Ch9. Stacks

Bird's-Eye View (1/2)

- Chapter 9: Stack
  - A kind of Linear list & LIFO(last-in-first-out) structure
  - Insertion and removal from one end
- Chapter 10: Queue
  - A kind of Linear list & FIFO(first-in-first-out) structure
  - Insertion and deletion occur at different ends of the linear list
- Chapter 11: Skip Lists & Hashing
  - Chains augmented with additional forward pointers

Bird's-Eye View (2/2)

- Stack Representation
  - Array-based class "ArrayStack"
  - Linked class "LinkedStack"

- Application using stack
  - Parenthesis Matching
  - Towers of Hanoi
  - Rearranging Railroad Cars
  - Switch Box Routing
  - Offline Equivalence Class Problem
  - Rat in a Maze

Table of Contents

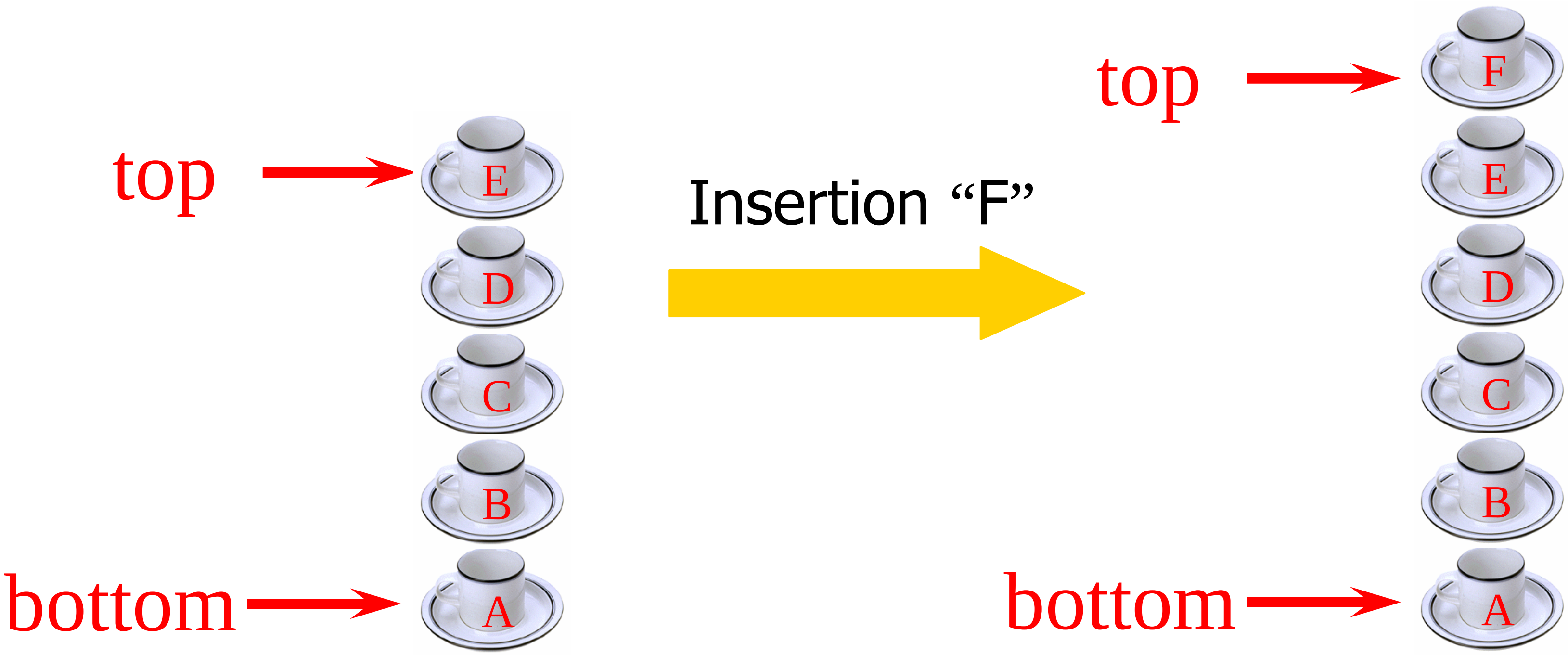
Definition

- A stack is
  - A kind of Linear list
  - One end is called "top"
  - Other end is called "bottom"
  - Insertions and removals take place at the top

- A stack is a LIFO list (Last In First Out)

Stack Application

- Stack of Cups



The ADT Stack

```
AbstactDataType Stack{
    instantces
        linear list of elements;
        bottom;
        top;
    operations
        empty() : Return true if the stack is empty,
                  Return false otherwise;
         peek() :  Return the top element;
        push(x) :  Add element x at the top;
          pop() :  Remove the top element  and return it;
}
```

Q: Can we think of any other core operation of the Stack?

The Interface Stack

```
public interface Stack
{
    public boolean empty();
    public Object   peek();
    public void      push(Object  theObject);
    public Object   pop();
}
```

Table of Contents

- Definition
- Array Representation
- Linked Representation
- Stack Applications

The Class DerivedArrayStack (1)

- Derived from ArrayLinearList class
- Implements Stack interface
- The top is right end of array linear list
  - push(theObject) → add (size(), theObject)
    - O(1) time
  - pop() → remove (size() - 1)
    - O(1) time

The Class DerivedArrayStack (2)

```
package dataStructures;
import java.util.*;  // has stack exception
public class DerivedArrayStack  extends       ArrayLinearList
                                implements  Stack
{
  public DerivedArrayStack(int initialCapacity)
         {super(initialCapacity);}  //ArrayLinearList's constructor

  public DerivedArrayStack()
            {this(10);}

  /*  Stack interface methods come here  */
}
```

The Class DerivedArrayStack (3)

```
public boolean   empty ()
   {return isEmpty();}

public Object    peek ()
  { if (empty())    throw new EmptyStackException();
     return get(size() - 1) }

public void        push (Object theElement)
  {    add(size(), theElement); }

public Object     pop ()
{ if (empty())   throw new EmptyStackException();
   return remove(size() - 1); }
```

The Class DerivedArrayStack (4)

- Stack as a subclass from ArrayLinearList class

- All public methods of ArrayLinearList may also be performed on a stack
  - For example, get(0)/remove(5)/add(3, x) are still alive
  - Thus, we do not have a true stack implementation

- So, need to override some undesired methods of ArrayLinearList
  - Public void add(int index, Object obj) {

```
        throw new UNsupportedOperationException ("Not supported")
    }
```

The Class ArrayStack (1)

- A faster implementation of array-based stack
- Implement Only the methods of the Stack interface
- Uses an one-dimensional array
- push()/pop() : O(1) time

The Class ArrayStack (2)

```
package dataStructures;
import java.util.EmptyStackException;
import utilities.*;  // ChangeArrayLength
public class ArrayStack implements Stack
{ int top;          // current top of stack
  Object [] stack;  // element array

  public ArrayStack(int initialCapacity){
  if (initialCapacity < 1)    throw new IllegalArgException ("initialCapacity must be >= 1");
  stack = new Object [initialCapacity];  // 1D array declaration
  top = -1;
  }
   public ArrayStack()     {this(10);}

  /* Stack interface methods come here */
}
```

The Class ArrayStack (3)

```
public bolean empty ()   {  return top== -1;    }

public Object peek () {
  if(empty()) throw new EmptyStackException();
  return stack[top];  }

 public void push (Object theElement){
  // increase(doubling) array size if necessary
   if  (top == stack.length - 1)     stack =   ChangeArrayLength.changeLength1D (stack, 2  * stack.length);
    stack[++top] = theElement; // put theElement at the top of the stack
}

public Object pop () {
  if (empty())      throw new EmptyStackException();
  Object topElement = stack[top];
  stack[top--] = null;   // enable garbage collection
  return topElement;
}
```

Performance Measurement

- Time to perform a 500,000 sequence

	InitialCapacity	
Class	10	500,000
ArrayStack	0.44	0.22
DerivedArrayStack	0.60	0.38
DerivedArrayStackWithCatch	0.55	0.33
DerivedVectorStack	1.27	1.04
Stack	1.15	-

(Times are in seconds)

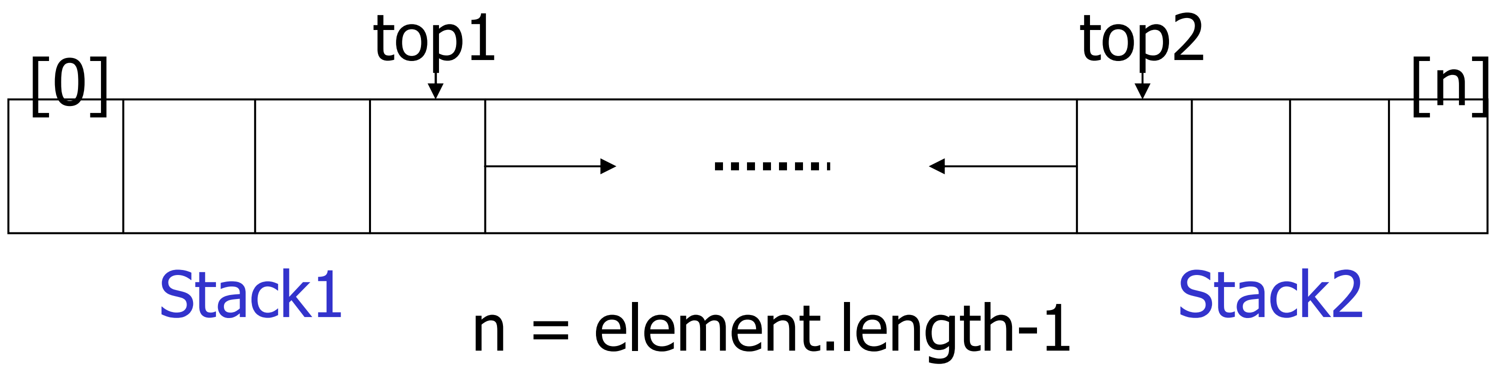
  - Stack took 2.6 times more of the time taken by ArrayStack
  - The time spent resizing the array is approximately the same for all implementation(0.2 second)

Table of Contents

- Definition
- Array Representation
- Linked Representation
- Applications

Multiple Stacks in an Array

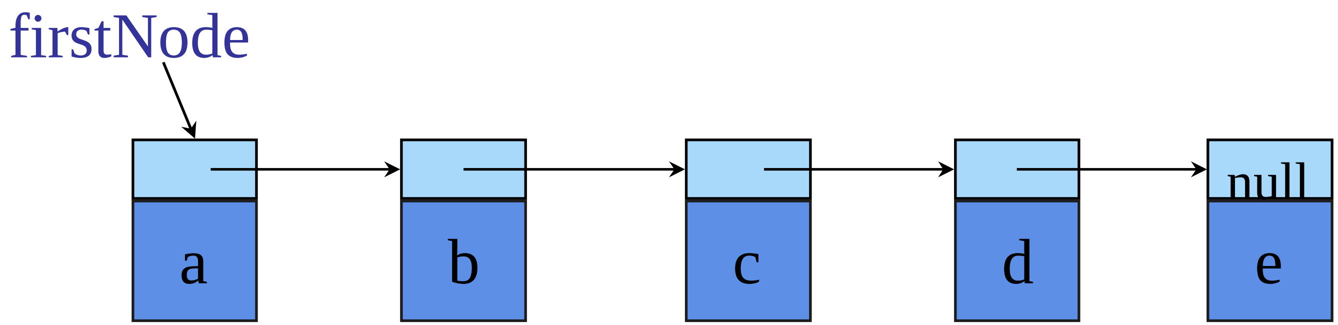
- Two stacks in an array (space efficient)



- When more than two Stacks in an array
  - push() : the worst case can be O(length) ➔ doubling the existing array may involve a lot of complications
  - pop() : O(1)
- If we have to save the space, why not pointers!

Chain for the Stack

- Using a chain for each stack



- Operations of peek, push, pop method
  - If top is the left-end of linear list
    - get(0), add(0,x), remove(0) : O(1) time
  - If top is the right-end of linear list
    - get(size() –1), add(size(),x), remove(size() –1) : O(size()) time

  ➔ Use the left-end as stack top

The Class DerivedLinkedStack

- One possible implementation: derived from the class Chain and implements the interface Stack

```
Public class DerivedLinkedStack extends Chain  implements Stack
{
// Replace the name DerivedArrayStack with the name DerivedLinkedStack
// Change the parameter of the methods get(), remove(), add() to 0
}
```

- As we experienced in DerivedArrayStack, some operations in Chain are redundant and mismatch with the stack
  - So, we better use only the Stack interface

The Class LinkedStack (1)

```
// want to implement only the stack interface
package dataStructures;
import java.util.EmptyStackException;
public class LinkedStack implements Stack
{ // data members
   protected ChainNode topNode;

   /** create an empty stack */
   public LinkedStack (int initialCapacity)
         { // the default initial value of topNode is null  }
   public LinkedStack ()
        {this(0);}

   /*   methods here       */
}
```

The Class LinkedStack (2)

```
public boolean empty() {
     return topNode == null;
    }

public Object peek() {
   if (empty())   throw new EmptyStackException();
   return topNode.element;
 }

 public void push(Object theElement) {
     topNode = new ChainNode(theElement, topNode);
}

public Object pop() {
      if (empty())  throw new EmptyStackException();
      Object topElement = topNode.element;
      topNode = topNode.next;
      return topElement;
   }
```

Performance Measurement

- LinkedStack is a little bit more efficient than DerivedLinkedStack because redundancies are removed
  - To perform 500,000 sequence
    - DerivedLinkedStack → 3.2 sec
    - LinkedStack → 2.96 sec

- In general, LinkedStack requires more memory and time than ArrayStack
- So, the use of pointers (LinkedStack) provides no benefit when we deal with only a single stack

Table of Contents

- Definition
- Array Representation
- Linked Representation
- Stack Applications
  - Parenthesis Matching
  - Towers of Hanoi
  - Rearranging Railroad Cars
  - Offline Equivalence Class Problem
  - Rat in a Maze

Parenthesis Matching

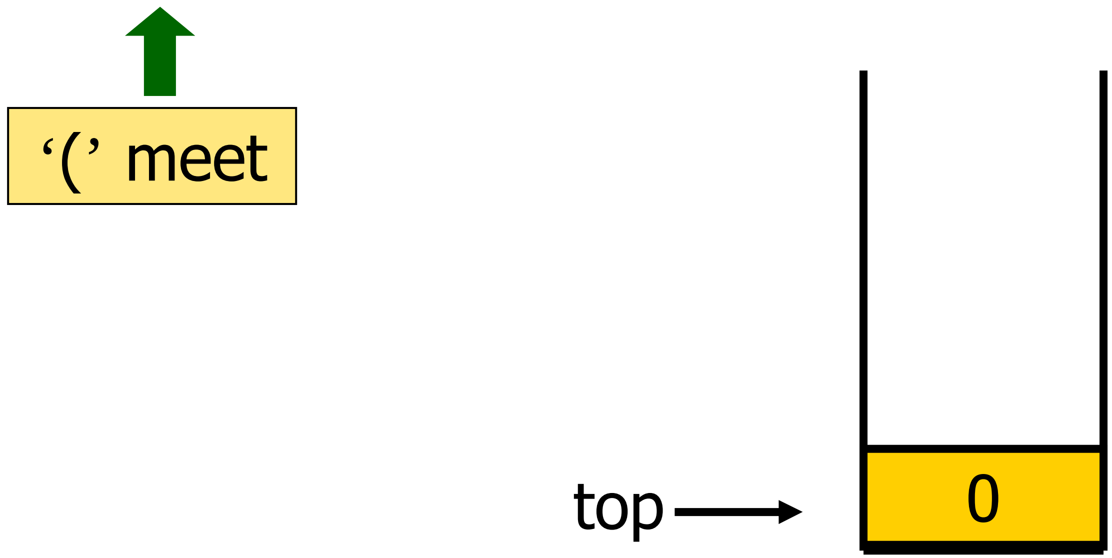
- Print out matching of the left and right parentheses in a character string
  - (a*(b+c)+d) : output ➔ (0,10), (3,7) match
  - (a+b))(
    - Output ➔ (0, 4) match
    - Output ➔ 5, 6 have no matching parentheses

- Solution steps
  - Scan the input expression from left to right
  - ‵ (‵ is encountered, add its position to the stack
  - ‵) ‵ is encountered, remove matching position from stack
- Complexity
  - Push / pop operations : O(n) time

Example: Parenthesis Matching (1)

- (a*(b+c)+d)

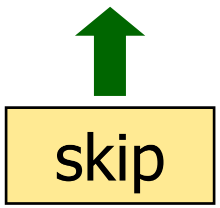
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

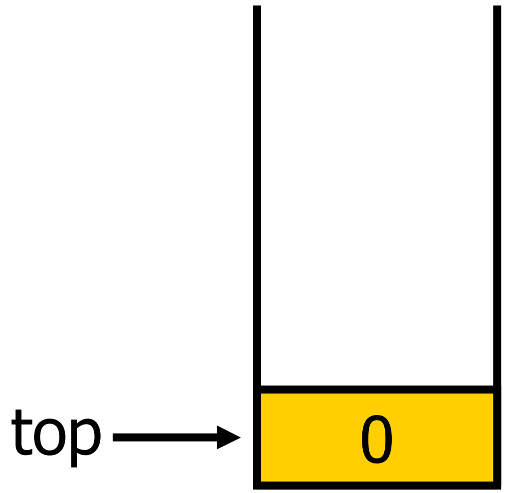


Example: Parenthesis Matching (2)

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



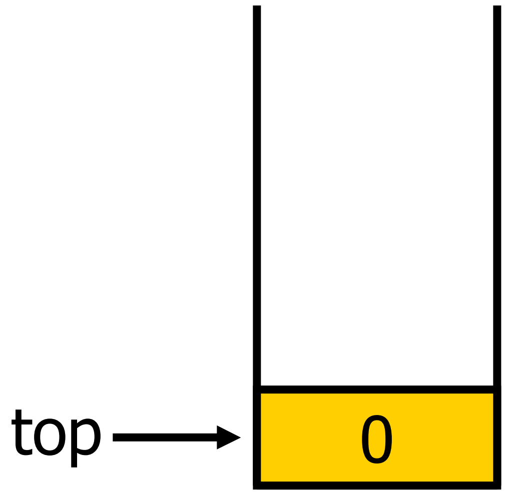


Example: Parenthesis Matching (3)

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

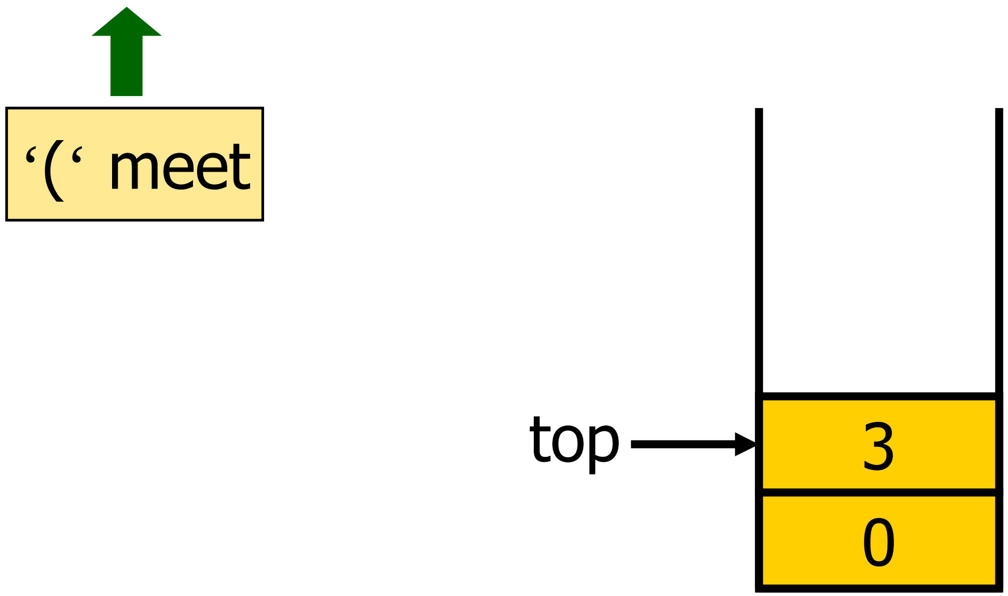
skip



Example: Parenthesis Matching (4)

- (a*(b+c)+d)

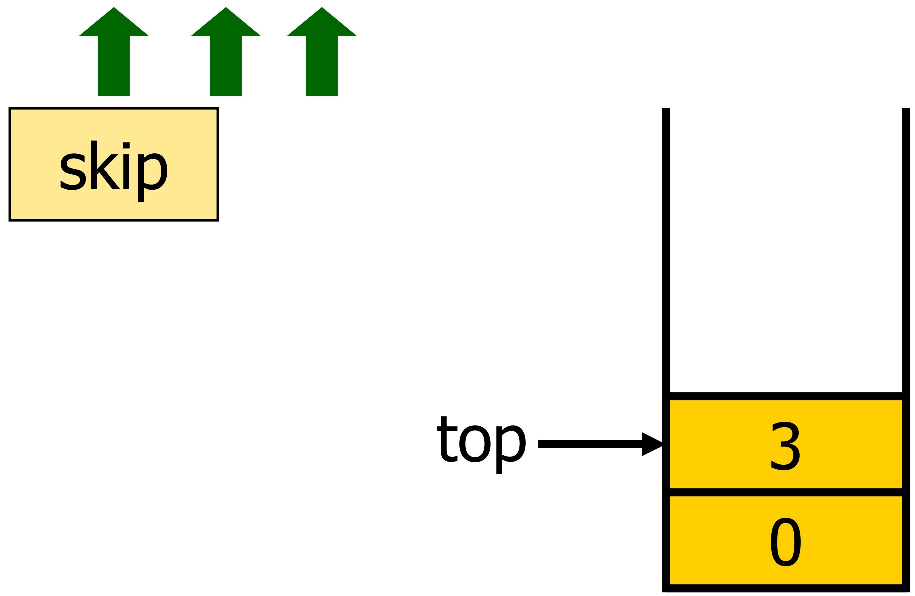
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Example: Parenthesis Matching (5)

- (a*(b+c)+d)

