Announcements

- Both TAs (Ron Tal and Serene Wong) will be present in the PRISM lab (CSEB 1006) tomorrow, Tues Feb 2 from 14:30-16:30 for the scheduled practice lab. So you will NOT find Ron in CSEB 2013 during his office hour (Tues 14:30-15:30) this week.
- **Assignment 1.** I have updated the comments for the test programs for each question to indicate the correct output.
- **Assignment 1, Question 2:** Clarification on error-checking. Your implementation need only support operands that are one-character lower-case letters (no numeric constants), and the operators +, -, * and /, as well as round brackets (and). Spaces between characters in the infix expression should be ignored. Any other symbol should generate an InvalidInfixExpressionStringException. However, you do NOT have to detect other kinds of invalid infix expression strings: you can assume the expressions are valid.
- **Assignment 1, Question 5:** The interface for push in Class minStack is public E push(E element). This is different from the push ADT defined by the book and in my slides, which does not return anything (i.e., public void push(E element)). I am following here the Stack class provided by java.util. Here, push returns the element just pushed onto the stack, i.e., element in push(E element). Please follow this convention.
- The statement **of Assignment 1, Question 2** was a bit misleading. The last expression should read z x y + *, not x y + z *.

Maps, Hash Tables and Dictionaries

Chapter 9

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database

The Map ADT

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its associated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M
  - entries(): returns an iterator of the entries in M

Example

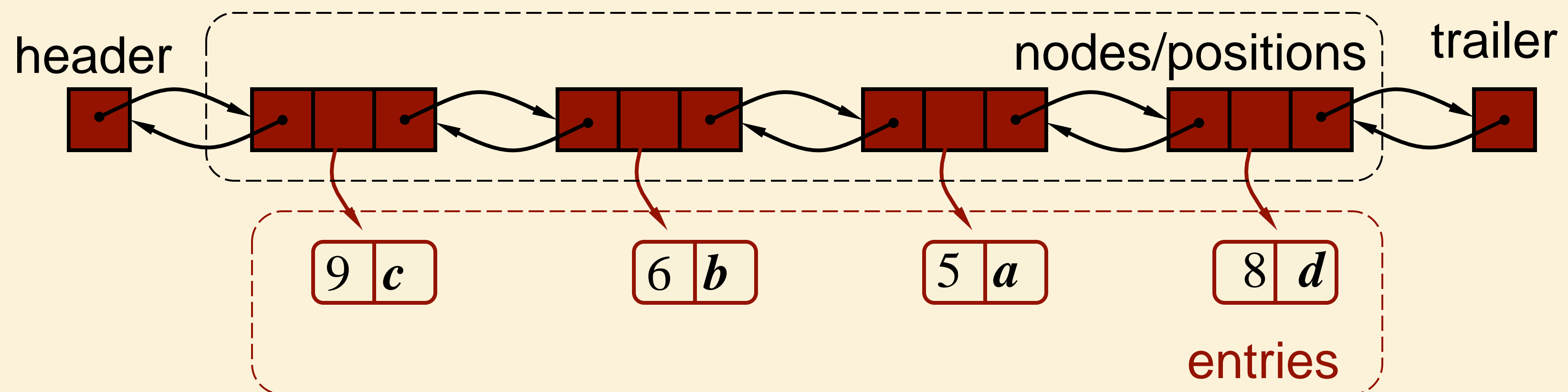
Operation	*Output*	*M*
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

Comparison to java.util.Map

Map ADT Methods	java.util.Map Methods
size()	size()
isEmpty()	isEmpty()
get(*k*)	get(*k*)
put(*k*,*v*)	put(*k*,*v*)
remove(*k*)	remove(*k*)
keys()	keySet()
values()	values()
entries()	entrySet()

A Simple List-Based Map

- We could implement a map using an unsorted list
  - We store the entries of the map in a doubly-linked list S, in arbitrary order



The get(k) Algorithm

```
Algorithm get(k):
    B = S.positions() {B is an iterator of the positions in S}
    while B.hasNext() do
        p = B.next()   // the next position in B
        if p.element().key() = k      then
            return p.element().value()
    return null {there is no entry with key equal to k}
```

The put(k,v) Algorithm

Algorithm put(k,v):

B = S.positions()

while B.hasNext() **do**

    p = B.next()

    if p.element().key() = k **then**

        t = p.element().value()

        B.replace(p,(k,v))

        return t {return the old value}

S.insertLast((k,v))

$n = n + 1$ {increment variable storing number of entries}

return null {there was no previous entry with key equal to k}

The remove(k) Algorithm

Algorithm remove(k):

B = S.positions()

while B.hasNext() **do**

  p = B.next()

  if p.element().key() = k **then**

    t = p.element().value()

    S.remove(p)

    $n = n - 1$ {decrement number of entries}

    return t {return the removed value}

return null {there is no entry with key equal to k}

Performance of a List-Based Map

- Performance:
  - put, get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for small maps

Hash Tables

- A hash table is a data structure that can be used to make map operations faster.
- While worst-case is still O(n), average case is typically O(1).

