Collections

Collections

- A collection is an object that helps us organize and manage other objects
- Chapter focuses on:
  - the concept of a collection
  - separating the interface from the implementation
  - dynamic data structures
  - linked lists
  - queues and stacks
  - trees and graphs
  - generics

Outline

Collections and Data Structures

Dynamic Representations

Queues and Stacks

Trees and Graphs

The Java Collections API

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*, containing all the same type of objects, and sometimes they are *heterogeneous*

Abstraction

- Collections can be implemented in many different ways
- Our data structures should be *abstractions*
- **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

Abstract Data Types

- 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**
- In one sense, as long as the ADT fulfills the promises of the interface, it doesn't matter how the ADT is implemented
- Objects are a perfect programming mechanism to create ADTs because their internal details are *encapsulated*

Outline

Collections and Data Structures

Dynamic Representations

Queues and Stacks

Trees and Graphs

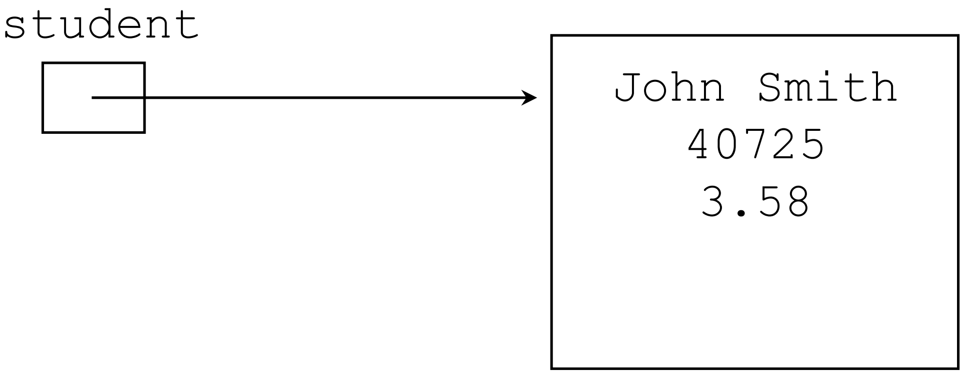
The Java Collections API

Dynamic Structures

- A *static* data structure has a fixed size
- This meaning is different from the meaning of the `static` modifier
- Arrays are static; once you define the number of elements it can hold, the size 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***

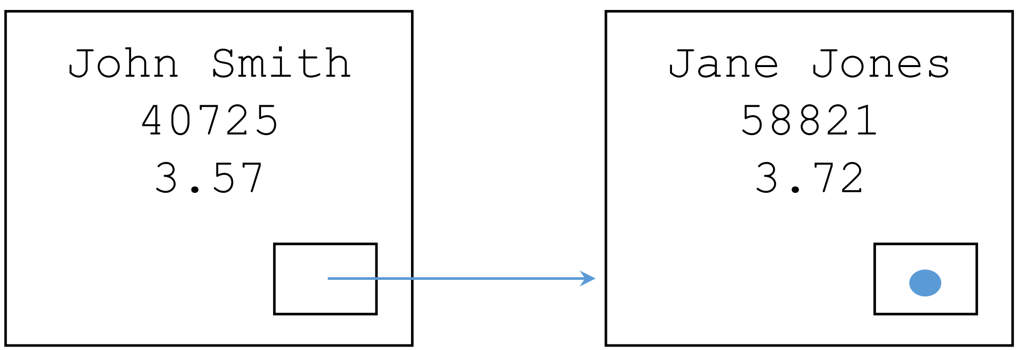
Object References

- 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:



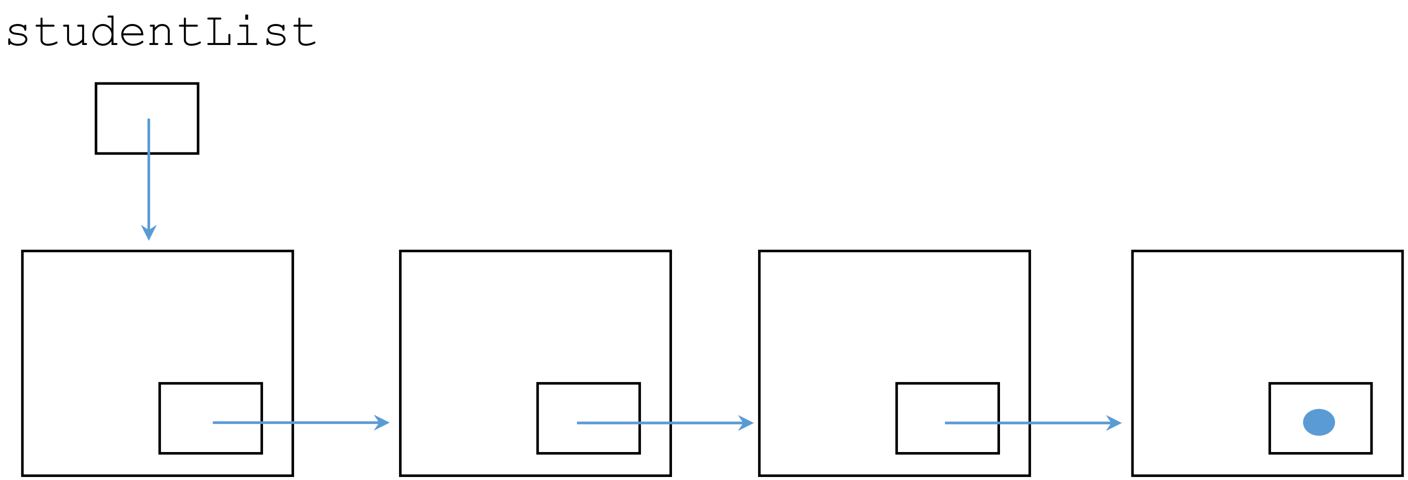
References as Links

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



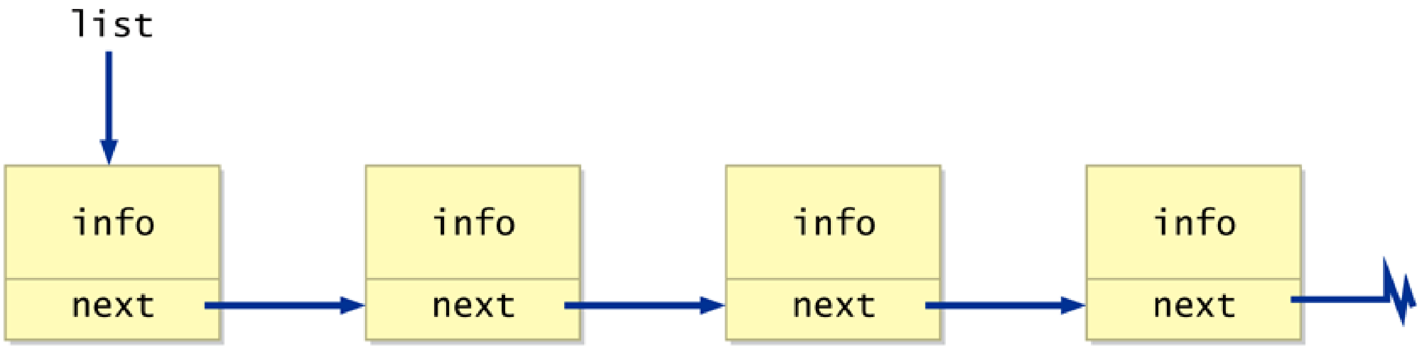
References as Links

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



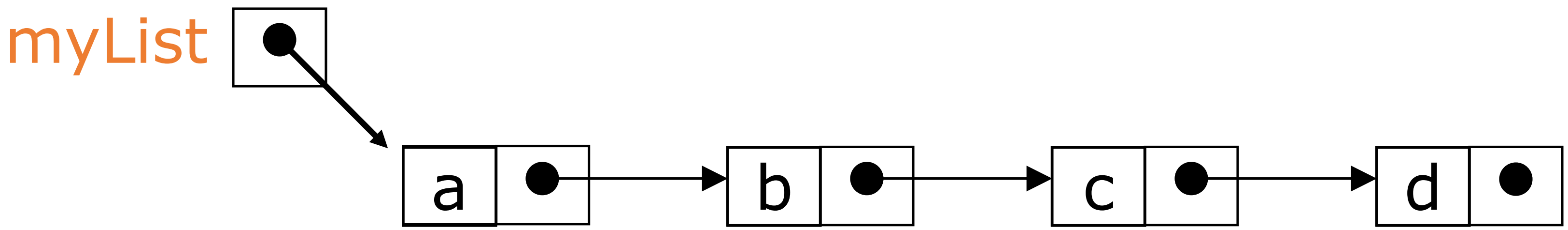
References as Links

```
class Node
{
    int info;
    Node next;
}
```



Anatomy of a linked list

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node

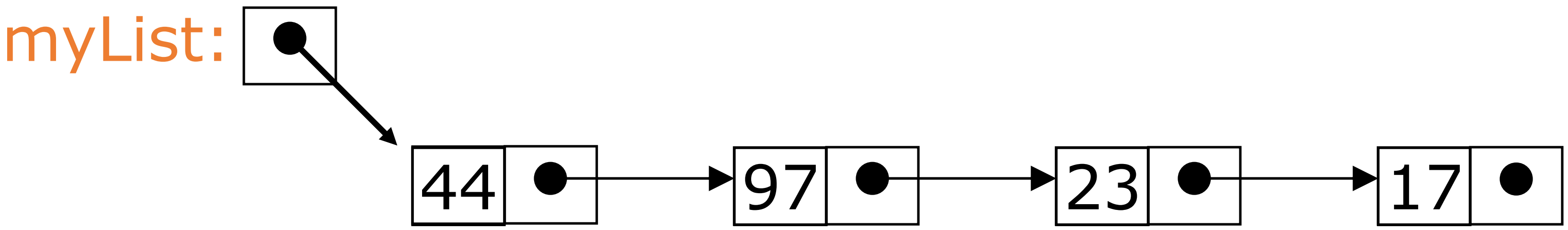
The last node contains a null link

The list may (or may not) have a header

More terminology

- A node's **successor** is the next node in the sequence
  - The last node has no successor
- A node's **predecessor** is the previous node in the sequence
  - The first node has no predecessor
- A list's **length** is the number of elements in it
  - A list may be empty (contain no elements)

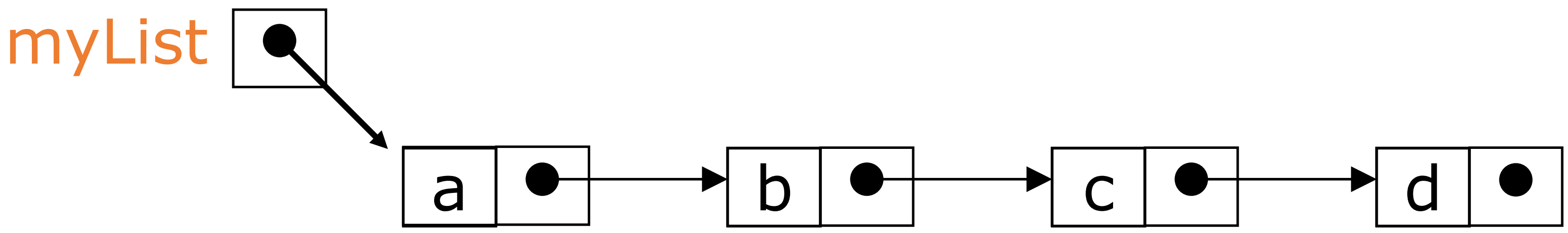
Creating links in Java




```
class Cell {
    int value;
    Cell next;
    Cell (int v, Cell n) { // constructor
        value = v;
        next = n;
    }
}
Cell temp = new Cell(17, null);
temp = new Cell(23, temp);
temp = new Cell(97, temp);
Cell myList = new Cell(44, temp);
```