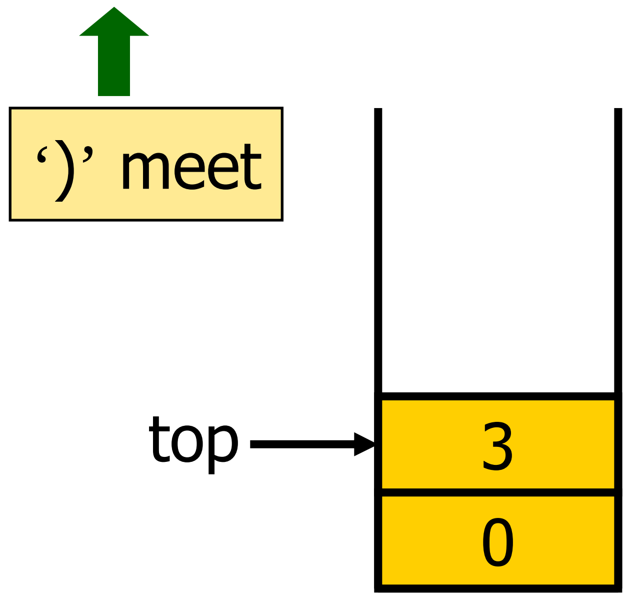
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Example: Parenthesis Matching (6)

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Example: Parenthesis Matching (7)

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)

‘)’ meet

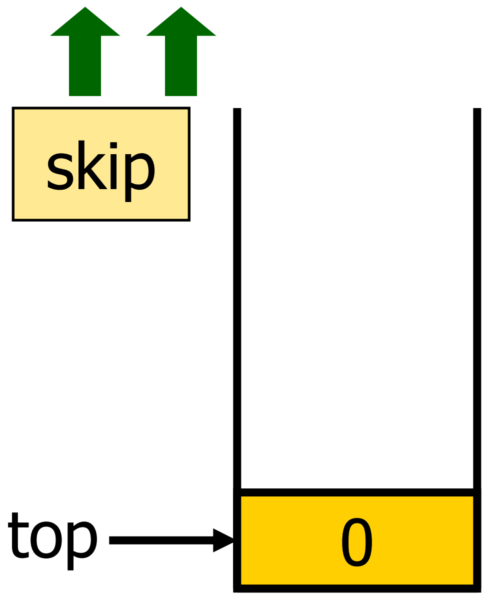
Output : (3, 7)

top → 0

Example: Parenthesis Matching (8)

- (a*(b+c)+d)

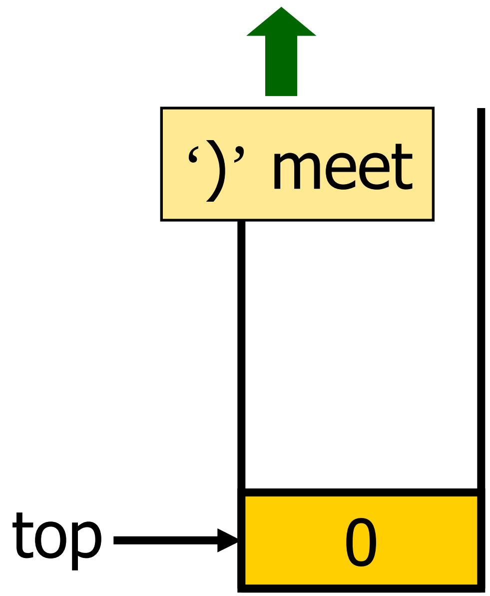
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Example: Parenthesis Matching (9)

- (a*(b+c)+d)

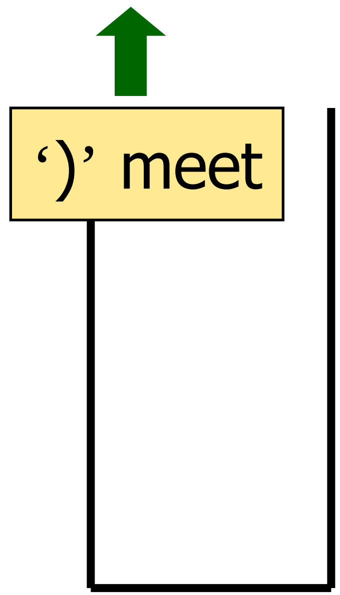
position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Example: Parenthesis Matching (10)

- (a*(b+c)+d)

position	0	1	2	3	4	5	6	7	8	9	10
character	(	a	*	(	b	+	c	)	+	d	)



Output : (3, 7), (0, 10)

Parenthesis Matching Code

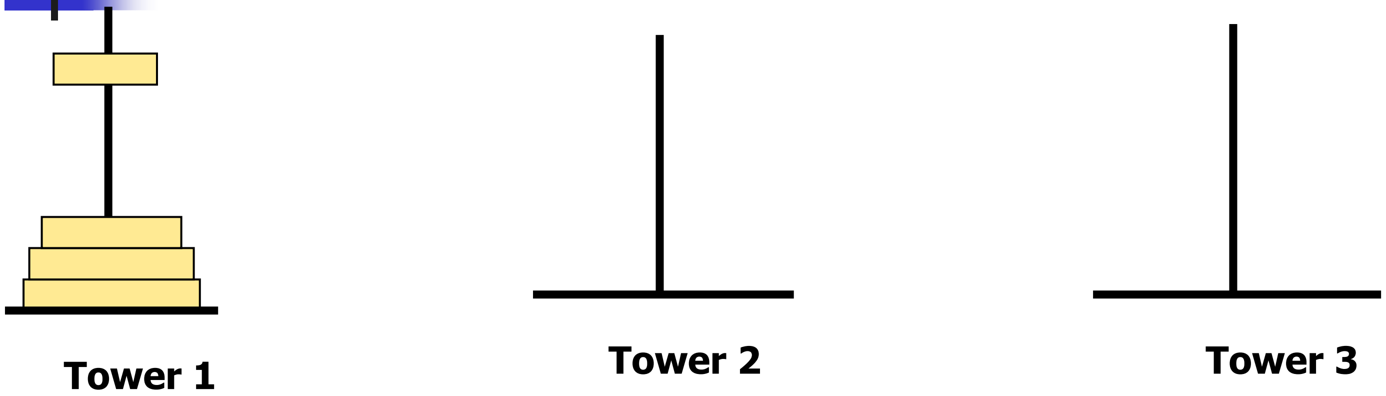
```
public static void printMatchedPairs (String expr) {
    /* data members */
    // scan expression expr for ( and )
     for (int i = 0; i < length; i++)
         if ( expr.charAt(i) == '(' )    s.push(new Integer(i));
         else  if (expr.charAt(i) == ')')
                 try{  // remove location of matching '(' from stack
                        System.out.println(s.pop() + "  " + i);  }
                  catch (Exception e)   { // stack was empty, no match exists   }

     // remaining '(' in stack are unmatched
     while (!s.empty())
         System.out.println("No match for left parenthesis at " + s.pop());
  }
```

Table of Contents

- Stack Applications
  - Parenthesis Matching
  - Towers of Hanoi
  - Rearranging Railroad Cars
  - Switch Box Routing
  - Offline Equivalence Class Problem
  - Rat in a Maze

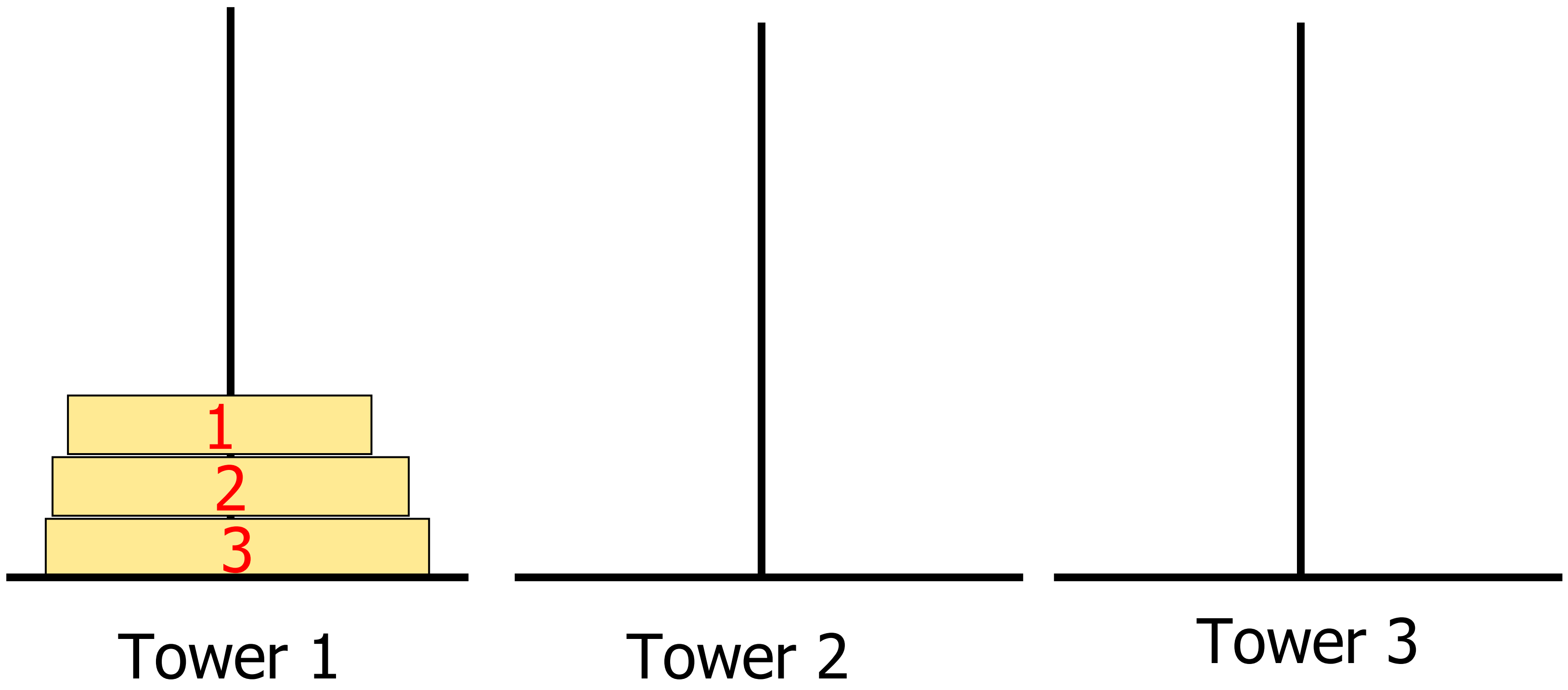
Towers of Hanoi



- Mission: Move the disks from tower1 to tower2
- Each tower operates as a stack
- Cannot place a big disk on top of a smaller one
  - Move n-1 disks to tower3 using tower2
  - Move the largest to tower2
  - Move the n-1 disks from tower3 to tower2 using tower1
- Use of Recursion

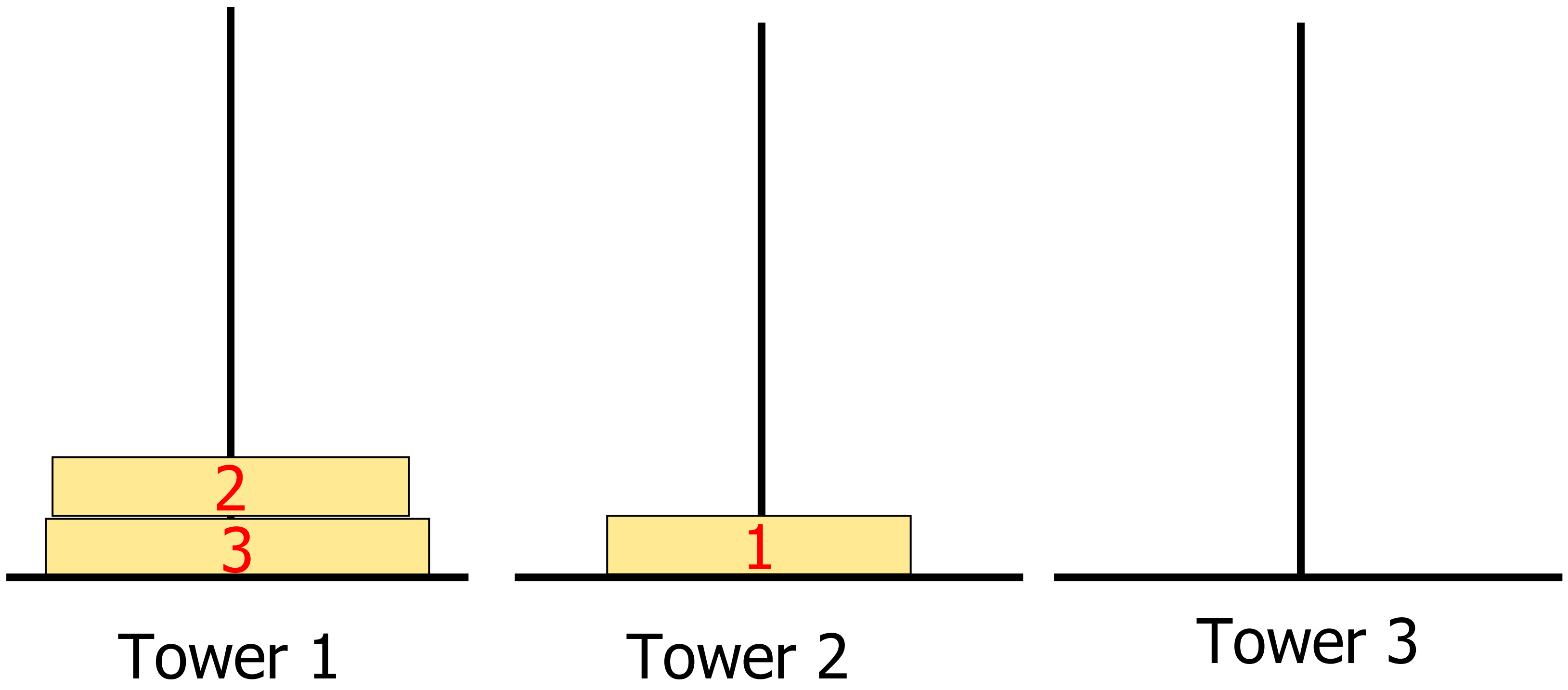
TOH Example (1/8)

- Mission: Move the disks from tower1 to tower2



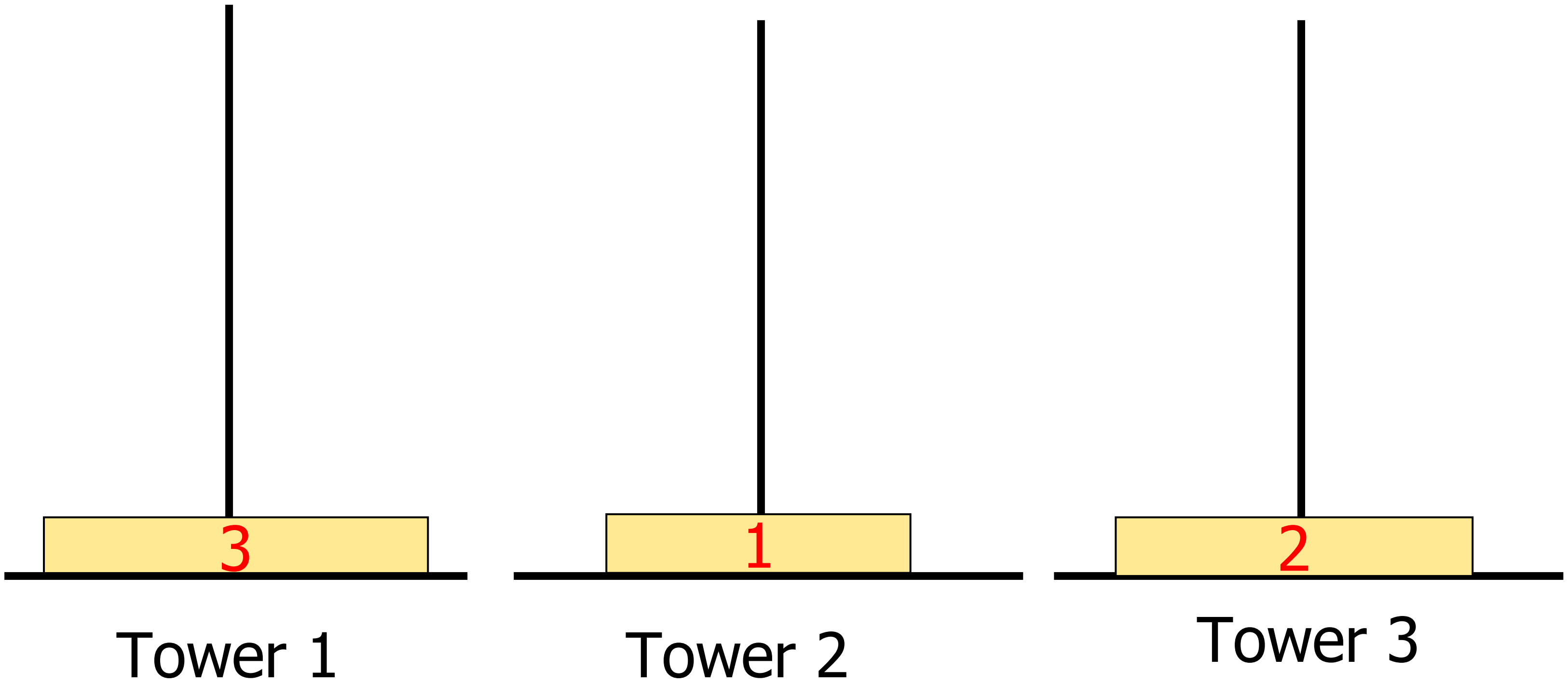
TOH Example (2/8)

- Mission: Move the disks from tower1 to tower2



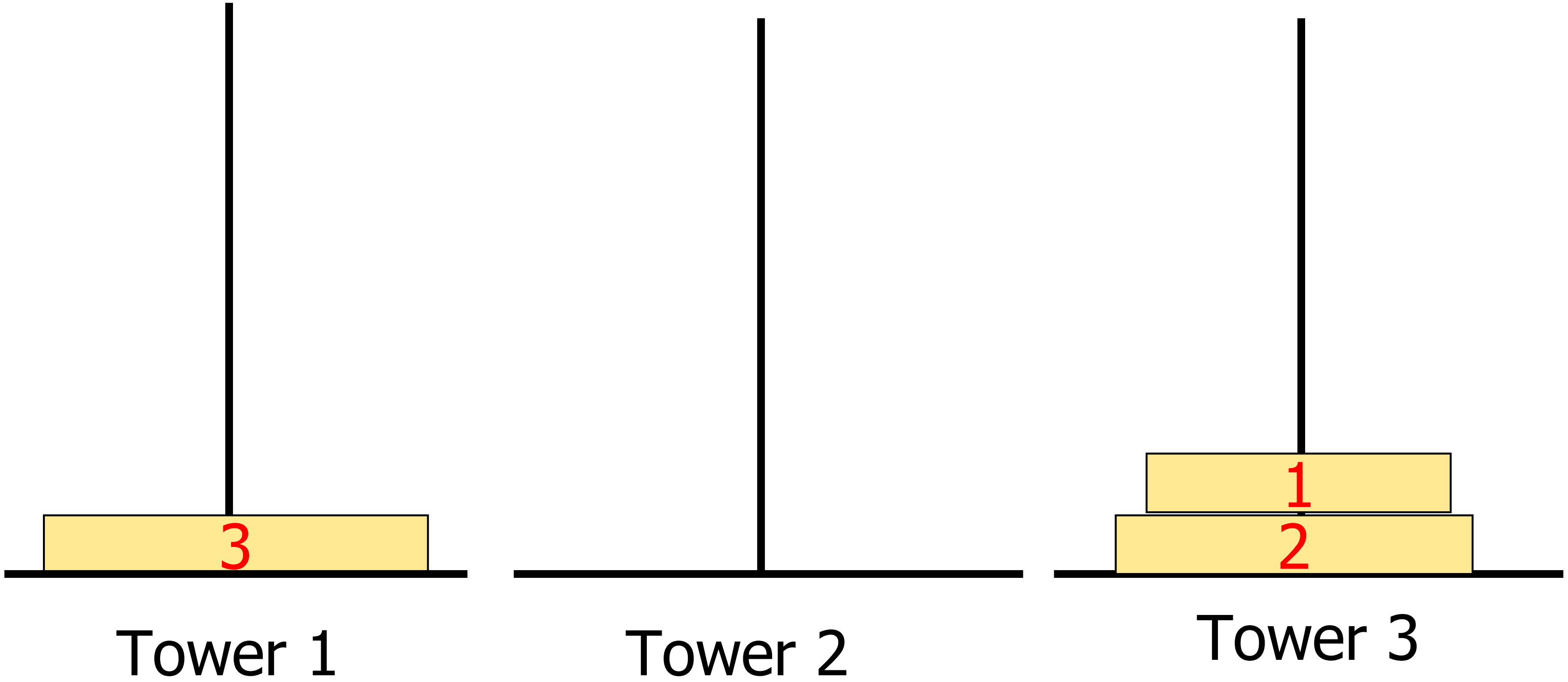
TOH Example (3/8)

