Maze exit finder (cont.)

- Solution must lead to smaller problems

```
boolean find_exit(int x, int y) /* 2nd try */
{  if (we have been here before)
      return false;   /* don't try same spot again */
   if ( x,y is an exit)
      return true;    /* success! */
   /* rest as before */
```

- So need a way to remember where we've been
  - e.g., mark square upon entering find_exit
  - Q: is it ever necessary to remove the mark?

Choosing maze data structures

- How to represent a maze square?
  - Okay, a class, but what data are stored?
    - Ways to know if exit or not, if has been visited yet or not
    - *Maybe* ways to know about neighboring squares
  - How about some helper methods?
    - e.g., isExit(), isMarked(), hasNeighbor(direction), …
- How to represent the whole maze?
  - Suggest: array of references to maze squares
  - Any other ways?

Towers of Hanoi (demo)

- Problem: move n disks from peg a to peg c, using peg b to hold disks temporarily; keep small on top
- Recursive solution: method with params `n`, `a`, `b`, `c`
  - Base case: just one disk – trivial:
    - If `n` is 1, move 1 disk from `a` to `c`
  - General case: *assume* a method that can move a tower of height n-1. This method!!!
    - Move top `n-1` disks from `a` to `b`, using `c` for holding purposes
    - Move the bottom disk from `a` to `c`
    - Move all `n-1` disks on `b` to `c`, using `a` for holding purposes
- Iterative solution much more difficult in this case

Decimal (value) to binary (string)

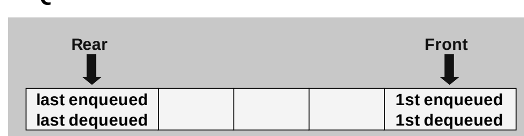
```
/**
 * Returns a String representation of the binary equivalent
 * of a specified integer.  The worstTime(n) is O(log n).
 * @param n – an int in decimal notation.
 * @return the binary equivalent of n, as a String
 * @throws IllegalArgumentException, if n is less than 0
 */  // (From Collins text's instructor resources)
public static String getBinary (int n) {
        if (n < 0)
                throw new IllegalArgumentException( );

        if (n <= 1)
                return Integer.toString (n);

        return getBinary (n / 2) + Integer.toString (n % 2);
}
```

Demo

Eliminating recursion

- Can *always* simulate recursion by explicit stack
  - Use iteration instead of recursion
    - Instead of recursive call: push key values onto stack
      - e.g., maze finder – push coordinates (x, y)
    - Instead of return: pop values from stack
      - e.g., back to square (x, y) in maze finder
- Sometimes an easy non-recursive translation without a stack – especially if "tail recursion"
  - e.g, factorial, fibonacci, ruler tick marks, …
  - Much harder for maze and Hanoi examples

Queues



- FIFO data structure – First In, First Out
- Typical operations similar to stacks
  - `enqueue` (an item at rear of queue)
  - `dequeue` (item at front of queue)
  - `peek` (at front item)
  - `isEmpty`, `isFull`, `size`, `clear`

Some queue applications

- Many operating system applications
  - Time-sharing systems rely on process queues
    - Often separate queues for high priority jobs that take little time, and low priority jobs that require more time
  - Printer queues and spoolers
    - Printer has its own queue with bounded capacity
    - Spoolers queue up print jobs on disk, waiting for print queue
  - Buffers – coordinate processes with varying speeds
- Simulation experiments
  - Models of queues at traffic signals, in banks, etc., used to "see what happens" under various conditions

Applying a queue – palindrome

- Palindrome - same forward and backward
  - e.g., Abba, and "Able was I ere I saw Elba."
- Lots of ways to solve, including recursive
- Can use a queue and a stack together
  - Fill a queue and a stack with copies of letters (only)
  - Then empty both together, verifying equality
- Reminder – we're using an *abstraction*
  - We still don't know how queues are implemented!!! To use them, it does not matter!
  - Abstraction is Good!

