CS 251 – Fall 2014

Week 1

PURDUE UNIVERSITY

1

Reminders

- If you have a PSO next Monday, attend another one next week if you can
- Course home on the web linked from my home page
- Register your i>Clicker in BBoard
- Sign up to Piazza
- Visit the course wiki to see slides and (soon) assignments

2

180/182 reviews continued

1. Recursion
2. Stacks & Queues – Text, section 1.3

3

Recursion

- Recursion is a powerful way of thinking about and solving a problem.
- Recursion means "defining something in terms of itself" at some smaller scale, perhaps multiple times.
- Recursion can lead to conceptually simple solutions.
  - Recursion is typically used when a problem can be broken down into independent sub-tasks that are combined after they have been completed.
- You have probably seen recursive definitions and a few recursive algorithms (180, 182, 240)
  - Definition of factorial, Fibonacci numbers
  - Euclid's algorithm for gcd
  - Binary search, Mergesort, Quicksort

4

Recursive definitions

A recursive definition is a definition with three parts:

1. Base case(s)
2. A recursive definition
3. A closure clause in many cases

- S = {2, 4, 8, 16, 32, …}
  - S(1) = 2 base case
  - S(n) = 2S(n – 1) for n ≥ 2
- n! = 1· 2· 3· ⋯ · (n-1)· n
  - f(0) = 1 base case
  - f(n) = n· f(n-1) for n > 0
- Recursive definitions and recursive programs lend themselves to proof by induction

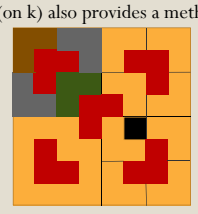
5

Recall: Tiling a board with triominos

Board size $2^k \times 2^k$, one square removed

Tiles L-shaped triominos

Induction proof (on k) also provides a method to tile the board

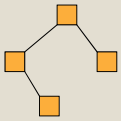


6

Binary tree from string

1. T = '( )' denotes the empty binary tree
2. If T1 and T2 denote binary trees, then so does '('T1' 'T2')'
3. Nothing else denotes a binary tree

Example: '((( ) (( ) ( ))) (( ) ( )))'

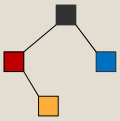


7

Binary tree from string

- T = '( )' denotes the empty binary tree
- If T1 and T2 denote binary trees, then so does '('T1' 'T2')'
- Nothing else denotes a binary tree

Example: **'((( ) (( ) ( ))) (( ) ( )))'**



8

Binary tree, all nodes labeled

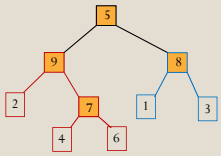
1. A leaf is a binary tree, and
   T = '(*n*)' denotes a leaf labeled with integer *n*
2. If T1 and T2 denote binary trees, then so do
   '(n , T1 , T2)'
   '(n , , T2)'
   '(n , T1 ,)'
3. Nothing else denotes a binary tree

9

Example

Top-down:

'(5 , (9 , (2) , (7 , (4) , (6))) , (8 , (1) , (3)))'



10

Pointer-based internal representation

- Tree node T = {ref T, int, ref T}
  T.left, T.label, T.right
- Tree is a pointer to the tree root, of type ref T

11

Useful?

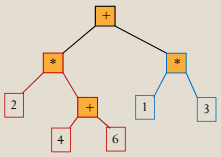
- String to tree conversion
- Recursive definition corresponds to the recursive structure of a program that makes that conversion
- A similar recursive program converts binary trees to strings (for archiving, or exchange with other programs,…)
- Prefix expressions yielding an expression tree

12

PL Example

Top-down:

'(+ , (* , (2) , (+ , (4) , (6))) , (* , (1) , (3)))'



+ * 2 + 4 6 * 1 3 equivalent to 2(4+6)+1*3

13

How to write recursive programs?

- Define the problem is terms of having solutions to smaller problem instances
  - Define the base case(s)
  - Identify the smaller problem instance(s)
  - Define how to combine the solutions returned from the smaller problems
- Define the methods in ways that facilitate recursion. This sometimes requires we define additional parameters that are passed to the method.
  - For example, define the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).

14

Example: Reversing entries in an array

- Input: An array A and nonnegative integer indices i and j
- Output: Reversal of the elements in A starting at index i and ending at j

```
public static void ReverseArray(A, i, j):
  if i < j then
    Swap A[i] and A[ j]
    ReverseArray(A, i + 1, j - 1)
  return
```

Where is the base case?