- Mission: Move the disks from tower1 to tower2



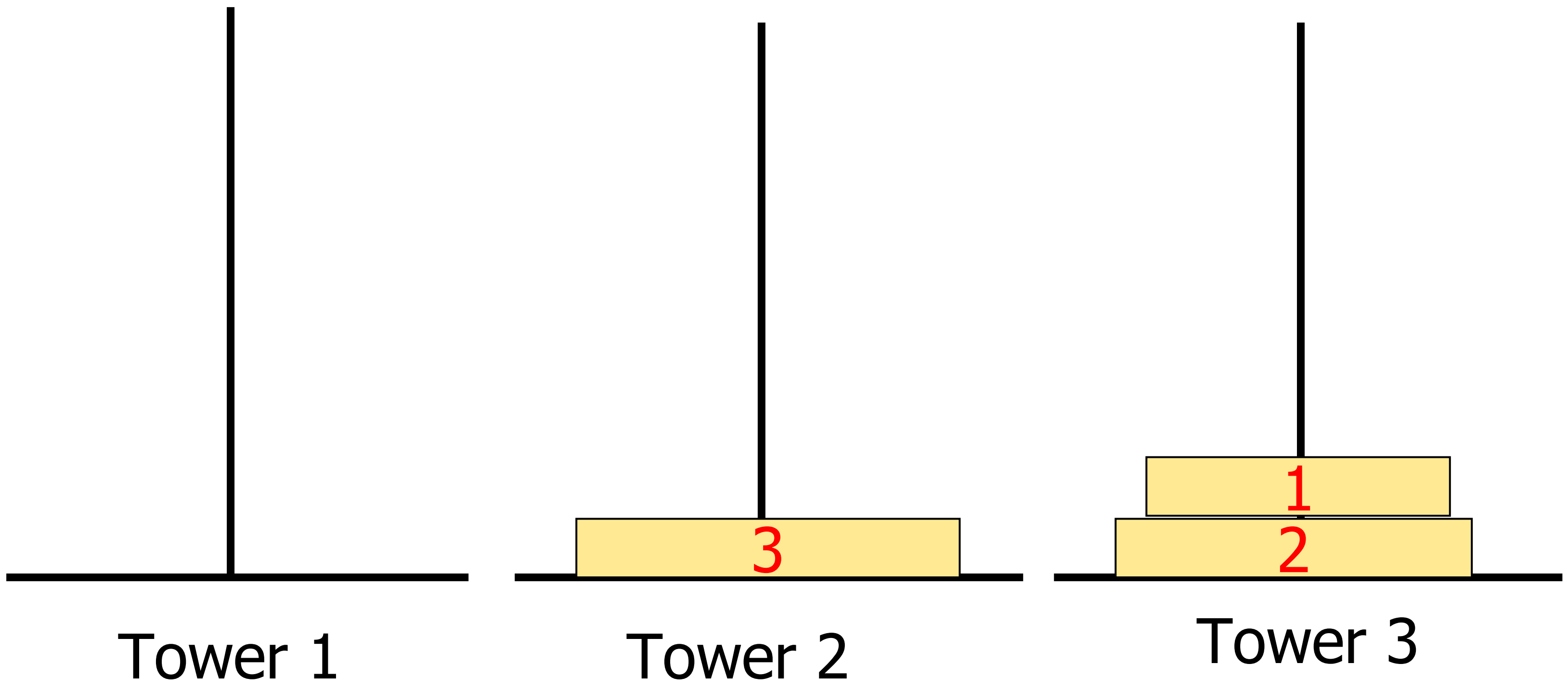
TOH Example (4/8)

- Mission: Move the disks from tower1 to tower2



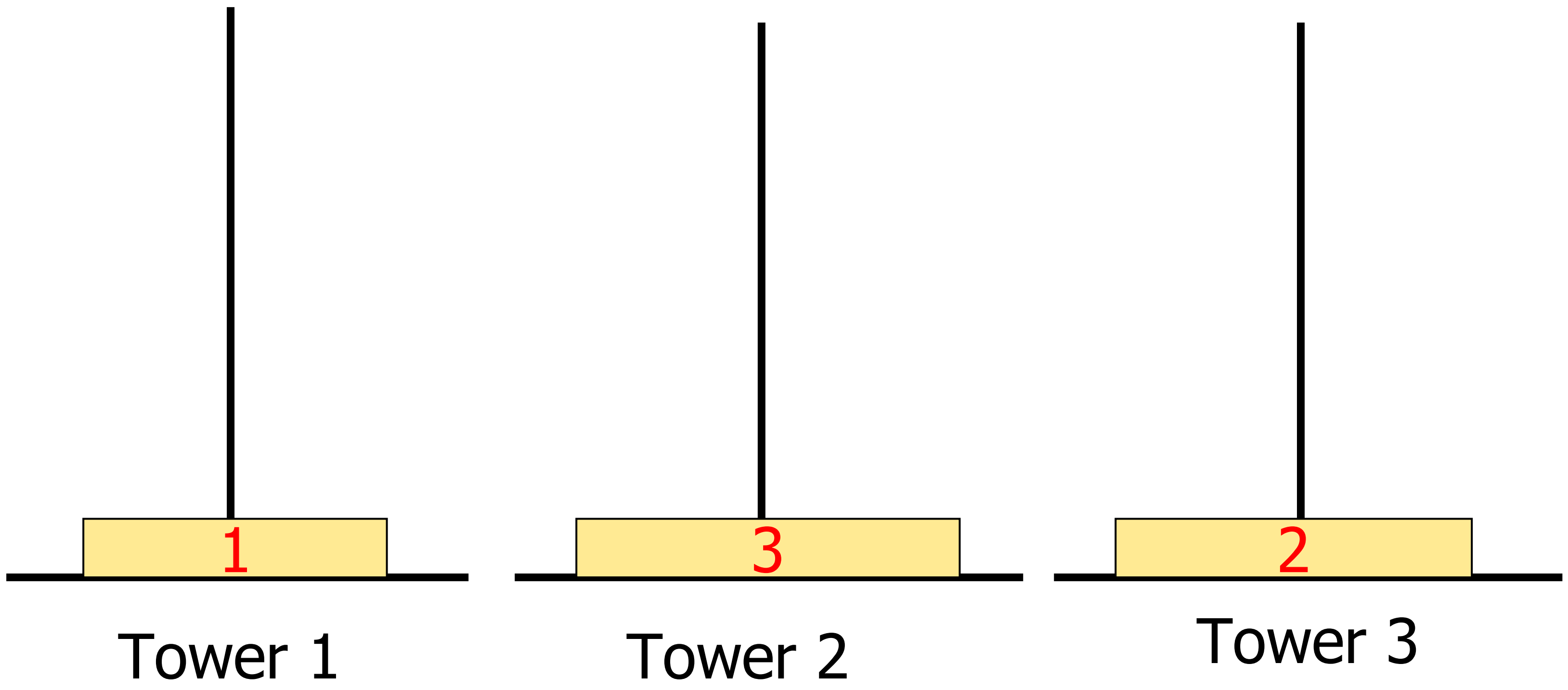
TOH Example (5/8)

- Mission: Move the disks from tower1 to tower2



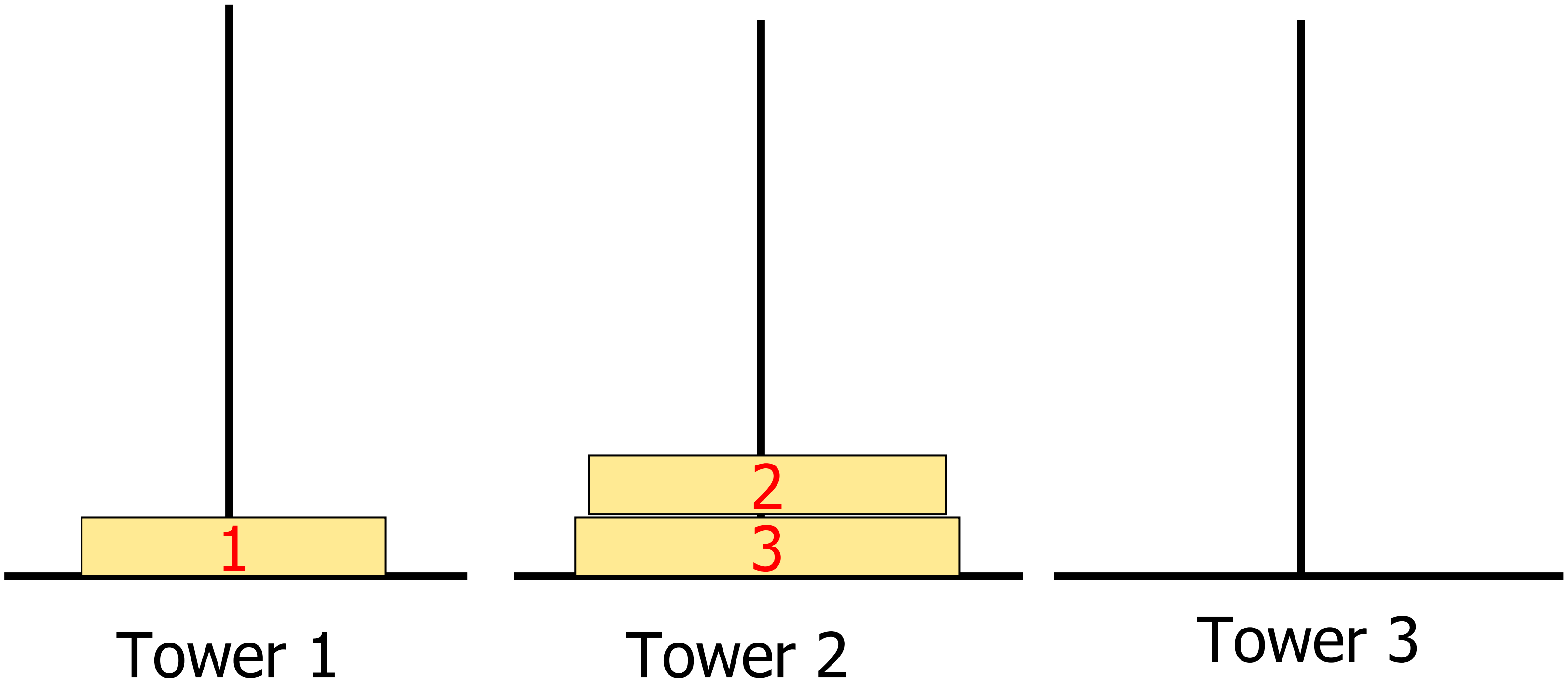
TOH Example (6/8)

- Mission: Move the disks from tower1 to tower2



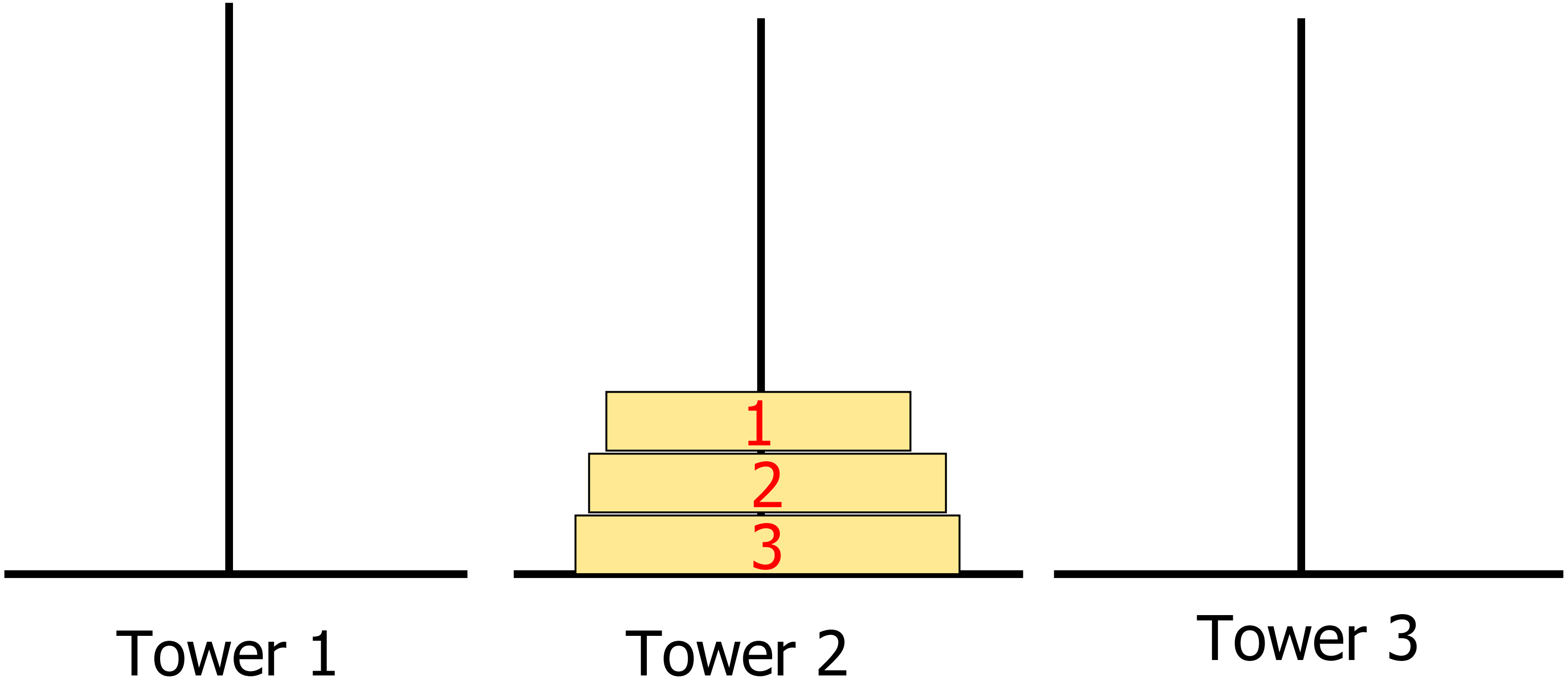
TOH Example (7/8)

- Mission: Move the disks from tower1 to tower2



TOH Example (8/8)

- Mission: Move the disks from tower1 to tower2



1st code: towersOfHanoi(m,1,2,3)

```
public static void towersOfHanoi (int n, int x, int y, int z)
  { // Move the top n disks from tower x to tower y.
   // Use tower z for intermediate storage.
     if (n > 0) {
        towersOfHanoi (n-1, x, z, y);
        System.out.println ("Move top disk from tower " + x +  " to top of tower " + y);
        towersOfHanoi (n-1, z, y, x);
     }
  }
```

Actual execution

TOH(3, x, y, z)

TOH(2, x, z, y)

TOH(1, x, y, z): <u>move 1 from x to y</u>

: <u>move 2 from x to z</u>

TOH(1, y, z, x): <u>move 1 from y to z</u>

<u>move 3 from x to y</u>

TOH(2, z, y, x)

TOH(1, z, x, y): <u>move 1 from z to x</u>

: <u>move 2 from z to y</u>

TOH(1, x, y, z): <u>move 1 from x to y</u>

Complexity: 1st TOH code

- The number of moves: moves(n)
  - n = 0 : moves(n) = 0
  - n > 0 : moves(n) = 2 * moves(n-1) + 1
- Therefore moves(n) = $2^n - 1$
  - Time Complexity : $\Theta(2^n)$

2nd Code: TOH (1/3)

- The 1st TOH code gives only the disk-move sequence
- What if we want to store the actual state of the 3 towers (the disk order bottom to top) ➔ use stacks!

```
public class TowersOfHanoiShowingStates
{
    // data member
    // the towers are tower[1:3]
    private static ArrayStack [] tower;

    code for  towersOfHanoi ( ) comes here;
    code for showTowerStates ( ) comes here;
}
```

2nd code: TOH (2/3)

```
/** n disk Towers of Hanoi problem */
public static void towersOfHanoi (int n) {
  // create three stacks, tower[0] is not used
  tower = new ArrayStack[4];
  for (int i = 1; i <= 3; i++)
      tower[i] = new ArrayStack();

  for (int d = n; d > 0; d--)
      tower[1].push(new Integer(d));  // add disk d to tower 1
  // move n disks from tower 1 to 2 using 3 as intermediate tower
  showTowerStates(n, 1, 2, 3);
 }
```

2nd code: TOH (3/3)

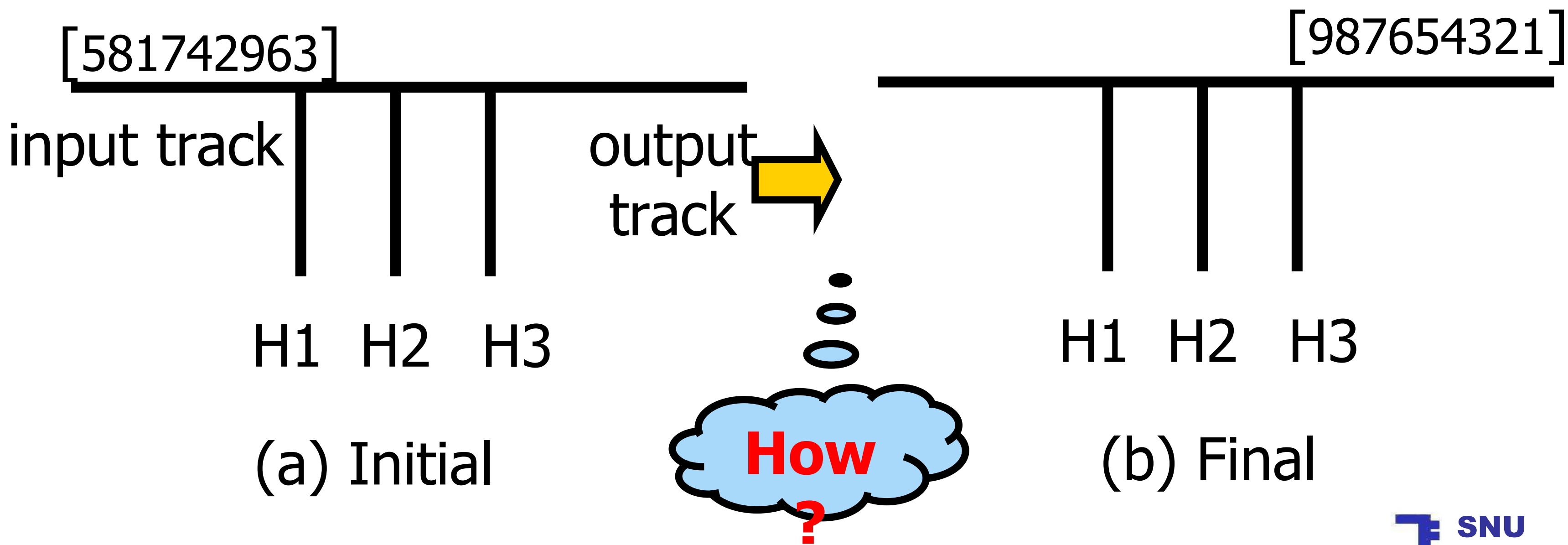
```
public static void showTowerStates (int n, int x, int y, int z)
{ // Move the top n disks from tower x to tower y.
  if (n > 0) {
     showTowerStates(n-1, x, z, y);
     Integer d = (Integer) tower[x].pop(); // move d from top of tower x
     tower[y].push(d);                     // to top of tower y
     System.out.println
              ("Move disk " + d + " from tower "+ x + " to top of tower " + y);
     showTowerStates(n-1, z, y, x);
     }
}
```

Table of Contents

- Stack Applications
  - Parenthesis Matching
  - Towers of Hanoi
  - Rearranging Railroad Cars
  - Offline Equivalence Class Problem
  - Rat in a Maze

Rearranging Railroad Cars (1)

- There are numbered N stations
- The freight train visits these stations in the order n through 1
- Must reorder the cars of the freight train to be in the order 1 through n from front to back
- Want to drop the x'th car into the station x and keep going

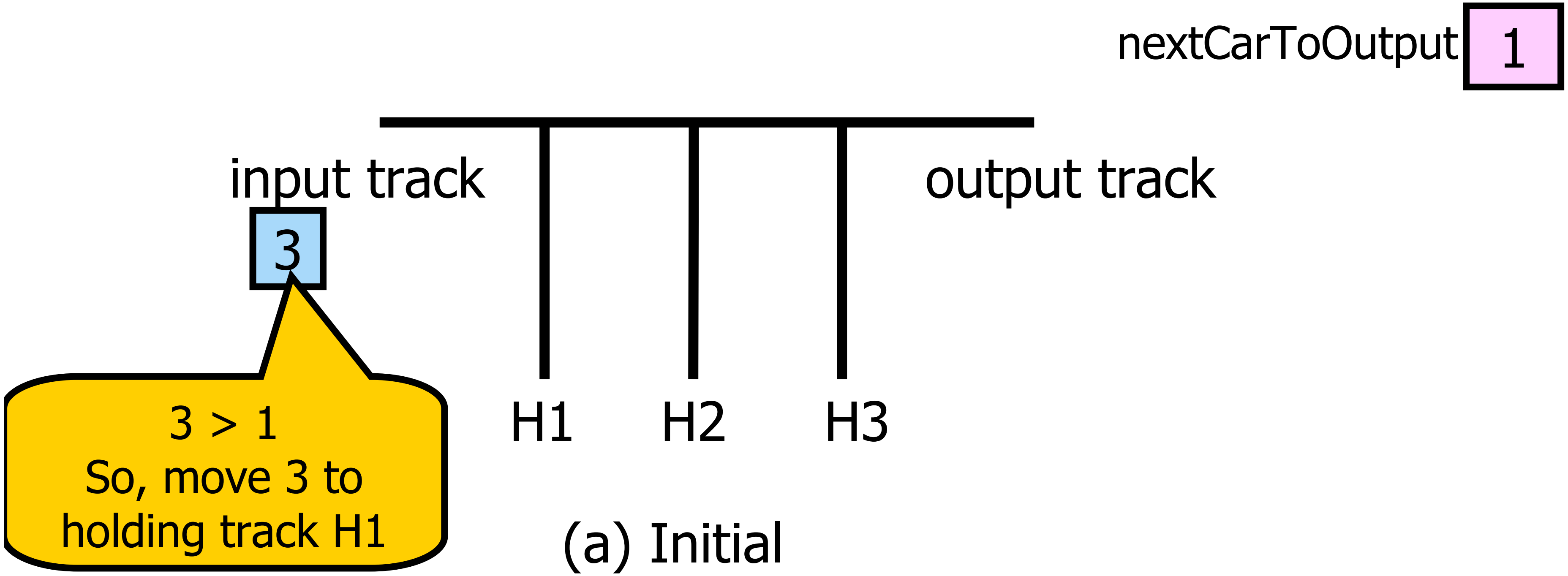


Rearranging Railroad Cars (2)

- Solution steps
  - If the car is the expected next one in the output track, move it directly to output track
  - If not, move it to a holding track
  - The holding tracks operate in a LIFO manner
  - Assignment rule : The new car u is moved to the holding track H that has at its top a car with smallest label v such that v > u
  - The bottom of each holding track has a big value

Rearranging Railroad Cars (1/17)

