Recursive Algorithms and Dynamic Data Structures

CS 18000
Sunil Prabhakar
Department of Computer Science
Purdue University

Problem

- Write a method that lists all the files contained in a directory (including all its sub-directories).
- How deep can the directory tree be?
- This is unknown *a priori*.
- To list the files in a directory:
  - list all the files in the directory
  - for each sub-directory, list the files in the (sub-)directory
- This is a *Recursive Definition*

Recursive Algorithms

- Within a given method, we are allowed to call other accessible methods.
- It is also possible to call the same method from within the method itself.
- This is called a recursive call.
- For certain problems a recursive solution is very natural and simple.
- It is possible to implement a recursive algorithm without using recursion, but the code can be more complex.

Recursive Method

- A *recursive method* is a method that contains a statement (or statements) that makes a call to itself.
- We can write a method for listDirectory(dir) using recursion:

Test to stop or continue.

End case: recursion stops.

Recursive case: recursion continues.

```
private static void listDirectory(File directory){
    File contentFiles[];

    contentFiles = directory.listFiles();
    if (contentFiles == null)
        return;
    for (int i = 0; i < contentFiles.length; i++ ) {
        if(contentFiles[i].isFile())
            listFile(contentFiles[i]);
        else
            listDirectory(contentFiles[i]);
    }
}
```

DirectoryListing.java

Example of Recursion

- The *factorial of N* is the product of the first N positive integers:

  $$n! = 1 * 2 * \ldots * (n-1) * n$$

- This is useful for many situations, e.g.,
  - there are ***n!*** possible sequences of ***n*** objects
  - there are ***n!(n-k)!/k!*** unique subsets of size ***k***, from a set of size ***n***.
- Note that: $n! = n * (n-1) * \ldots * 2 * 1 = n * (n-1)!$
  - this is a **recursive** definition

$$\text{factorial}(n) = \begin{cases} n * \text{factorial}(n-1) & \text{if } n>1 \\ 1 & \text{otherwise} \end{cases}$$

Recursive Method

- An *recursive method* is a method that contains a statement (or statements) that makes a call to itself.
- We can write a method for factorial(n) using recursion:

Test to stop or continue.

End case: recursion stops.

Recursive case: recursion continues.

```
public int factorial( int n ) {

    if ( n == 1 ) {
        return 1;

    } else {
        return n * factorial( n-1 );
    }
}
```

The Details …

- As with any call, a recursive call results in the creation of temporary workspace for the called method and copying of parameters.
- Each call to a method results in the creation of a separate workspace with new copies of variables (local and formal parameters).
- When a recursive call ends, flow returns to the caller.

Example

```
factorial( 3 );
```

```
public int factorial( int n ) {
    if ( n == 1 ) {
        return 1;

    } else {
        return n * factorial( n-1 );
    }
}
```

n 3

```
public int factorial( int n ) {
    if ( n == 1 ) {
        return 1;

    } else {
        return n * factorial( n-1 );
    }
}
```

n 2

```
public int factorial( int n ) {
    if ( n == 1 ) {
        return 1;

    } else {
        return n * factorial( n-1 );
    }
}
```

n 1

Example

6

```
factorial( 3 );
```

2

```
public int factorial( int n ) {
    if ( n == 1 ) {
        return 1;

    } else {
        return n * factorial( n-1 );
    }
}
```

n 3

1

```
public int factorial( int n ) {
    if ( n == 1 ) {
        return 1;

    } else {
        return n * factorial( n-1 );
    }
}
```

n 2

```
public int factorial( int n ) {
    if ( n == 1 ) {
        return 1;

    } else {
        return n * factorial( n-1 );
    }
}
```

n 1

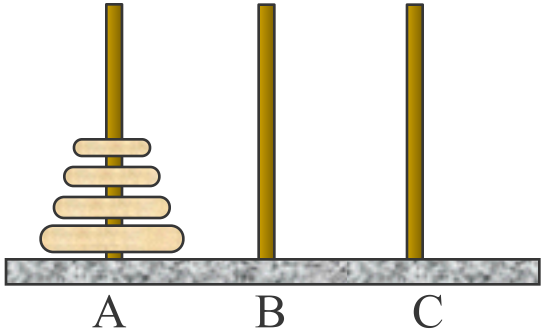
Towers of Hanoi

- The goal of the Towers of Hanoi puzzle is to move N disks from Column A to Column C:

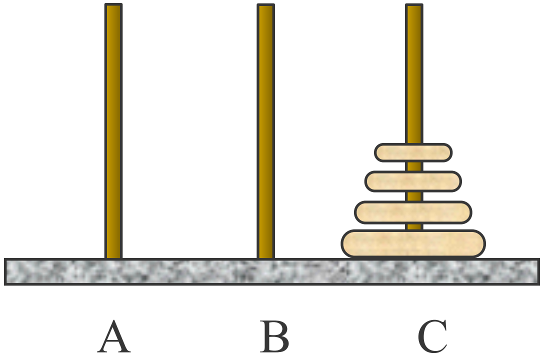
- Only two rules:

1. Move one disk at a time
2. A disk cannot rest on a smaller disk

From:

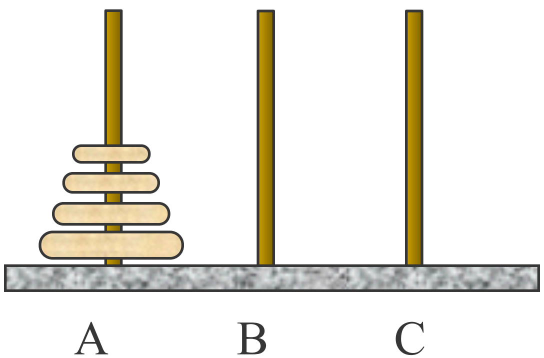


End:

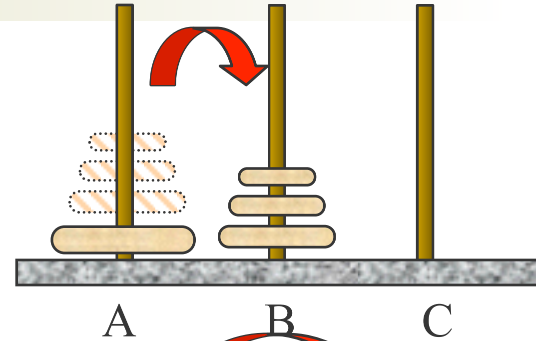


Towers of Hanoi Solution

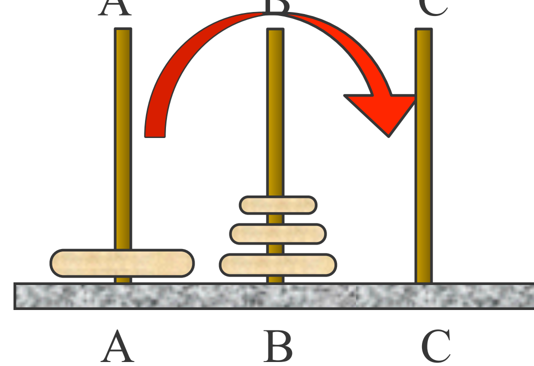
To move 4 disks from A to C (using B as a temporary spot):



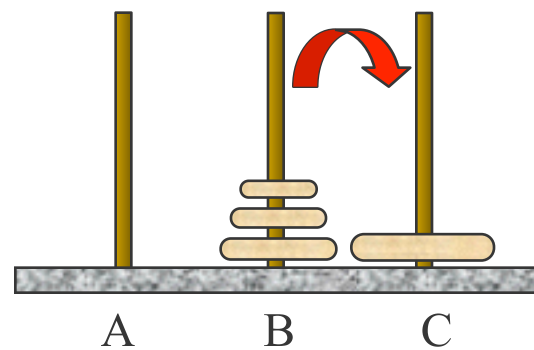
1. Move 3 disks from A to B (using C)



2. Move one disk from A to C



3. Move 3 disks from B to C (using A)



Towers of Hanoi Solution

Test

End case

Recursive case

```
public void moveDisks(int start,  //source column
                      int end,    //destination column
                      int using, //helper column
                      int n   ) {  //number of disks
    if ( n == 1 ) {
        moveOneDisk( start, end );
    } else {
        moveDisks( start, using, end, n-1 );
        moveOneDisk(start, end );
        moveDisks( using, end, start, n-1 );
    }
}

private void moveOneDisk( int from, int to ) {
    System.out.println( "Move from " + from + " to " + to );
}
```

Recursion vs Iteration

- Recursion has greater overhead due to method calls, variable setups etc.
- Recursion provides cleaner solutions
- Can be simulated using iteration and a stack-like structure
  - may add too much complexity (consider an iterative solution for Towers of Hanoi)