Applications of Hash Tables

- databases
- compilers
- browser caches

Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

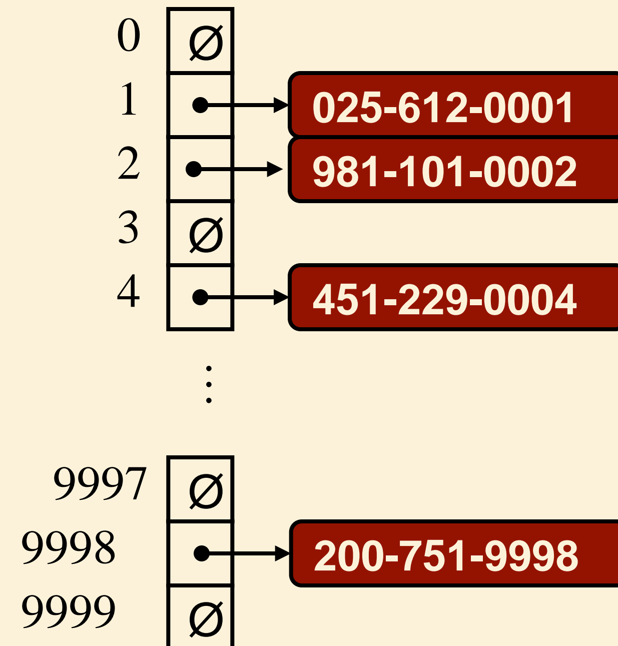
  $$h(x) = x \bmod N$$

  is a hash function for integer keys

- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a map storing entries as (SIN, Name), where SIN (social insurance number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) = \text{last four digits of SIN } x$



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:
  h_1: keys ➔ integers

  Compression function:
  h_2: integers ➔ $[0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

  $$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Codes

- **Memory address**:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Does not work well when copies of the same object may be stored at different locations.
- **Integer cast**:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- **Component sum**:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Problems with Component Sum Hash Codes

- Hashing works when
  - the number of different common keys is small relative to the hashing space (e.g., 2^{32} for a 32-bit hash code).
  - the hash codes for common keys are well-distibuted (do not collide) in this space.
- Component Sum codes ignore the ordering of the components.
  - e.g., using 8-bit ASCII components, ‘stop’ and ‘pots’ yields the same code.
- Since common keys are often anagrams of each other, this is often a bad idea!

Polynomial Hash Codes

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0 \, a_1 \, \ldots \, a_{n-1}$$

  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$ at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
  - Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
    - The following polynomials are successively computed, each from the previous one in $O(1)$ time

      $$p_0(z) = a_{n-1}$$

      $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$

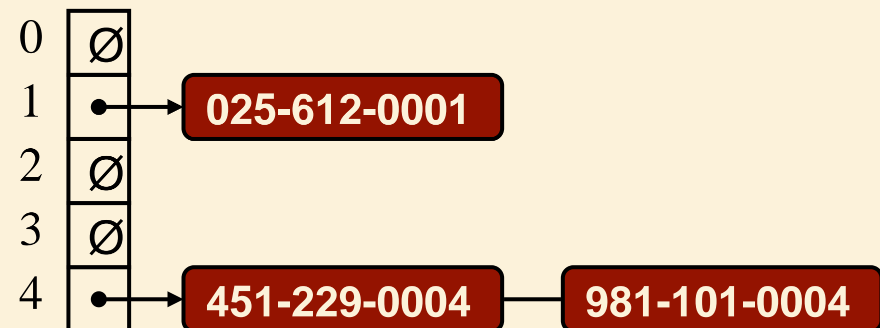
  - We have $p(z) = p_{n-1}(z)$

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Collision Handling

- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**:
  - Let each cell in the table point to a linked list of entries that map there
  - Separate chaining is simple, but requires additional memory outside the table



Map Methods with Separate Chaining

- Delegate operations to a list-based map at each cell:

Algorithm get(k):

Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map

return $A[h(k)]$.get(k) {delegate the get to the list-based map at $A[h(k)]$}

Map Methods with Separate Chaining

- Delegate operations to a list-based map at each cell:

Algorithm put(k,v):

Output: Store the new (key, value) pair. If there is an existing entry with key equal to k, return the old value; otherwise, return **null**

$t = A[h(k)].\mathrm{put}(k,v)$ {delegate the put to the list-based map at $A[h(k)]$}

if t = **null then** {k is a new key}

$n = n + 1$

return t

Map Methods with Separate Chaining

- Delegate operations to a list-based map at each cell:

Algorithm remove(k):

Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map

$t = A[h(k)]$.remove(k) {delegate the remove to the list-based map at $A[h(k)]$}

if $t \neq$ **null then** {k was found}

  $n = n - 1$

return t

Linear Probing

- **Open addressing**: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, so that future collisions cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44	59	32	22	31	73	

Get with Linear Probing

- Consider a hash table **A** of length **N** that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

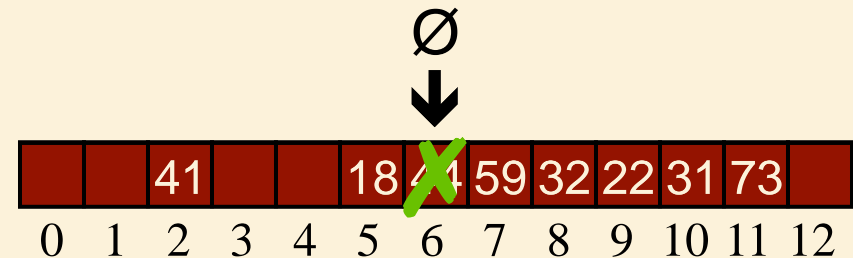
```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = Ø
            return null
         else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until  p = N
    return null
```

Remove with Linear Probing

- Suppose we receive a **remove(44)** message.
- What problem arises if we simply remove the key = 44 entry?

k	h(k)	i
18	5	5
41	2	2
22	9	9
44	5	6
59	7	7
32	6	8
31	5	10
73	8	11

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Removal with Linear Probing

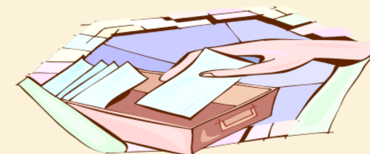
- To address this problem, we introduce a special object, called **AVAILABLE** , which replaces deleted elements
- **AVAILABLE** has a null key
- No changes to get(k) are required.

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = Ø
            return null
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until  p = N
    return null
```

Updates with Linear Probing

- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item *AVAILABLE* and we return element o
  - Else, we return *null*
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores *AVAILABLE*, or
    - N cells have been unsuccessfully probed
  - We store entry (k, o) in cell i

Double Hashing



- Double hashing uses a **secondary hash function h'(k)** in addition to the primary hash function h(x).
- Suppose that the primary hashing i=h(k) leads to a collision.
- We then iteratively probe the locations
  $(i + jh'(k)) \bmod N$ for $j = 0, 1, \dots, N - 1$
- The secondary hash function ***h'(k)*** cannot have zero values
- The table size ***N*** must be a prime to allow probing of all the cells
- Common choice of secondary hash function h'(k):
  - $h'(k) = q - k \bmod q$, where
    - $q < N$
    - q is a prime
- The possible values for h'(k) are
  $1, 2, \dots, q$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $h'(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73

k	$h(k)$	$h'(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\lambda = n/N$ affects the performance of a hash table
  - For separate chaining, performance is typically good for $\lambda < 0.9$.
  - For open addressing , performance is typically good for $\lambda < 0.5$.
  - java.util.HashMap maintains $\lambda < 0.75$
- Separate chaining is typically as fast or faster than open addressing.

Rehashing

- When the load factor λ exceeds threshold, the table must be **rehashed**.
  - A larger table is allocated (typically at least double the size).
  - A new hash function is defined.
  - All existing entries are copied to this new table using the new hash function.

Java Example (MAD hash function)