- Example: Input : 581742963

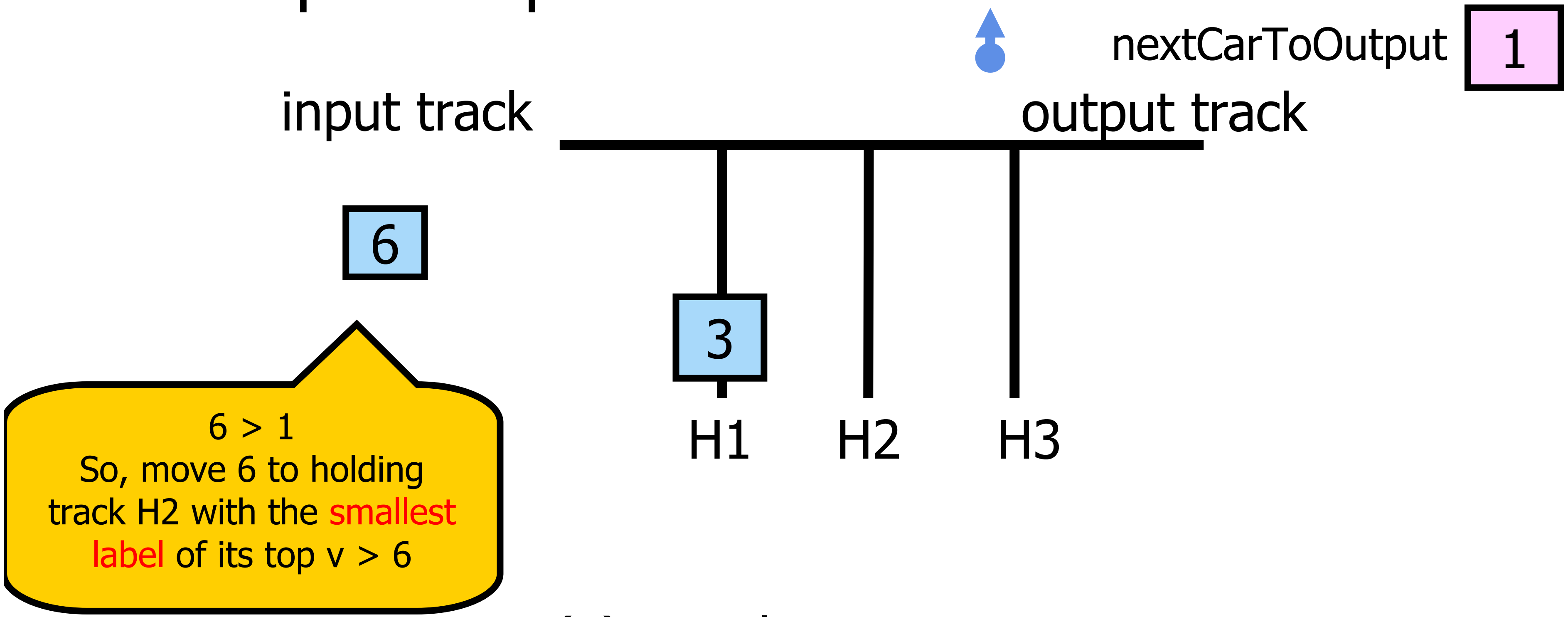


(a) Initial

Assume: the bottom of holding block has a big number v

Rearranging Railroad Cars (2/17)

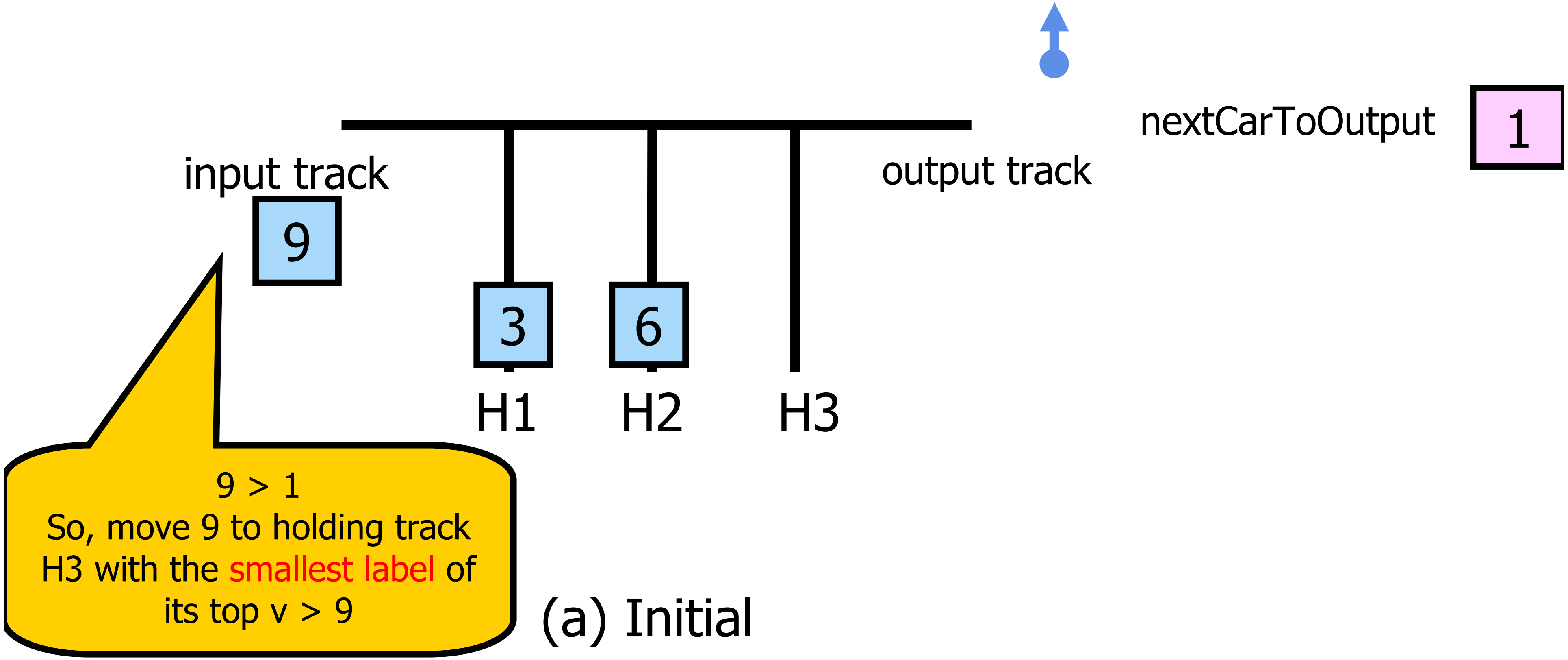
- Example : Input : 581742963



(a) Initial

Rearranging Railroad Cars (3/17)

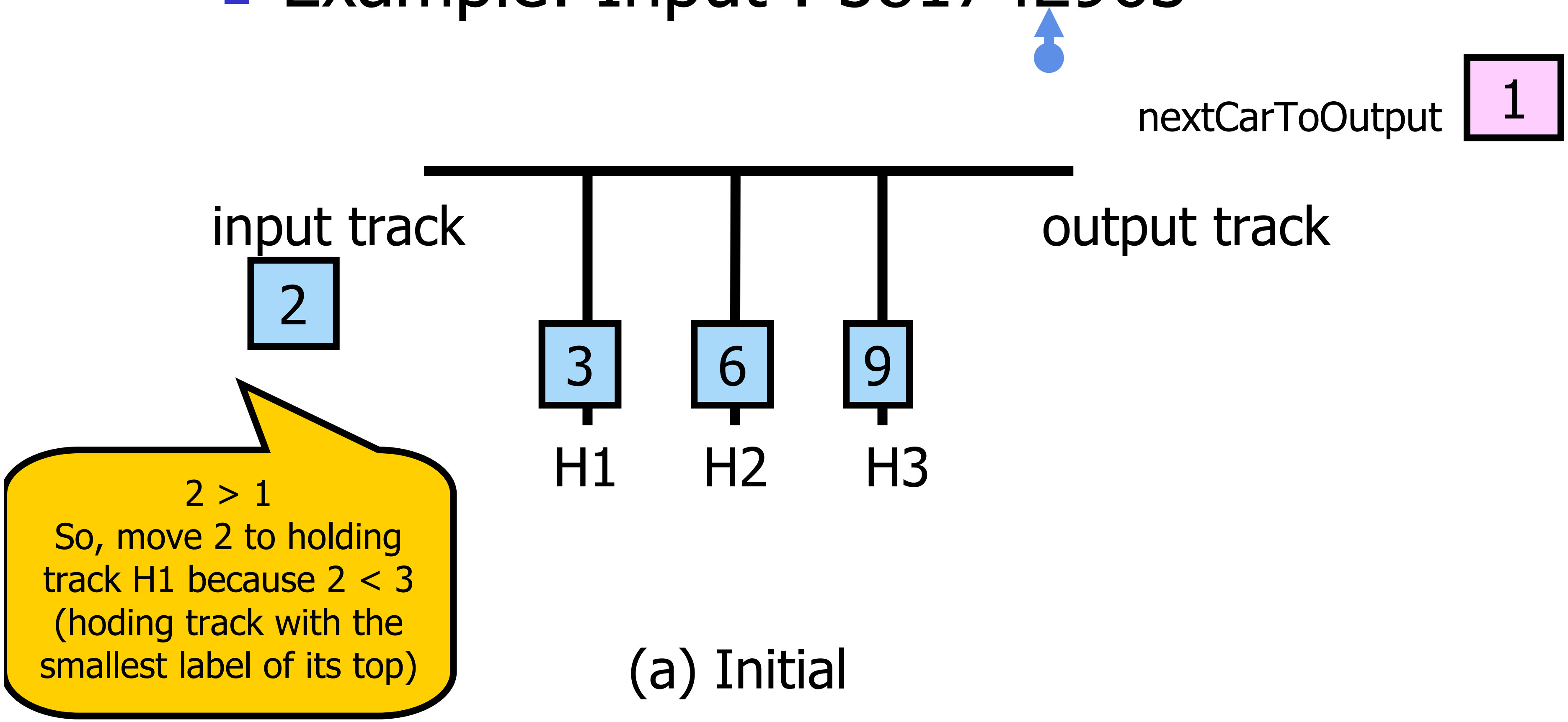
- Example: Input : 581742963



(a) Initial

Rearranging Railroad Cars (4/17)

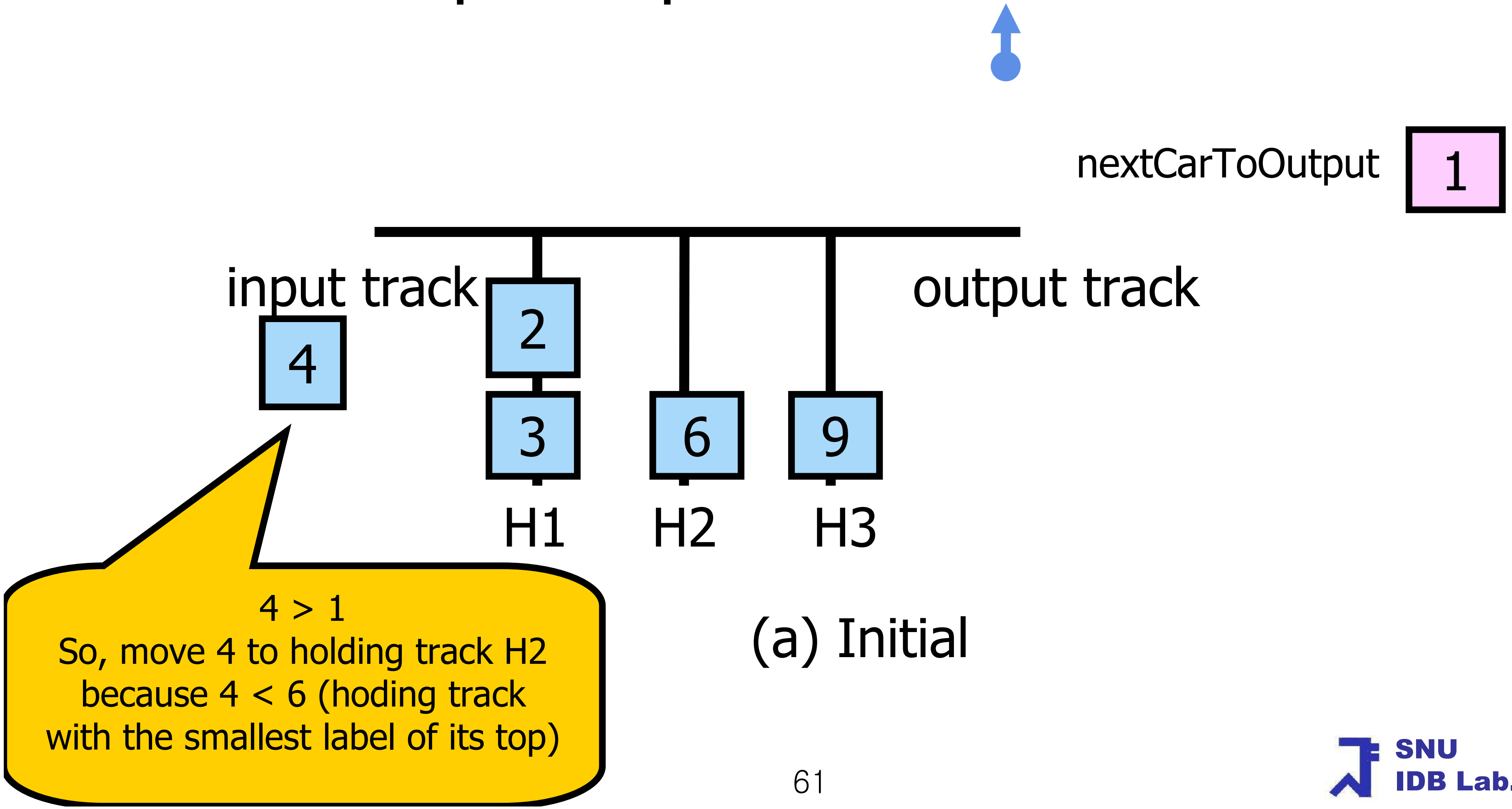
- Example: Input : 581742963



(a) Initial

Rearranging Railroad Cars (5/17)

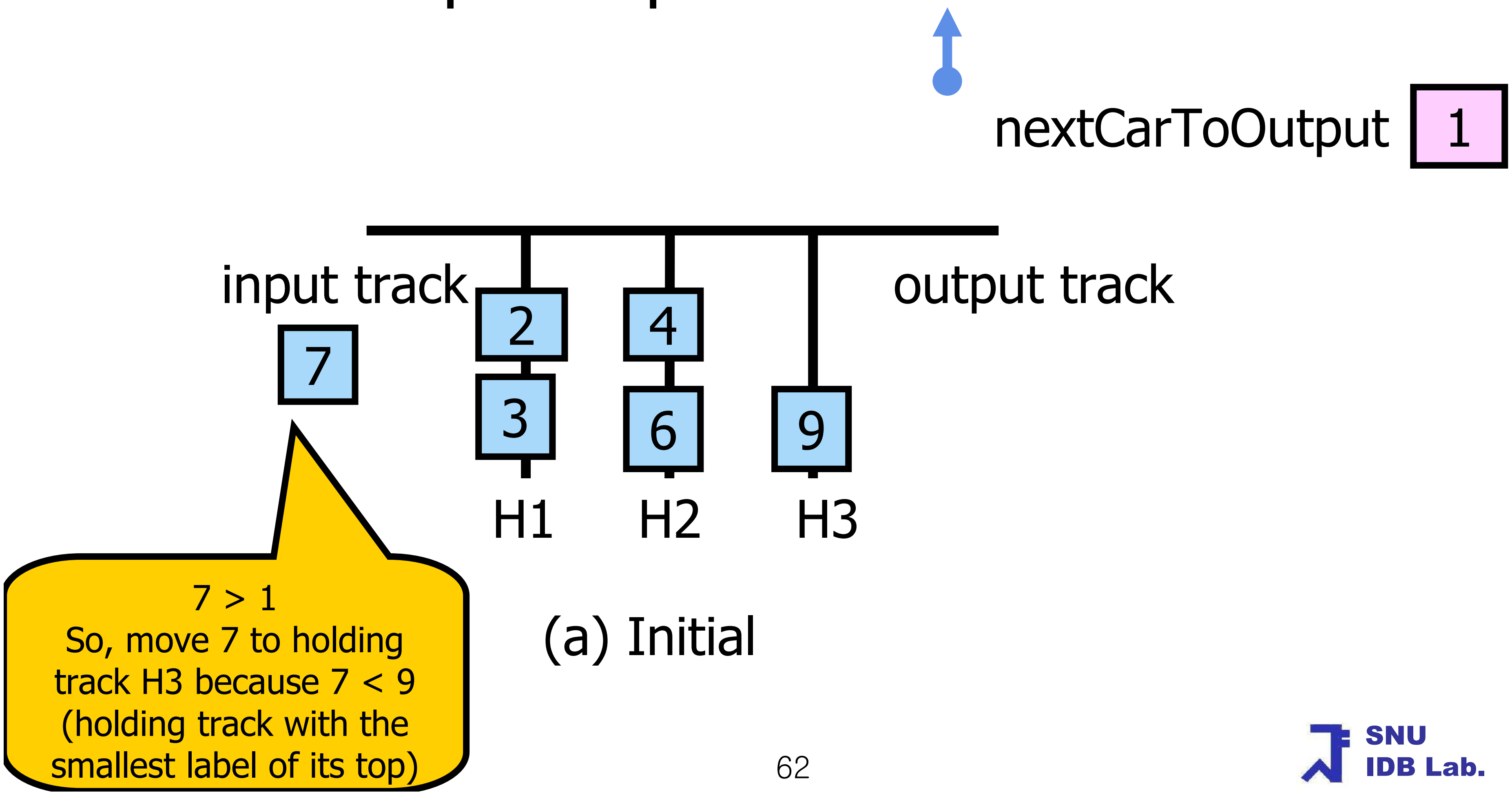
- Example: Input : 581742963



(a) Initial

Rearranging Railroad Cars (6/17)

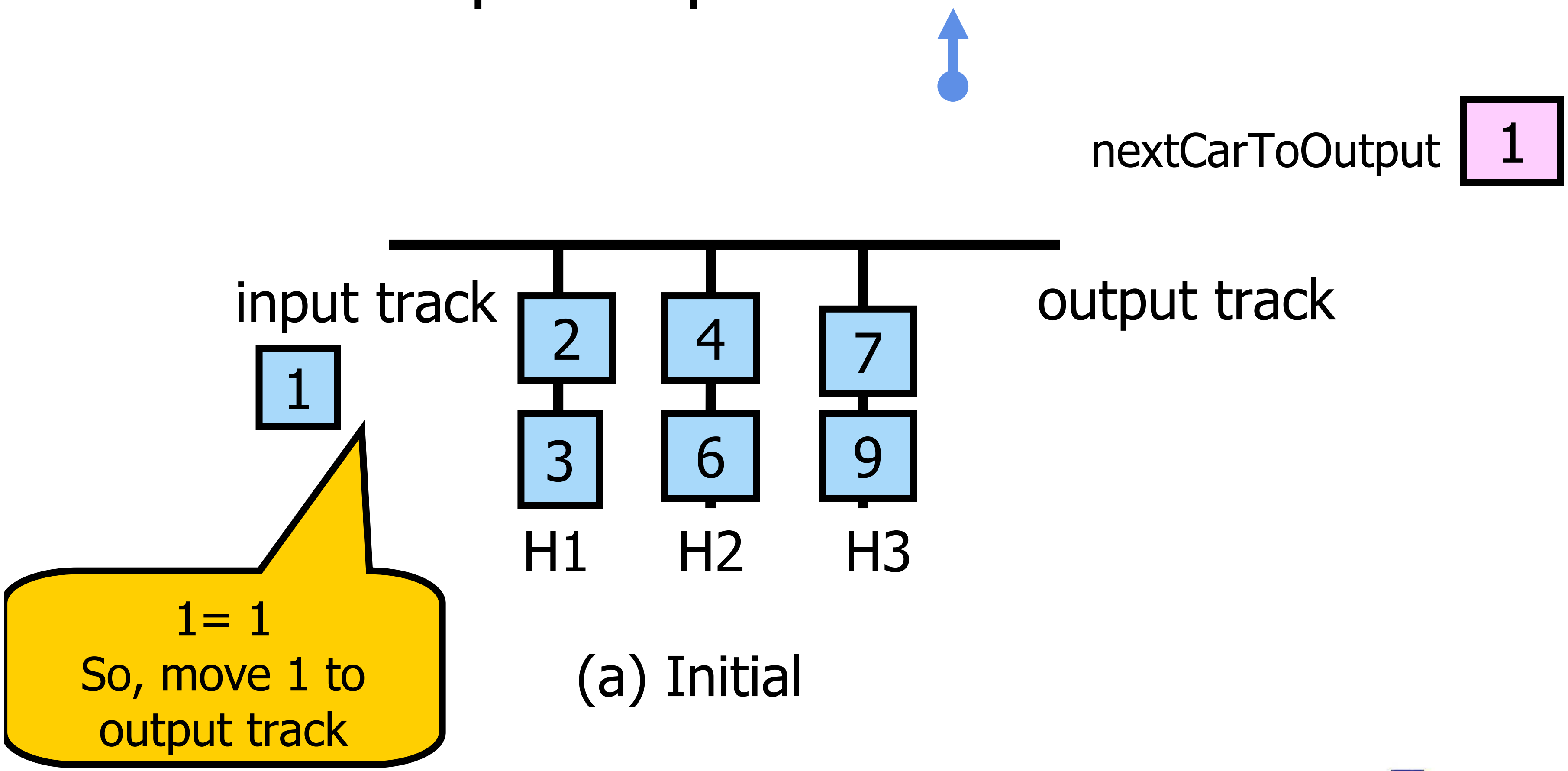
- Example: Input : 581742963



(a) Initial

Rearranging Railroad Cars (7/17)

