APA 254

Data Structures

## Lecture 5.1 (Stacks)

Dept. of Information Systems Hanyang University

# Stacks

* Linear list
* One end is called top.
* Other end is called bottom.
* Additions to and removals from the top end only.

# Stack of Cups

* Add a cup on the stack.
* Remove a cup from the stack.
* A stack is a LIFO (Last-In, First-Out) list.
* Read Example 8.1

# Observations on Stack & Linear List

* Stack is a restricted version of linear list
  + All the stack operations can be performed as linear list operations
* If we designate the left end of the list as the stack bottom and the right as the stack top
  + Stack add (push) operation is equivalent to inserting at the right end of a linear list
  + Stack delete (pop) operation is equivalent to deleting from the right end of a linear list

# Stack ADT

AbstractDataType stack {

**instances**

linear list of elements; one end is the bottom; the other is the top.

**operations**

empty() : Return true if stack is empty, return false otherwise; size() : Return the number of elements in the stack;

top() : Return top element of stack;

pop() : Remove the top element from the stack; push(x) : Add element x at the top of the stack;

}

* See the C++ abstract class stack definition in Program 8.1

# Derived Classes and Inheritance

* + Stack can be defined as a special type of linear list
  + Class B may inherit members of class A in one of three basic modes: public, protected and private

### class B: public A

* + - protected members of A become protected members of B
    - public members of A become public members of B
    - cannot access private members of A

### class B: public A, private C

* + - public and protected members of C become private members of B

# Array-based Representation of Stack

* + Since stack is a restricted version of linear list, we can use an array-based representation of linear list (given in Section 5.3.3) to represent stack
  + Top of stack  **element[length-1]**
  + Bottom of stack  **element[0]**
  + The class derivedArrayStack defined in Program 8.2 is a derived class of arrayList<T> (Program 5.3)

# Derived Array-based class Stack

template<class T> // program 8.2

class derivedArrayStack : private arrayList<T>, public stack<T>

{

public:

derivedArrayStack(int initialCapacity = 10)

: arrayList<T> (initialCapacity) {} bool empty() const

{return arrayList<T>::empty();} int size() const

{return arrayList<T>::size();}

T& top()

{

if (arrayList<T>::empty()) throw stackEmpty();

return get(arrayList<T>::size() - 1);

}

void pop()

{

if (arrayList<T>::empty()) throw stackEmpty();

erase(arrayList<T>::size() - 1);

}

void push(const T& theElement)

{insert(arrayList<T>::size(), theElement);}

};

# Base Class arrayStack

* + We learned that a stack class can be defined by extending an arrayList class as in Program 8.2
  + However, this is not a very efficient implementation
  + A faster implementation is to develop a base class that uses an array stack to hold the stack elements
  + Program 8.4 is a faster implementation of an array stack
  + Read Section 8.3.2

### Efficiency of Array-based Representation

* + Array-based representation of a stack can waste space when multiple stacks are to coexist
  + An exception is when only two stacks are to coexist
  + How do we implement two stacks in an array?
    - Fix the bottom of one stack at position 0
    - Fix the bottom of the other at position MaxSize-1
    - The two stacks grow toward the middle of the array
  + See Figure 8.4

11

# Linked Representation of Stack

* Multiple stacks can be represented efficiently using a chain for each stack
* Which end of chain should be the stack top?
* If we use the right end of the chain as the stack top, then stack operations push and pop are implemented using chain operations *insert(n,x)* and *erase(n-1)* where n is the number of nodes in the chain  **(n)** time
* If we use the left end of the chain as the stack top, then

*insert(0,x)* and *erase(0)*  **(1)** time

* What should the answer be?

– **Left end!**

* See Program 8.5 for linked stack definition

# Application: Parenthesis Matching

* + Problem: match the left and right parentheses in a character string
  + (a\*(b+c)+d)
    - Left parentheses: position 0 and 3
    - Right parentheses: position 7 and 10
    - Left at position 0 matches with right at position 10
  + (a+b))\*((c+d)

– (0,4)

* + - Right parenthesis at 5 has no matching left parenthesis

– (8,12)

* + - Left parenthesis at 7 has no matching right parenthesis

# Parenthesis Matching

* + (((a+b)\*c+d-e)/(f+g)-(h+j)\*(k-1))/(m-n)
    - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v. (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)

### How do we implement this using a stack?

1. Scan expression from left to right
2. When a left parenthesis is encountered, add its position to the stack
3. When a right parenthesis is encountered, remove matching position from the stack

# Example of Parenthesis Matching

## (((a+b)\*c+d-e)/(f+g)-(h+j)\*(k-1))/(m-n)

stack …

2

1

0

1

0

0

15

0

0

21

0

0

output

(2,6)

(1,13)

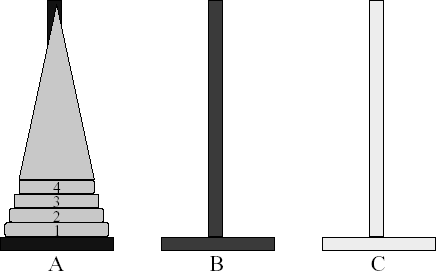
(15,19)

(21,25)…

* See Program 8.6
* Do the same for (a-b)\*(c+d/(e-f))/(g+h)