Singly-linked lists

- Here is a singly-linked list (SLL):



- Each node contains a value and a link to its successor (the last node has no successor)
- *The header points to the first node in the list (or contains the null link if the list is empty)*

Singly-linked lists in Java

```
public class SLL {

    private SLLNode first;

    public SLL() {
        this.first = null;
    }

    // methods...
}
```

- This class actually describes the ***header* of a singly-linked list**
- However, the entire list is accessible from this header
- Users can think of the SLL as *being* the list
    - Users shouldn't have to worry about the actual implementation

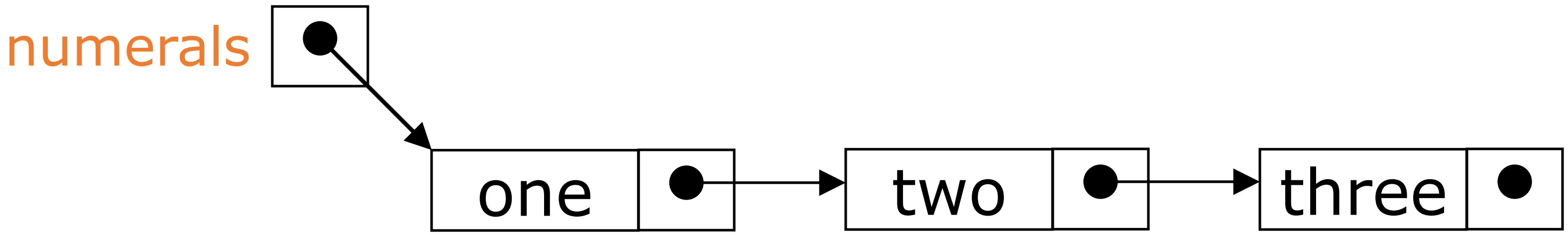
SLL nodes in Java

```
public class SLLNode {
    protected Object element;
    protected SLLNode succ;

    protected SLLNode(Object elem,
                      SLLNode succ) {
        this.element = elem;
        this.succ = succ;
    }
}
```

Creating a simple list

- To create the list ("one", "two", "three"):

```
SLL numerals = new SLL();
numerals.first =
    new SLLNode("one",
        new SLLNode("two",
            new SLLNode("three", null)));
```



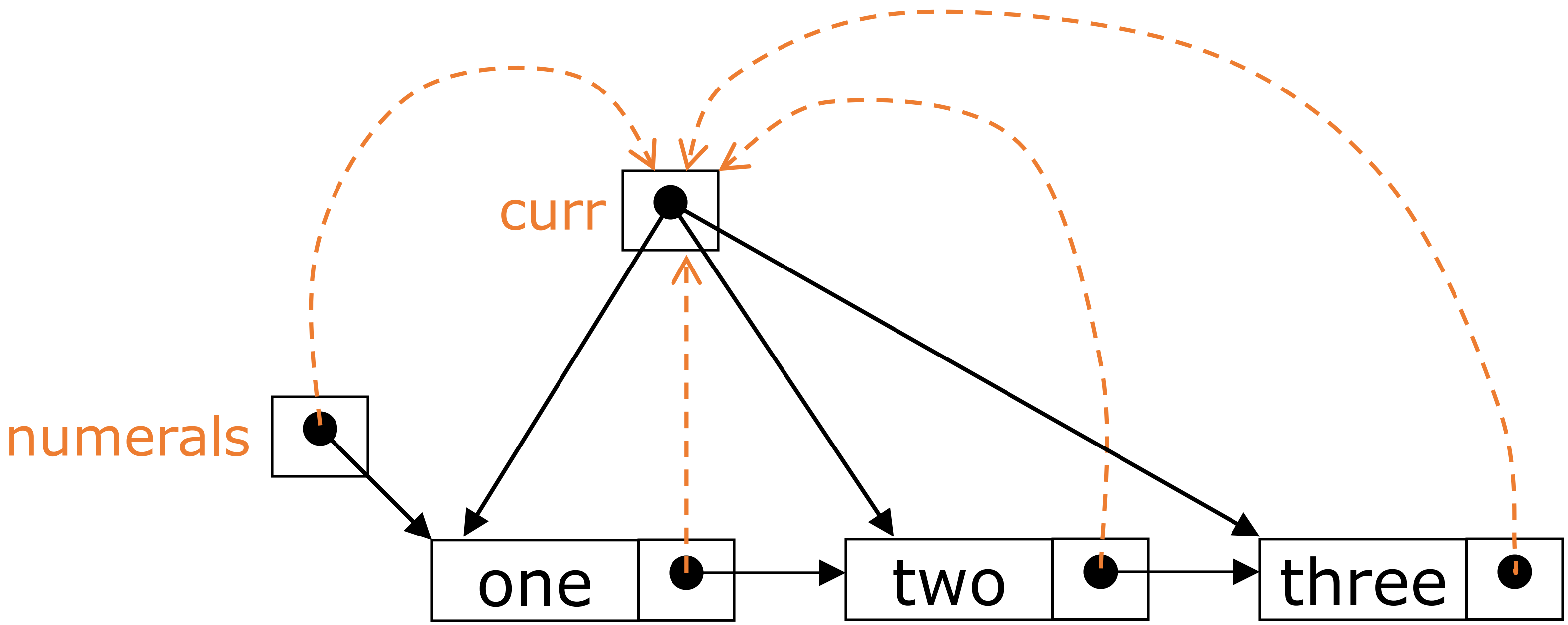
Traversing a SLL

- The following method traverses a list (and prints its elements):

```
public void printFirstToLast() {
   for (SLLNode curr = first;
         curr != null;
         curr = curr.succ) {
      System.out.print(curr.element + "  ");
   }
}
```

- You would write this as an instance method of the SLL class

Traversing a SLL (animation)



Inserting a node into a SLL

- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a *reference*)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

Inserting as a new first element

- This is probably the easiest method to implement
- In class SLL (not SLLNode):

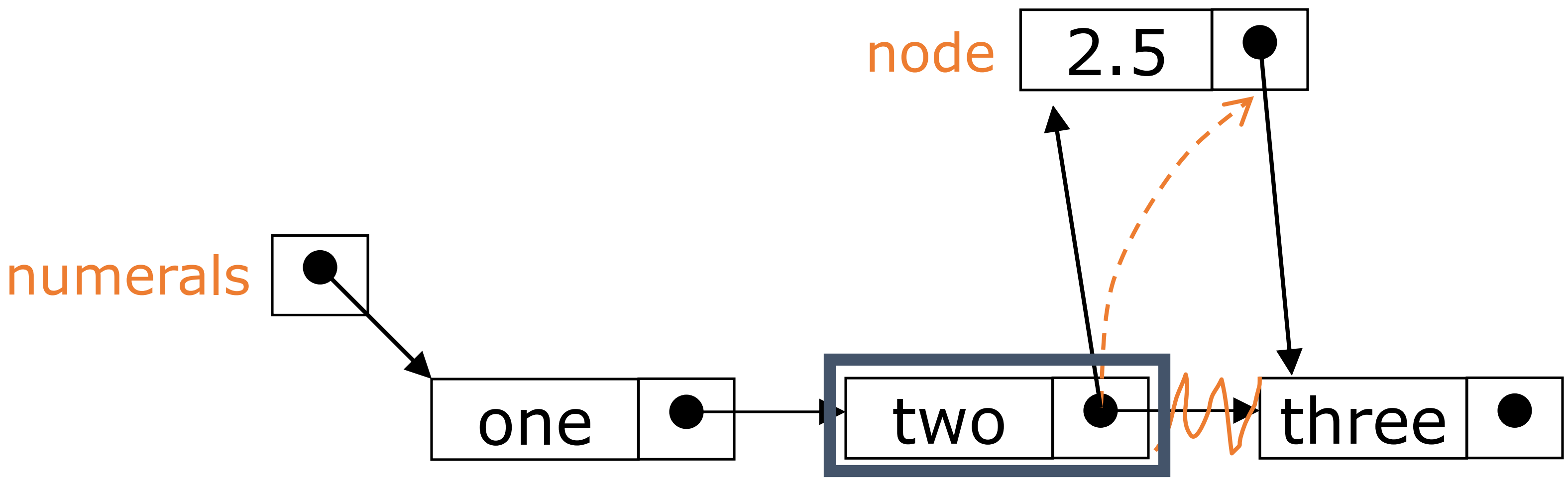
```
void insertAtFront(SLLNode node) {
    node.succ = this.first;
    this.first = node;
}
```

- Notice that this method works correctly when inserting into a previously empty list

Inserting a node after a given value

```
void insertAfter(Object obj, SLLNode node) {
  for (SLLNode here = this.first ; here != null ; here = here.succ) {
      if (here.element.equals(obj)) {
         node.succ = here.succ;
         here.succ = node;
         return;
      } // if
  } // for
  // Couldn't insert--do something reasonable!
}
```

Inserting after (animation)



Find the node you want to insert after

First, copy the link from the node that's already in the list

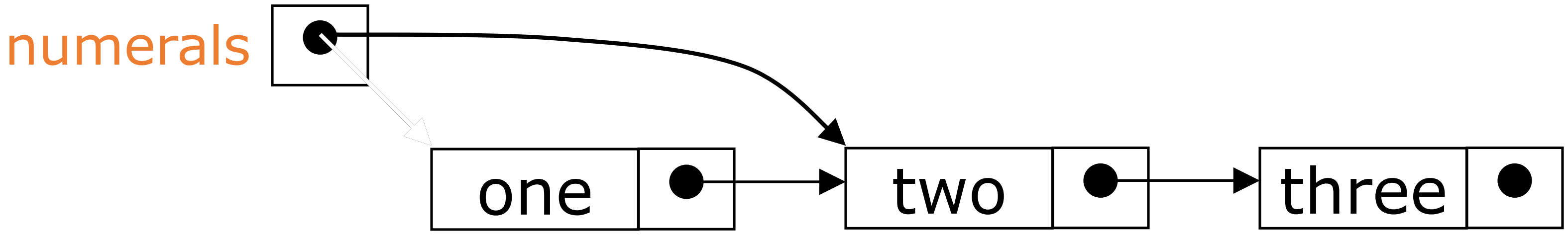
Then, change the link in the node that's already in the list