15

Every recursive program has a corresponding non-recursive one

In some cases, the non-recursive programs are easy to generate and look very similar (e.g., in programs with "tail recursion")

```
public static void IterativeReverseArray(A, i, j ):
    while i < j do
        Swap A[i ] and A[ j ]
        i = i + 1
        j = j - 1
    return
```

16

Computing Fibonacci numbers

Recursive algorithm (first attempt):

- Input: Nonnegative integer k
- Output: The k-th Fibonacci number

```
public static int calcFibRec(k):
  if k = 1 then
    return 1
  else
    return calcFibRec(k - 1) + calcFibRec(k - 2)
```

17

#Version 1

```
def calcFib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        an_0 = 0
        an_1 = 1
        for i in range(n):
            temp = an_0
            an_0 = an_1
            an_1 = temp + an_1
        return an_0
```

#Version 2

```
def calcFibRecur(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return calcFibRecur(n-1) + calcFibRecur(n-2)
```

#Version 3

```
memo = {0:0, 1:1}
def improvedFibRecur(n):
    if not n in memo:
        memo[n] = improvedFibRecur(n-1) + improvedFibRecur(n-2)
    return memo[n]
```

18

	Version 2 (recursive)	Version 1 For-loop
n=10	0.000109	0.000006
n=11	0.000147	0.000006
n=12	0.000236	0.000006
n=13	0.000354	0.000006
n=14	0.000785	0.000008
n=15	0.001114	0.000007
n=16	0.001788	0.000006
n=17	0.002507	0.000006
n=18	0.003686	0.000007
n=19	0.005941	0.000007
n=20	0.010097	0.000007
n=21	0.014925	0.000007
n=22	0.024313	0.000009
n=23	0.038561	0.000009
n=24	0.081620	0.000008
n=25	0.103057	0.000018
n=26	0.186741	0.000010
n=27	0.275817	0.000009
n=28	0.423574	0.000012
n=29	0.705648	0.000010

19

	Version 1 For-loop	Version 3 (recursive)
n=10	0.000007	0.000004
n=11	0.000019	0.000003
n=12	0.000009	0.000005
n=13	0.000009	0.000005
n=14	0.000009	0.000005
n=15	0.000009	0.000004
n=16	0.000009	0.000006
n=17	0.000007	0.000003
n=18	0.000011	0.000006
n=19	0.000012	0.000005
n=20	0.000008	0.000003
n=21	0.000008	0.000005
n=22	0.000011	0.000005
n=23	0.000012	0.000005
n=24	0.000008	0.000004
n=25	0.000012	0.000005
n=26	0.000012	0.000004
n=27	0.000013	0.000005
n=28	0.000010	0.000004
n=29	0.000010	0.000003

Most common recursion mistakes

- Base case for terminating the recursion is missing or is not correct
- No termination
  - Problem size does not decrease properly
  - Will typically have a stack overflow
- Combining the data returned by recursive calls is not done correctly (often a logical error)
- Slow performance due to poor programming style
  - Make sure there are no unnecessary recomputations
  - Understand the number of subproblems created

20

Recursive programs you will see/write

- Binary search
- Sorting algorithms
- Computations on trees
- Queries on search trees
- Traversals of graphs
- … and more

21

CQ: Which statement(s) are true?

1. Every decimal integer can be represented in binary.
2. Every decimal number can be represented in binary.

A. Both
B. only 1
C. only 2
D. Neither

required reading

1.3 (Bags,) Queues, and Stacks

- queues
- stacks
- dynamic resizing
- doubly-linked lists
- iterators
- applications

Review: Arrays

- In Java, an array is an indexed collection of data values of the same type.
- Array declaration and creation

```
<data type> [ ] <variable>
<variable> = new <data type> [ <size> ]
```

An array is like an object

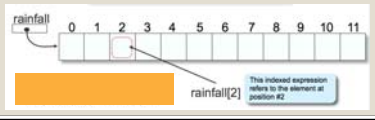
- Example:

```
double[ ] rainfall;
rainfall = new double[12];
```

- Like other data types, it is possible to declare and initialize an array at the same time:

```
int[] number = { 2, 4, 6, 8 };
```

- Individual elements in an array are accessed with an index expression.



