COMP9024: Data Structures and Algorithms

Week Three: Linked Lists and Recursion

Hui Wu

Session 2, 2015

http://www.cse.unsw.edu.au/~cs9024

Outline

- Singly Linked Lists
- Doubly Linked Lists
- Recursions

Data Structures and Algorithms

- Data structures
  - A **data structure** is a way of storing data in a computer so that it can be used efficiently.
- Algorithms
  - An **algorithm** is a finite sequence of well-defined instructions for solving a problem and guaranteed to terminate in a finite time.
  - An algorithm does not necessarily need to be executable in a computer, but can be converted into a program.

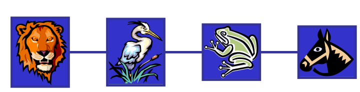
Algorithms versus Heuristics

- Heuristics
  - A **heuristic** is a finite sequence of well-defined instructions for partially solving a hard problem and guaranteed to terminate in a finite time.
  - A heuristic is not always guaranteed to find a solution while an algorithm is.
- Descriptions of algorithms
  - We will use Java-like pseudo code mixed with English to describe algorithms.

Abstract Data Types

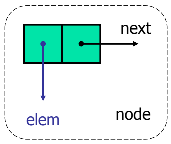
- An **Abstract Data Type** (**ADT**) is a specification of a set of data and the set of operations that can be performed on the data.
  - Such a data type is abstract in the sense that it is independent of various concrete implementations.
- The definition can be mathematical, or it can be programmed as an interface.
- We need to implement an ADT by using a class to make it usable.

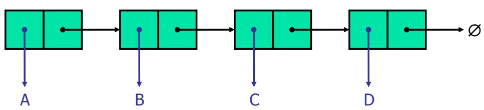
Linked Lists



Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





The Node Class for List Nodes

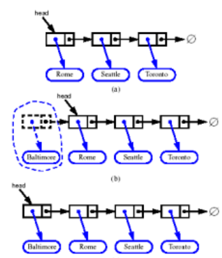
```
/** Node of a singly linked list of strings. */
public class Node {
   private String element; // we assume elements are character strings
   private Node next; /** Creates a node with the given element and next node. */
   public Node(String s, Node n) {
      element = s;
      next = n;                          }

    /** Returns the element of this node. */
   public String getElement() { return element; }

   /** Returns the next node of this node. */
   public Node getNext() { return next; } // Modifier methods:

   /** Sets the element of this node. */
   public void setElement(String newElem) { element = newElem; }

   /** Sets the next node of this node. */
   public void setNext(Node newNext) { next = newNext; } }
```


The Class for a Singly Linked List

```
/** Singly linked list .*/
public class SLinkedList {
   protected Node head; // head node of the list
   protected long size; // number of nodes in the list
   /** Default constructor that creates an empty list */
   public SLinkedList() {
     head = null; size = 0;
                       }
       // ... update and search methods would go here ...
                         }
```


Inserting at the Head (1/2)

1. Allocate a new node
2. Insert the new element
3. Have the new node point to old head
4. Update head to point to the new node

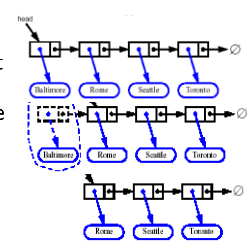


Inserting at the Head (2/2)

```
Algorithm addFirst(v)
 {
    v.setNext(head);   // make v  point  to the old head node
    head = v;          // make head point to the new node
    size =size+1;      // increment the node count
 }
```


Removing at the Head (1/2)

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

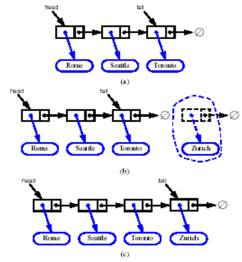


Removing at the Head (2/2)

```
Algorithm removeFirst()
 {
  if ( head = null )
     Indicate an error: the list is empty;
  t = head ;
   /* make head point to the next node (or null) */
  head = head.getNext();
  t.setNext(null);   //null out the next pointer of the removed node
  size = size - 1;  //decrement the node count
 }
```