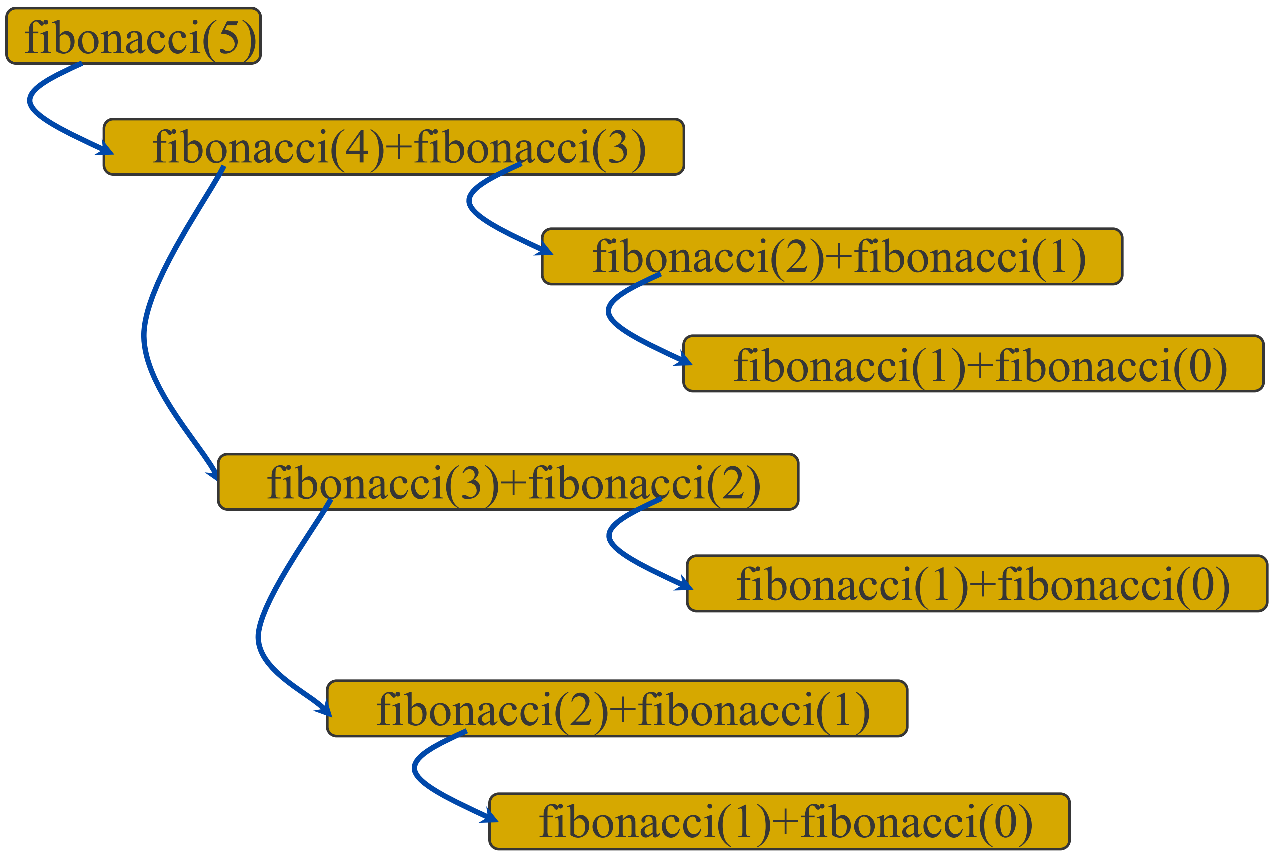
When Not to Use Recursion

- When recursive algorithms are designed carelessly, it can lead to very inefficient solutions.
- For example, consider the following:

```
public int fibonacci( int n ) {

    if (n == 0 || n == 1) {
        return 1;

    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Excessive Repetition

- Recursive Fibonacci ends up repeating the same computation numerous times.



Non-recursive Fibonacci

```
public long fibonacci( int n ) {

    long fib[] = {1,1,1};

    if (n < 2)
        return 1;

    for (int i=2; i<=n;i++ ) {
        fib[0]=fib[1];
        fib[1]=fib[2];
        fib[2]=fib[0]+fib[1];
    }

    return fib[2];
}
```

Overhead of Recursion

- Remember that each recursive call is expensive
  - create local variables for each call
  - copy arguments into local variables
  - track the execution point for each call
  - returned values are copied back
  - local space is reclaimed

When Not to Use Recursion

- In general, use recursion if
  - A recursive solution is natural and easy to understand.
  - A recursive solution does not result in excessive duplicate computation.
  - The equivalent iterative solution is too complex.

Recursive Data Structures

- As with recursive methods, recursive data structures can be very useful
- A recursive data structure contains a data member of its own type

Problem

- Consider a program that has to read in an unknown number of names from input
- How do we store these in our program?
  - We could guess the number and use an array of this size
  - May waste memory if guess is too large
  - What if guess is too small?

Changing Array Size

- If our guess for the size is too small, we would run out of space
- We should resize the array in this situation
- But an allocated array can't be resized
  - have to create a new bigger array and COPY!
- How much do we grow by?
  - if too small will grow and copy again
  - if too big, we may be wasting memory!
- Is there a better solution?

Linked Lists

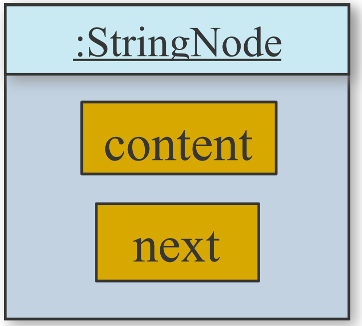
- Linked lists are an option for this problem.
- Instead of creating new fixed size arrays and copying, organize the data to reflect the actual size at any time.
- Each stored item is created as a separate object
- We organize the objects in the list as a chain.
- Objects are added or deleted as needed

Linked List

- Our linked list will be implemented as a chain of Node objects.
  - No predetermined size - add/delete as needed
- Each Node object will have
  - A String data member that is the value stored at that position, and
  - A reference to the next node in the list.
  - This is a recursive data structure.
- No indexes for entries
- Addition takes place at one end

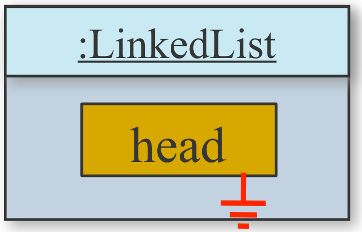
The StringNode Class

```
public class StringNode {
    private StringNode next;
    private String content;