Queue interface

- **e.g., java.util.Queue:**

```
public interface Queue<E> extends Collection<E> {
    boolean offer(E o); // enqueue
    E poll();      // dequeue (null if empty)
    E remove();    // dequeue (exception if empty)
    E peek();      // peek (null if empty)
    E element();   // peek (exception if empty)
}
```

- **All Known Implementing Classes:**
  - AbstractQueue, ArrayBlockingQueue, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingQueue, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

Implementing queues

- Easy to do with a list (e.g., ArrayList):
  - Mostly same as stack implementation
  - Enqueue – add to end – `list.add(item);`
  - Then to dequeue and peek: refer to first item
    - e.g., to dequeue – `list.remove(0);`
- Array implementation is trickier:
  - Must keep track of front and rear indices
  - Increment front/rear using modulus arithmetic
    - Indices cycle past last index to first again – idea is to reuse the beginning of the array after dequeues

See demos in `~mikec/cs20/demo03/queue/` at CSIL

Queue operation complexity

- Implementing a queue with an array
  - enqueue(object) – add to end and increment tail index
    - O(1) if array is not full; otherwise O(n) to resize/copy
  - dequeue() – remove front and increment front index
    - O(1) – does not depend on size of queue
- Implementing with single-linked list
  - enqueue(object) – add a *last* item
    - O(n) – for single-linked list with just a first pointer
    - But O(1) if also have a pointer to last element – an easy fix
  - dequeue() – remove *first* item
    - O(1) – point first at first.next – not affected by n size
  - Why not enqueue first and dequeue last?

What are iterators?

- Objects for iterating over elements of structure
- e.g., java.util.Iterator:

```
interface Iterator<E> {
    boolean hasNext(); // true if more objects
    E next(); // return object and increment
        // throws NoSuchElementException if !hasNext()
    void remove(); // optional – and potentially dangerous
        // may throw UnsupportedOperationException
}
```

- Handy to implement as inner class of structure
  - Has reference to all data structure fields/methods
  - Could be anonymous/local to `getIterator` method

Why iterators?

- Provide ability to traverse list (or other structure) without direct access to nodes
- Easy to use – e.g., print list with while loop:

```
Iterator it = list.getIterator();
while (it.hasNext())
    print(it.next());
```

- Even shorter with a for loop:

```
for(Iterator it=list.getIterator(); it.hasNext();)
    print(it.next()); // the increment step happens here
```

  - And simpler with enhanced for loop:

```
for (DataType d : list) print(d);
```

Implementing linked lists

- e.g., a method to insert a new second node – imagine list now is (DUS→ORD→SAN),
  want (DUS→BRU→ORD→SAN)
  or now (DUS), want (DUS→BRU)
  or now (), want (BRU)
  - Any other special cases?
- A strategy:

```
create new node to hold BRU – call it n;
if empty list – set first to n; return;
else   set n.next to first.next;
       set first.next to n; return;
```

Code to insert new 2nd node

- Assume instance variable for first node:
  `ListNode first;` // refers to first node or null if list is empty
- So use that fact to write "is empty" method:
  `boolean isEmpty() { return first == null; }`
- Then easy to code insert 2nd node method:

```
void insertNewSecondNode(Object data){
    ListNode n = new ListNode(); // null data and next
    n.data = data;
    if (isEmpty()) first = n; // leave next null
    else {
        n.next = first.next;
        first.next = n;
    }
}
```

Searching for a node

- Idea: *return reference to the node that contains particular info*, or return `null` if the info is not in the list (Note: probably a private method – returns node reference)
- Strategy:

```
declare local node reference - call it n;
point n at first node in list;
while (n points to non-null node) {
    if (n's referent has the info)
        return n;
    else advance n to n.next;
}
return null if get this far;
```

List traversal without iterators

- Search strategy typifies list-hopping activity:

```
start by referencing first node;
process that node;
change reference to that node's next link;
keep going until success (e.g., found info), or
   until end (i.e., reference is null);
```

  - Same idea works for lots of list operations
    - e.g., `print` list – immediately applicable
    - To `append` (add last), first must link-hop to last node
    - To `remove` a node, must link-hop to node that precedes it
- But also usually consider potential special cases
  - e.g., first node, last node, empty list, just one node, …