Deleting a node from a SLL

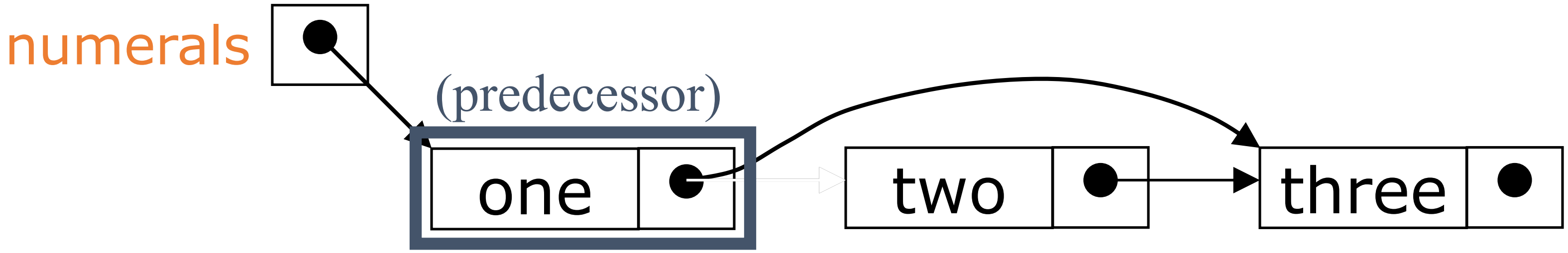
- In order to delete a node from a SLL, you have to **change the link in its *predecessor***
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

- To delete the first element, change the link in the header



- To delete some other element, change the link in its predecessor



- **Deleted nodes will eventually be garbage collected**

Deleting from a SLL

```
public void delete(SLLNode del) {
    SLLNode succ = del.succ;
    // If del is first node, change link in header
    if (del == first) first = succ;
    else { // find predecessor and change its link
        SLLNode pred = first;
        while (pred.succ != del) pred = pred.succ;
        pred.succ = succ;
    }
}
```

Magazine Collection

- Let’s explore an example of a collection of `Magazine` objects, managed by the `MagazineList` class, which has an private inner class called `MagazineNode`
- Because the `MagazineNode` is private to `MagazineList`, the `MagazineList` methods can directly access `MagazineNode` data without violating encapsulation
- See `MagazineRack.java`
- See `MagazineList.java`
- See `Magazine.java`

```java
//********************************************************************
//  MagazineRack.java       Author: Lewis/Loftus
//
//  Driver to exercise the MagazineList collection.
//********************************************************************

public class MagazineRack
{
   //----------------------------------------------------------------
   //  Creates a MagazineList object, adds several magazines to the
   //  list, then prints it.
   //----------------------------------------------------------------
   public static void main (String[] args)
   {
      MagazineList rack = new MagazineList();

      rack.add (new Magazine("Time"));
      rack.add (new Magazine("Woodworking Today"));
      rack.add (new Magazine("Communications of the ACM"));
      rack.add (new Magazine("House and Garden"));
      rack.add (new Magazine("GQ"));

      System.out.println (rack);
   }
}
```

```java
//*****************                                    *****************
//  MagazineRack.
//
//  Driver to exe
//*****************                                    *****************

public class Maga
{
   //-----------------------------------------------------------------
   //  Creates a MagazineList object, adds several magazines to the
   //  list, then prints it.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      MagazineList rack = new MagazineList();

      rack.add (new Magazine("Time"));
      rack.add (new Magazine("Woodworking Today"));
      rack.add (new Magazine("Communications of the ACM"));
      rack.add (new Magazine("House and Garden"));
      rack.add (new Magazine("GQ"));

      System.out.println (rack);
   }
}
```

Output

```
Time
Woodworking Today
Communications of the ACM
House and Garden
GQ
```

```java
//********************************************************************
//  MagazineList.java       Author: Lewis/Loftus
//
//  Represents a collection of magazines.
//********************************************************************

public class MagazineList
{
   private MagazineNode list;

   //-----------------------------------------------------------------
   //  Sets up an initially empty list of magazines.
   //-----------------------------------------------------------------
   public MagazineList()
   {
      list = null;
   }
```

continue

continue

```java
   //-----------------------------------------------------------------
   //  Creates a new MagazineNode object and adds it to the end of
   //  the linked list.
   //-----------------------------------------------------------------
   public void add (Magazine mag)
   {
      MagazineNode node = new MagazineNode (mag);
      MagazineNode current;

      if (list == null)
         list = node;
      else
      {
         current = list;
         while (current.next != null)
            current = current.next;
         current.next = node;
      }
   }
```

continue

continue

```java
   //-----------------------------------------------------------------
   //  Returns this list of magazines as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      String result = "";

      MagazineNode current = list;

      while (current != null)
      {
         result += current.magazine + "\n";
         current = current.next;
      }

      return result;
   }
```

continue

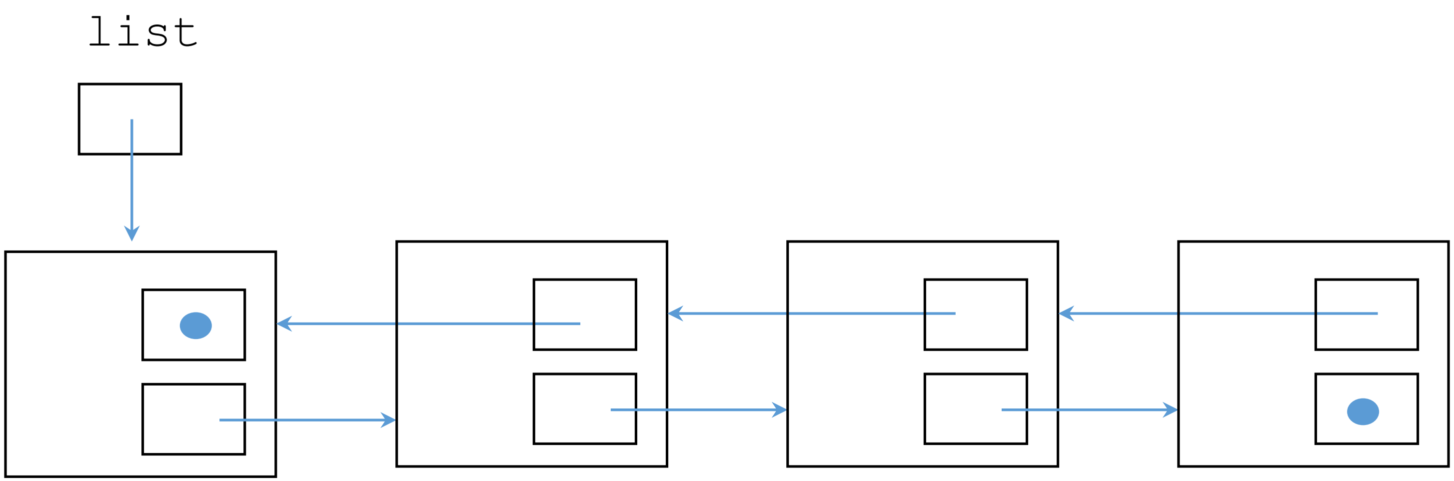
continue

```
   //*****************************************************************
   //  An inner class that represents a node in the magazine list.
   //  The public variables are accessed by the MagazineList class.
   //*****************************************************************
   private class MagazineNode
   {
      public Magazine magazine;
      public MagazineNode next;

      //--------------------------------------------------------------
      //  Sets up the node
      //--------------------------------------------------------------
      public MagazineNode (Magazine mag)
      {
         magazine = mag;
         next = null;
      }
   }
}
```

```java
//********************************************************************
//  Magazine.java       Author: Lewis/Loftus
//
//  Represents a single magazine.
//********************************************************************

public class Magazine
{
   private String title;

   //-----------------------------------------------------------------
   //  Sets up the new magazine with its title.
   //-----------------------------------------------------------------
   public Magazine (String newTitle)
   {
      title = newTitle;
   }

   //-----------------------------------------------------------------
   //  Returns this magazine as a string.
   //-----------------------------------------------------------------
   public String toString ()
   {
      return title;
   }
}
```

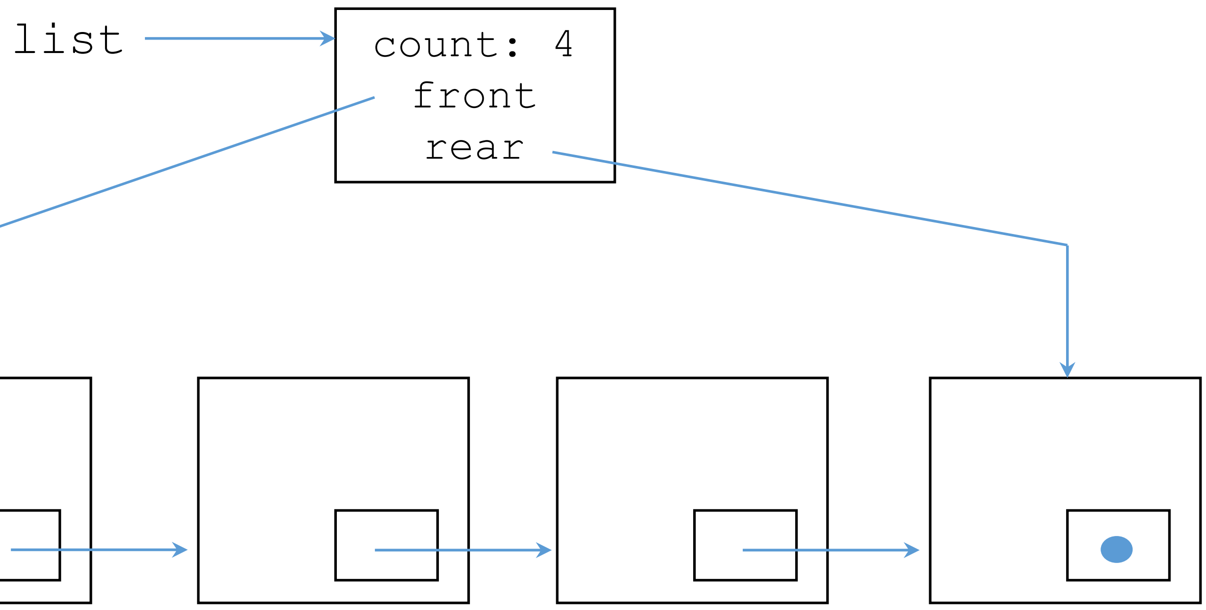
Other Dynamic Representations

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



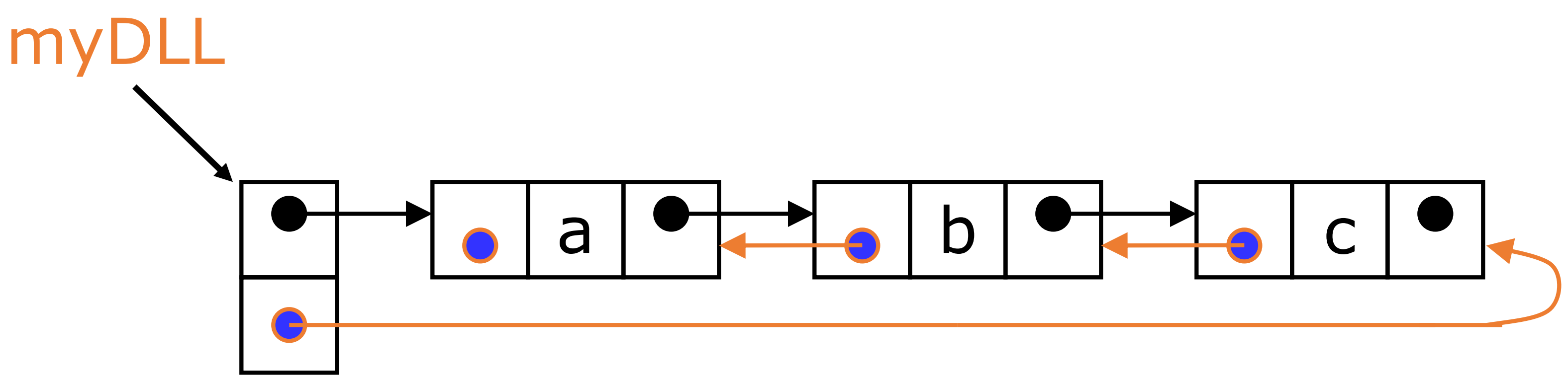
Other Dynamic Representations

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



Doubly-linked lists

- Here is a doubly-linked list (DLL):



- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Constructing SLLs and DLLs

```java
public class SLL {

    private SLLNode first;

    public SLL() {
        this.first = null;
    }

    // methods...
}
```

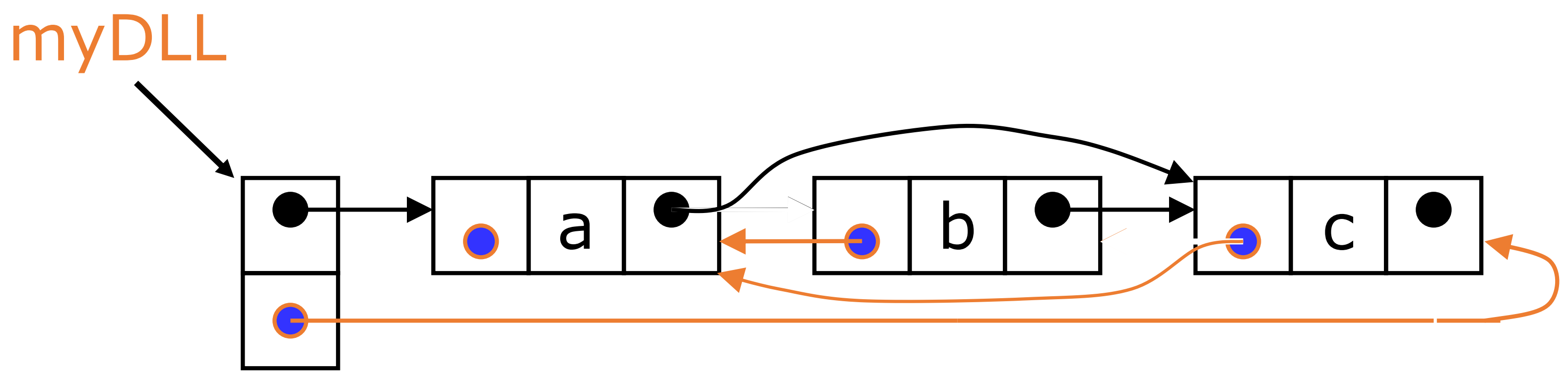
```java
public class DLL {

    private DLLNode first;
    private DLLNode last;

    public DLL() {
        this.first = null;
        this.last = null;
    }

    // methods...
}
```

DLL nodes in Java

```
public class DLLNode {
    protected Object element;
    protected DLLNode pred, succ;

    protected DLLNode(Object elem,
                      DLLNode pred,
                      DLLNode succ) {
        this.element = elem;
        this.pred = pred;
        this.succ = succ;
    }
}
```

Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



- We don’t have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

Other operations on linked lists

- Most "algorithms" on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- **Sorting a linked list is just messy, since you can't directly access the n^{th} element—you have to count your way through a lot of other elements**

Circular Linked Lists

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*

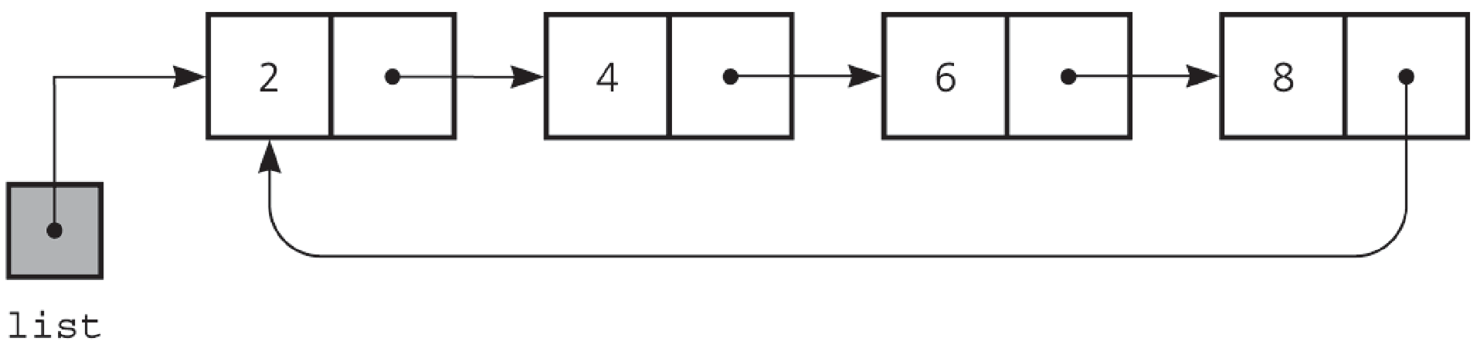


Figure 4.25 A circular linked list

Outline

Collections and Data Structures

Dynamic Representations

Queues and Stacks

Trees and Graphs

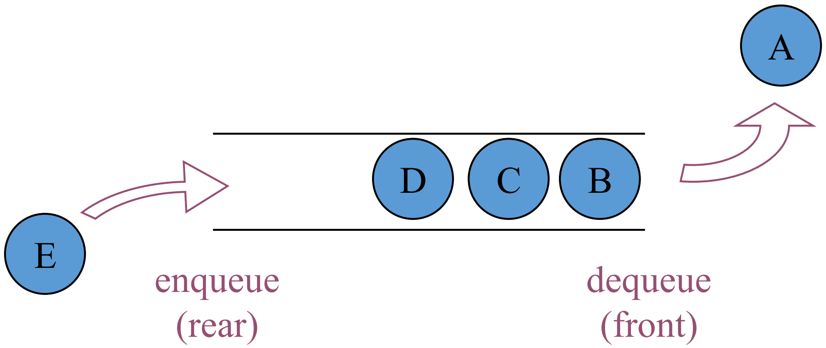
The Java Collections API

Classic Data Structures

- Now we'll examine some classic data structures
- Classic *linear data structures* include *queues* and *stacks*
- Classic *nonlinear data structures* include *trees* and *graphs*

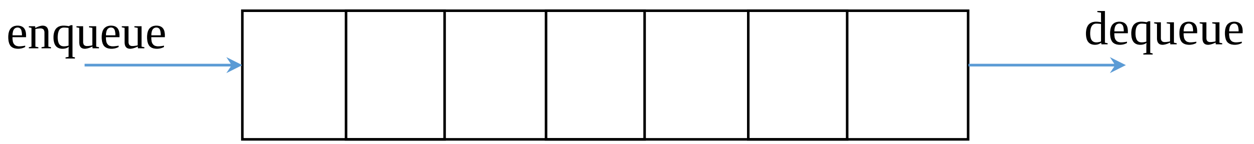
Queues

- Abstract – FIFO (First In, First out)
- Methods
    - enqueue, dequeue & isEmpty
    - isFull & constructor
- Uses
    - simulations
    - in event handling
- Implementation
    - Arrays & linked lists



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
  - empty - returns true if the queue is empty
- As with our linked list example, by storing generic `Object` references, any object can be stored in the queue
- Queues often are helpful in simulations or any situation in which items get “backed up” while awaiting processing

Queues

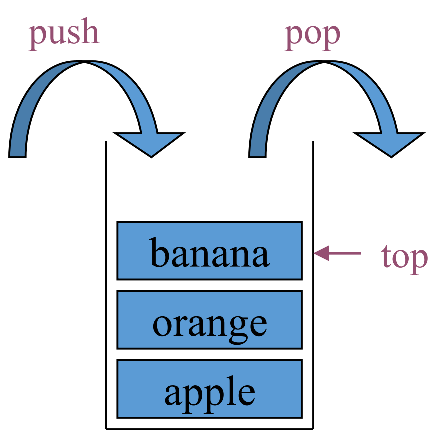
- A queue can be represented by a singly-linked list; it is most efficient if the references point **from the front toward the rear of the queue**
- A queue can be represented by an array, using the remainder operator (%) to “wrap around” when the end of the array is reached and space is available at the front of the array

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

- Abstract - LIFO (Last In, First Out)
- Methods
  - push, pop & isEmpty
  - isFull & constructor
- Uses
  - in method calling,
    in interrupt handling,
    calculator *(postfix expressions!)*
- Implementation
  - Java Stack class
  - arrays & linked-lists



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 (or top) - retrieves the top item without removing it
  - empty - 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, but the new item should be placed in the next available place in the array rather than at the end

Stacks

- The `java.util` package contains a `Stack` class
- Like `ArrayList` operations, the `Stack` operations operate on `Object` references
- See [`Decode.java`]

```java
//********************************************************************
//  Decode.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the Stack class.
//********************************************************************

import java.util.*;

public class Decode
{
   //-----------------------------------------------------------------
   //   Decodes a message by reversing each word in a string.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Scanner scan = new Scanner (System.in);

      Stack word = new Stack();

      String message;
      int index = 0;

      System.out.println ("Enter the coded message:");
      message = scan.nextLine();
      System.out.println ("The decoded message is:");
```

continue

continue

```
      while (index < message.length())
      {
         // Push word onto stack
         while (index < message.length() && message.charAt(index) != ' ')
         {
            word.push (new Character(message.charAt(index)));
            index++;
         }

         // Print word in reverse
         while (!word.empty())
            System.out.print (((Character)word.pop()).charValue());
         System.out.print (" ");
         index++;
      }

      System.out.println();
   }
}
```

continue

```
      while (index
      {
         // Push w
         while (in                                                    harAt(index) != ' ')
         {
            word.push (new Character(message.charAt(index)));
            index++;
         }

         // Print word in reverse
         while (!word.empty())
            System.out.print (((Character)word.pop()).charValue());
         System.out.print (" ");
         index++;
      }

      System.out.println();
   }
}
```

Sample Run

```
Enter the coded message:
artxE eseehc esaelp
The decoded message is:
Extra cheese please
```

How to implement a queue using two stacks

- Let queue to be implemented be q and stacks used to implement q be stack1 and stack2
- Implement the enQueue and dnQueue operations

Method 1 (costly enQueue operation)

Makes sure that oldest entered element is always at the top of stack 1

deQueue operation just pops from stack1

To put the element at top of stack1, stack2 is used.

enQueue(q, x)

1) While stack1 is not empty, push everything from stack1 to stack2.
2) Push x to stack1 (assuming size of stacks is unlimited).
3) Push everything back to stack1.

dnQueue(q)

1) If stack1 is empty then error
2) Pop an item from stack1 and return it

How to implement a queue using two stacks

Method 2 (By making deQueue operation costly)

In enqueue operation, the new element is entered at the top of stack1

In dequeue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned

enQueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

deQueue(q)

1) If both stacks are empty then error.