    public StringNode() {
        next = null;
        content = null;
    }
    public String getContent(){
        return content;
    }
    public void setContent(String s){
        content = s;
    }
    public StringNode getNext(){
        return next;
    }
    public void setNext(StringNode nextNode){
        next = nextNode;
    }
}
```



Linked List Behavior

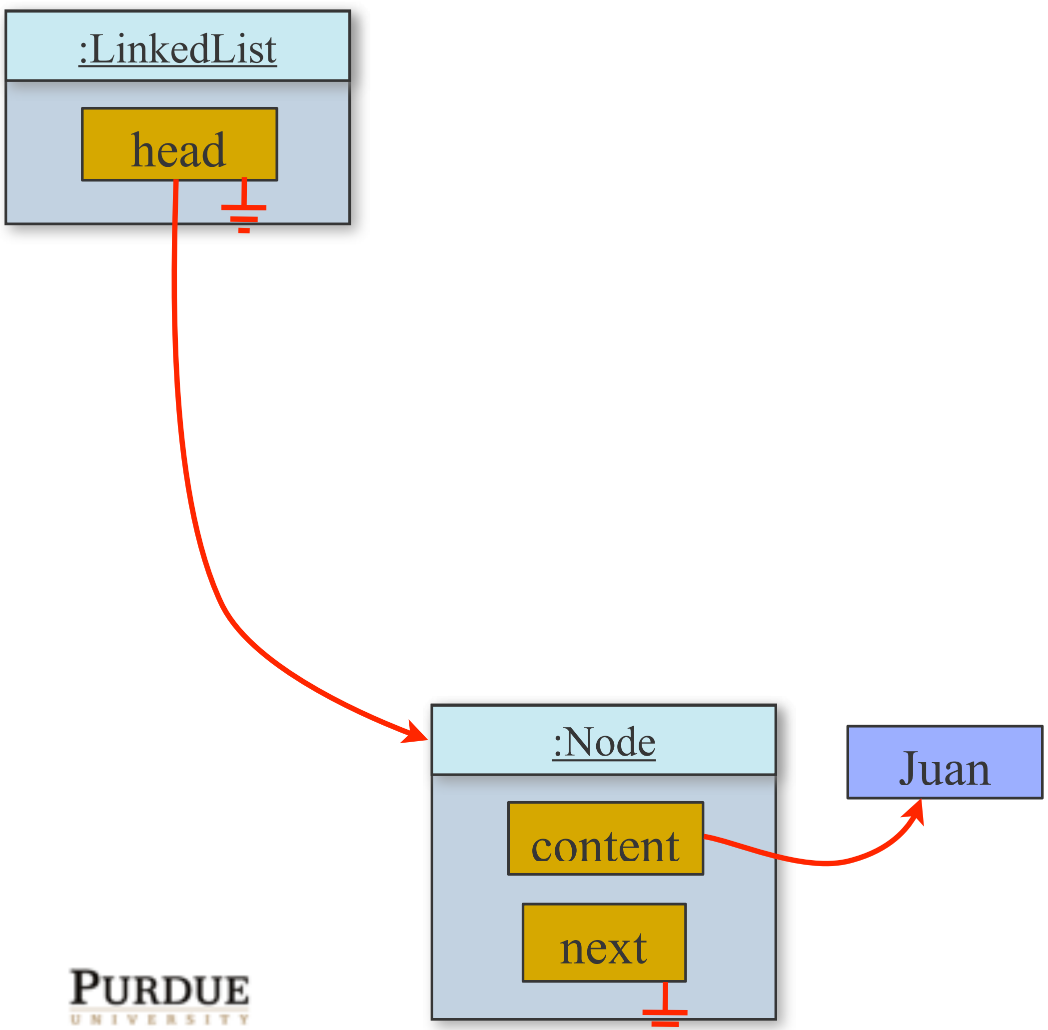
Empty List



 `null` reference

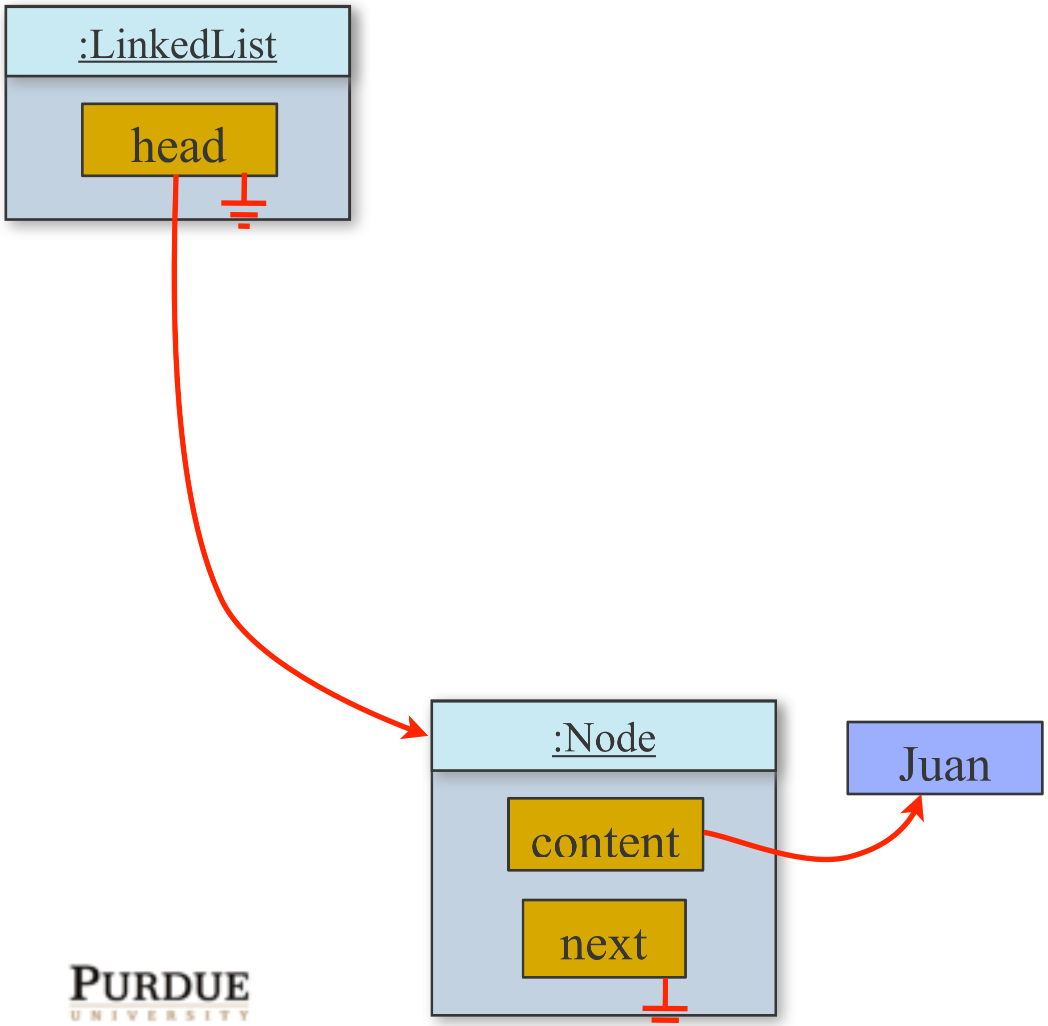
