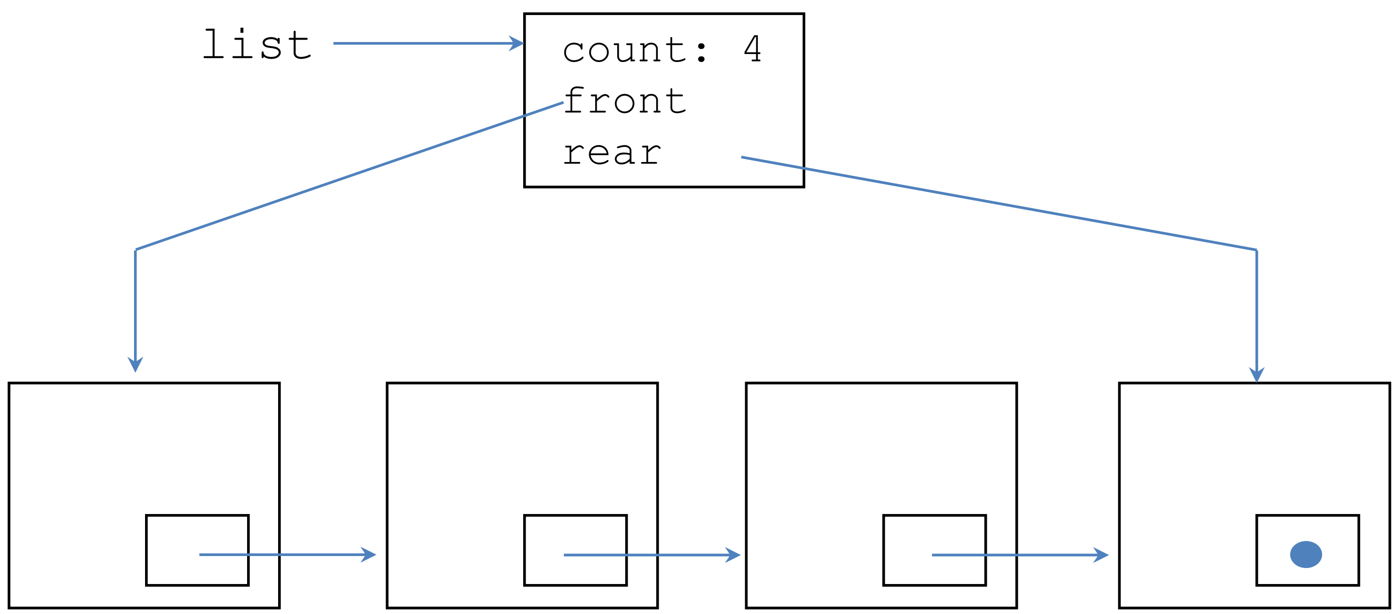
Other Dynamic List Representations

- It may be convenient to implement as list as a *doubly linked list*, with `next` and `previous` references



Other Dynamic List Implementations

- It may be convenient to use a separate *header node*, with a count and references to both the front and rear of the list



Other Dynamic List Implementations

- A linked list can be *circularly linked* in which case the last node in the list points to the first node in the list

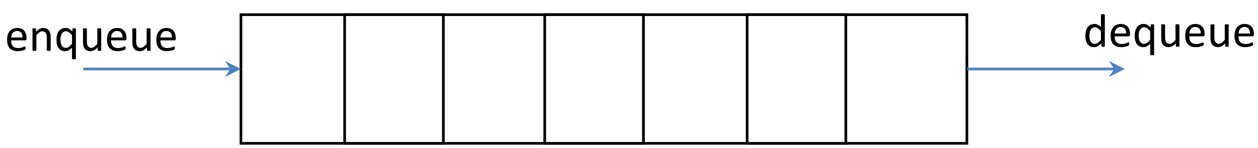
- If the linked list is doubly linked, the first node in the list also points to the last node in the list

- The representation should facilitate the intended operations and should make them easy to implement

Queues

- A *queue* is similar to a list but adds items only to the rear of the list and removes them only from the front