Inserting at the Tail (1/2)

1. Allocate a new node
2. Insert the new element
3. Have the new node point to null
4. Have the old last node point to new node
5. Update tail to point to new node

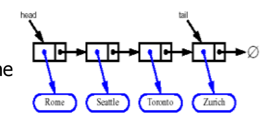


Inserting at the Tail (2/2)

```
Algorithm addLast(v)
 {
  v.setNext(null);    //make the new node v point to null object
  tail.setNext(v);    //make the old tail node point to the new node
  tail = v ;          //make tail point to the new node
  size = size +1;     //increment the node count
 }
```


Removing at the Tail (1/2)

- Removing the tail node of a singly linked list cannot be done in constant time.
- We need to traverse the whole linked list to remove the tail node, which takes O(n) time, where n is the number of nodes in the linked list.

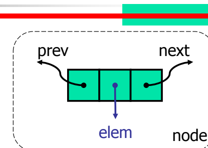


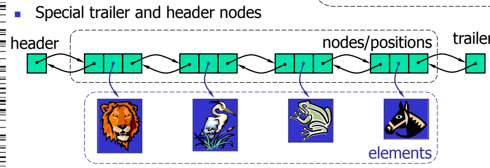
Removing at the Tail (2/2)

```
Algorithm removeLast()
 {
  if ( size = 0 )
    Indicate an error: the list is empty;
  else
   { if ( size = 1 ) // only one node
       head = null;
     else // at least two nodes
      { t = head;          /* make t  point to the head */
        while ( t.getNext != null ) /* find the last node */
           { s = t;
             t = t.getNext(); }
        s.setNext(null);  //null out the next pointer of the last node
      }
     size = size - 1;       //decrement the node count
   }
 }
```


Doubly Linked List

- A doubly linked list is a concrete data structure consisting of a sequence of nodes.
- Each node stores:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

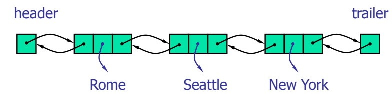




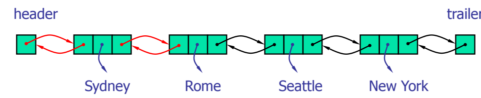
The Node Class of Doubly Linked List

```
/** Node of a doubly linked list of strings */
  public class DNode {
    protected String element; // String element stored by a node
    protected DNode next, prev; // Pointers to next and previous nodes
     /** Constructor that creates a node with given fields */
    public DNode(String e, DNode p, DNode n) {
        element = e;
        prev = p;
        next = n; } /** Returns the element of this node */
    public String getElement() { return element; } /** Returns the previous node of this
     node */
    public DNode getPrev() { return prev; } /** Returns the next node of this node */
    public DNode getNext() { return next; } /** Sets the element of this node */
    public void setElement(String newElem) { element = newElem; }
     /** Sets the previous node of this node */
    public void setPrev(DNode newPrev) { prev = newPrev; }
     /** Sets the next node of this node */
    public void setNext(DNode newNext) { next = newNext; }
                              }
```


Inserting at the Head (1/2)



(a) Before the insertion

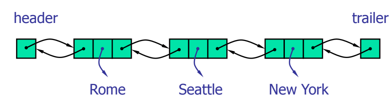


(b) After the insertion

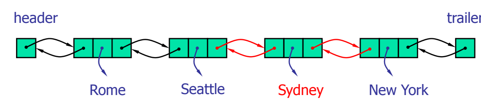
Inserting at the Head (2/2)

```
Algorithm addFirst(v)
  {
    w = header.getNext();  // make w point to the first node
    v.setNext(w);           // make v point to the old first node
    v.setPrev(header);     // make v point to the header
    w.setPrev(v);          // make the old first node point to v
    header.setNext(v);     // make header point to v
    size = size+1;         // increment the node count
  }
```