Linked List Behavior

Add “Juan”



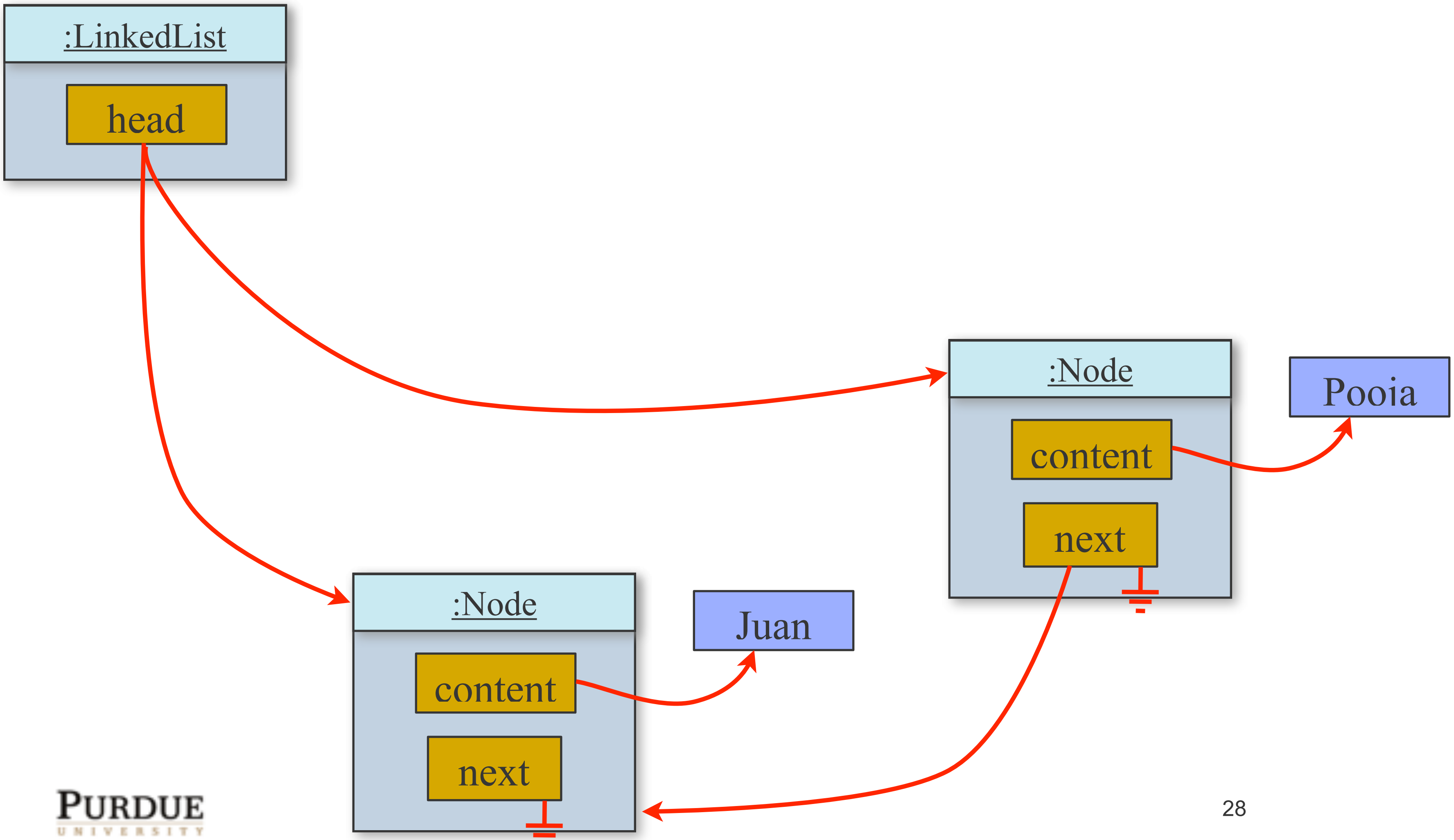
Linked List Behavior

Add “Juan”



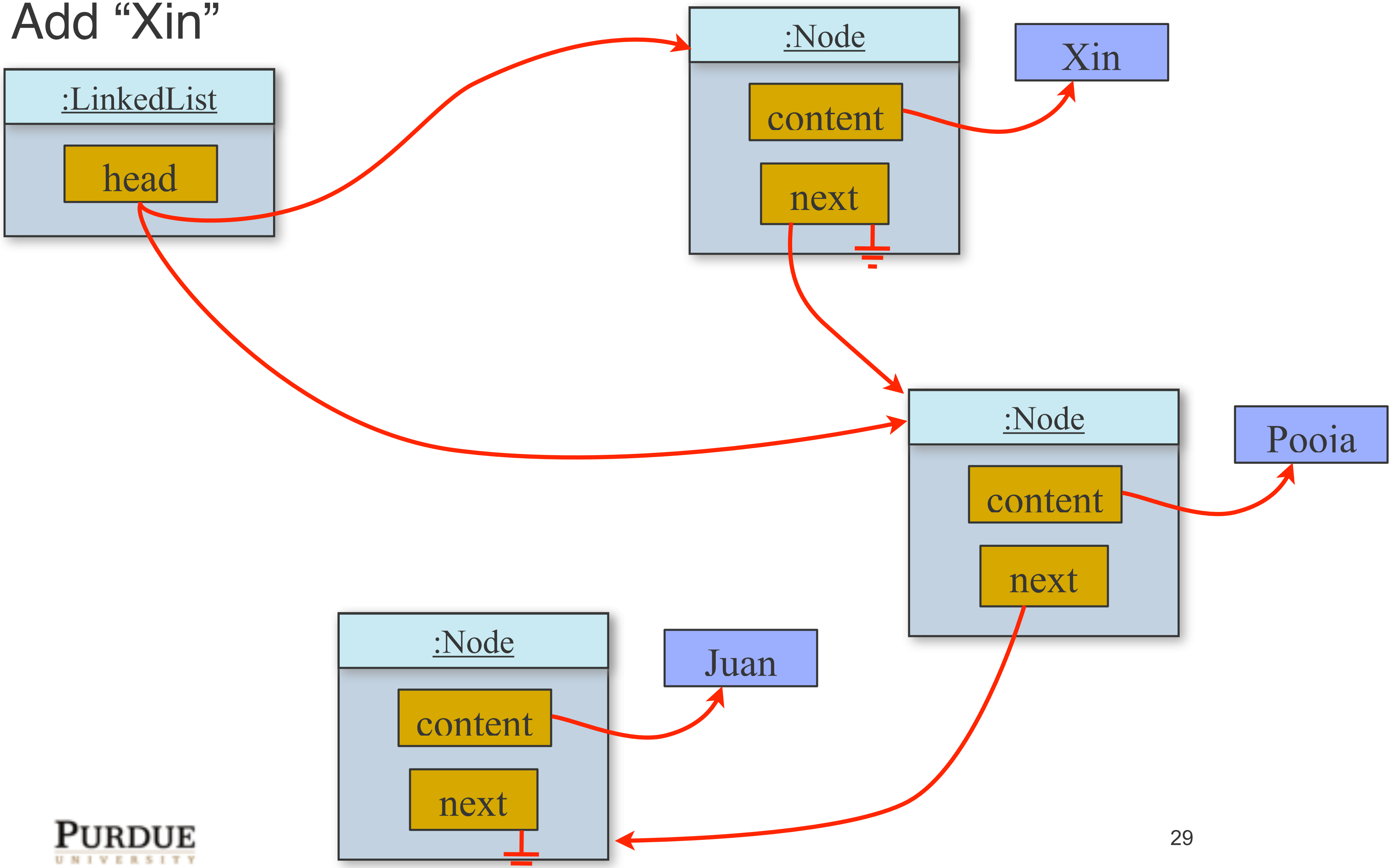
Linked List Behavior

Add “Pooja”



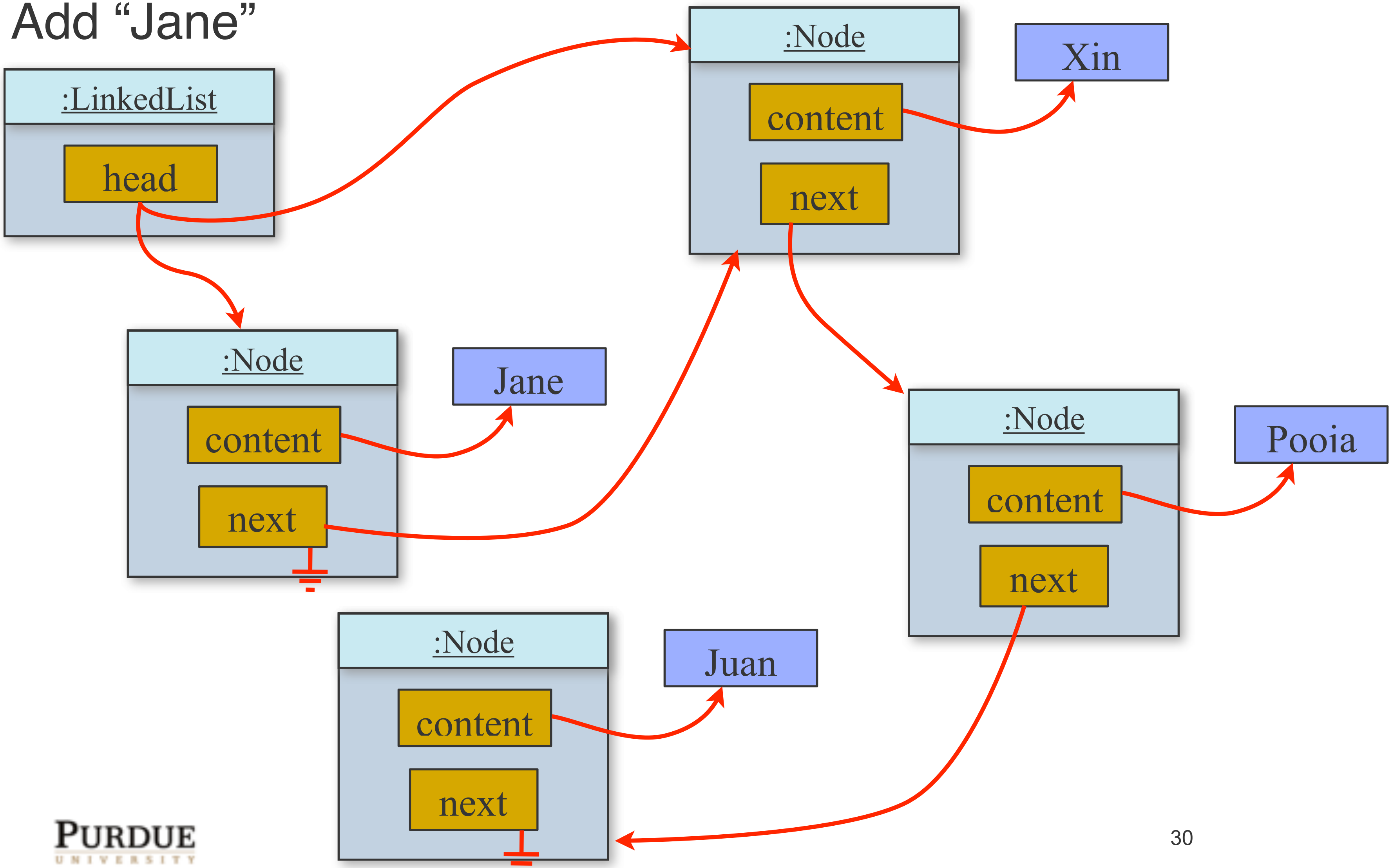
Linked List Behavior

Add “Xin”



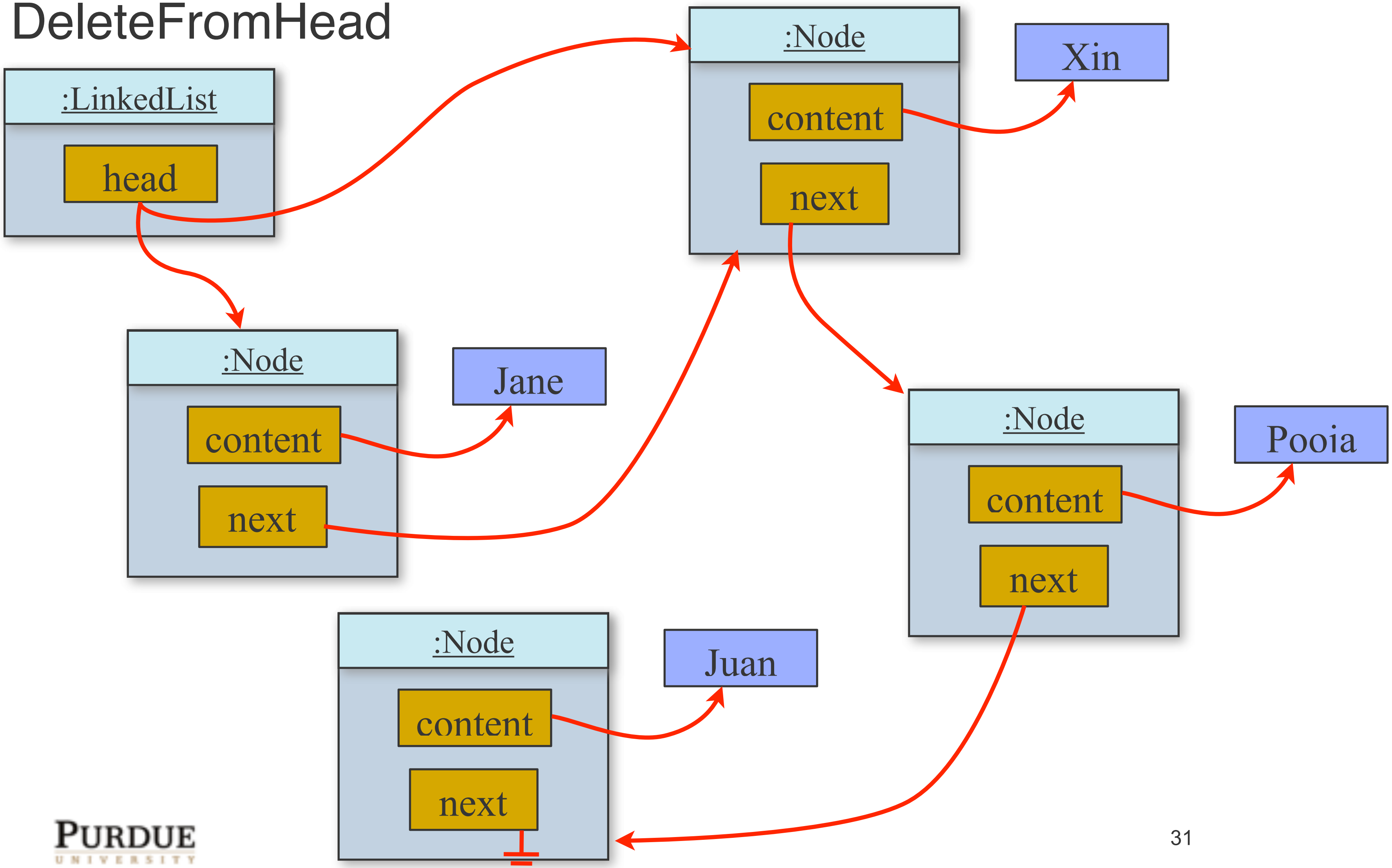
Linked List Behavior

Add “Jane”



Linked List Behavior

DeleteFromHead



Reading into Linked List

Allocate linked list

Add each string to the linked list