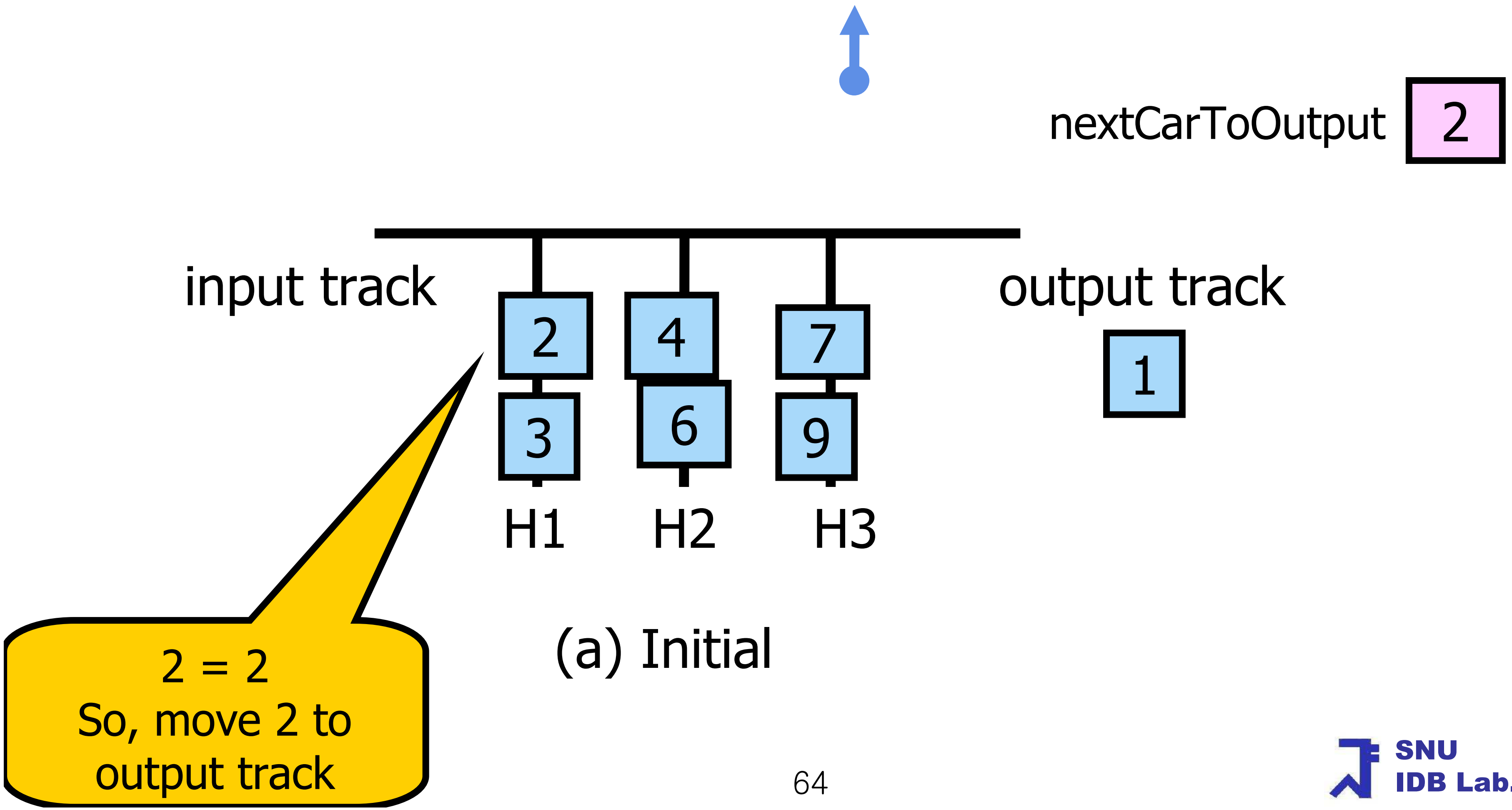
- Example: Input : 581742963



(a) Initial

Rearranging Railroad Cars (8/17)

- Example: Input : 581742963

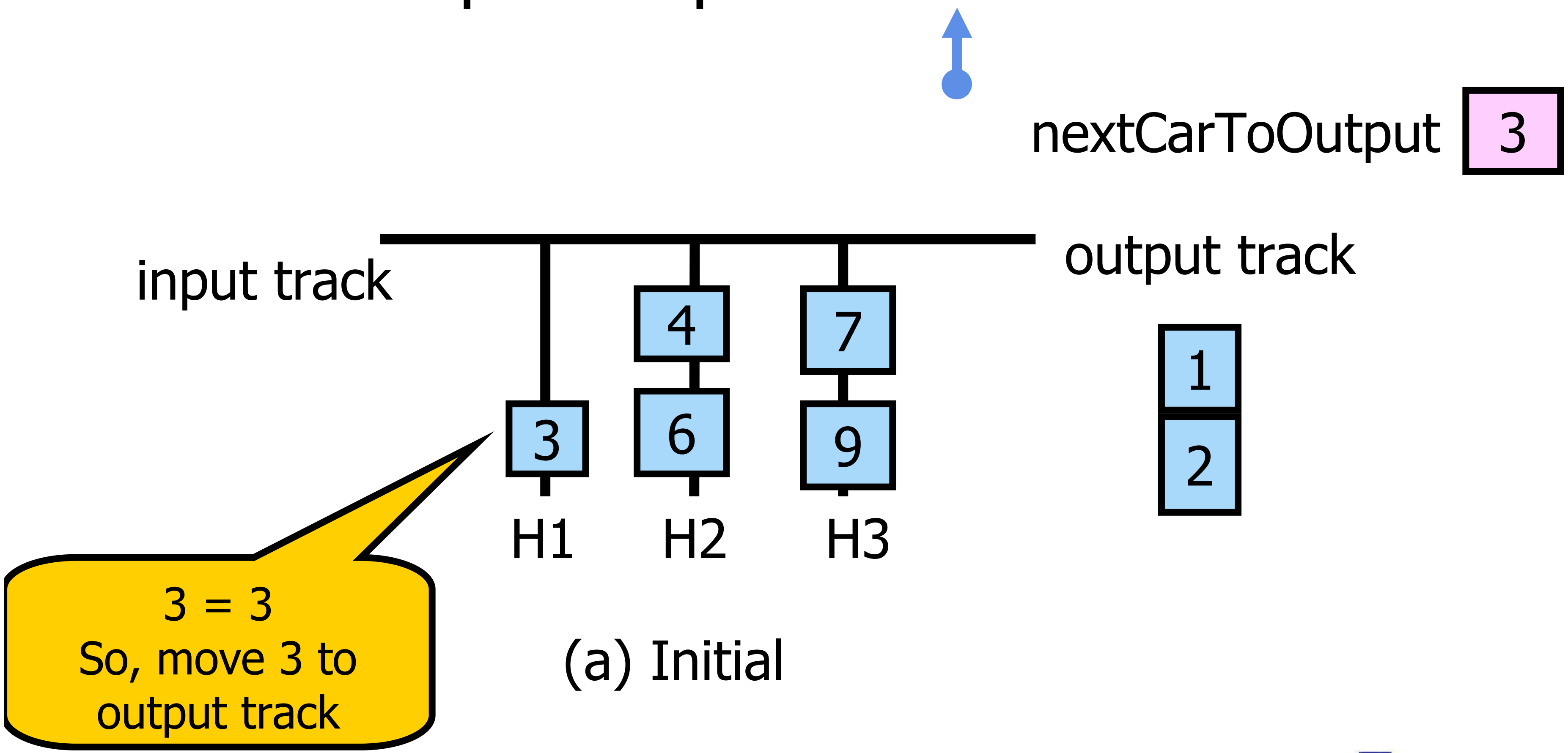


(a) Initial

SNU IDB Lab.

Rearranging Railroad Cars (9/17)

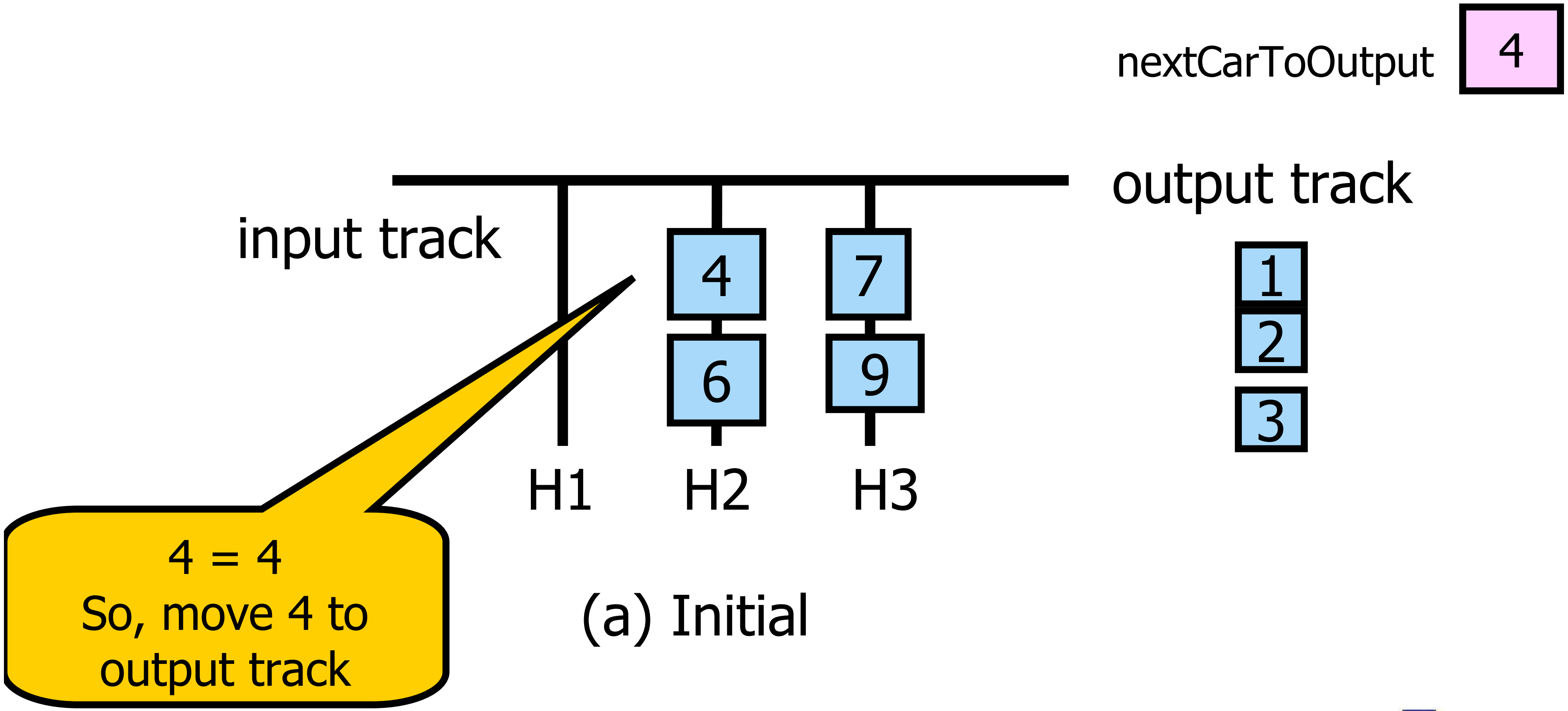
- Example : Input : 581742963



(a) Initial

Rearranging Railroad Cars (10/17)

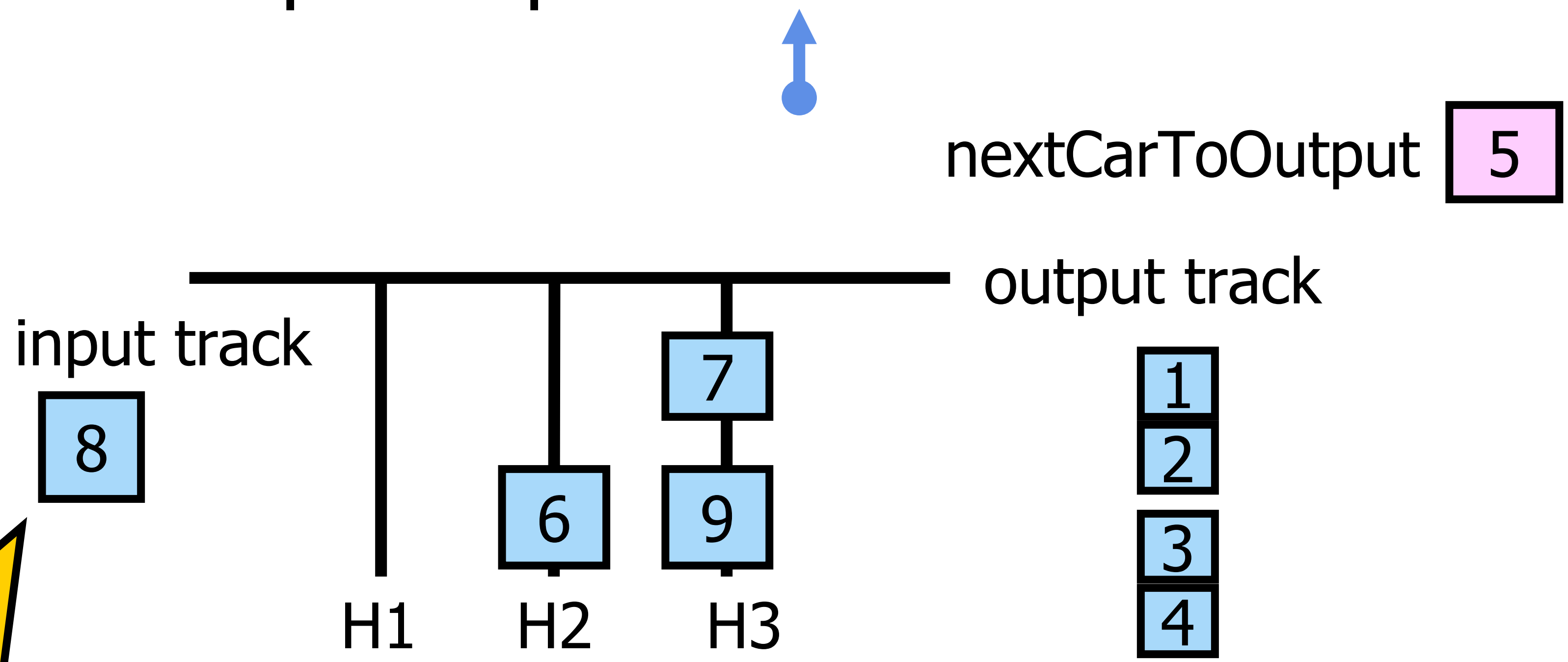
- Example : Input : 581742963



(a) Initial

Rearranging Railroad Cars (11/17)

- Example: Input : 581742963

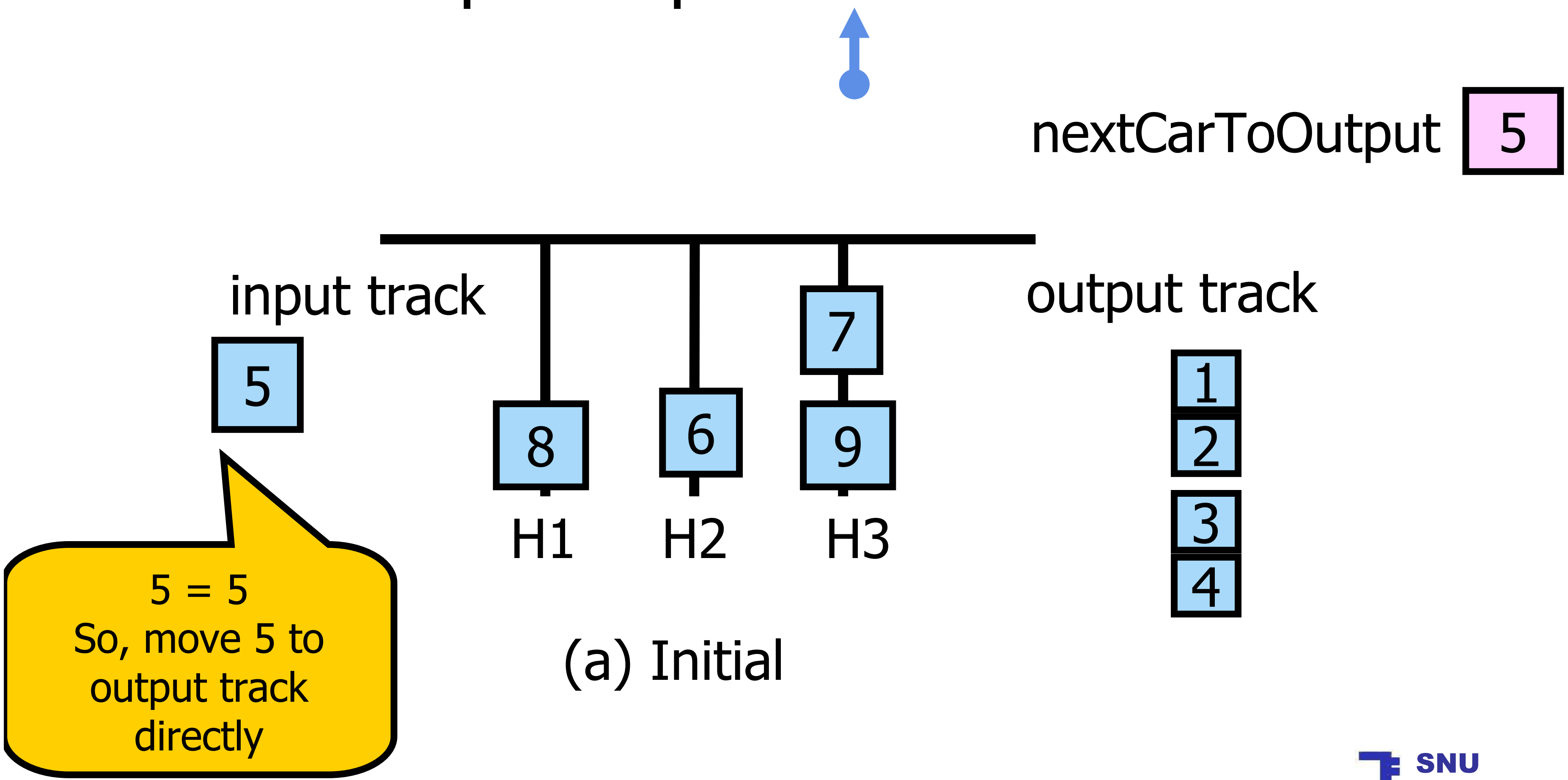


8 > 5
So, move 8 to holding track H1
(hoding track with the smallest
label of its top)

(a) Initial

Rearranging Railroad Cars (12/17)

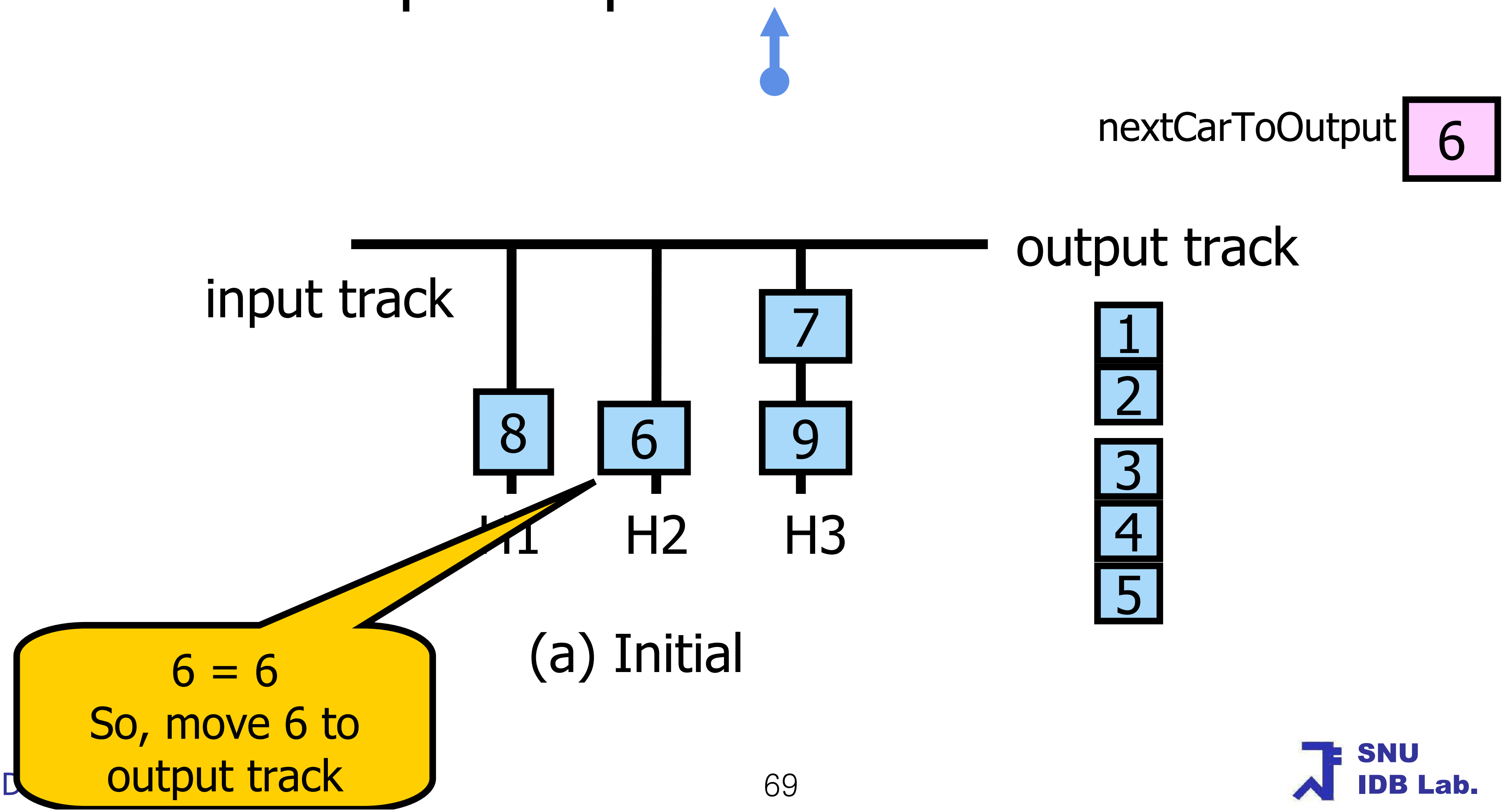
- Example: Input : 581742963



(a) Initial

Rearranging Railroad Cars (13/17)