2) If stack2 is empty

   While stack1 is not empty, push everything from satck1 to stack2.

3) Pop the element from stack2 and return it.

Method 1 moves all the elements twice in enQueue operation

Method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty

How to implement a stack using two queues

- Let stack to be implemented be ‘s’ and queues used to implement be ‘q1’ and ‘q2’
- Implement the push and pop operations

Method 1 (By making push operation costly)

Makes sure that newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’

‘q2’ is used to put every new element at front of ‘q1’.

push(s, x) // x is the element to be pushed and s is stack

1) Enqueue x to q2
2) One by one dequeue everything from q1 and enqueue to q2.
3) Swap the names of q1 and q2

// Swapping of names is done to avoid one more movement of all elements

// from q2 to q1.

pop(s)

1) Dequeue an item from q1 and return it.

How to implement a stack using two queues

Method 2 (By making pop operation costly)

In push operation, the new element is always enqueued to q1

In pop() operation, if q2 is empty then all the elements except the last, are moved to q2

Finally the last element is dequeued from q1 and returned.

push(s, x)

1) Enqueue x to q1 (assuming size of q1 is unlimited).

pop(s)

1) One by one dequeue everything except the last element from q1 and enqueue to q2.

2) Dequeue the last item of q1, the dequeued item is result, store it.

3) Swap the names of q1 and q2

4) Return the item stored in step 2.

// Swapping of names is done to avoid one more movement of all elements
// from q2 to q1.

Outline

Collections and Data Structures

Dynamic Representations

Queues and Stacks

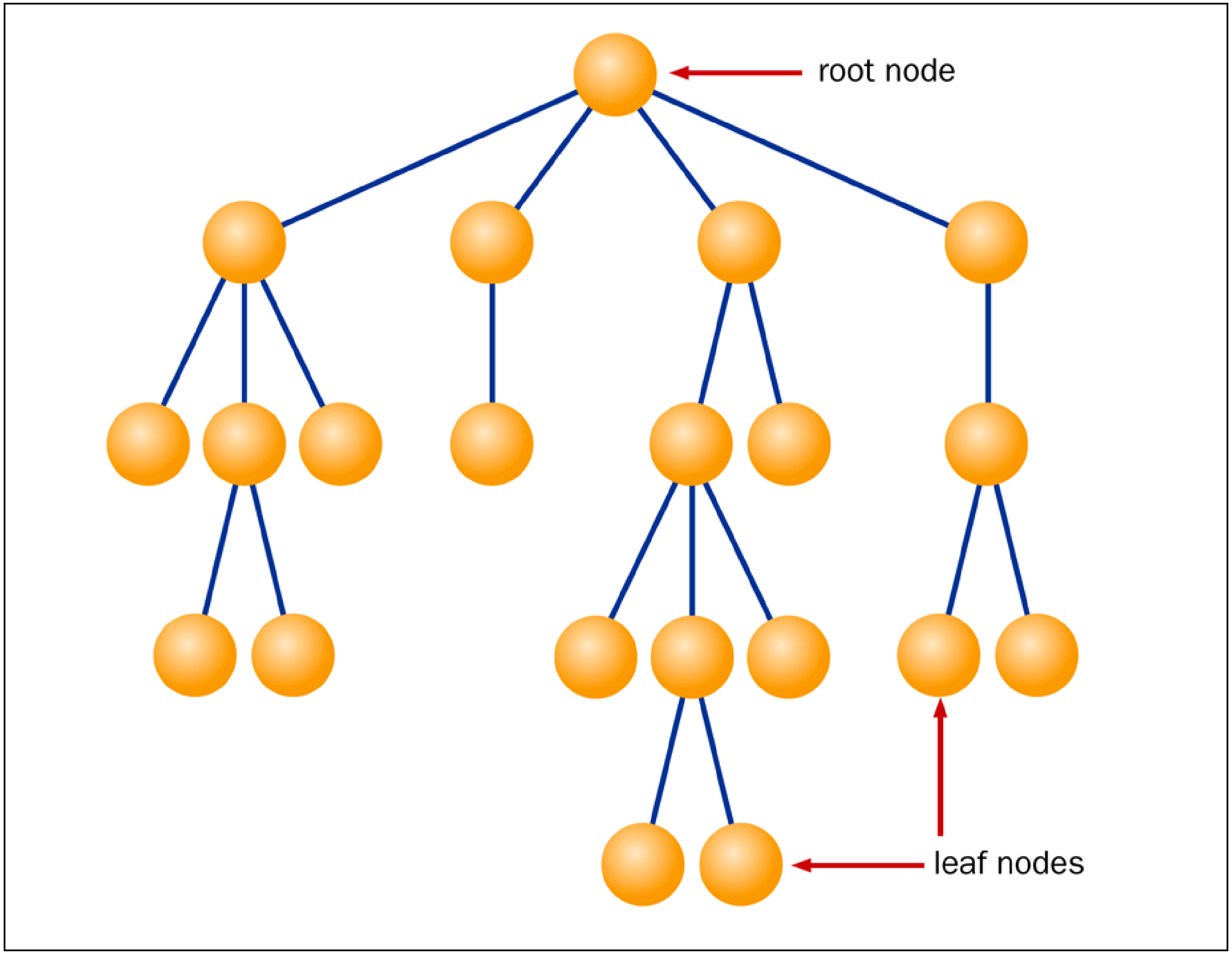
Trees and Graphs

The Java Collections API

Trees

- A *tree* is a non-linear data structure that consists of a *root node* and potentially many levels of additional nodes that form a hierarchy
- Nodes that have no children are called *leaf nodes*
- Nodes except for the root and leaf nodes are called *internal nodes*
- In a general tree, each node can have many child nodes

A General Tree

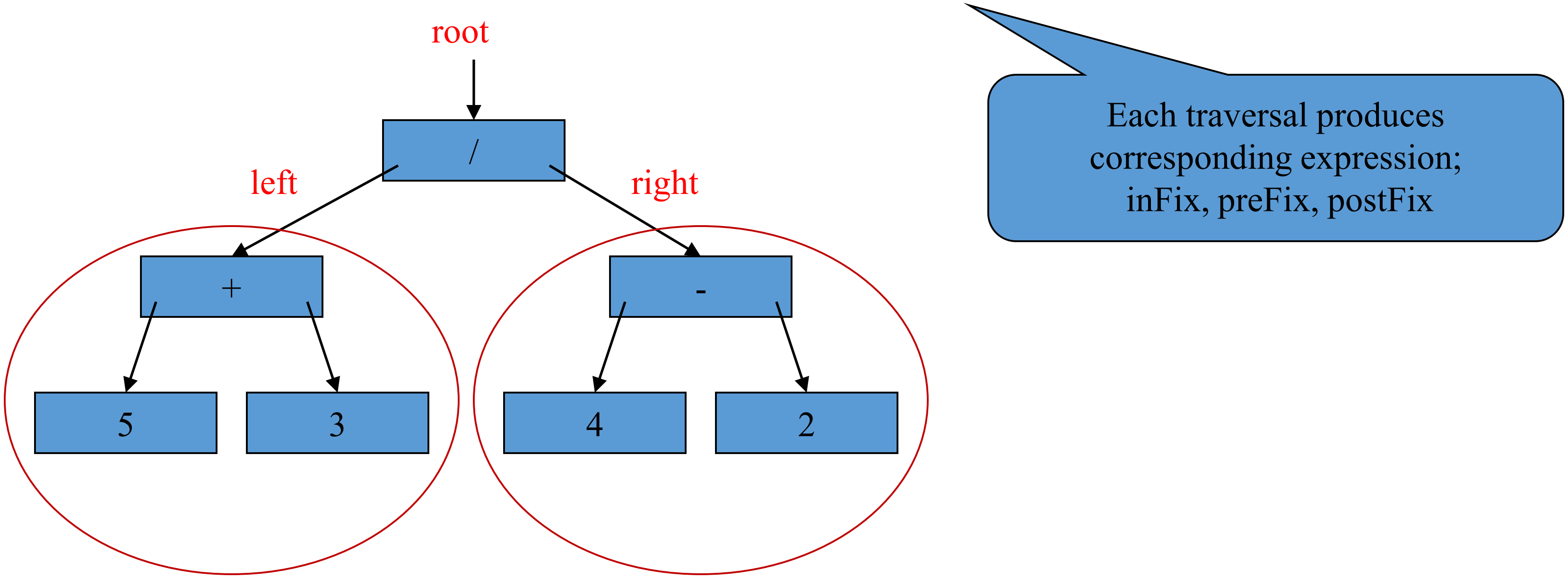


Binary Trees

- In a *binary tree*, each node can have **no more than two child nodes**
- A binary tree **can be defined recursively**
  - Either it is empty (the base case) or it consists of a *root* and two *subtrees*, each of which is a binary tree
- Trees are typically are represented using references as dynamic links, though it is possible to use fixed representations like arrays
- For binary trees, this requires storing only **two links per node** to the left and right child

Binary Trees

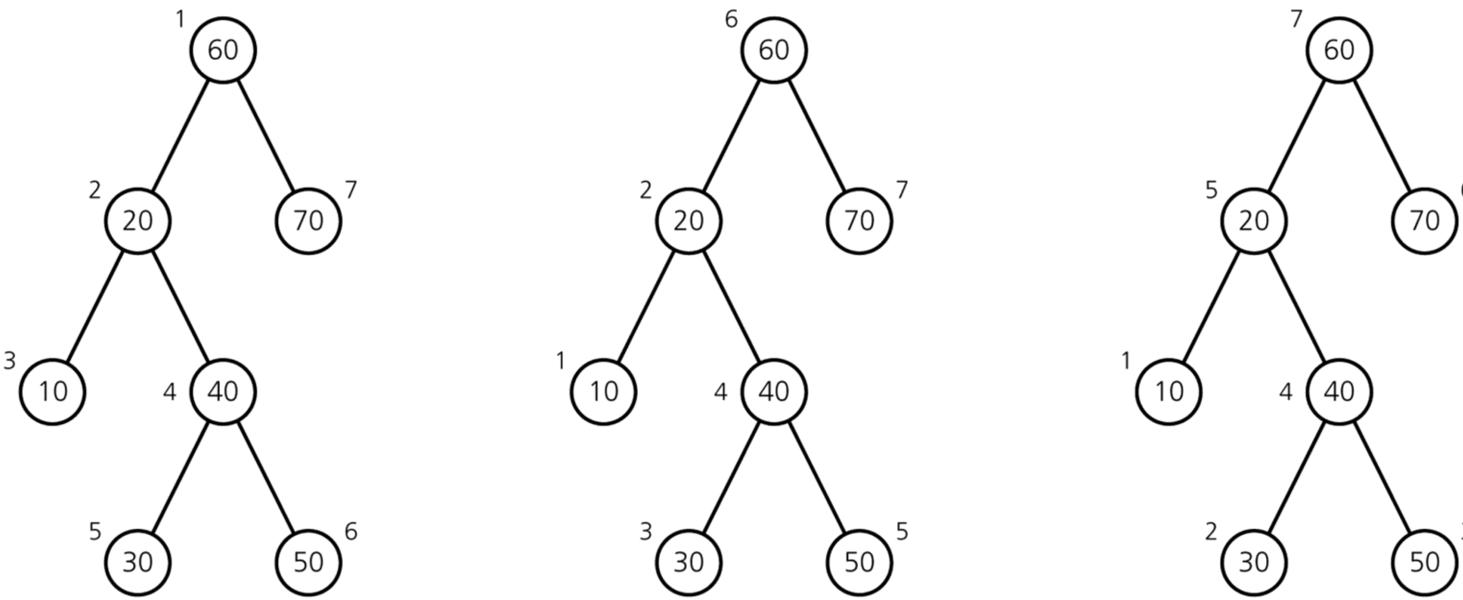
- Nodes with 0, 1 or 2 children
- Recursive – children are trees too!
- Traversals - *inOrder, preOrder, postOrder*



Binary Tree Traversals

- **Preorder Traversal**
  - The node is visited before its left and right subtrees,
- **Postorder Traversal**
  - The node is visited after both subtrees.
- **Inorder Traversal**
  - The node is visited between the subtrees,
  - Visit left subtree, visit the node, and visit the right subtree.