```
int numStrings = 0;
String input;
Scanner scanner = new Scanner(System.in);
StringLinkedList listOfStrings;

listOfStrings = new StringLinkedList();


input = scanner.nextLine();

while (input.length() > 0) {
    listOfStrings.addToHead(input);
    numStrings++;
    input = scanner.nextLine();
}
```

Reading Out a Linked List

```
String value[];

. . .
value = new String[numStrings];

for (int i = 0; i < numStrings; i++) {
    value[i] = listOfStrings.deleteFromHead();
    System.out.println(value[i]);
}
```

The LinkedList Class

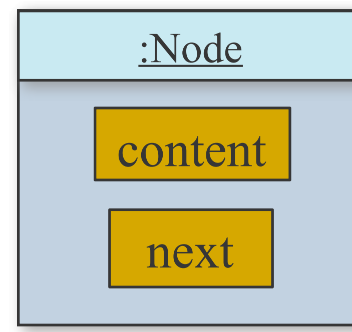
- The need for a linked list occurs often.
- It is beneficial to separate out the code for handling a linked list into a separate class.
- This class can then be reused.
- In general our list may
  - store other types of objects, not just strings
  - delete entries from the list
  - add entries to the other end of the list
  - add entries in the middle of the list
  - create an empty list

Abstract Data Types

- By creating a separate class for a general linked list, we are creating a new type
- We call these Abstract Data Types (ADTs) in general.
- An ADT is defined in terms of its public behavior
- The ADT hides (encapsulates) the details of the implementation

The Generic Node Class

```
public class Node {
    private Node next;
    private Object content;

    public Node() {
        next = null;
        content = null;
    }
    public Object getContent(){
        return content;
    }
    public void setContent(Object c){
        content = c;
    }
    public Node getNext(){
        return next;
    }
    public void setNext(Node nextNode){
        next = nextNode;
    }
}
```



The LinkedList class

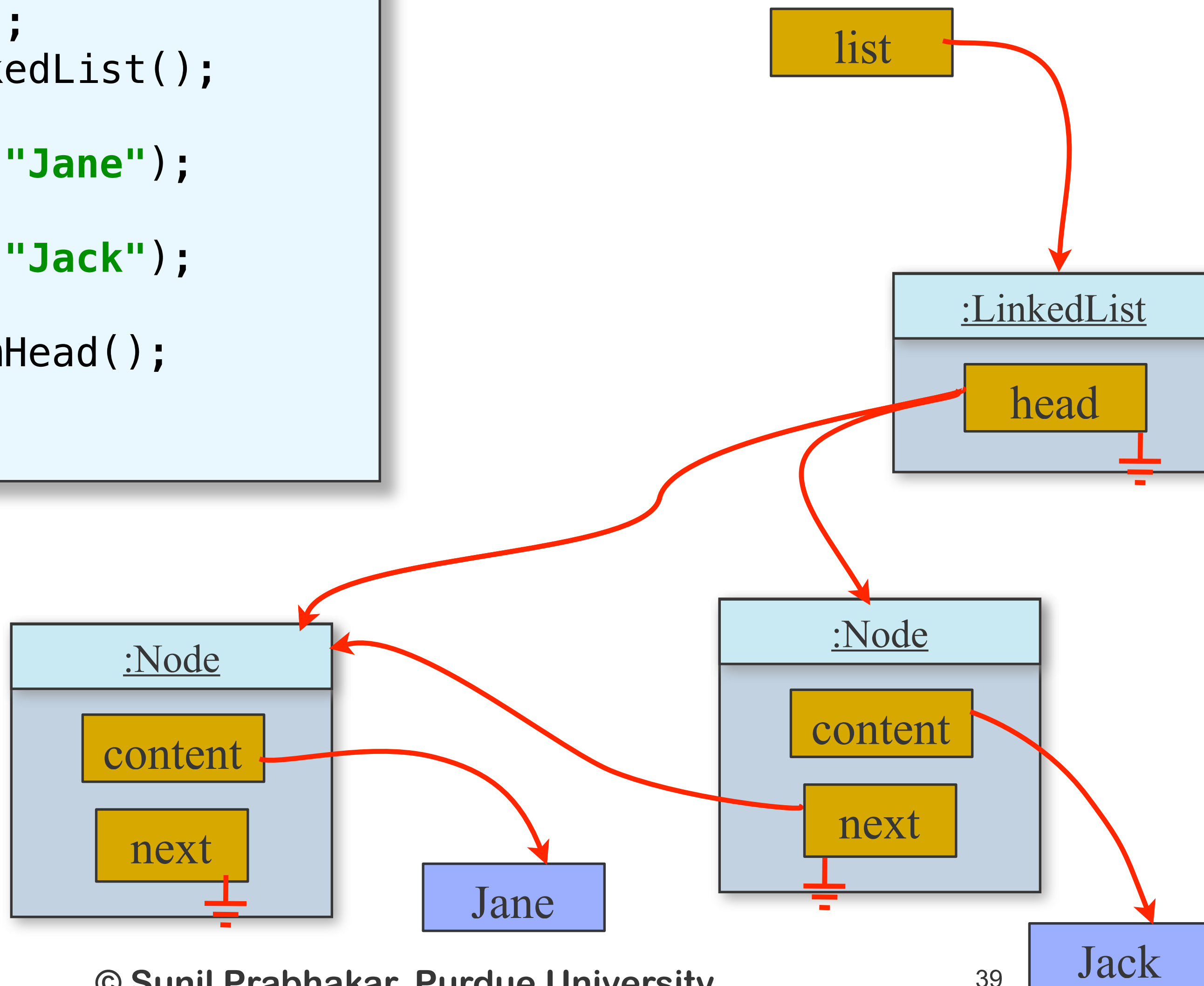
```
public class LinkedList {
    private Node head;

    public void LinkedList() {
        head = null;
    }

    public void addToHead(Object c) {
        Node n = new Node();
        n.setContent(c);
        n.setNext(head);
        head = n;
    }

    ...
```

The LinkedList class (cont.)

```
    . . .
    public void deleteFromHead() throws Exception {
        if(head == null)
            throw new Exception("Empty List");
        else
            head = head.getNext();
    }

    public Object getFromHead() throws Exception {
        if(head == null)
            throw new Exception("Empty List");
        else
            return head.getContent();
    }
}
```

Using the LinkedList class

```
public void main (String[] args){
    LinkedList list;
    list = new LinkedList();

    list.addToHead("Jane");

    list.addToHead("Jack");
    . . .
    list.deleteFromHead();
}
. . .
```

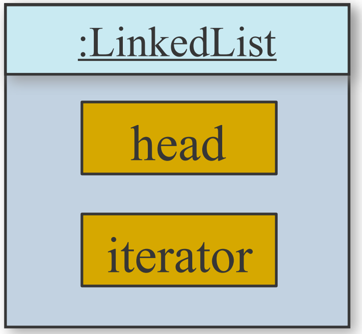


Accessing Other Elements

- The current version of linked list only allows access to the element at the head, and adding and deleting from head
- Let us add the ability for a client class to traverse the list
- This is a common feature for most collections
  - we use a reference to a node called an iterator

Adding an iterator

```
public class LinkedList {
    private Node head;
    private Node iterator;

    public void LinkedList() {
        head = null;
        iterator = null;
    }
    public void startScan throws Exception (){
        if(head == null)
            throw new Exception("Empty List");
        else
            iterator = head;
    }
    . . .
```



The LinkedList class iterators

```
    . . .
    public boolean hasMore(){
        if(iterator.getNext() == null)
            return false;
        else
            return true;
    }

    public void moveAhead(){
        iterator = iterator.getNext();
    }

    public Object getCurrentItem throws Exception (){
        if(iterator == null)
            throw new Exception("No Current Item");
        return iterator.getContent();
    }
}
```

Traversing the list