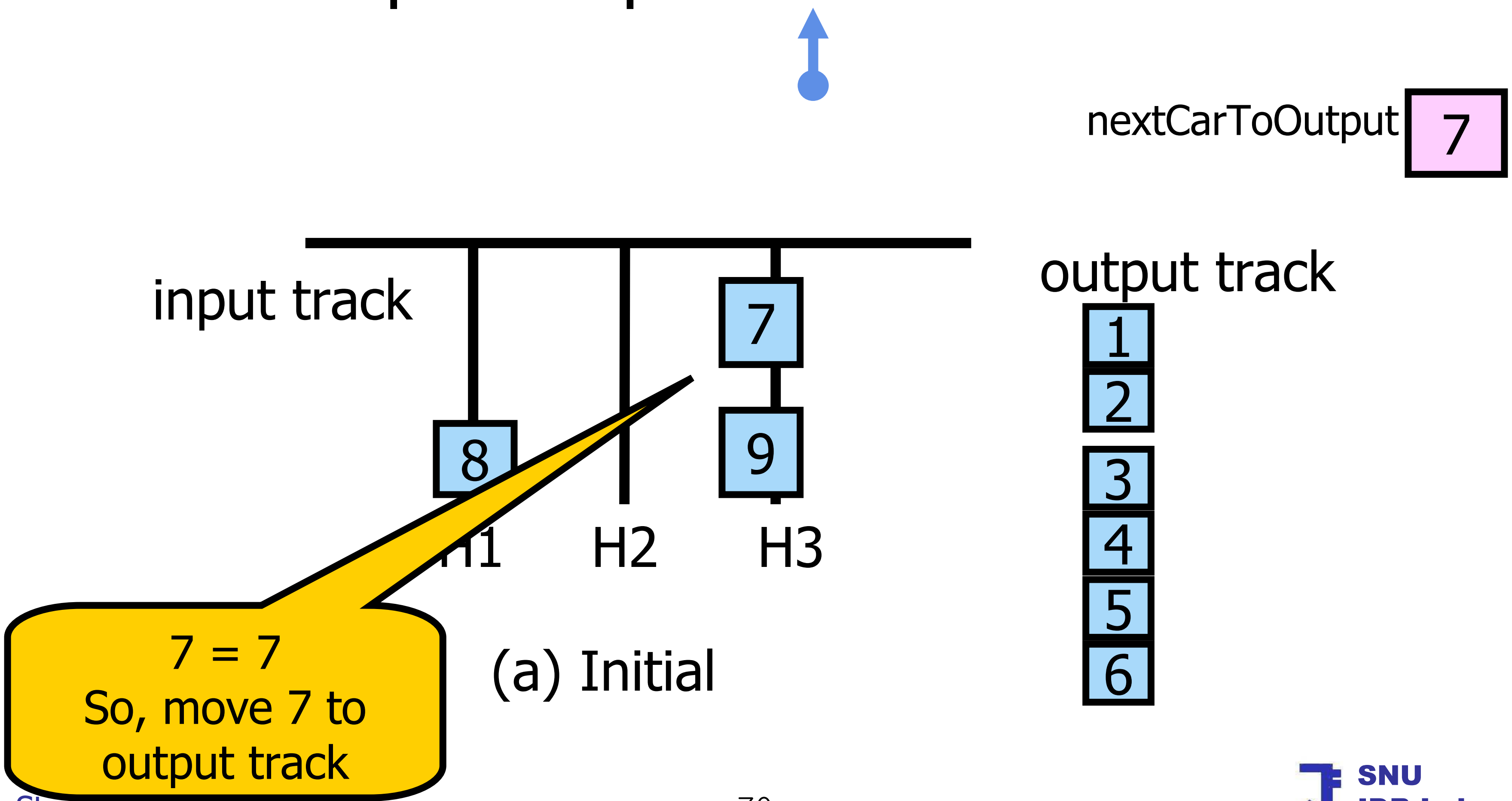
- Example: Input : 581742963



(a) Initial

Rearranging Railroad Cars (14/17)

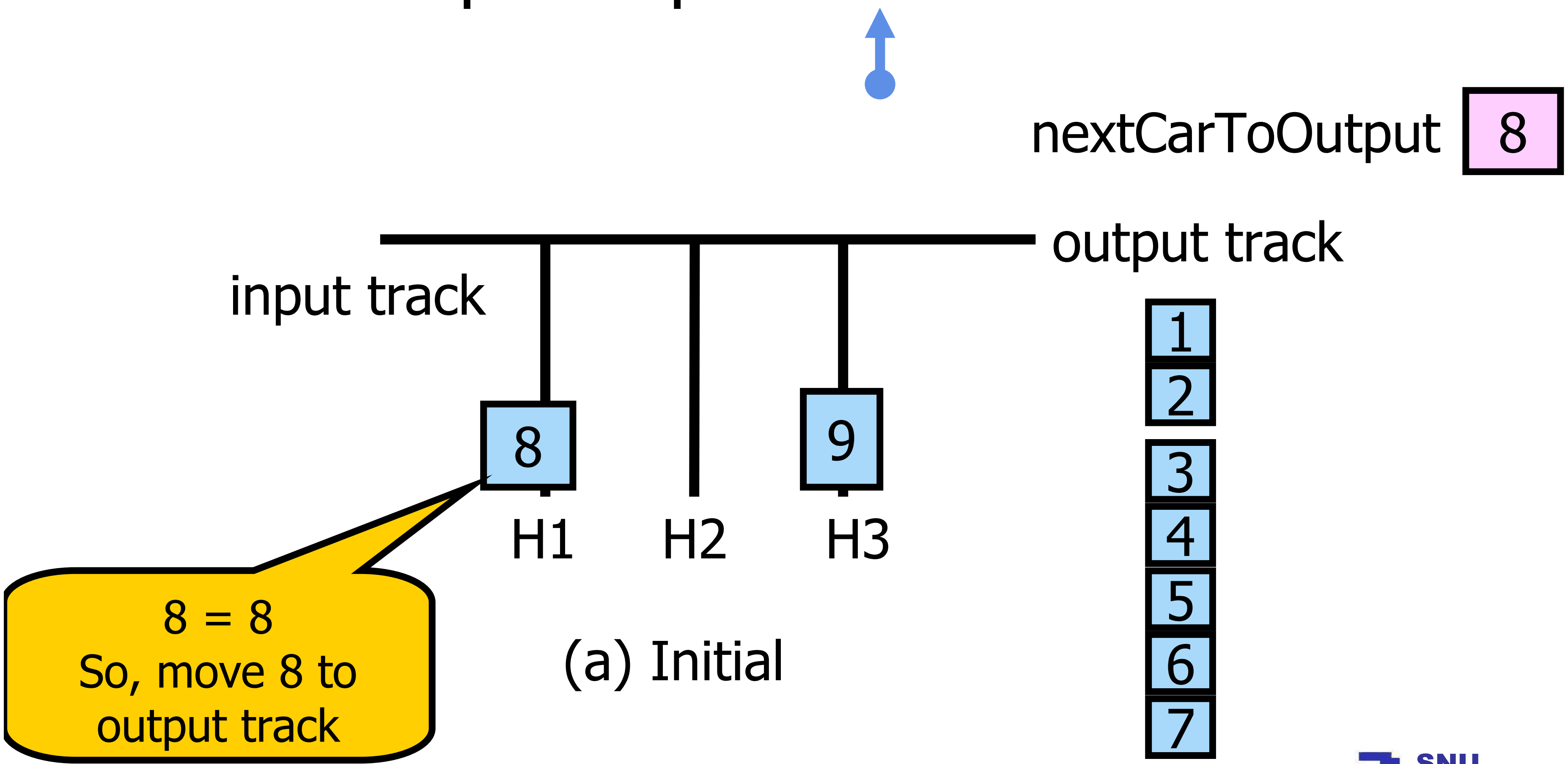
- Example : Input : 581742963



(a) Initial

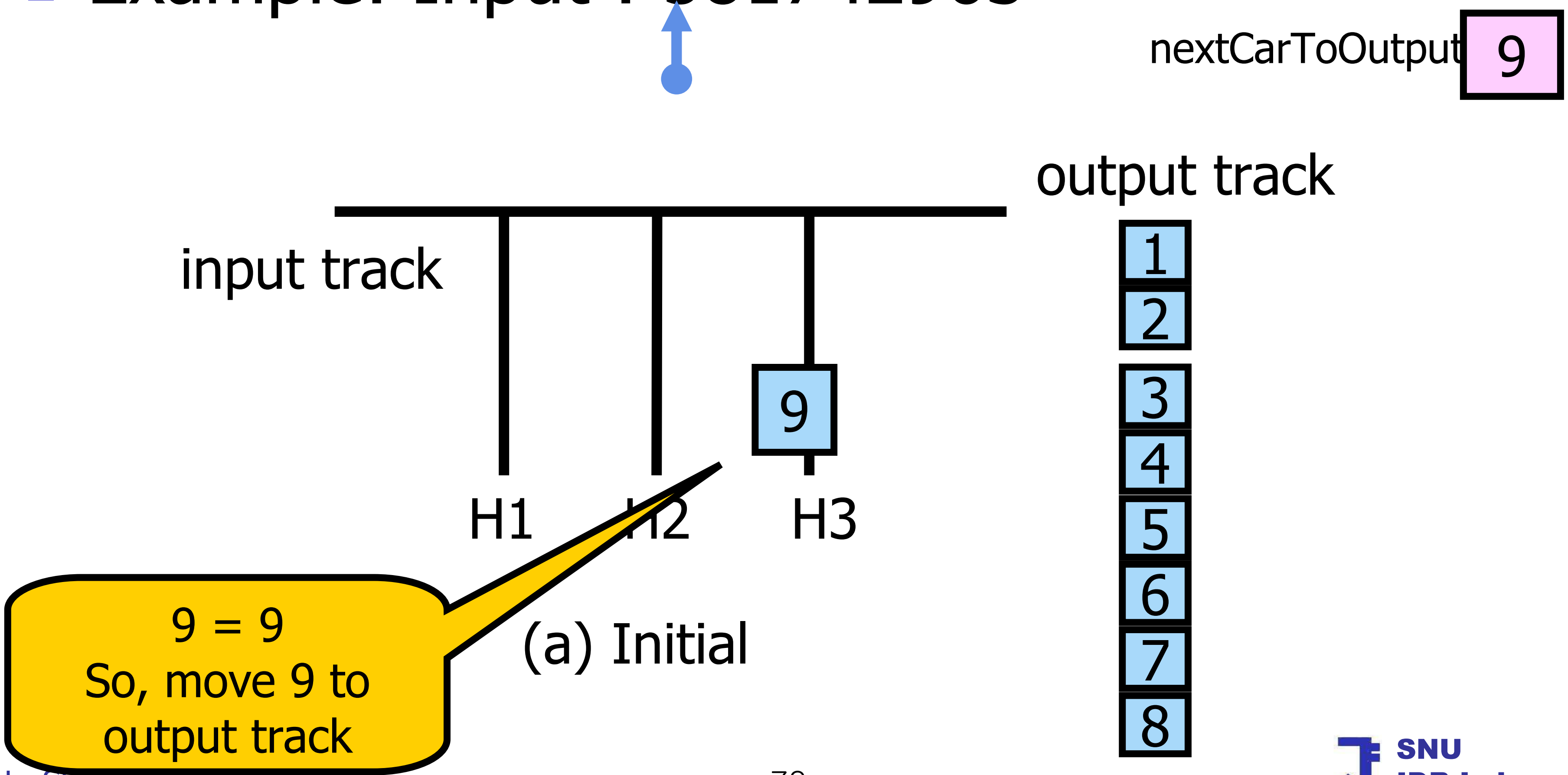
Rearranging Railroad Cars (15/17)

- Example: Input : 581742963



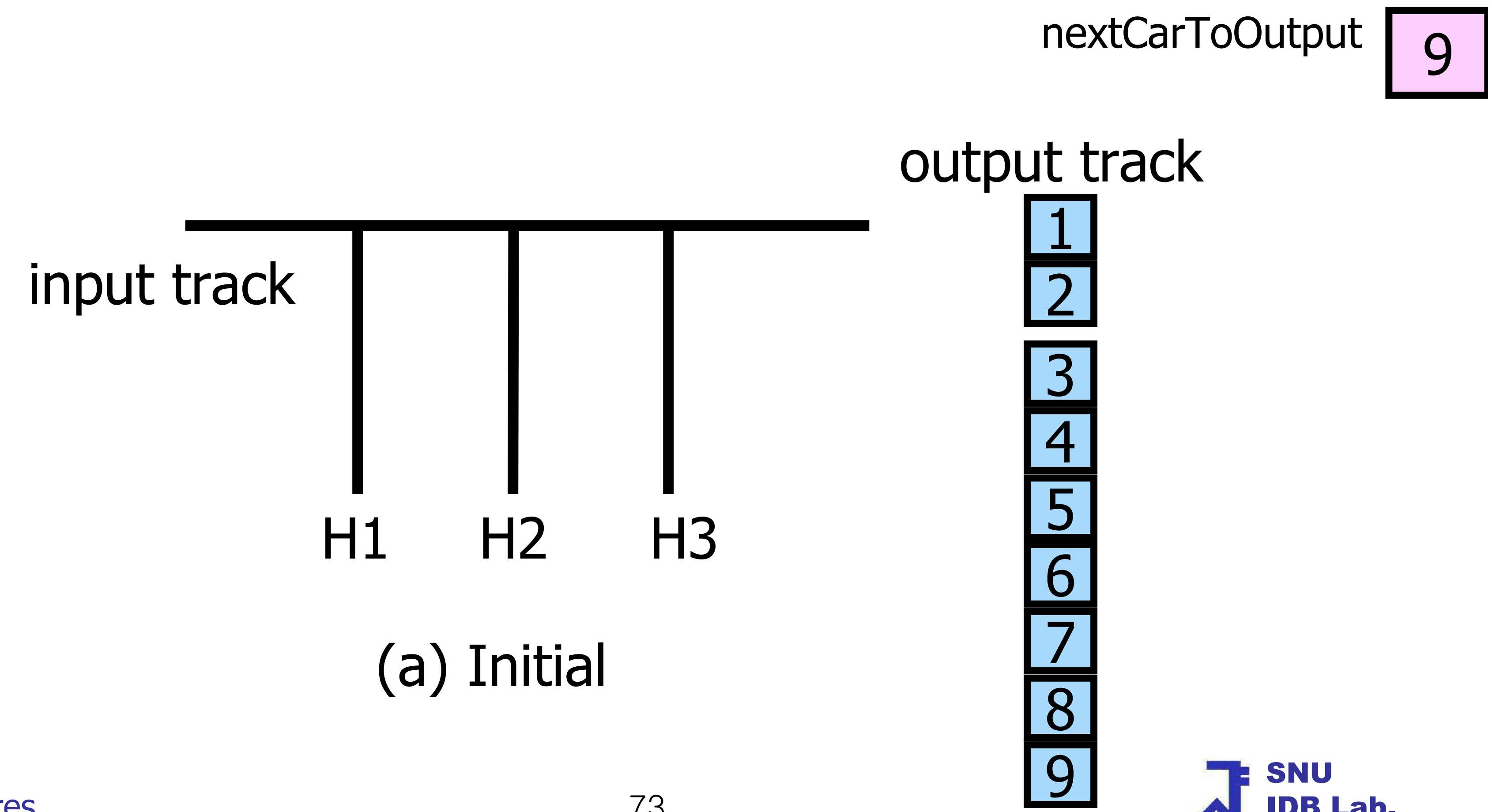
Rearranging Railroad Cars (16/17)

- Example: Input : 581742963



Rearranging Railroad Cars (17/17)

- Example: Input : 581742963



(a) Initial

RailRoadwithStacks (1)

Use array-based stacks

```
public class RailRoadWithStacks {
// data members:
ArrayStack[], inputOrder[], numOfCars, numOfTracks, smallestCar, itsTrack

//methods
/* * rearrange railroad cars beginning with the initial order inputOrder[]
railroad (int [] inputOrder, int NoOfCars, int NoOfTracks)
/* * output the smallest car from the holding tracks
outputFromHoldingTrack ()
/* * put car c into a holding track
putInHoldingTrack (int c)
}
```

RailRoadwithStack (2)

```
public static boolean railroad (int [] inputOrder, int NoOfCars, int NoOfTracks){
 numOfCars     =  NoOfCars;
 numOfTracks =  NoOfTracks;
 track = new ArrayStack [numOfTracks + 1]; // create stacks as holding tracks
 for (int i = 1; i <= numOfTracks; i++)  track[i] = new ArrayStack();
 int nextCarToOutput = 1;
 smallestCar              = numOfCars + 1; // no car in holding tracks

 for (int i = 1; i <= numOfCars; i++) { // rearrange cars
        if (inputOrder[i] == nextCarToOutput) {  // send car to output track
            nextCarToOutput++;
            while (smallestCar == nextCarToOutput){ //output from holding tracks
                    outputFromHoldingTrack();
                    nextCarToOutput++;}
        } else   if ( !putInHoldingTrack(inputOrder[i]) )     return false;
  }    return true;
}
```

RailRoadwithStack (3)

- Move a car from a holding track to the output track

```
private static void outputFromHoldingTrack() {
    track[itsTrack].pop(); // remove smallestCar from itsTrack

    // find new smallestCar and itsTrack by checking top of all stacks
    smallestCar = numOfCars + 2;
    for (int i = 1; i <= numOfTracks; i++)
      if (!track[i].empty() &&  ((Integer) track[i].peek()).intValue() < smallestCar) {
         smallestCar = ((Integer) track[i].peek()).intValue();
         itsTrack = i;
      }
}
```

RailRoadwithStack (4)

- Put car c into a holding track using the assignment rule

```
private static boolean putInHoldingTrack(int c) { // find best holding track for car c
    int bestTrack  = 0
    int bestTop     = numOfCars + 1;
    for (int i = 1; i  <=  numOfTracks; i++)  { // scan tracks
      if (!track[i].empty()) { // track i not empty
         int topCar = ((Integer) track[i].peek()).intValue();
         if (c < topCar && topCar < bestTop){   // track i has smaller car at top
           bestTop = topCar;
           bestTrack = i;}
      }  else if (bestTrack == 0) bestTrack = i; // track i empty
    }
    if (bestTrack == 0)   return false;              // no feasible track
    track[bestTrack].push(new Integer(c));   // add c to bestTrack
    if (c < smallestCar) {   // update smallestCar and itsTrack if needed
        smallestCar = c;
        itsTrack = bestTrack; }
    return true;
}
```

Complexity: RailRoadWithStack

- Complexity of RailRoad()
  - outputFromHoldingTrack() ➔ O(numOfCars * numOfTracks)
  - putInHoldingTrack() ➔ O(numOfCars * numOfTracks)
  - The rest of the code ➔ Θ(numOfCars)
  - The overall complexity ➔ O(numOfCars * numOfTracks)

- If a balanced binary search tree is used for storing the labels of the cars at the top of the holding tracks, finding the holding track can be performed efficiently
  - The complexity can be reduced to

    O(numOfCars * log(numOfTracks))

Table of Contents

- Stack Applications
  - Parenthesis Matching
  - Towers of Hanoi
  - Rearranging Railroad Cars
  - Offline Equivalence Class Problem
  - Rat in a Maze

Offline Equivalence Class Problem (1)

- Problem
  - Input :
    - the number of element: n
    - the number of relation pairs: r
    - the r relation pairs
  - Output : partition the n elements into equivalence classes
- Example
  - N=9, r=11
  - R = 11 relation pairs

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

Offline Equivalence Class Problem (2)

- Solution Strategy
  - The 1st phase : Input the data and set up n lists to represent the relation pairs
    - Each relation pair(i,j), i is put on list[j] and j is put on list[i] :
      { (1,5), (1,6), (3,7), (4,8), (5,2),(6,5),
      (4,9), (9,7), (7,8), (3,4), (6,2) }

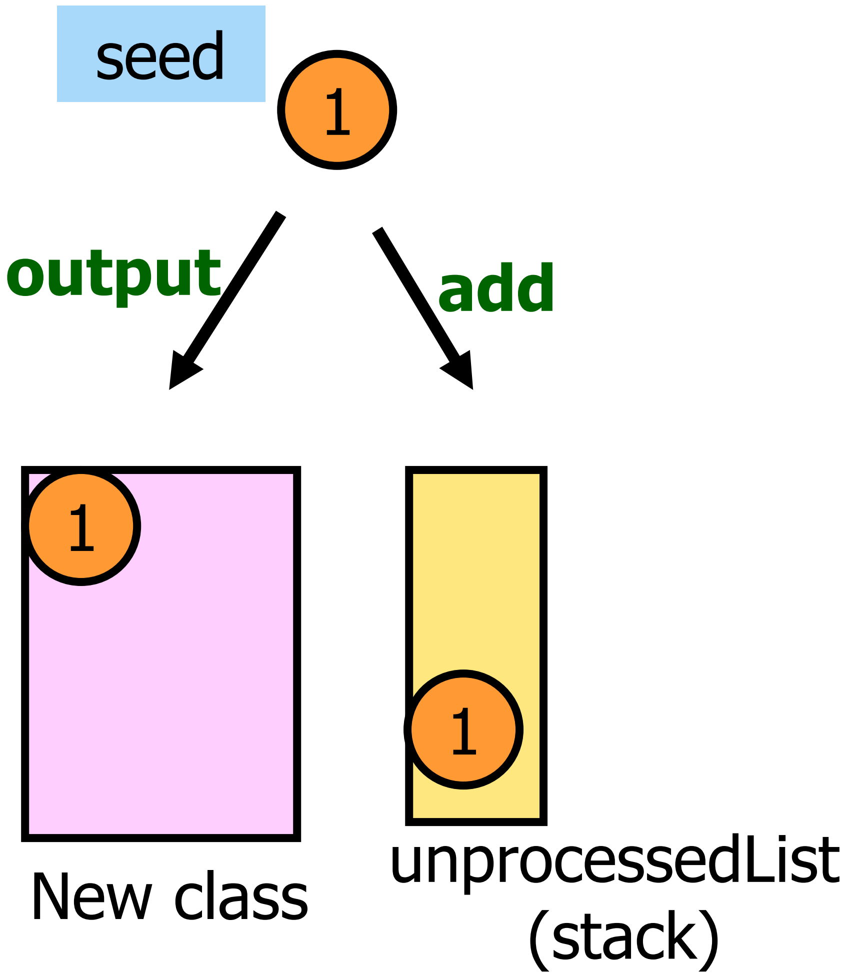
list[1] = [5, 6], list[2] = [5, 6], list[3] = [7, 4],
list[4] = [8, 9, 3], list[5] = [1, 2, 6], list[6] = [1, 2, 5],
list[7] = [3, 9, 8], list[8] = [4, 7], list[9] = [4, 7]