```
/** Doubles the size of the hash table and rehashes all the entries. */
protected void rehash() {
  capacity = 2*capacity;
  Entry<K,V>[] old = bucket;
  bucket = (Entry<K,V>[]) new Entry[capacity]; // new bucket is twice as big
  java.util.Random rand = new java.util.Random();
  scale = rand.nextInt(prime-1) + 1;      // new hash scaling factor
  shift = rand.nextInt(prime);          // new hash shifting factor
  for (int i=0; i<old.length; i++) {
     Entry<K,V> e = old[i];
    if ((e != null) && (e != AVAILABLE)) { // if we have a valid entry
          int j = - 1 - findEntry(e.getKey());  // find the appropriate spot
          bucket[j] = e;                // copy into the new array
    }
  }
}
```

DICTIONARIES

Dictionary ADT

- The dictionary ADT models a searchable collection of key-element entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key **are** allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

- Dictionary ADT methods:
  - find(k): if the dictionary has at least one entry with key k, returns one of them, else, returns null
  - findAll(k): returns an iterator of all entries with key k
  - insert(k, o): inserts and returns the entry (k, o)
  - remove(e): remove the entry e from the dictionary
  - entries(): returns an iterator of the entries in the dictionary
  - size(), isEmpty()

Example

Operation	***Output***	***Dictionary***
insert(5,*A*)	(5,*A*)	(5,*A*)
insert(7,*B*)	(7,*B*)	(5,*A*),(7,*B*)
insert(2,*C*)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*)
insert(8,*D*)	(8,*D*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
insert(2,*E*)	(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(7)	(7,*B*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(4)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(2)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
findAll(2)	(2,*C*),(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
size()	5	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
remove(find(5))	(5,*A*)	(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(5)	**null**	(7,*B*),(2,*C*),(8,*D*),(2,*E*)

A List-Based Dictionary

- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- Performance:
  - insert takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - find and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

The findAll(k) Algorithm

Algorithm findAll(*k*):

Input: A key *k*

Output: An iterator of entries with key equal to *k*

Create an initially-empty list *L*

B = *D*.entries()

while *B*.hasNext() **do**

  e = *B*.next()

  if *e*.key() = *k* **then**

    L.insertLast(*e*)

return *L*.elements()

The insert and remove Methods

Algorithm insert(*k*,*v*):
Input: A key *k* and value *v*
Output: The entry (*k*,*v*) added to *D*
Create a new entry *e* = (*k*,*v*)
S.insertLast(*e*) {*S* is unordered}
return *e*

Algorithm remove(*e*):
Input: An entry *e*
Output: The removed entry *e* or **null** if *e* was not in *D*
{We don't assume here that *e* stores its location in *S*}
B = *S*.positions()