```
. . .
list.startScan();
s = (String) list.getCurrentItem();
System.out.println(s);
while(list.hasMore()){
    list.moveAhead();
    s = (String) list.getCurrentItem();
    System.out.println(s);
}
. . .
```

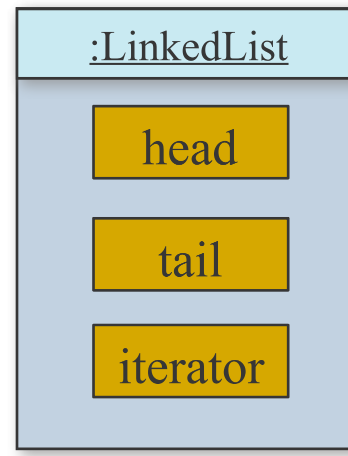
Adding to the End of a Linked List

- We can add to the end of a list, not just the beginning (head).
- To do this efficiently, we keep a reference to the end (tail).

- How about adding to the middle of a list
  - e.g., in sorted order of entries?

Adding at other end

```
class LinkedList {
    private Node head;
    private Node tail;
    private Node iterator;

    public void LinkedList() {
        head = null;
        iterator = null;
        tail = null;
    }
    public void addToTail(Object c){
        Node n = new Node();
        n.setContent(c);
        tail.setNext(n);
        tail = n;
    }
    . . .
```



Common ADTs

- Some data structures are used very often in programming. We will see three of these
  - Stack
  - Queue
  - Binary Tree
- An ADT is defined in terms of its public behavior
- The internal implementation can vary
  - affects performance, but not behavior

Stack

- A stack is analogous to a stack of books
- We add and remove one book at a time from the top of the stack
- A stack supports these operations:
  - `void push(Object)`
    - add Object to top of stack
  - `Object pop()`
    - delete Object from top of stack and return it
  - `Object top()`
    - return Object from top of stack without deleting
  - `boolean isEmpty()`

The Stack class

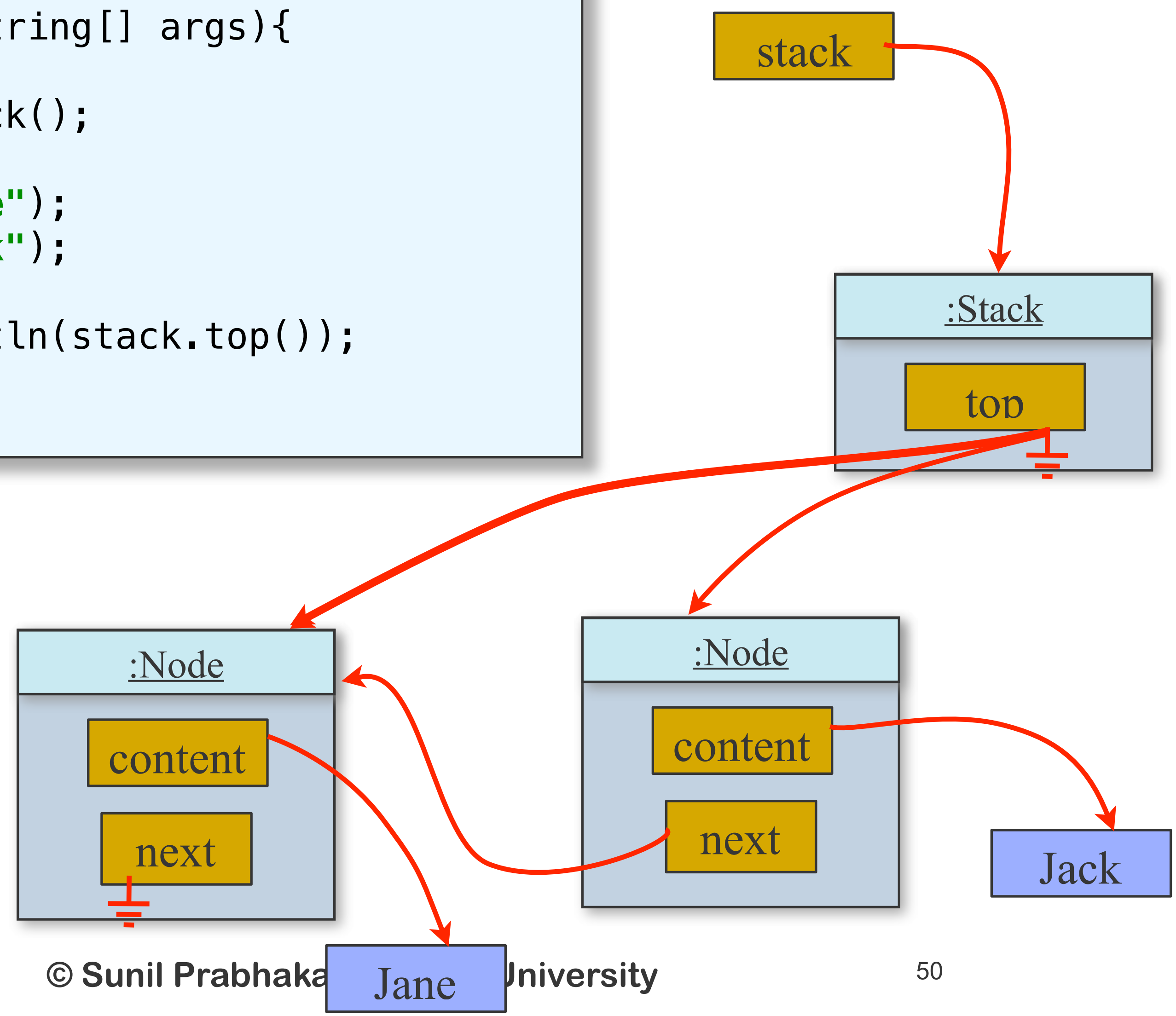
```
public class Stack {
    private Node top;

    public void Stack() {
        top = null;
    }
    public void push(Object c){
        Node n = new Node();
        n.setContent(c);
        n.setNext(top);
        top = n;
    }

    public Object pop () throws Exception{
        if(isEmpty())
            throw new Exception(“Empty Stack”);
        Object value = top.getContent();
        top = top.getNext();
        return value;
    }
    . . .
```

The Stack class (cont.)

```
   . . .

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

    public Object top () throws Exception{
        if(isEmpty())
        throw new Exception("Empty Stack");
        return top.getContent();
    }
}
```

Using the Stack class

```
. . .
public void main (String[] args){
    Stack stack;
    stack = new Stack();

    stack.push("Jane");
    stack.push("Jack");
    stack.pop();
    System.out.println(stack.top());
}
. . .
```



Getting input with a Stack

```
. . .
Stack list= new Stack();
int n = 0;
while(in.hasNextLine()){
     list.push(nextLine());
     n++;
}

String value[] = new String[n];
for(int i=0; i<n; i++)
value[i] = (String)list.pop();

. . .
```

Queue

- A First-In-First-out (FIFO) Queue is often used by operating systems for processes, requests, etc.
- Similar to a regular queue for service
- Queue ADT:
  - void **put**(Object)
    - add object to end of queue
  - Object **get**()
    - get object at head of queue
  - boolean **isEmpty**()

Queue

- A First-In-First-out (FIFO) Queue is often used by operating systems for processes, requests, etc.
- Similar to a regular queue for service
- Queue ADT:
  - `void put(Object)`
    - add object to end of queue
  - `Object get()`
    - get object at head of queue
  - `boolean isEmpty()`

The Queue class

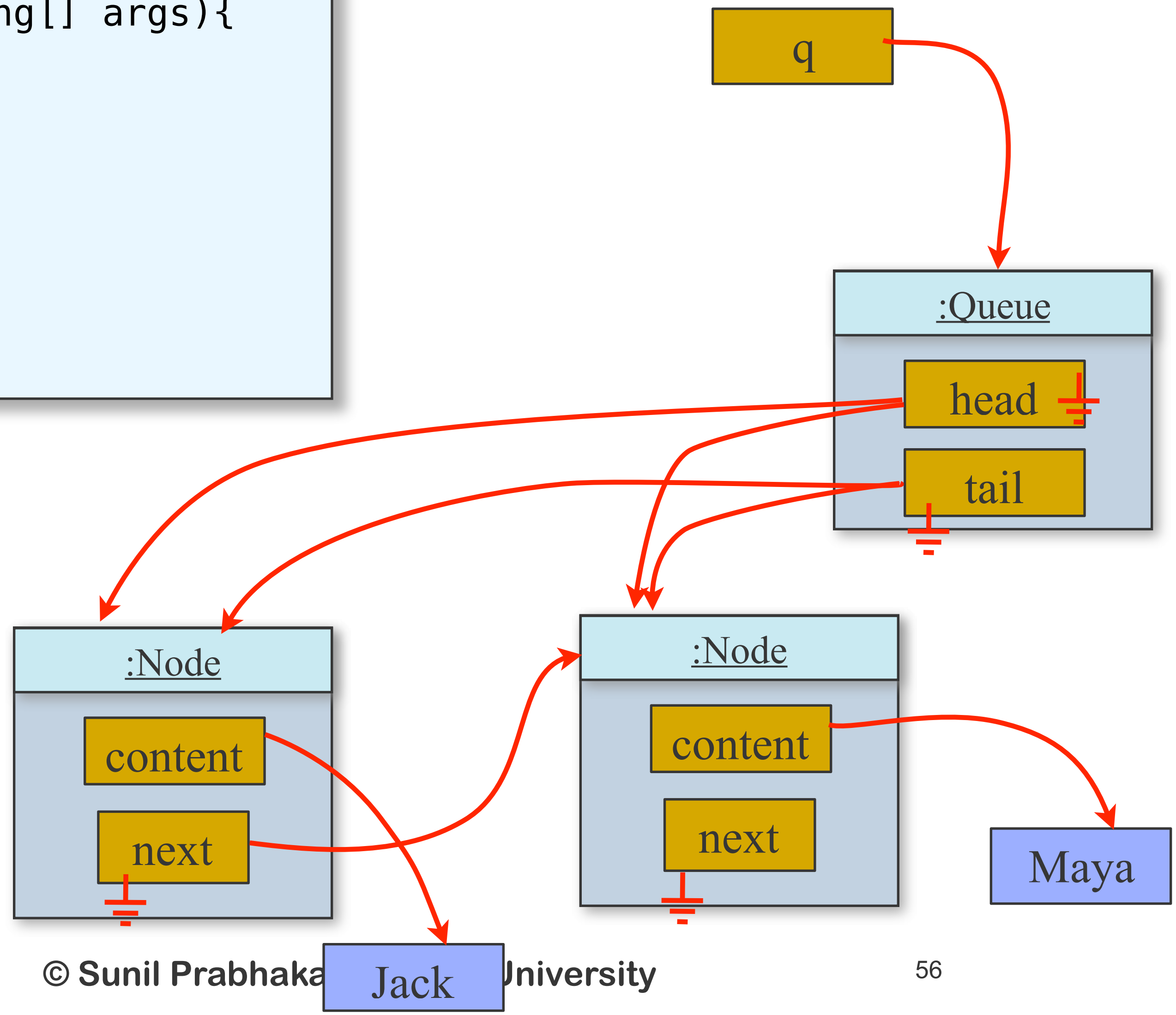
```
public class Queue {
    private Node head, tail;