Binary Tree Traversals

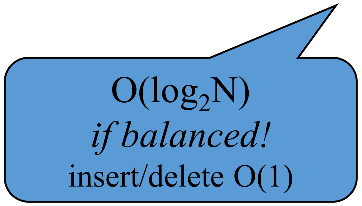


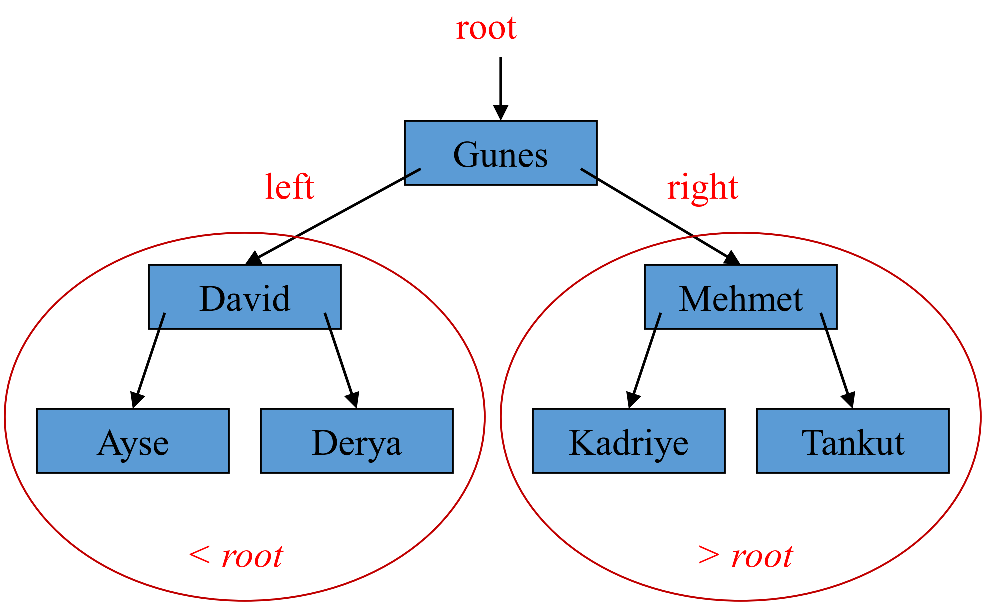
(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

Binary Trees

- Efficient insert/delete
- & search! *(binary search tree)*

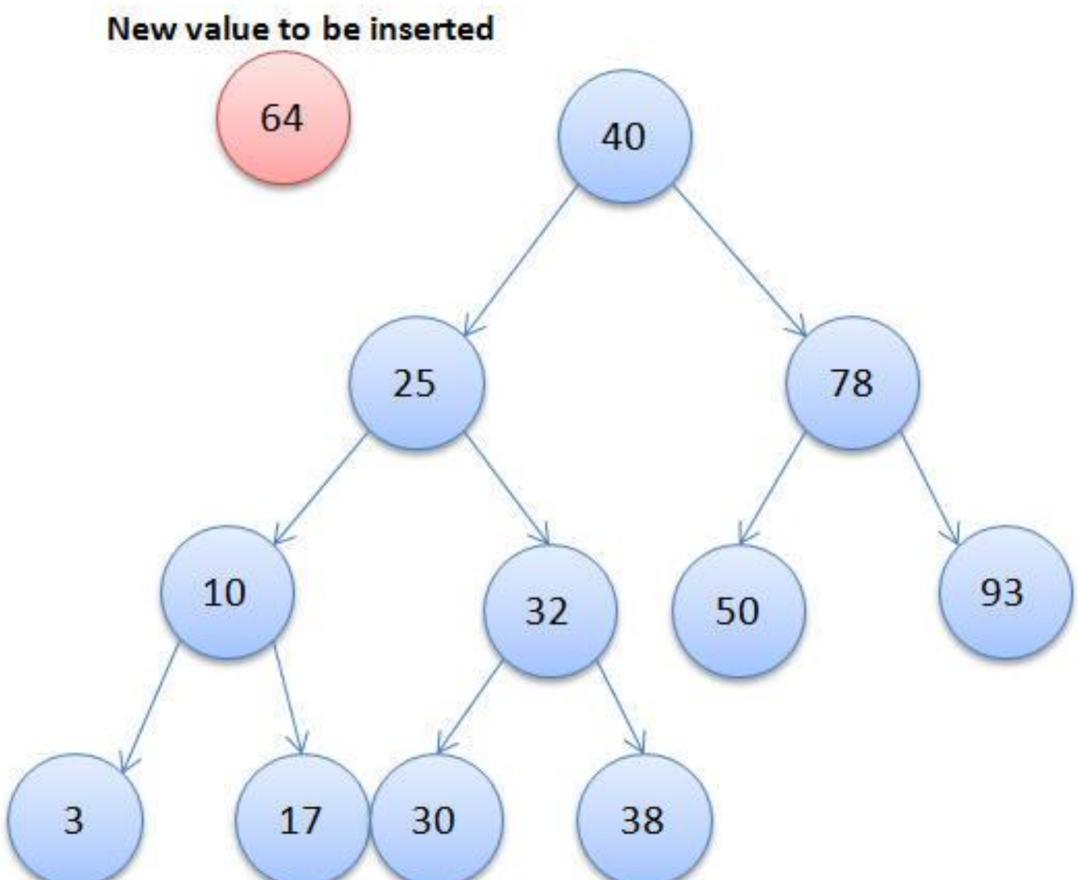




Binary Tree Node

```
public class Node<T> {
  public int value;
  public Node left;
  public Node right;

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

}
```

Binary Search Tree - Insert

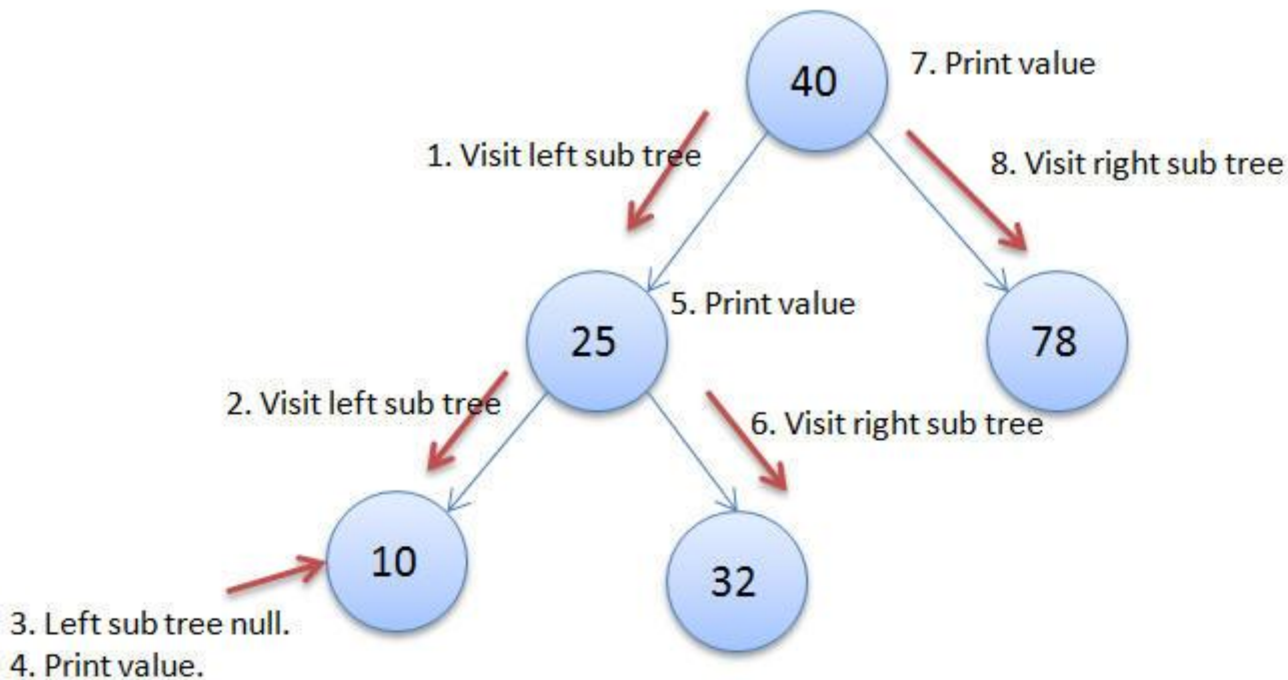


Binary Search Tree - Insert

```
public class BinarySearchTree {
 public Node root;

 public void insert(int value){
  Node node = new Node<>(value);

  if ( root == null ) {
   root = node;
   return;
  }

  insertRec(root, node);

 }
```

```
private void insertRec(Node latestRoot, Node node){

  if ( latestRoot.value > node.value){

   if ( latestRoot.left == null ){
    latestRoot.left = node;
    return;
   }
   else{
    insertRec(latestRoot.left, node);
   }
  }
  else{
   if (latestRoot.right == null){
    latestRoot.right = node;
    return;
   }
   else{
    insertRec(latestRoot.right, node);
   }
  }
 }
}
```