24

Review: Arrays (2)

Most common pitfalls

- Index out of bounds
  - The index for an array A, must evaluate to a value between 0 and A.length-1.
  - If it does not, an "ArrayIndexOutOfBoundsException" is thrown
- Elements of an array are not initialized by default
  - Make sure it happens before using its values

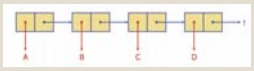
To review, read your 180 book or some other resource

- **We will use 1- and 2-dimensional arrays**
  - Higher dimensional arrays are used in many applications
  - For sparse data, they become expensive

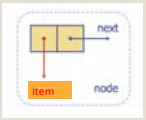
25

Review: Linked lists

- A singly linked list is a data structure consisting of a sequence of nodes



- Each node stores:
- Item
- Link to the next node



26

Review: Linked list

- When creating a linked list
- Need a reference variable, **`first`**, that identifies the first node in the list

- Traversing a linked list
- Once you are at the first node, you can use **`node.getNext()`** to get to the next node
- Scan a linked list by assigning a variable **`curr`** to the value of **`first`**, then use the **`node.getNext()`** method of each node to proceed down the list
- Conclude when **`curr == null`**

- Other details
- If a linked list is empty, then **`first`** value is **`null`**
- Inserting or deleting an element at the front of the list is easy, because the list maintains a reference that points to the first element

27

Stacks and queues

- Values: collections of objects.
  - Operations: insert, remove, test if empty.
  - Insertion is clear; removal is different

LIFO = "last in first out"

Stack. Remove the item most recently added.

- Analogy. Cafeteria trays, Web surfing.

FIFO = "first in first out"

Queue. Remove the item least recently added.

- Analogy. Registrar's line.

28

Client, implementation, interface

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find, ...

Benefits.

- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

Client: program using operations defined in interface.
Implementation: actual code implementing operations.
Interface: description of data type, basic operations.

Queue API

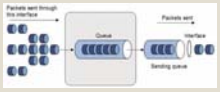
public class QueueOfStrings	
`QueueOfStrings()`	*create an empty queue*
`void enqueue(String s)`	*insert a new item onto queue*
`String dequeue()`	*remove and return the item least recently added*
`boolean isEmpty()`	*is the queue empty?*
`int size()`	*number of items on the queue*

30

Queue applications

- Familiar applications.
  - iTunes playlist.
  - Data buffers (iPod, TiVo).
  - Asynchronous data transfer (file IO, pipes, sockets).
  - Dispensing requests on a shared resource (printer, processor).
- Modeling and simulations of the real world.
  - Traffic analysis.
  - Waiting times of customers at call center.
  - Determining number of cashiers to have at a supermarket.

It is not always just first-in first-out!



31

Queue: linked-list implementation

- Maintain one pointer `first` to first node in a singly-linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.

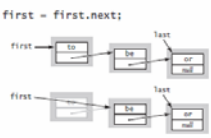
Queue dequeue: linked-list implementation

```
inner class
private class Node
{
   String item;
   Node next;
}
```

```
save item to return
   String item = first.item;

delete first node
   first = first.next;

return saved item
   return item;
```



Remark. Identical code to linked-list stack `pop()`.

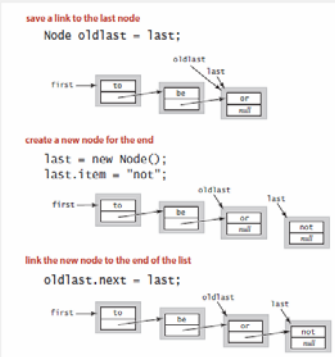
Queue enqueue: linked-list implementation

```
inner class
private class Node
{
   String item;
   Node next;
}
```

```
save a link to the last node
   Node oldlast = last;

create a new node for the end
   last = new Node();
   last.item = "not";

link the new node to the end of the list
   oldlast.next = last;
```



Queue: linked-list implementation in Java

```
public class LinkedQueueOfStrings
{
   private Node first, last;

   private class Node
   { /* same as in LinkedStackOfStrings */ }

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

   public void enqueue(String item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

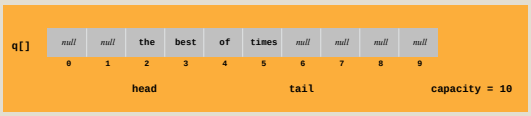
special cases for empty queue

35

Queue: dynamic array implementation

Array implementation of a queue.

- Use array q[] to store items in queue.
- enqueue(): add new item at q[tail].
- dequeue(): remove item from q[head].
- Update head and tail modulo the capacity.
- Add dynamic resizing.



Dynamic arrays; e.g., for stacks

Have client give an estimate of maximum size?

- unrealistic!

Increase/decrease array size by 1 as needed?

Too expensive:

- Need to copy all item to a new array.
- Inserting N items (no dequeue operations) into an empty array takes time proportional to $1 + 2 + \ldots + N \sim N^2 / 2$.

Grow and shrink by more than 1…

37

Dynamic-array implementation

- Q. How to grow array?
- A. If array is full, create a new array of **twice the size**, and copy items.

"repeated doubling"