    public void Queue() {
        head = tail = null;
    }
    public void put(Object c){
        Node n = new Node();
        n.setContent(c);
        if (isEmpty())
            head = tail = n;
        else {
            tail.setNext(n);
            tail = n;
        }
    }
    . . .
```

The Queue class (cont.)

```
    . . .

    public Object get(Object c){
        Object val;
        if isEmpty()
        throw new Exception("Empty Queue");
        else {
            val = head.getContent();
            head = head.getNext();
            return val;
        }
    }

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

Using the Queue class

```
. . .
public void main (String[] args){
 ➜ Queue q;
    q = new Queue();

    q.put("Jack");
    q.put("Maya");
    q.get();
}
. . .
```



Getting input with a Queue

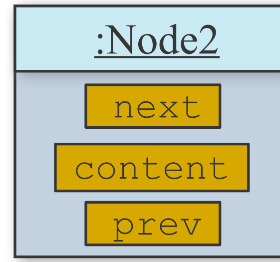
```
. . .
Queue list= new Queue();

int n = 0;

while(in.hasNextLine()){
    list.put(nextLine());
    n++;
}

String value[] = new String[n];
for(int i=0; i<n; i++)
value[i] = (String)list.get();
. . .
```

Deleting from Both Ends

- What if we need a linked list that allows addition and deletion at both ends?
- Using the Node class, deleting at the tail is expensive
  - we have to traverse the entire list to delete!
  - why?
- A similar problem arises if we want to delete a node in a linked list
- Solution: doubly linked nodes

The Doubly Linked Node2 Class

```java
class Node2 {
    private Node2 next, prev;
    private Object content;

    public void Node2() {
        next = null;
        prev = null;
        content = null;
    }

    public Object getContent() {
        return content;
    }

    public void setContent(Object c) {
        content = c;
    }

    ...
```




```java
    ...
    public Node2 getNext() {
        return next;
    }

    public void setNext(Node2 nextNode) {
        next = nextNode;
    }

    public Node getPrev() {
        return prev;
    }

    public void setPrev(Node2 prevNode) {
        prev = prevNode;
    }
}
```

The DoubleEndedQ class

```
public class DoubleEndedQ {
    private Node2 head, tail;

    public void DoubleEndedQ() {
        head = null;
        tail = null;
    }
    public void addToHead(Object c){
        Node2 n = new Node2();
        n.setContent(c);
        n.setPrev(null);
        n.setNext(head);
        head.setPrev(n);
        head = n;
    }
    . . .
```

```
. . .
public void deleteFromHead(){
    if(head== null)
        throw new Exception("Empty Queue");
    if(head==tail){
        head = tail = null;
    } else {
        head = head.getNext()
        head.setPrev(null);
    }
}
. . .
```

The DoubleEndedQ class (cont.)

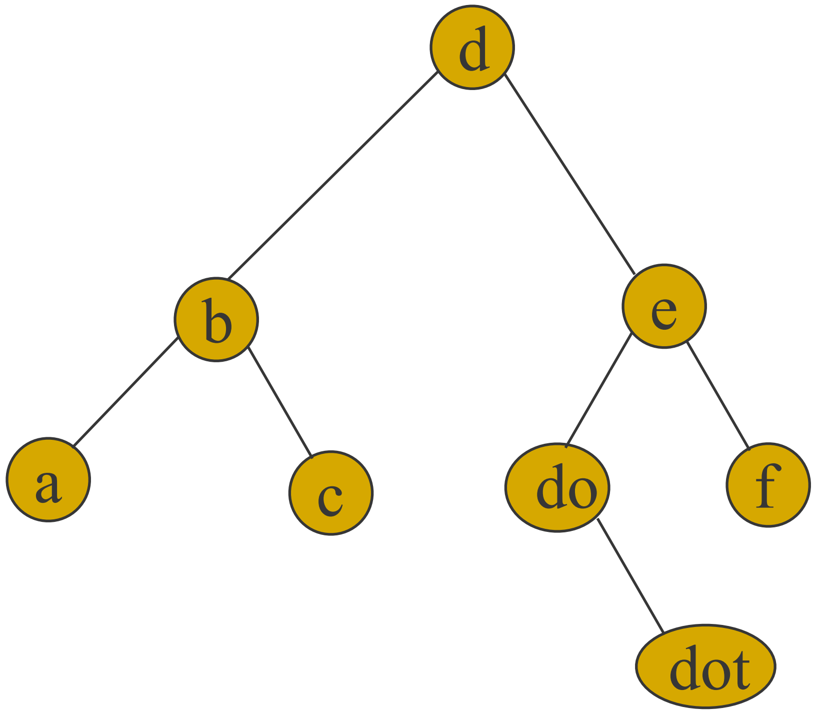
```
public void isEmpty() {
        return(head==null);
    }
    public void addToTail(Object c){
        Node2 n = new Node2();
        n.setContent(c);
        n.setNext(null);
        tail.setNext(n);
        n.setPrev(tail);
        tail = n;
    }

    public void deleteFromTail(){
        if(isEmpty())
            throw new Exception("Empty
Queue");
        if(head==tail){
            head = tail = null;
        } else {
            tail = tail.getPrev()
            tail.setNext(null);
        }
    }
}
```

Trees

- Trees are a very commonly used data structure in Computer Science
- For example, simple binary trees can be used to maintain a sorted list of strings
- Suppose we had an unknown number of strings to input, sort, then output
- We could use linked lists
  - Have to modify insert to ensure sorted order
- Or, we can use trees
  - more efficient

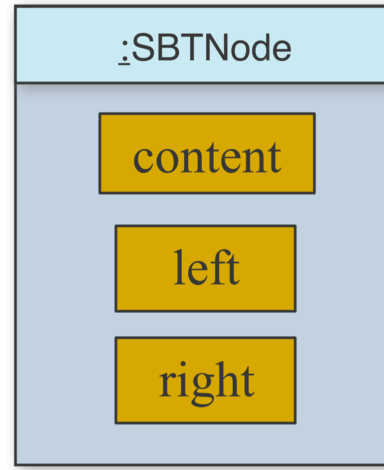
A Sorted Binary Tree

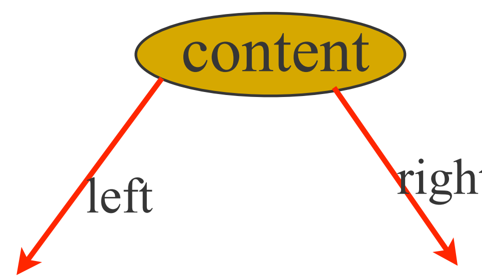
d, b, a, e, f, c, do, dot



The Sorted Binary Tree Node

```
public class SBTNode {
    private SBTNode left, right;
    private String content;

    public SBTNode (String c) {
        left = right = null;
        content = c;
    }
    public void insert(String c){
        if(c.compareTo(this.content) <= 0)
            if(left==null)
                left = new SBTNode(c);
            else
                left.insert(c);
        else
        if(right==null)
            right = new SBTNode(c);
        else
            right.insert(c );
    }
    . . .
```





Binary Tree Node Class (cont.)