Inserting in the Middle (1/2)



(a) Before the insertion

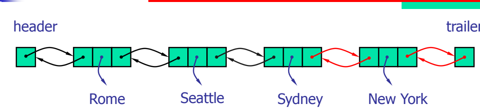


(b) After the insertion

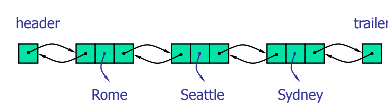
Inserting in the Middle (2/2)

```
Algorithm addAfter(v,z)
 {
   w=v.getNext();   // make w point to the node after v
   z.setPrev(v);    // link z  to its predecessor, v
   z.setNext(w);    // link z to its successor, w
   w.setPrev(z);    // link w  to its new predecessor, z
   v.setNext(z);    // link v  to its new successor, z
   size = size+1;   // increment the node count
 }
```


Removing at the Tail (1/2)



(a) Before the deletion



(b) After the deletion

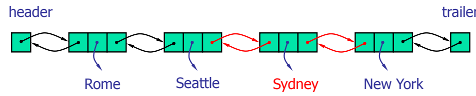
Removing at the Tail (2/2)

Algorithm removeLast():

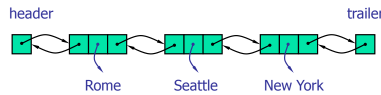
```
{
   if ( size = 0 )
      Indicate an error: this list is empty;
   v = trailer.getPrev()   // make v point to the last node
   u = v.getPrev();        // u points to the node before the last node
   trailer.setPrev(u);
   u.setNext(trailer);
   v.setPrev(null);
   v.setNext(null);
   size = size -1;
}
```

25

Removing in the Middle (1/2)



(a) Before the deletion



(b) After the deletion

26

Removing in the Middle (2/2)

Algorithm remove(v):

```
{
   u = v.getPrev()     // make u point to the node before v
   w = v.getNext();    // w points to the node after v
   w.setPrev(u);       // link out v
   u.setNext(w);
   v.setPrev(null);    // null out the pointers of v
   v.setNext(null);
   size = size -1;     // decrement the node count
}
```

27

Programming with Recursion

28

The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$$

- As a Java method:

```
// recursive factorial function
public static int recursiveFactorial(int n) {
   if (n == 0) return 1;        // basis case
   else return n * recursiveFactorial(n- 1);        // recursive case
}
```

29

Content of a Recursive Method

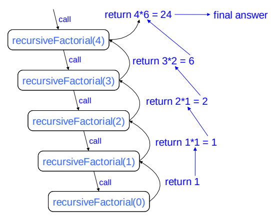
- **Base case(s)**
  - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  - Every possible chain of recursive calls **must** eventually reach a base case.
- ***Recursive calls***
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

30

Visualizing Recursion

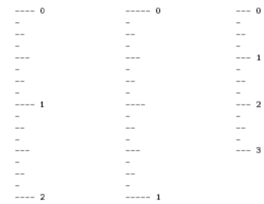
- Recursion trace
- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

Example recursion trace:



Example – English Rulers

- Define a recursive way to print the ticks and numbers like an English ruler:



```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

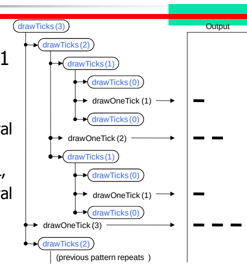
```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```