```
public StackOfStrings() {  s = new String[1];  }
public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}
private void resize(int capacity)
{
   String[] copy = new String[capacity];
   for (int i = 0; i < N; i++)
      copy[i] = s[i];
   s = copy;
}
```

cost of array resizing is now $2 + 4 + 8 + \ldots + N \sim 2N$

- Consequence. Inserting first N items takes time proportional to N (not N^2).

38

Important concepts underlying many data structures and algorithms

Repeated (recursive) doubling

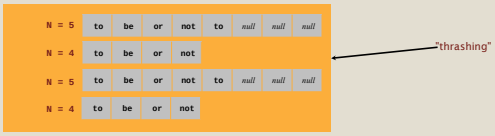
- Effort is doubling at every step when starting with 1 and reaching N and "paying" according to current size. What is the total effort?
- $1 + 2 + 4 + 8 + \ldots + N/2 + N \sim 2N$
- $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \sim 2N$

- Repeated halving
- Starting with N pebbles and giving half away in each step, after how many steps are we left with one pebble?
- When N is a power of 2: N, N/2, N/4, N/8, N/16, … 4, 2, 1
- $\log_2 N$ steps

39

Dynamic-array implementation

- Q. How to shrink array?
  - First try – same approach as in growing the array .
    - Insertion: double size of `s[]` when array is full.
    - Deletion: halve size of `s[]` when array is one-half full.



- Too expensive.
  - Consider push-pop-push-pop… sequence in a stack when array is full.
  - Takes time proportional to N per operation in the worst case.

40

Dynamic-array implementation

- Q. How to shrink array?

- Efficient solution.
- **`Insertion:`** double size of `s[]` when array is full.
- **`Deletion:`** halve size when array is one-quarter full.

```
public String pop()
{   String item = s[--N];    s[N] = null;
   if (N > 0 && N == s.length/4) resize(s.length / 2);
   return item;
}
```

- Invariant. Array is between 25% and 100% full.

41

Dynamic-array implementation: performance

- Amortized analysis. Average running time per operation over a worst-case sequence of operations.

- Claim 1: Starting from empty stack (with dynamic resizing), any sequence of M push and pop operations takes time proportional to M.

- Claim 2: Starting from empty queue (with dynamic resizing), any sequence of M enqueue and dequeue operations takes time proportional to M.

doubling and shrinking

	best	worst	amortized
construct	1	1	1
push	1	N	1
pop	1	N	1
size	1	1	1

running time for doubling stack with N items

42

Stack dynamic array implementation: memory usage

- Proposition. Uses between ~ 4 N and ~ 16 N bytes to represent a stack with N items.
- ~ 4 N when full.
- ~ 16 N when one-quarter full.

```
public class DoublingStackOfStrings
{
   private String[] s;
   private int N = 0;   …
}
```

4 bytes × array size

4 bytes

- Remark. Analysis includes memory for the stack (but not the strings themselves, which the client owns).

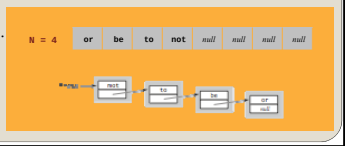
43

Dynamic array vs. linked List

Tradeoffs. Can implement a stack or queue with either dynamic array or linked list; client can use interchangeably.

Which one is better?

- Linked-list implementation.
  - Every operation takes constant time in the worst case.
  - Uses extra time and space to deal with the links.
- Dynamic-array implementation.
  - Every operation takes constant amortized time.
  - Less wasted space.



Stack API

- Warmup. Stack of strings objects.

public class StackOfStrings	
StackOfStrings()	*create an empty stack*
void push(String s)	*insert a new item onto stack*
String pop()	*remove and return the item most recently added*
boolean isEmpty()	*is the stack empty?*
int size()	*number of items on the stack*

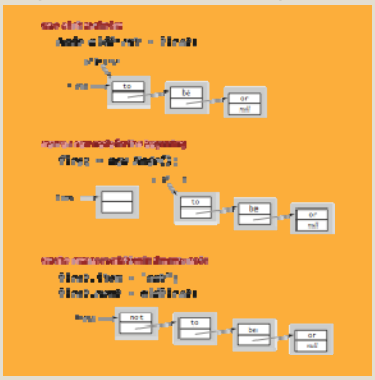
- Challenge. Reverse sequence of strings from standard input.

45

Stack pop: linked-list implementation

46

Stack push: linked-list implementation



47

Stack: linked-list implementation in Java

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

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

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      if (isEmpty()) throw new RuntimeException();
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

inner class

stack underflow

48

Stack: linked-list implementation performance

- Proposition. Using a linked-list implementation of a stack, every operation takes constant time in the worst case.
- Proposition. Uses ~ 16 N bytes to represent a stack with N items.