Offline Equivalence Class Problem (3)

- The 2nd phase : The equv classes(EC) are identified by locating a seed
  1. Find a seed that is output as the first member of the next EC
  2. The seed is put onto a list, *unprocessedList (Stack)*, of elements that are in the same EC
  3. Remove an element i from *unprocessedList*
  4. Output and add to *unprocessedList* elements on *list[i]* that haven't already been identified as class members
  5. Until the *unprocessedList* becomes empty, Continues the process 3&4.
     - If it is empty, we have completed a class
  6. Proceed to find a seed for the next class

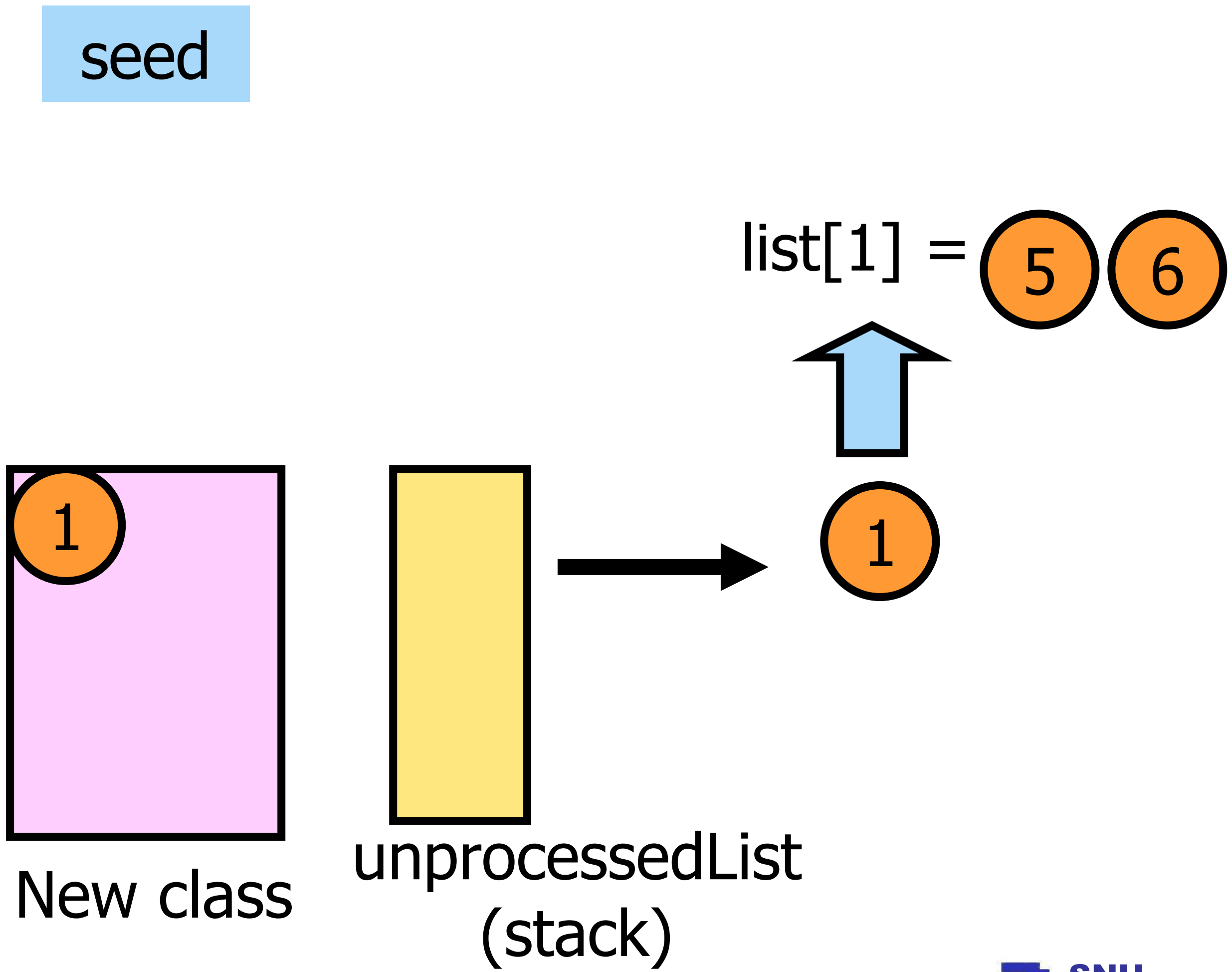
Equivalence Class with Stack (1)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]



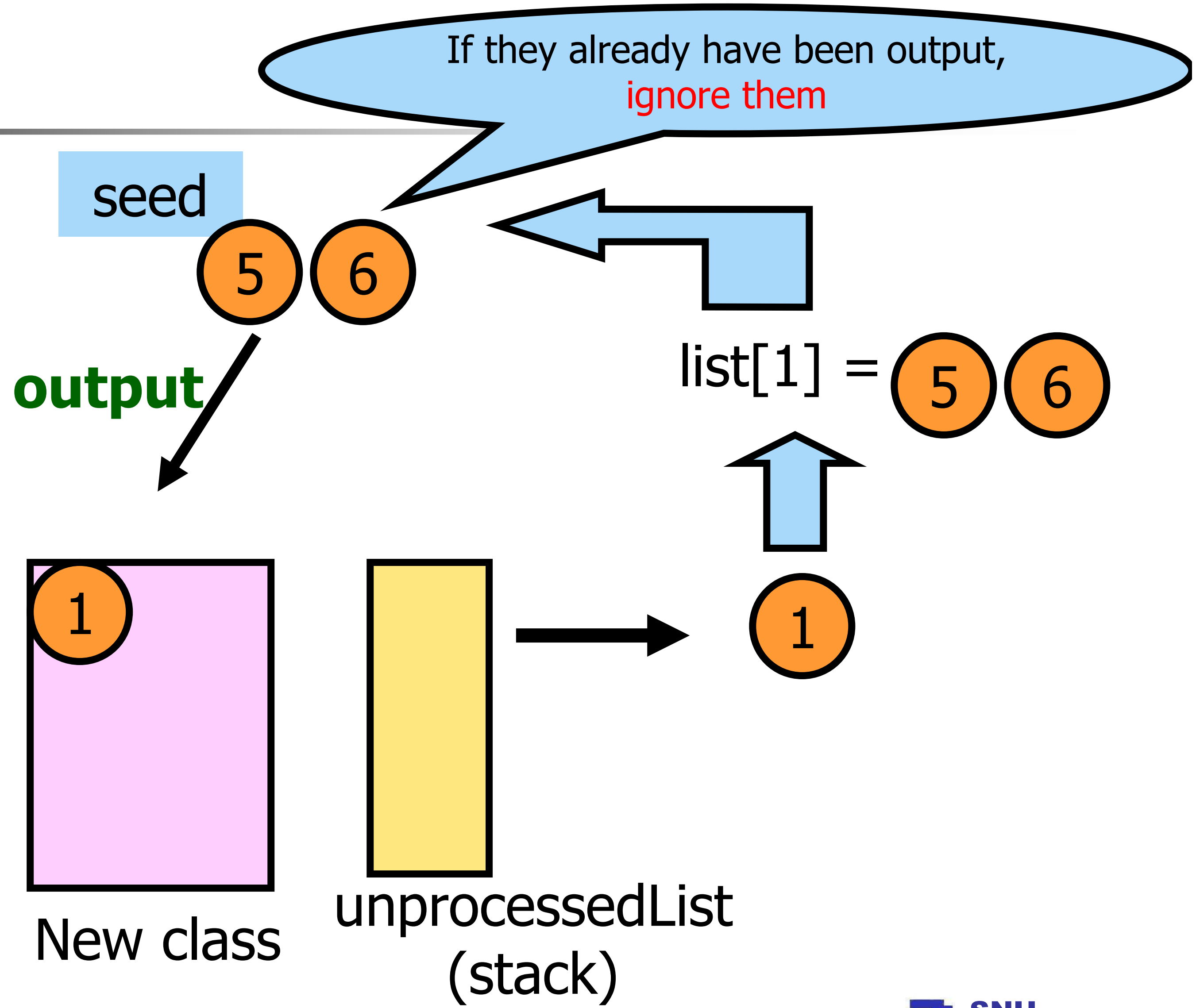
Equivalence Class with Stack (2)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]



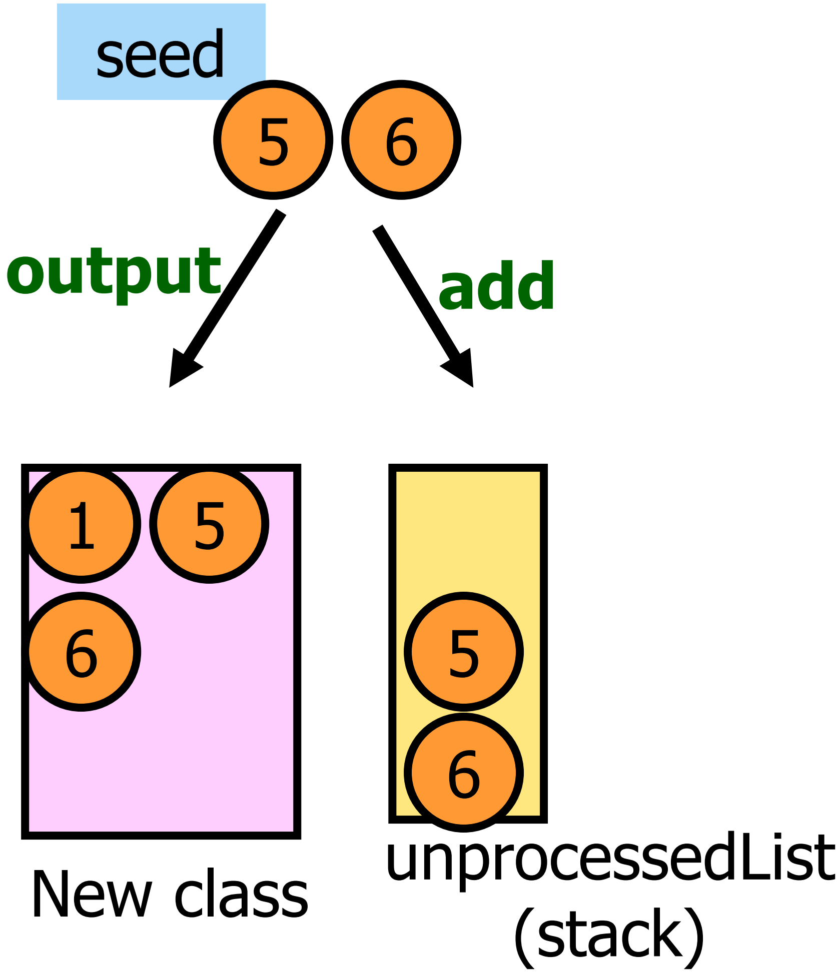
Equivalence Class with Stack (3)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]

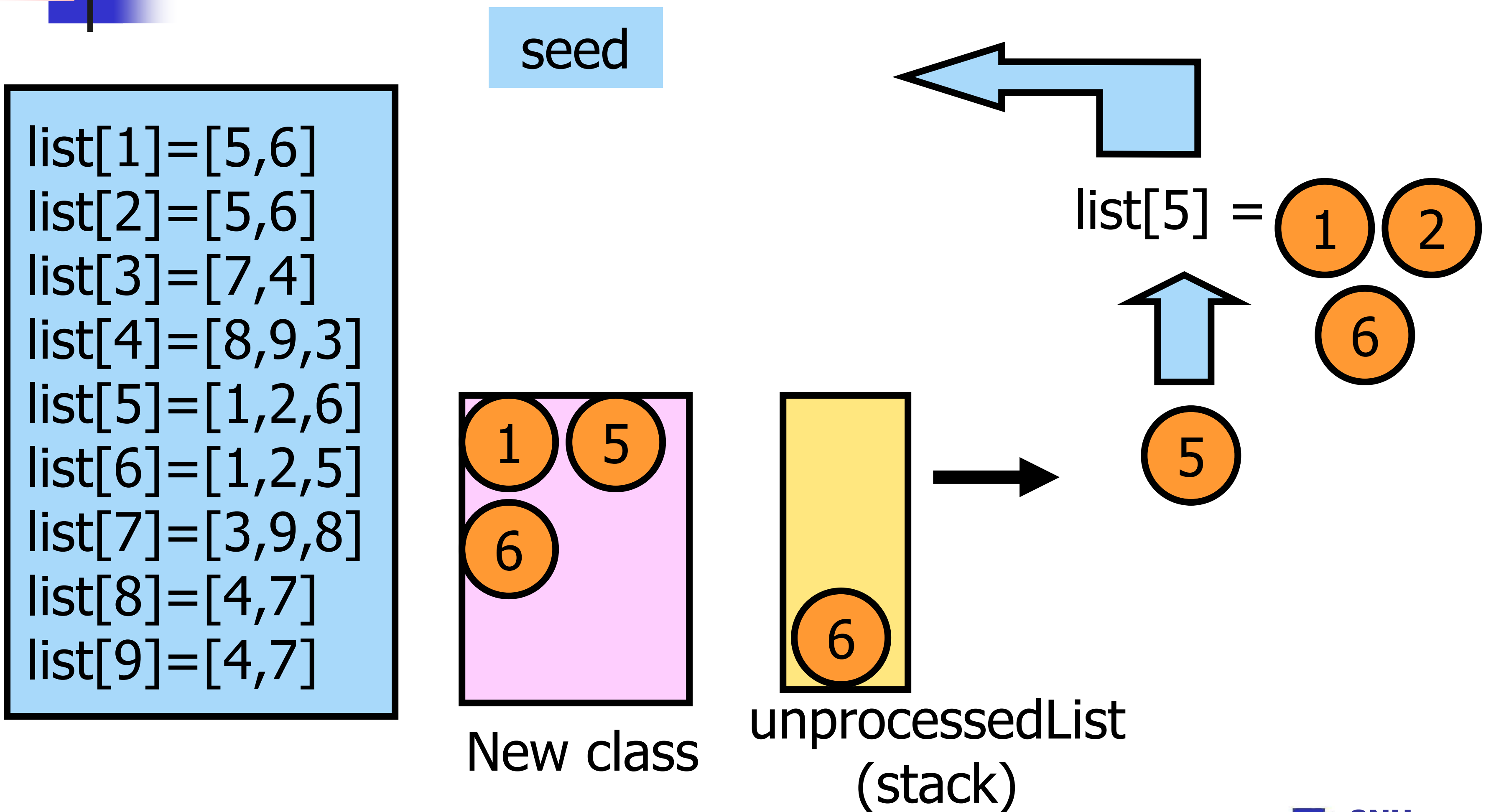


Equivalence Class with Stack (4)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]

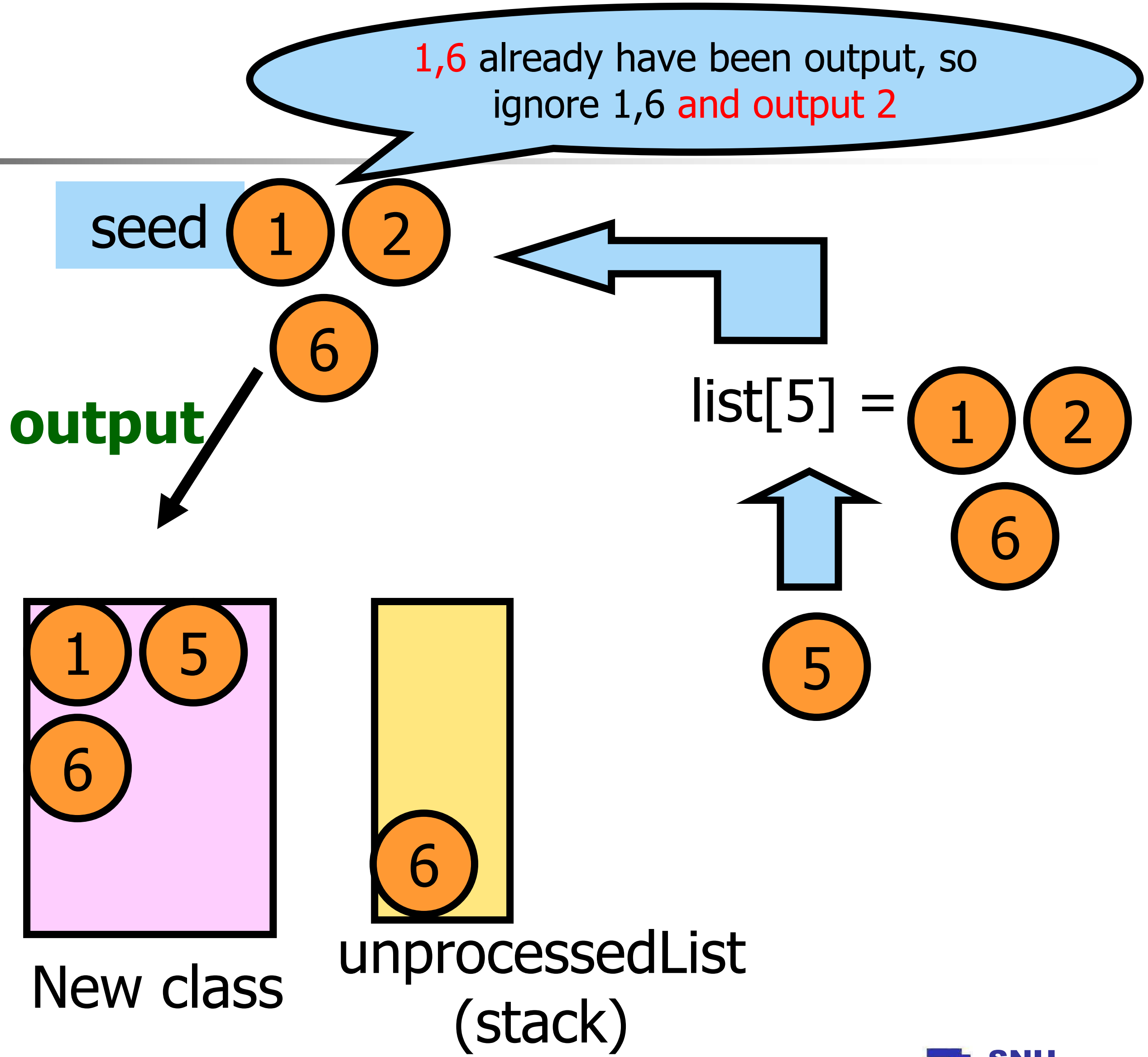


Equivalence Class with Stack (5)



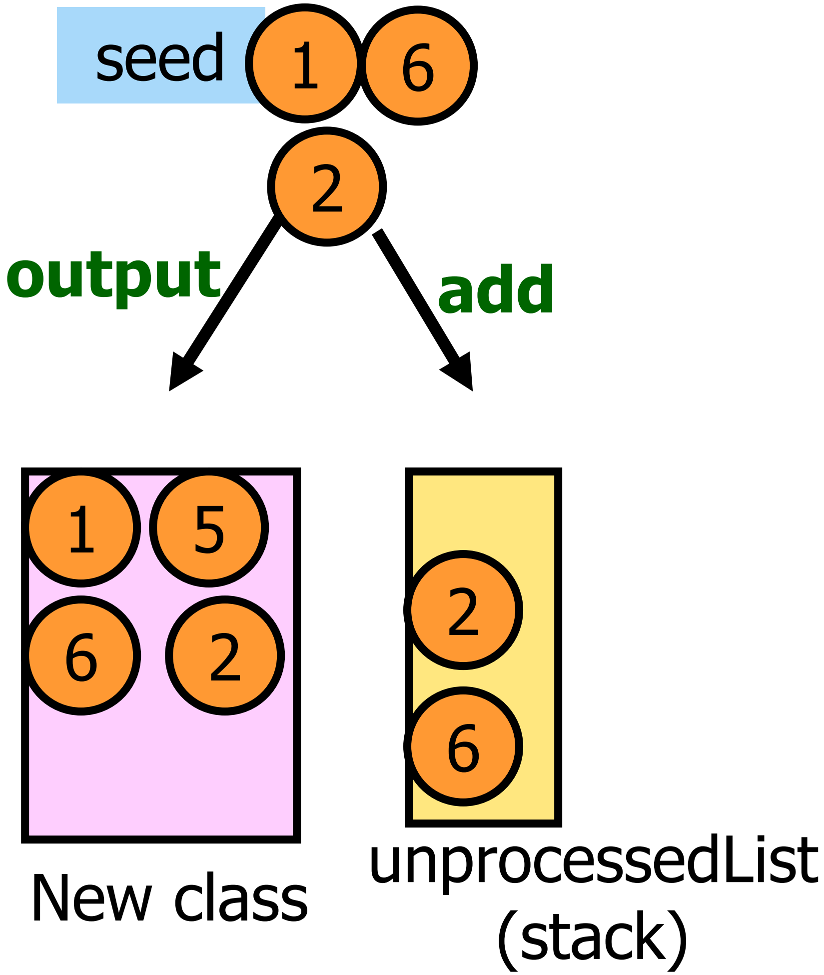
Equivalence Class with Stack (6)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]