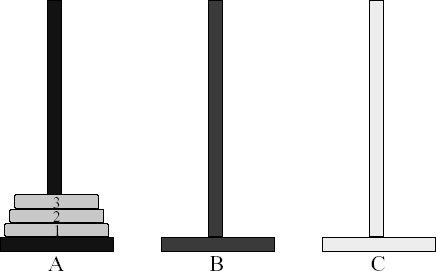
16

# Application: Towers of Hanoi

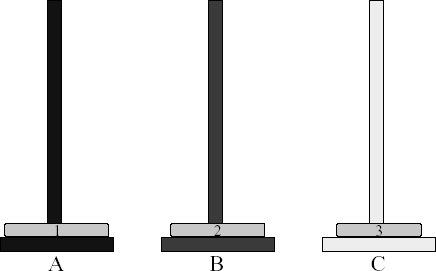
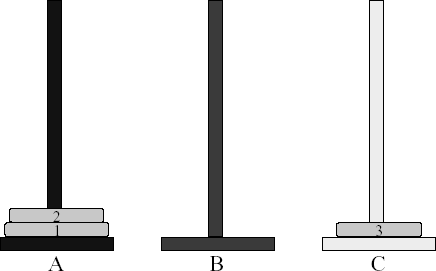


* + Read the ancient Tower of Brahma ritual (p. 285)
  + ***n*** disks to be moved from tower **A** to tower **C** with the following restrictions:
    - Move 1 disk at a time
    - Cannot place larger disk on top of a smaller one

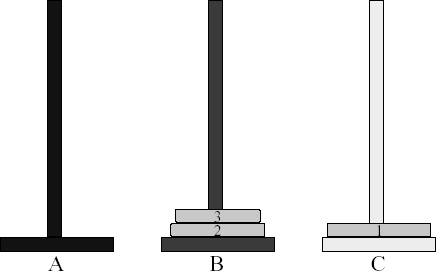
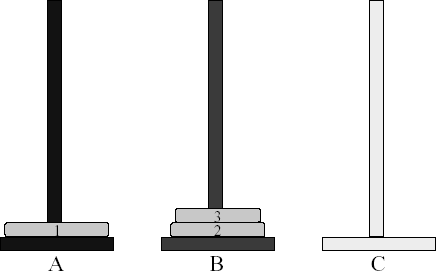
# Let’s solve the problem for 3 disks



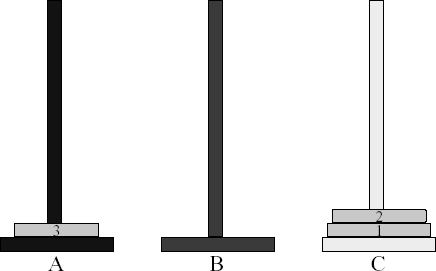
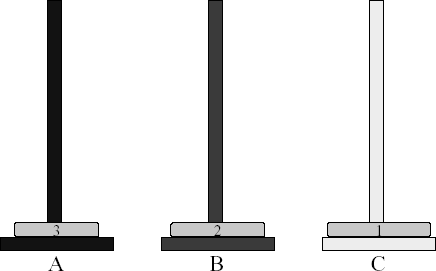
**Towers of Hanoi (1, 2)**



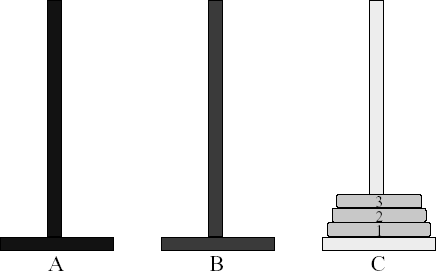
# Towers of Hanoi (3, 4)



**Towers of Hanoi (5, 6)**



# Towers of Hanoi (7)



* + So, how many moves are needed for solving 3-disk Towers of Hanoi problem?

 7

# Time complexity for Towers of Hanoi

* A very elegant solution is to use **recursion.** See Program 8.7 for Towers of Hanoi recursive function
* The minimum number of moves required is ***2n-1***
* Time complexity for Towers of Hanoi is

(***2n***), which is exponential!

* Since disks are removed from each tower in a LIFO manner, each tower can be represented as a stack
* See Program 8.8 for Towers of Hanoi using stacks

# Recursive Solution

1

A B C

* + **n > 0** gold disks to be moved from A to C using B
  + move top **n-1** disks from A to B using C

# Recursive Solution

1

A B C

* + move top disk from A to C

# Recursive Solution

1

A B C

* + move top n-1 disks from B to C using A

# Recursive Solution

1

A B C

# 64-disk Towers of Hanoi Problem

* How many moves is required to move 64 disks?

 moves(64) = 264-1 = 1.8 \* 1019 (approximately)

* How long would it take to move 64 gold disks by the Brahma priests?

 At 1 disk move/min, the priests will take about **3.4 \* 1013 years**.

 “According to legend, the world will come to an end when the priests have competed their task”

* How long would it take to move 64 disks?

 hundreds of years to complete.

29

# Application: Rearranging Railroad Cars

* + Read the problem on p. 289
  + Rearrange the cars at a shunting yard that has an input track, and output track, and *k* holding tracks between the input and output tracks
  + See Figure 8.6 for a three-track example
  + See Figure 8.7 for track states
  + See Program 8.9, 8.10 and 8.11