Binary Search Tree – Inorder Traversal

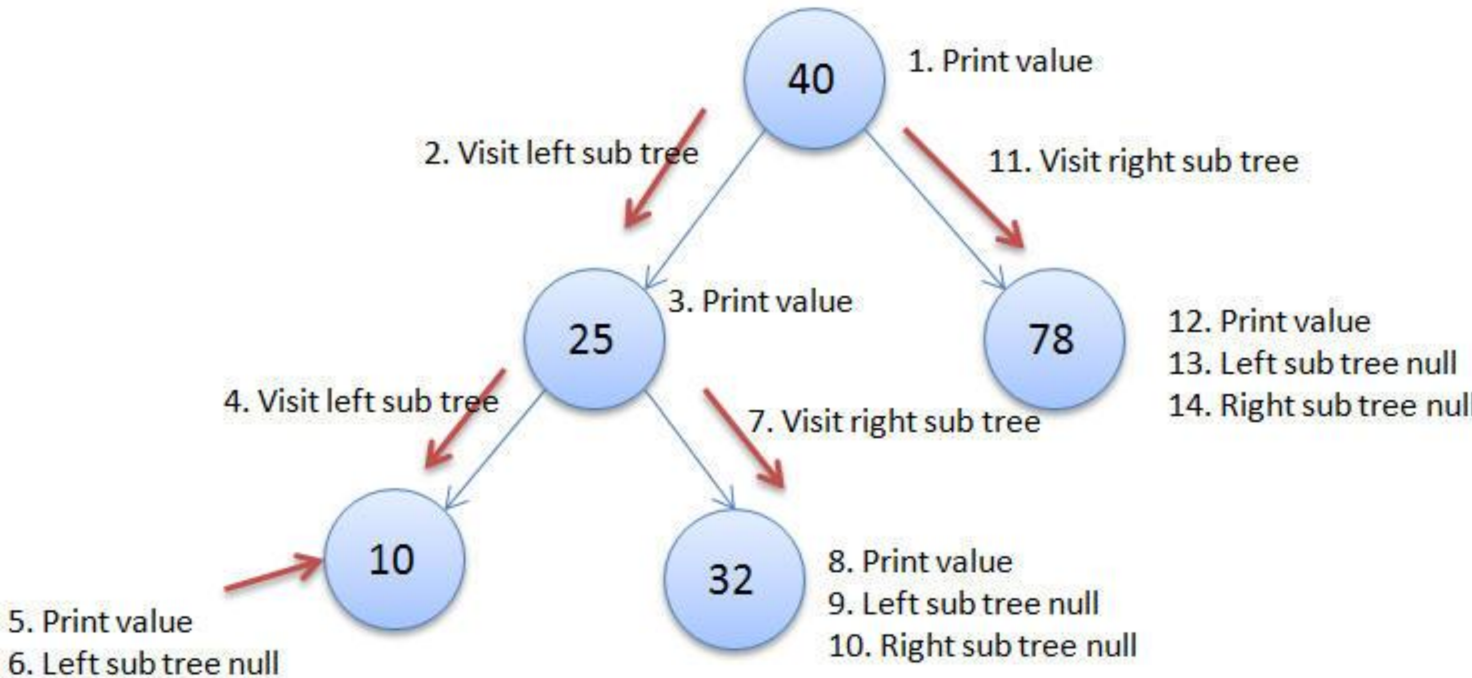


The above INORDER traversal gives: **10, 25, 32, 40, 78**

Binary Search Tree – Inorder Traversal

```
/**
 * Printing the contents of the tree in an inorder way.
 */
public void printInorder(){
  printInOrderRec(root);
  System.out.println("");
}

/**
 * Helper method to recursively print the contents in an inorder way
 */
private void printInOrderRec(Node currRoot){
  if ( currRoot == null ){
    return;
  }
  printInOrderRec(currRoot.left);
  System.out.print(currRoot.value+", ");
  printInOrderRec(currRoot.right);
}
```

Binary Search Tree – Preorder Traversal

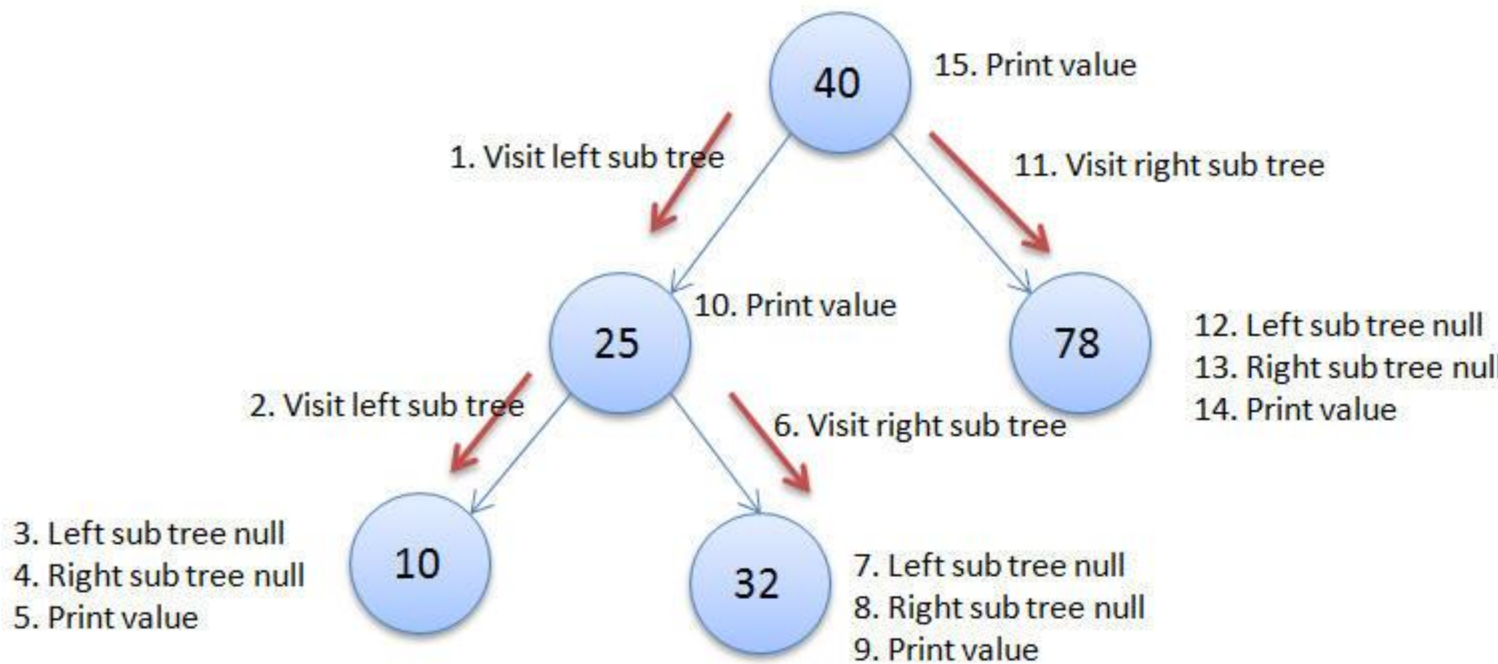


The above PREORDER traversal gives: **40, 25, 10, 32, 78**

Binary Search Tree – Preorder Traversal

```
/**
 * Printing the contents of the tree in a Preorder way.
 */
public void printPreorder() {
  printPreOrderRec(root);
  System.out.println("");
}

/**
 * Helper method to recursively print the contents in a Preorder way
 */
private void printPreOrderRec(Node currRoot) {
  if (currRoot == null) {
    return;
  }
  System.out.print(currRoot.value + ", ");
  printPreOrderRec(currRoot.left);
  printPreOrderRec(currRoot.right);
}
```

Binary Search Tree – Postorder Traversal



The above POSTORDER traversal gives: **10, 32, 25, 78, 40**

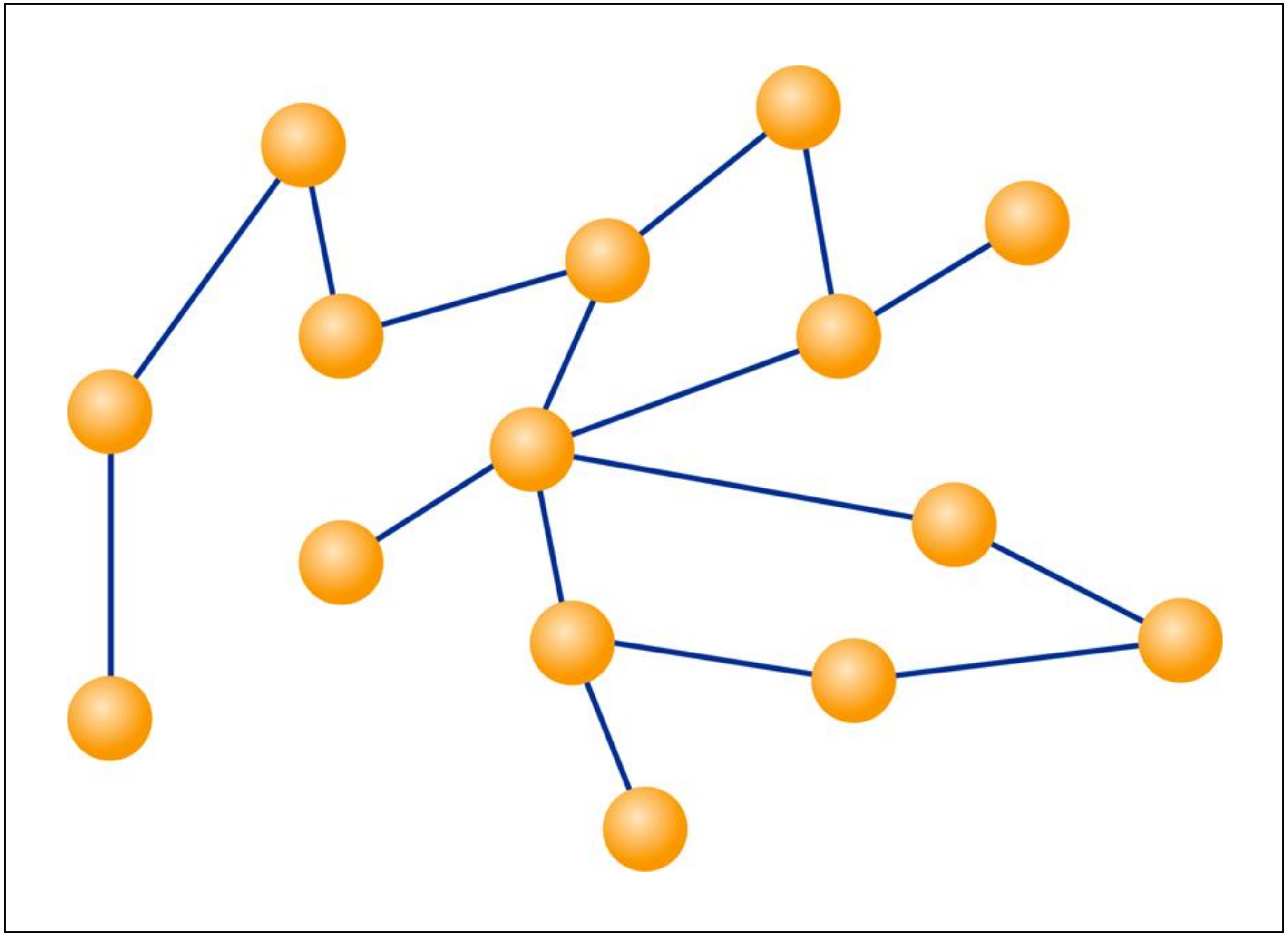
Binary Search Tree – Postorder Traversal

```
/**
 * Printing the contents of the tree in a Postorder way.
 */
public void printPostorder() {
  printPostOrderRec(root);
  System.out.println("");
}

/**
 * Helper method to recursively print the contents in a Postorder way
 */
private void printPostOrderRec(Node currRoot) {
  if (currRoot == null) {
    return;
  }
  printPostOrderRec(currRoot.left);
  printPostOrderRec(currRoot.right);
  System.out.print(currRoot.value + ", ");

}
```