- It is called a FIFO data structure: First-In, First-Out



- Analogy: a line of people at a bank teller's window

Queues

- We can define the operations for a queue
  - enqueue - add an item to the rear of the queue
  - dequeue (or serve) - remove an item from the front of the queue
  - isEmpty - returns true if the queue is empty
- Queues often are helpful in simulations or any situation in which items get “backed up” while awaiting processing
- Java provides a `Queue` interface, which the `LinkedList` class implements:

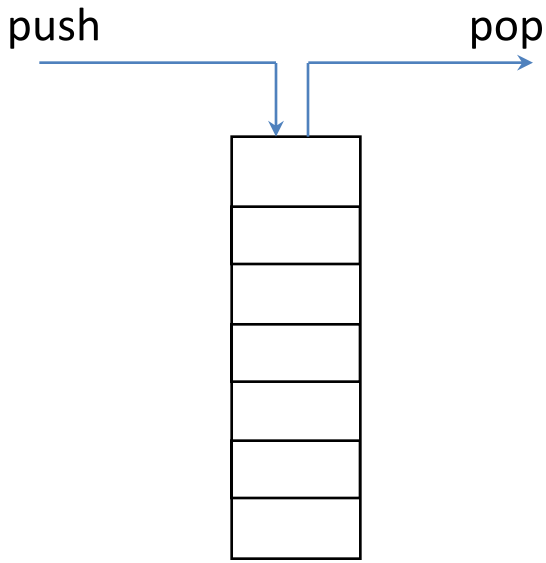
```
Queue<String> q = new
  LinkedList<String>();
```

Stacks

- A *stack* ADT is also linear, like a list or a queue
- Items are added and removed from only one end of a stack
- It is therefore LIFO: Last-In, First-Out
- Analogies: a stack of plates in a cupboard, a stack of bills to be paid, or a stack of hay bales in a barn

Stacks

- Stacks often are drawn vertically:



Stacks

- Some stack operations:
  - push - add an item to the top of the stack
  - pop - remove an item from the top of the stack
  - peek - retrieves the top item without removing it
  - isEmpty - returns true if the stack is empty

- A stack can be represented by a singly-linked list; it doesn't matter whether the references point from the top toward the bottom or vice versa

- A stack can be represented by an array

Stacks

- The `Stack` class is part of the Java Collections API and thus is a generic class

```
Stack<String> strStack = new Stack<String>();
```

Dictonary / Dynamic Set Operations

Perhaps the most important class of data structures maintain a set of items, indexed by keys.

- *Search(S,k)* – A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x]$ = k, or nil if no such element belongs to S.
- *Insert(S,x)* – A modifying operation that augments the set S with the element x.
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.

- *Min(S)*, *Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.
- *Next(S,x)*, *Previous(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.

There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.

Array Based Sets: Unsorted Arrays

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot, $O(1)$
- Delete(S,x) - copy nth item to the xth spot, $O(1)$
- Min(S,x), Max(S,x) - sequential search, $O(n)$
- Successor(S,x), Predecessor(S,x) - sequential search, $O(n)$

Array Based Sets: Sorted Arrays

- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$
- Delete(S,x) - move to fill up the hole, $O(n)$
- Min(S,x), Max(S,x) - first or last element, $O(1)$
- Successor(S,x), Predecessor(S,x) - Add or subtract 1 from pointer, $O(1)$

Problem of the Day

What is the asymptotic worst-case running times for each of the seven fundamental dictionary operations when the data structure is implemented as

- A singly-linked unsorted list,
- A doubly-linked unsorted list,
- A singly-linked sorted list, and finally
- A doubly-linked sorted list.

Solution Blank

	singly unsorted	singly sorted	doubly unsorted	doubly sorted
Search(L, k)				
Insert(L, x)				
Delete(L, x)				
Successor(L, x)				
Predecessor(L, x)				
Minimum(L)				
Maximum(L)				

Solution

Dictionary operation	singly unsorted	double unsorted	singly sorted	doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$