A Recursive Method for Drawing Ticks on an English Ruler

```
    // draw a tick with no label
public static void drawOneTick(int tickLength) { drawOneTick(tickLength, - 1); }
    // draw one tick
public static void drawOneTick(int tickLength, int tickLabel) {
  for (int i = 0; i < tickLength; i++)
     System.out.print("-");
  if (tickLabel >= 0) System.out.print(" " + tickLabel);
  System.out.print("\n");
}
public static void drawTicks(int tickLength) { // draw ticks of given length
  if (tickLength > 0) {                 // stop when length drops to 0
     drawTicks(tickLength- 1);           // recursively draw left ticks
     drawOneTick(tickLength);            // draw center tick
     drawTicks(tickLength- 1);           // recursively draw right ticks
  }
}
public static void drawRuler(int nInches, int majorLength) { // draw ruler
  drawOneTick(majorLength, 0);          // draw tick 0 and its label
  for (int i = 1; i <= nInches; i++)    {
     drawTicks(majorLength- 1);          // draw ticks for this inch
     drawOneTick(majorLength, i);        // draw tick i and its label
  }
}
```


Visualizing the DrawTicks Method

- An interval with a central tick length $L \geq 1$ is composed of the following:
  - an interval with a central tick length L-1,
  - a single tick of length L,
  - an interval with a central tick length L-1.



Using Recursion

Recall the Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a Java method:

```
// recursive factorial function
public static int recursiveFactorial(int n) {
   if (n == 0) return 1;          // basis case
   else return n * recursiveFactorial(n- 1);      // recursive case
}
```

Linear Recursion

- **Test for base cases.**
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls **must** eventually reach a base case, and the handling of each base case should not use recursion.
- ***Recur once.***
  - Perform a single recursive call. (This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.)
  - Define each possible recursive call so that it makes progress towards a base case.

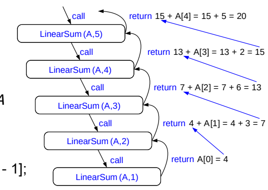
37

A Simple Example of Linear Recursion

Algorithm LinearSum(A, n)
{
Input :
A integer array A and an integer n such that A has at least n elements
Output :
The sum of the first n integers in *A*

if (n = 1) **return** A[0];
else
return LinearSum(A, n - 1) + A[n - 1];
}

Example recursion trace:



38

Reversing an Array

Algorithm ReverseArray(A, i, j)
{
Input : An array A and nonnegative integer indices i and j
Output : The reversal of the elements in A starting at index i and ending at j
if (i < j)
{
swap A[i] and A[j];
ReverseArray(A, i + 1, j - 1);
}
}

39

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(*A, i, j*), not ReverseArray(*A*).

40

Computing Powers

- The power function, p(x,n)=x^n, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to an power function that runs in O(n) time (for we make n recursive calls).
- We can do better than this, however.

41

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

- For example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$
$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$
$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$
$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$

42

A Recursive Squaring Method

Algorithm Power(*x, n*)
{
Input : A number *x* and integer *n* = 0
Output : The value x^n
if (n = 0) **return** 1;
if (n is odd)
{ y = Power(x, (n - 1)/ 2);
return x*y*y; }
else
{ y = Power(x, n/ 2);
return y*y; }
}

Analyzing the Recursive Squaring Method

Algorithm Power(*x, n*)
{
Input : A number *x* and integer *n* = 0
Output : The value x^n
if (n = 0) **return** 1;
if (n is odd)
{ y = Power(x, (n - 1)/ 2);
return x*y*y; }
else
{ y = Power(x, n/ 2);
return y*y; }
}

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we used a variable twice here rather than calling the method twice.

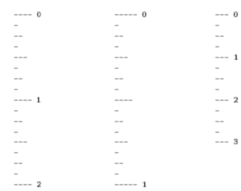
Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j)
{ **Input :** An array A and nonnegative integer indices i and j
Output : The reversal of the elements in *A* starting at index i and ending at j
while (i < j)
{ swap A[i] and A[j];
i = i + 1;
j = j – 1;
}
}

Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Example: the DrawTicks method for drawing ticks on an English ruler.