Graphs

- A *graph* is a non-linear structure
- Unlike a tree or binary tree, a graph does not have a root
- Any node in a graph can be connected to any other node by an *edge*
- Analogy: the highway system connecting cities on a map

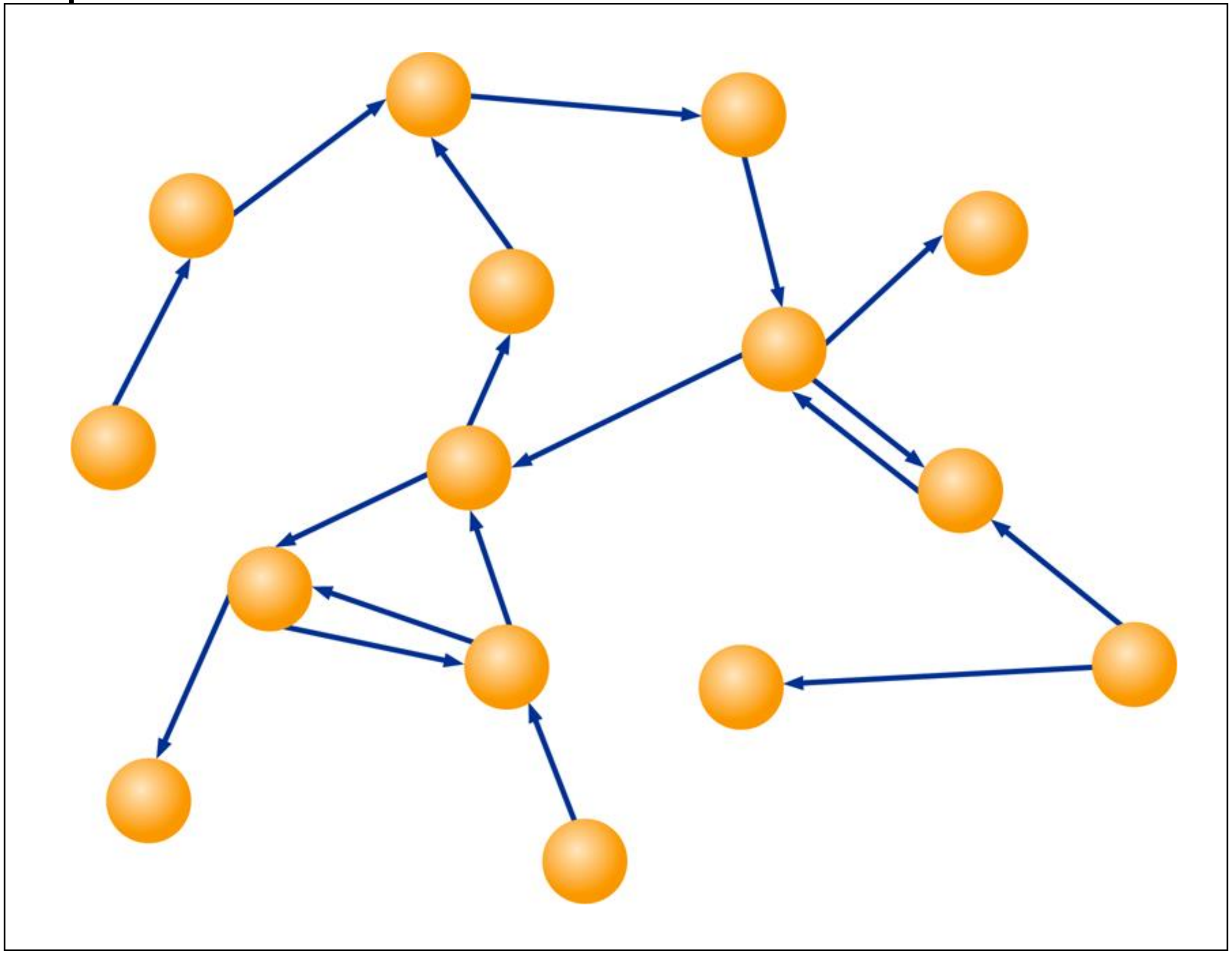
Graphs



Digraphs

- In a *directed graph* or *digraph*, each edge has a specific direction.
- Edges with direction sometimes are called *arcs*
- Analogy: airline flights between airports

Digraphs

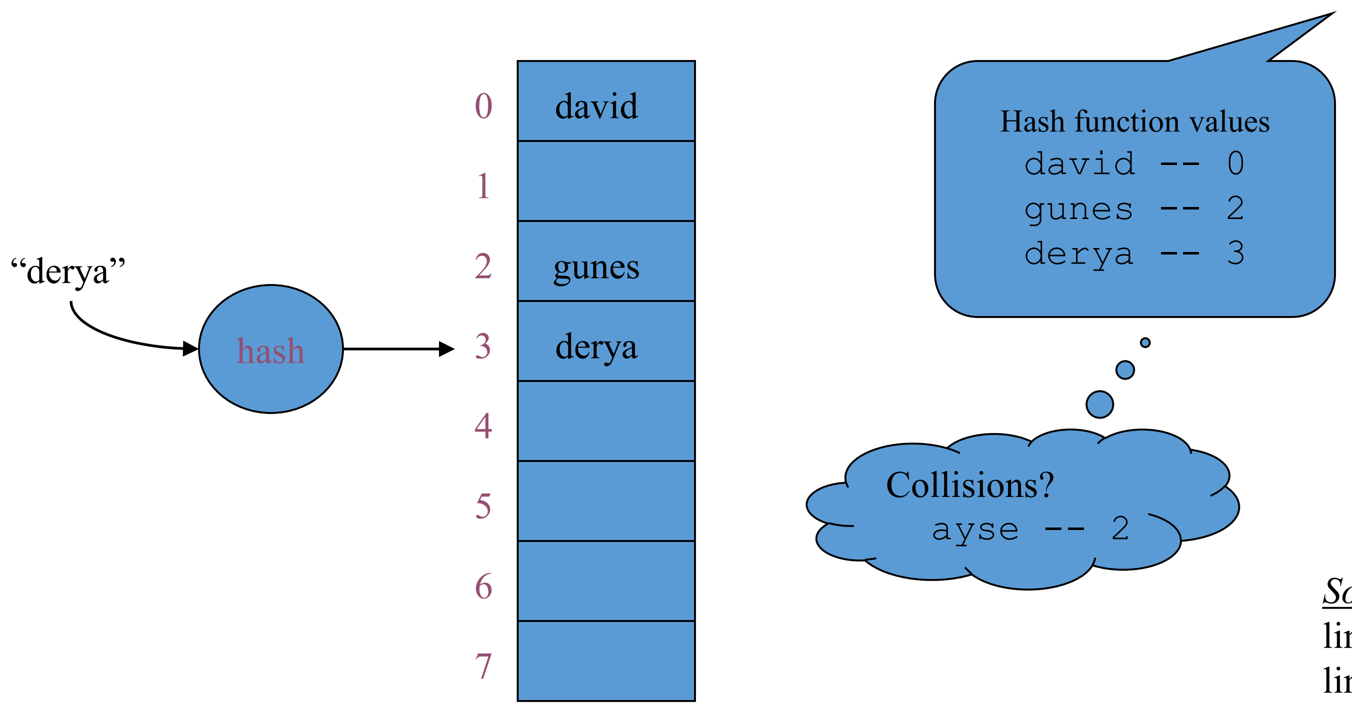


Representing Graphs

- Both graphs and digraphs can be represented using dynamic links or using arrays.
- As always, the representation should facilitate the intended operations and make them convenient to implement

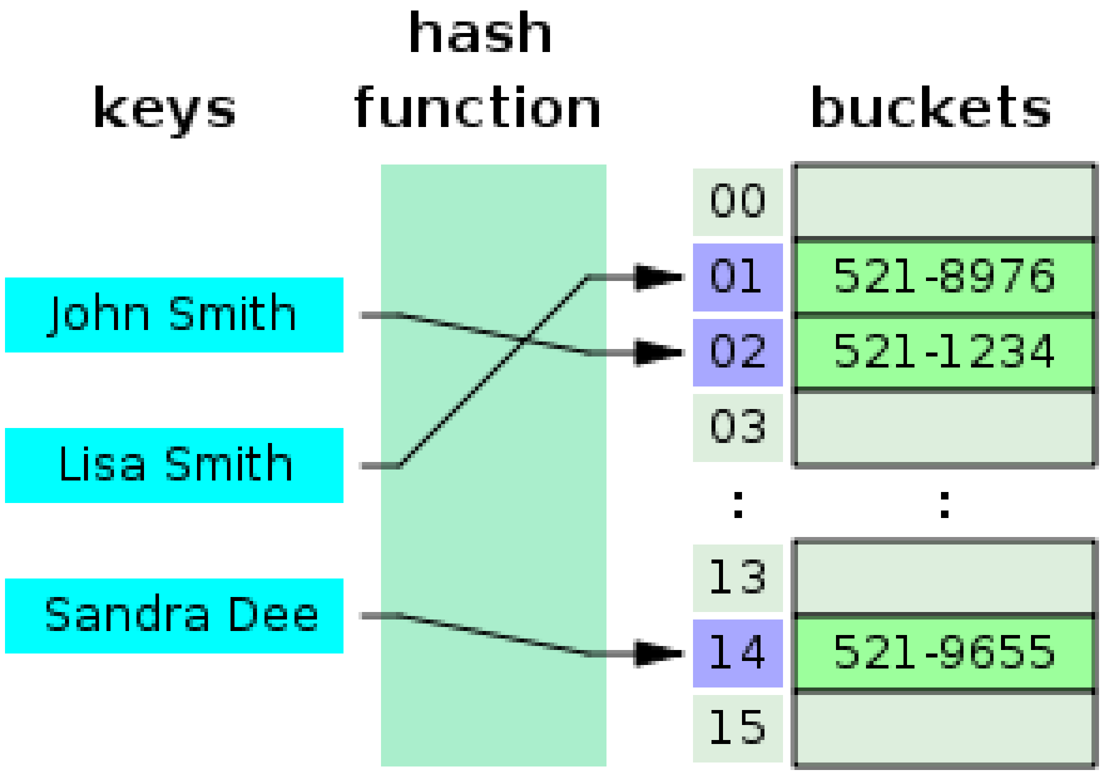
Hash

- What's the fastest way to find something?
  - Remember where you put it & look there!
- Hashing - computes location from data



Solutions:
linear probing
linked lists

Hash Tables


hash
function
keys
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
:
13
14
15
521-8976
521-1234
:
521-9655

Hash Tables

- This example creates a hashtable of numbers. It uses the names of the numbers as keys:

```
Hashtable<String, Integer> numbers = new Hashtable<String, Integer>();
numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);
```

- To retrieve a number, use the following code:

```
Integer n = numbers.get("two");
if (n != null) {
        System.out.println("two = " + n);
}
```

Hash Tables – Collusion