```
private class Node{
   String item;
   Node next;
}
```

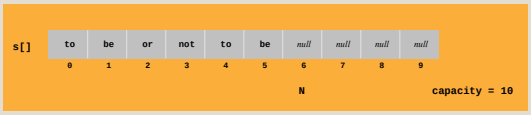
8 bytes (object overhead)
4 bytes (reference to String)
4 bytes (reference to Node)
16 bytes per stack item

- Remark. Analysis includes memory for the stack (but not the strings themselves, which the client owns).

49

Stack: array implementation

- Array implementation of a stack.
- Use array `s[]` to store `N` items on stack.
- `push()`: add new item at `s[N]`.
- `pop()`: remove item from `s[N-1]`.



- Defect. Stack overflows when `N` exceeds capacity. [stay tuned]

50

Stack: array implementation

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;      public
StackOfStrings(int capacity)
   {  s = new String[capacity];  }
public boolean isEmpty()
   {  return N == 0;  }      public void
push(String item)
   {  s[N++] = item;  }      public String
pop()
   {  return s[--N];  }}
```

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   return item;
}
```

this version avoids "loitering":
garbage collector reclaims memory
only if no outstanding references

decrement N;
then use to index into array

51

Stack applications

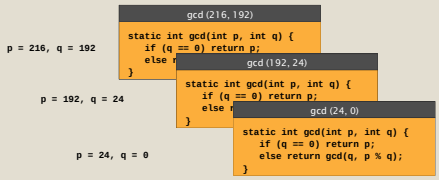
- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.
- …

52

Function calls

- How a compiler implements a function.
- Function call: push local environment and return address.
- Return: pop return address and local environment.

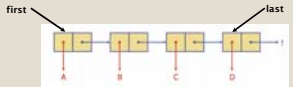
- Recursive function. Function that calls itself.
- Note. Can always use an explicit stack to remove recursion.



53

More on singly-linked lists

- Note that the queue implementation needed a pointer to the end of the list (last) as well as the beginning of the list (first)
- It is easy to maintain the pointers to both ends of the list if all we do is remove from the front and insert at the end

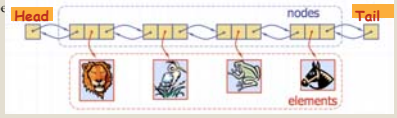


- But what if we want to delete at the end of the list?
- There is no constant time way to update the tail to point to the previous!

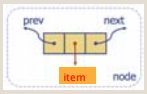
54

Doubly-linked lists

- A doubly linked list is a data structure consisting of a sequence of nodes
- Spe



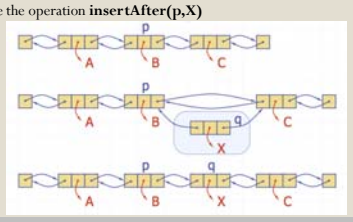
- Each node stores:
- Item
- Link to the next node
- Link to the previous node



55

Insertion

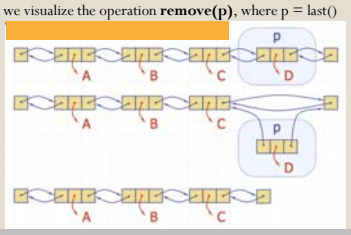
- Below we visualize the operation **insertAfter(p,X)**




```
public static void insertAfter(p,x):
  Node v = new Node()     //Create a new node v
  v.setItem(x)
  v.setPrev(p)            //link v to its predecessor
  v.setNext(p.getNext())  //link v to its successor
 (p.getNext()).setPrev(v) //link p's old successor to v
  p.setNext(v)             //link p to its new successor
```

Deletion

- Below we visualize the operation **remove(p)**, where p = last()




```
public static Object remove(p):
   t = p.item                        //tmp variable to hold the return value
  (p.getPrev()).setNext(p.getNext())       //linking over p
  (p.getNext()).setPrev(p.getPrev())
   p.setPrev(null)                   //invalidating the position p
   p.setNext(null)
   return t
```

57