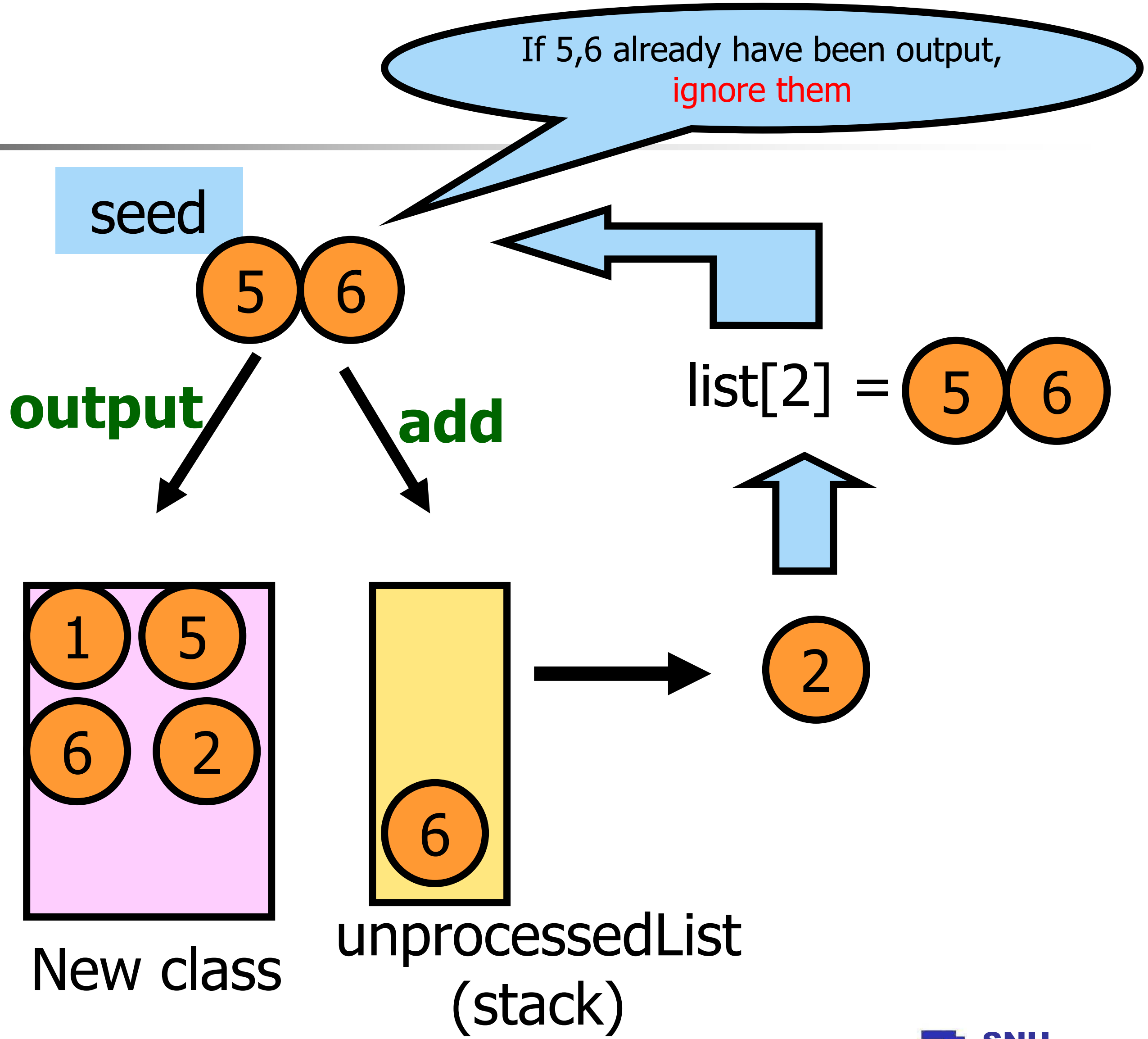
Equivalence Class with Stack (7)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]



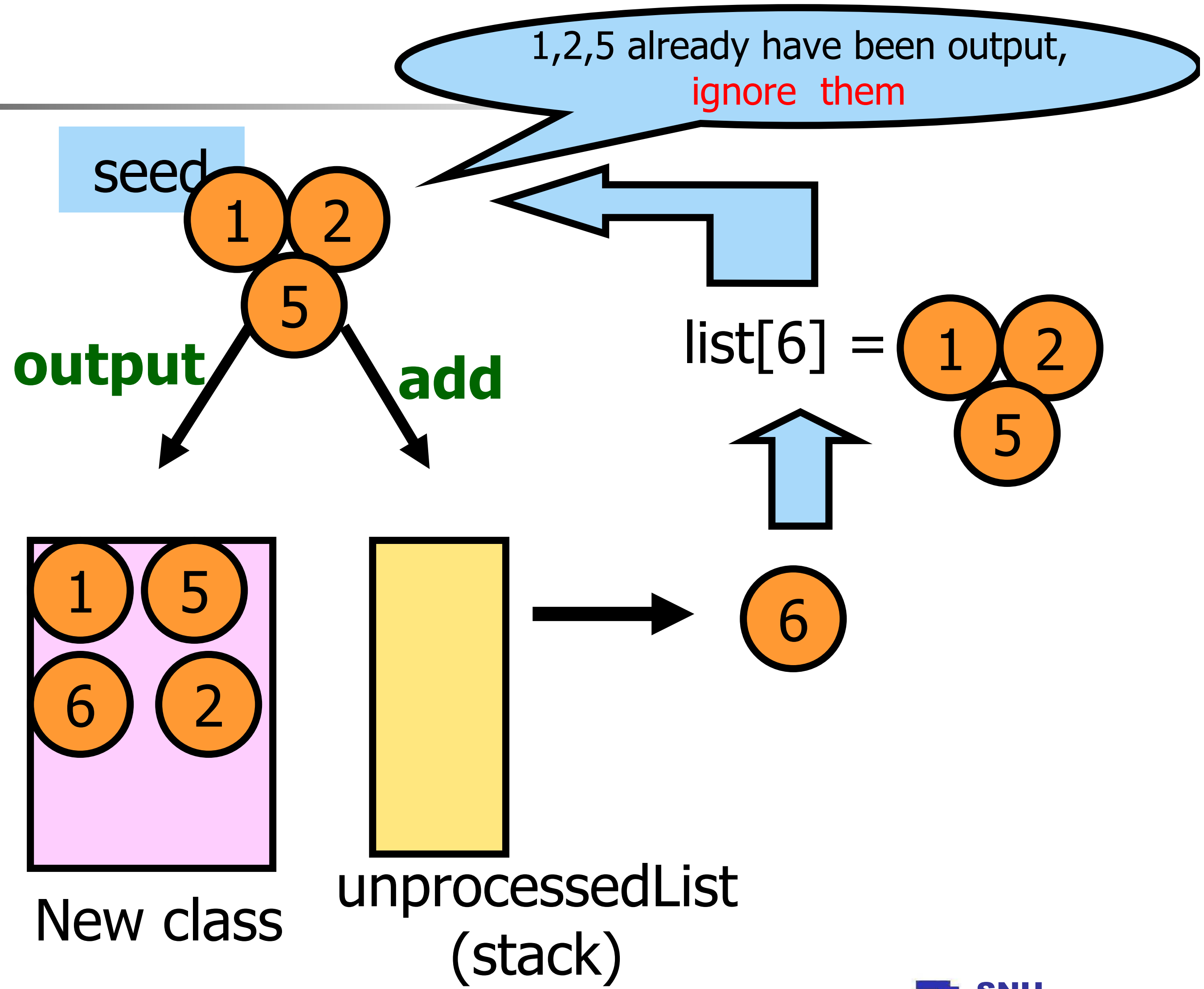
Equivalence Class with Stack (8)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]



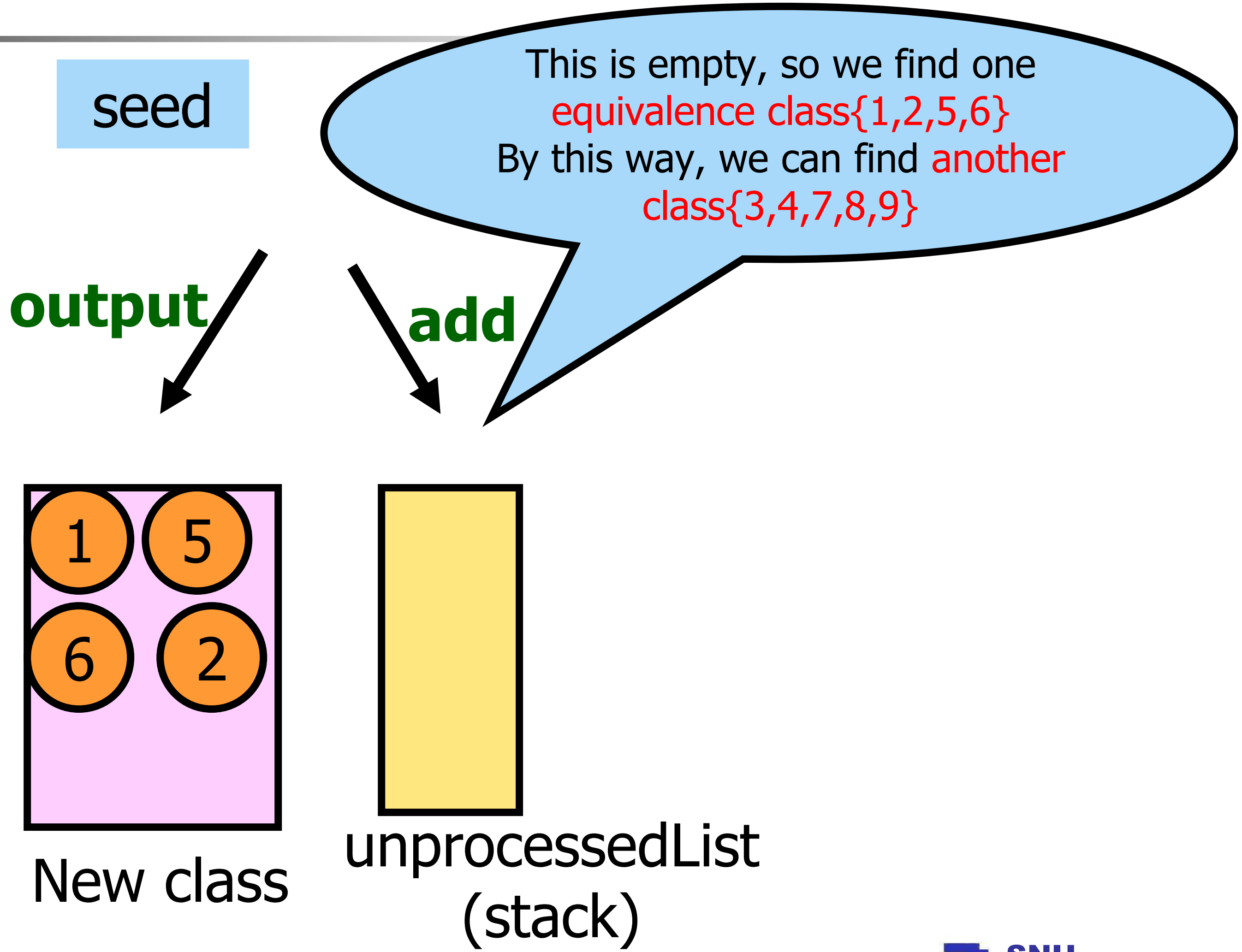
Equivalence Class with Stack (9)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]



Equivalence Class with Stack (10)

list[1]=[5,6]
list[2]=[5,6]
list[3]=[7,4]
list[4]=[8,9,3]
list[5]=[1,2,6]
list[6]=[1,2,5]
list[7]=[3,9,8]
list[8]=[4,7]
list[9]=[4,7]



Equivalence Class with Stack: Code (1)

- Operations
  - Insert and examine all elements
- Select a representation for *list* and *unprocessedList*
  - Space requirement
    - Between 2r (stack) and 4r (linked list) references
  - Time performance
    - The linked implementations are slower than array-based counterparts.
  - So, *unprocessedList* and *list* is implemented as an ArrayStack

Equivalence Class with Stack: Code (1)

```
public static void main (String [] args) {
    // input the number of elements, n
    // input the number of relations, r
    // create an array of empty stacks, list[0] not used
    ArrayStack [] list = new ArrayStack [n + 1];
    for (int i = 1; i <= n; i++)    list[i] = new ArrayStack();

    for (int i = 1; i <= r; i++) { // input the r relations and put on stacks
       int a = keyboard.readInteger();
       int b = keyboard.readInteger();
       list[a].push(new Integer(b));
       list[b].push(new Integer(a));  }

     // initialize to output equivalence classes
     ArrayStack unprocessedList = new ArrayStack();
     boolean [] out                  = new boolean [n + 1];
```

Equivalence Class with Stack: Code (2)

```
for (int i = 1; i <= n; i++)  // output equivalence classes
  if  (!out[i]) { // start of a new class
          out[i] = true;
          unprocessedList.push(new Integer(i)) ;
   while (!unprocessedList.empty()) { // get rest of class from unprocessedList
       int j = ((Integer) unprocessedList.pop()).intValue();
       while  (!list[j].empty()) { // elements on list[j] are in the same class
             int q = ((Integer) list[j].pop()).intValue();
             if (!out[q]) { // q not yet output
                System.out.print(q + " ");
                out[q] = true;
                unprocessedList.push(new Integer(q));  } //end of if
           } //end of while
         }  //end of while
      } //end of if
} // end of main
```

Complexity: Equv Class with Stack

- Input and initialize the array list[]
  - $O(n+r)$
- Pop and examine all elements
  - $\Theta(r)$

- So, overall complexity
  - $O(n+r)$
  - If no exception, $\Theta(n+r)$

Table of Contents

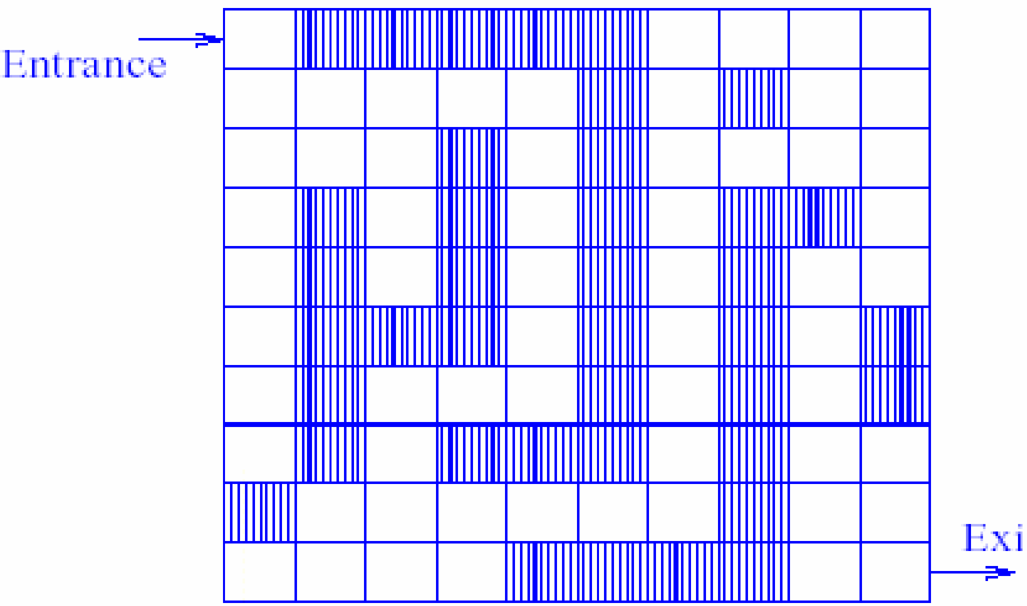
- Stack Applications
  - Parenthesis Matching
  - Towers of Hanoi
  - Rearranging Railroad Cars
  - Offline Equivalence Class Problem
  - Rat in a Maze

Rat in a Maze: Problem

- What is maze?
  - A rectangular area with an entrance and an exit
  - The interior of maze contains obstacles
- Suppose that maze is to be modeled as an n x m matrix
  - Position (1, 1) : entrance
  - Position (n, m) : exit
  - Each maze position : row and column intersection
  - Position (i, j) = 1 iff there is an obstacle
  - Position (i, j) = 0 otherwise
- Problem
  - Find a path from the entrance to the exit of a maze
  - A path is a sequence of position

Rat in a Maze: Representation

- Example



(a) A maze

0	1	1	1	1	1	0	0	0	0
0	0	0	0	0	1	0	1	0	0
0	0	0	1	0	1	0	0	0	0
0	1	0	1	0	1	0	1	1	0
0	1	0	1	0	1	0	1	0	0
0	1	1	1	0	1	0	1	0	1
0	1	0	0	0	1	0	1	0	1
0	1	0	1	1	1	0	1	0	0
1	0	0	0	0	0	0	1	0	0

(b) Matrix representation of maze

Find a path: Idea

```
{ Begin with the entrance
   If (the present position is the exit)
         { pop the all entries in the path stack;
            return }
   Else { We block the current position;
            If (the current position is surrounded by the obstacles)
                   { pop the stack; back-up }
           While  ( there is unblocked adjacent maze position )
                 { Move to this new adjacent position;
                    Push the current position into the path stack;
                    Attempt to find a path }
            If ( tried all adjacent unblocked positions and no path is found )
                      {  There is no path in maze;
                          return }
}
```

Rat in a Maze: Strategy (1)

- Maze
  - Storing 1 or 0
  - 2D array of type *byte* or 2D array of type *Boolean*

- Each maze position : Position(i,j) ➔ maze[i][j]
  - Use objects of type *Position*, which is defined a class with private members *row* and *col*
- Path :
  - Use array-based stack that maintains the path from entrance to the current position

Rat in a Maze: Strategy (2)

- Four moves are possible: right, down, left, up
  - To avoid to handle positions on the boundaries of maze differently from interior positions
    - Surround the maze with a wall of obstacles.
    - For an m x m maze, 0th row & m+1th row and 0th cloumn & m+1th column are added
    - see figure 9.15 (page 343)

- Theoretically a long path passes m x m positions (worst case) while a short path passes 2m positions (best case) in a maze with no blockages

Rat in a Maze: Strategy (3)

- Have to select the order of moves from "here"
  - For example, move right, then down, then left, and up
  - The coordinates to move to are computed by maintaining a table of offsets.

Move	Direction	offset[move].row	offset[move].col
0	right	0	1
1	down	1	0
2	left	0	−1
3	up	−1	0

Figure 9.18 Table of offsets

- To avoid moving to positions that we have been through before
  - we place an obstacle on a visited position, i.e, set maze[i][j] =1
- If we have to back up, the next move option is computed by the followings:

```
if   (next.row == here.row)    option = 2 + next.col – here.col;
else                           option = 3 + next.row – here.row;
```

Rat In a Maze: Example (1)

- Maze M[3,3]
  - 0 0 1
  - 0 1 0
  - 0 0 0

- Start: M[1,1] =1, here = M[1,1]
- Move to M[1,2] : pushStack M[1,1], here=M[1,2], blocking M[1,2]
- M[1,2] is surrounded by "1" ➔ Back-up: popStack, next = M[1,1]
  ➔ option = 2 + next.col – here.col = 1, here = next = M[1,1]
- Move to M[2,1] : pushStack M[1,1], here=M[2,1], blocking M[2,1]

Rat In a Maze: Example (2)

- Maze M[3,3]
  - 1 1 1
  - 1 1 0
  - 0 0 0
- Move to M[3,1] : pushStack M[2,1], here=M[3,1], blocking M[3,1]
- Move to M[3,2] : pushStack M[3,1], here=M[3,2], blocking M[3,2]
- Move to M[3,3] : pushStack M[3,2], here=M[3,3], blocking M[3,3]
- You arrived at the entrance: pushStack M[3,3]
- Pop all the entries from the stack & print in a reverse order:

  M[1,1], M[2,1], M[3,1], M[3,2], M[3,3]
- return true;

Rat in a Maze: Main()

```
//private static method welcome
//private static method inputMaze
//private static method findPath
//private static method outputPath

public static void main(String[] args){
   welcome();
   inputMaze();
   if (findPath())
         outputPath();
   else  System.out.println("No path");
}
```

FindPath() code using stack (1)

```
private static boolean findPath(){
    path = new ArrayStack();
    // initialize offsets
    Position [] offset = new Position [4];
    offset[0] = new Position(0, 1);   // right
    offset[1] = new Position(1, 0);   // down
    offset[2] = new Position(0, -1);  // left
    offset[3] = new Position(-1, 0);  // up

   // initialize wall of obstacles around maze
    for (int i = 0; i <= size + 1; i++) {
      maze[0][i] = maze[size + 1][i] = 1; // bottom and top
      maze[i][0] = maze[i][size + 1] = 1; // left and right
    }
    Position here = new Position(1, 1);
    maze[1][1] = 1; // prevent return to entrance
    int option = 0;     // next move
    int lastOption = 3;
```

FindPath() code using stack (2)

```
// search for a path
while (here.row != size || here.col != size) { // not at exit  // find a neighbor to move to
           int r = 0, c = 0;   // row and column of neighbor
          while (option <= lastOption) {
                r = here.row + offset[option].row;
                c = here.col + offset[option].col;
                if (maze[r][c] == 0) break; // find a neightbor to proceed!
                option++;   // next option  }
          // was a neighbor found?
          if (option <= lastOption) { // move to maze[r][c]
                path.push(here);
                here = new Position(r, c);
                maze[r][c] = 1;  // set to 1 to prevent revisit
                option = 0; }
          else {  // no neighbor to move to, back up
                 if (path.empty()) return false;  // no place to back up to
                 Position next = (Position)   path.pop();
                 if (next.row == here.row)     option = 2 + next.col - here.col;
                 else                          option = 3 + next.row - here.row;
                  here = next; } //end of else
          } return true;  // at exit
 }
```

Complexity: Rat in a Maze

- Complexity
  - For the time complexity, we move to each unblocked position of the input maze in worst case
    - O(*unblocked*), where *unblocked* is the number of unblocked positions in the input maze
    - Once we visited a position, we block it ➔ no revisit
  - That is, in m *m maze, the worst case of time complexity is $O(size^2) = O(m^2)$

Summary

- Stack is
  - A kind of Linear list
  - Insertion and removal from one end "top"
  - LIFO(last-in-first-out) structure
- Representation
  - ArrayStack class
  - LinkedStack class

- Stack Applications
  - Parenthesis Matching
  - Towers of Hanoi
  - Rearranging Railroad Cars
  - Offline Equivalence Class Problem
  - Rat in a Maze