A Binary Recursive Method for Drawing Ticks

```
    // draw a tick with no label
public static void drawOneTick(int tickLength) { drawOneTick(tickLength, - 1); }
    // draw one tick
public static void drawOneTick(int tickLength, int tickLabel) {
  for (int i = 0; i < tickLength; i++)
     System.out.print("-");
  if (tickLabel >= 0) System.out.print(" " + tickLabel);
  System.out.print("\n");
}
public static void drawTicks(int tickLength) { // draw ticks of given length
  if (tickLength > 0) {                 // stop when length drops to 0
     drawTicks(tickLength- 1);          // recursively draw left ticks
     drawOneTick(tickLength);           // draw center tick
     drawTicks(tickLength- 1);          // recursively draw right ticks
   }
}
public static void drawRuler(int nInches, int majorLength) { // draw ruler
  drawOneTick(majorLength, 0);          // draw tick 0 and its label
  for (int i = 1; i <= nInches; i++)    {
     drawTicks(majorLength- 1);         // draw ticks for this inch
     drawOneTick(majorLength, i);       // draw tick i and its label
   }
}
```

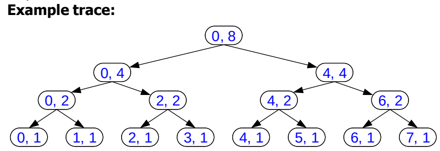
Note the two recursive calls

Another Binary Recursive Method

- Problem: add all the numbers in an integer array A:

Algorithm BinarySum(A, i, n)
{ **Input :** An array A and integers i and n
Output : The sum of the n integers in A starting at index i
if (n = 1) **return** A[i];
return BinarySum(A, i, n/ 2) + BinarySum(A, i + n/ 2, n/ 2);
}

- **Example trace:**



Computing Fibanacci Numbers

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- As a recursive algorithm (first attempt):

```
Algorithm BinaryFib(k)
  { Input : Nonnegative integer k
    Output : The kth Fibonacci number F_k
     if ( k = 1) return k;
     else
        return BinaryFib(k - 1) + BinaryFib(k - 2);
  }
```


Analyzing the Binary Recursion Fibonacci Algorithm

- Let n_k denote number of recursive calls made by BinaryFib(k). Then
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that the value at least doubles for every other value of n_k. That is, $n_k > 2^{k/2}$. It is exponential!

A Better Fibonacci Algorithm

- Use linear recursion instead:

```
Algorithm LinearFibonacci(k):
  { Input : A nonnegative integer k
    Output : Pair of Fibonacci numbers (F_k, F_k-1)
     if  ( k = 1) return (k, 0);
     else
       {
         (i,  j)  =  LinearFibonacci(k - 1);
         return (i +j, i);
       }
  }
```

- Runs in O(k) time.

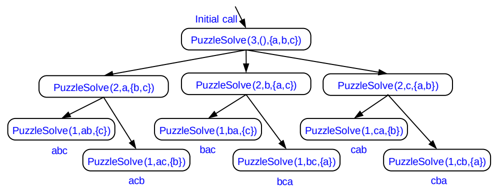
Multiple Recursion

- Motivating example: summation puzzles
  - *pot + pan = bib*
  - *dog + cat = pig*
  - *boy + girl = baby*
- Multiple recursion: makes potentially many recursive calls (not just one or two).

Algorithm for Multiple Recursion

```
Algorithm PuzzleSolve(k,S,U)
{
  Input: An integer k, sequence S, and set U (the universe of elements to test)
  Output:  An enumeration of all k-length extensions to S using elements in U
     without repetitions
  for all e  in U do
   { Remove e from U;          // e is now being used
     Add e to the end of S;
     if  (k = 1)
        {
         Test whether S is a configuration that solves the puzzle;
         if ( S solves the puzzle ) return "Solution found: " S;
        }
     else
          PuzzleSolve(k - 1, S,U);
     Add e back to U;          // e is now unused
     Remove e from the end of S;
   }
}
```


Visualizing PuzzleSolve



References

1. Chapter 3, Data Structures and Algorithms by Goodrich and Tamassia.
2. Garbage Collection by Bill Venners (http://www.artima.com/insidejvm/ed2/gc.html).