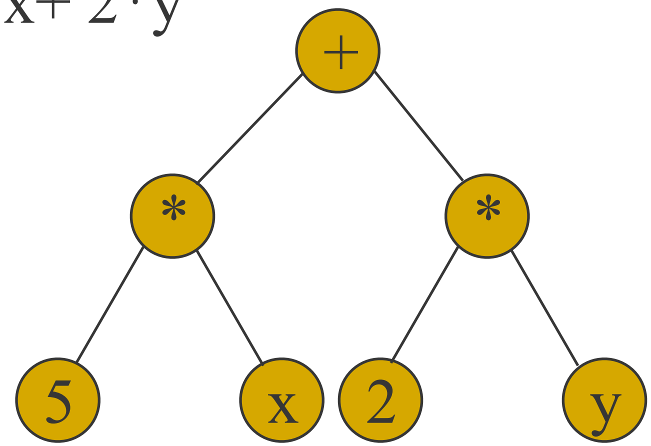
```
    . . .
    public void sortedPrint(){
        if(left!=null)
            left.sortedPrint();

        System.out.println(content);

        if(right!=null)
            right.sortedPrint();
    }
}
```

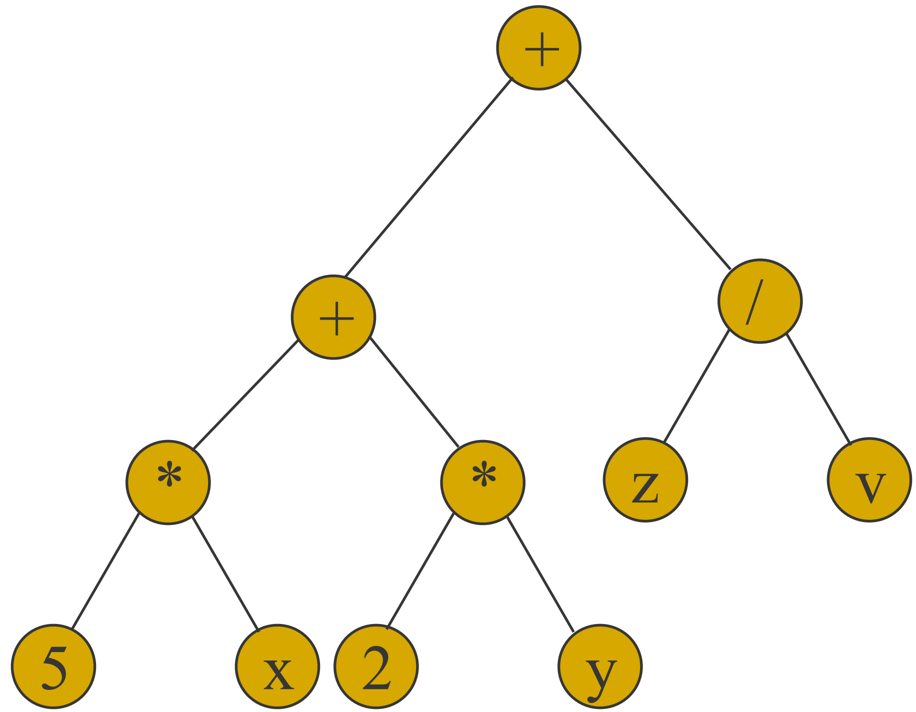
Example Use

```
    . . .
    public static void  main (String[] args) {

        SBTNode root = null;
        String input;

        input = JOptionPane.showInputDialog(null, "Enter String");
        if(input.length() > 0) {
             root = new SBTNode(input);

        while(true){
            input= JOptionPane.showInputDialog(null, "Enter String");
            if(input.length()<1)
                break;
            root.insert(input);
        }
       root.sortedPrint();
    }
}
```

Expression Trees

5*x+ 2*y



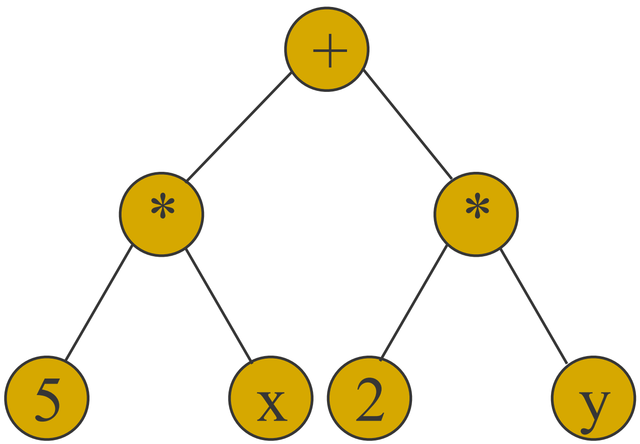


5*x+ 2*y + z/v

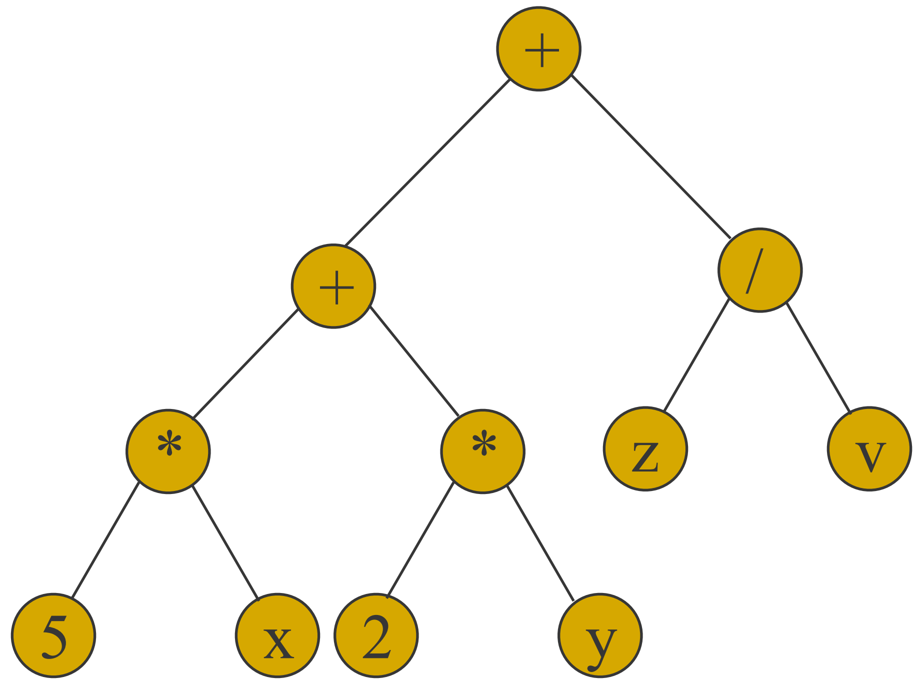
Tree traversal

- Visiting all the nodes of a tree is called a tree traversal
  - e.g., in earlier example, we visited and printed out each node
- Important types of traversals
  - In order: left child, parent, right child
    - e.g., sorted output
  - Post order: left child, right child, parent
    - e.g., to get postfix version of expression
    - useful for expression evaluation using a stack
  - Pre order: parent, left child, right child
    - e.g., to get prefix version of expression

Expression Tree Traversals



In: 5*x+ 2*y
Post: 5x*2y*+
Pre: +*5x*2y



In: 5*x+ 2*y + z/v
Post: 5x*2y*+zv/+
Pre: ++*5x*2y/zv

General Trees

- In general, trees may have
  - more than two children
  - no ordering among children
- They don't have:
  - cycles, multiple parents,
- More general structure: graph
- Applications:
  - class hierarchy,
  - index structures (databases)

Tree Traversal

```
public void inOrderPrint(){
        if(left!=null)
            left.print();
        System.out.println(content);
        if(right!=null)
            right.print();
    }

    public void preOrderPrint(){
        System.out.println(content);
        if(left!=null)
            left.print();
        if(right!=null)
            right.print();
    }
}
```

Searching

- Searching through a collection to see if a particular value is included is a common problem.
- Consider the case of an array of integers.
- We want to know if a particular integer value (key) is somewhere in this array.
- We could do
  - Linear search -- check each entry in turn (may have to see all entries)
  - Binary search -- similar to searching a dictionary

Binary Search

- Think about how we search for a word in a dictionary
- Binary search is similar:
  - Check the “middle” entry of the collection
  - If not found, search before or after
  - With each test we cut the task in half
  - Allows us to quickly search through very large collections
- But: data must be sorted and accessible by index (e.g., a sorted array)

Binary Search

```
boolean binarySearch(int[] data, int key) {

    int start = 0, end = data.length - 1, mid;

    while (start <= end) {
        mid = (start + end) / 2;

        if (data[mid] == key)
            return true;

        if (data[mid] < key)
            start = mid + 1;
        else
            end = mid - 1;
    }
    return false;
}
```

Recursive Binary Search

```
boolean binarySearch (int[] data, int key, int start,  int end) {

    int mid = (start + end)/2;

    if(start <= end)
        return false;

    if(data[mid] == key)
        return true;

    if(data[mid] < key)
        return binarySearch(data, key, mid+1, end);
    else
        return binarySearch(data, key, start, mid-1);
}
```

Search Performance

- Given N items, how many comparisons before we find the data in the worst case
- Linear Search
  - may have to test all entries: N
- Binary Search
  - each test halves the remaining data to test
  - worst case: $\log_2 N$
  - for 1000,000 items, only 20 comparisons at most!
  - but, data must already be sorted!