while *B*.hasNext() **do**
  p = *B*.next()
  if *p*.element() = *e* **then**
    S.remove(*p*)
    return *e*
return null {there is no entry *e* in *D*}

Hash Table Implementation

- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

Dictionaries & Ordered Search Tables

- If keys obey a total order relation, can represent dictionary as an ordered search table stored in an array.
- Can then support a fast find(k) using **binary search.**
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps

Ordered Search Tables

- Performance:
  - find takes $O(\log n)$ time, using binary search
  - insert takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
  - remove takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal
- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)
- Binary search can interact poorly with the memory hierarchy (i.e. caching), because of its random-access nature. A common technique is to abandon binary searching for linear searching as soon as the size of the remaining span falls below a small value such as 8 or 16 or even more in recent computers.

Define Problem: Binary Search

- PreConditions
  - Key 25
  - Sorted List

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- PostConditions
  - Find key in list (if there).

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

More on Binary Search

Define Loop Invariant

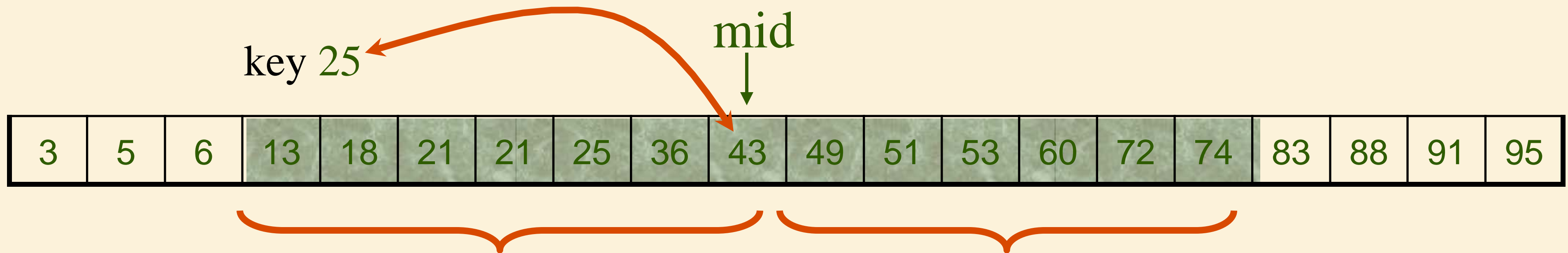
- Maintain a sublist.
- If the key is contained in the original list, then the key is contained in the sublist.

key 25

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Define Step

- Cut sublist in half.
- Determine which half the key would be in.
- Keep that half.

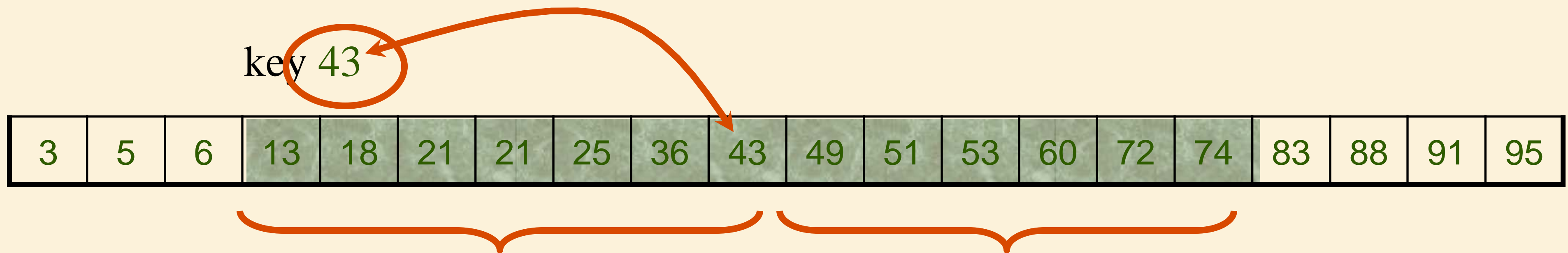


If key ≤ mid, then key is in left half.

If key > mid, then key is in right half.

Define Step

- It is faster not to check if the middle element is the key.
- Simply continue.

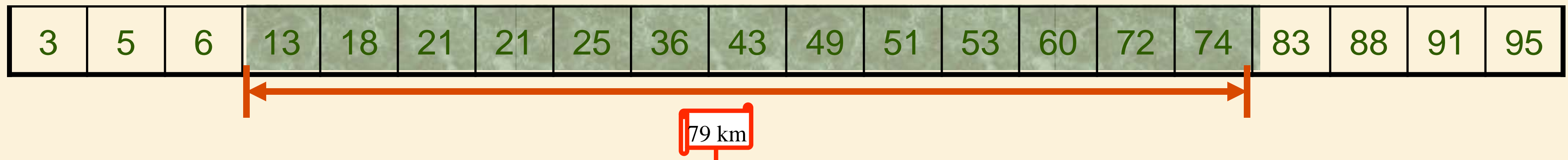


If key ≤ mid, then key is in left half.

If key > mid, then key is in right half.

Make Progress

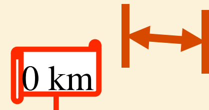
- The size of the list becomes smaller.



Ending Algorithm

key 25

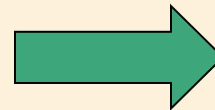
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95



- ➢ If the key is contained in the original list,

  then the key is contained in the sublist.

- ➢ Sublist contains one element.



- If the key is contained in the original list,

  then the key is at this location.

If key not in original list

- If the key is contained in the original list,

  then the key is contained in the sublist.

- Loop invariant true, even if the key is not in the list.

key 24

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- If the key is contained in the original list, then the key is at this location.

- Conclusion still solves the problem.

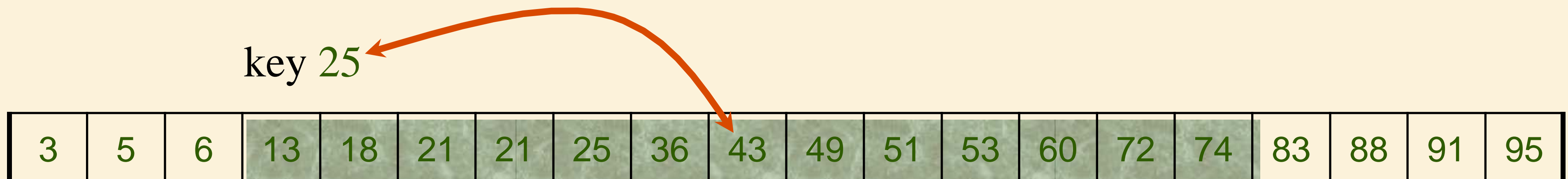
  Simply check this one location for the key.

Running Time

The sublist is of size n, $^n/_2$, $^n/_4$, $^n/_8$,…,1

Each step O(1) time.

Total = O(log n)



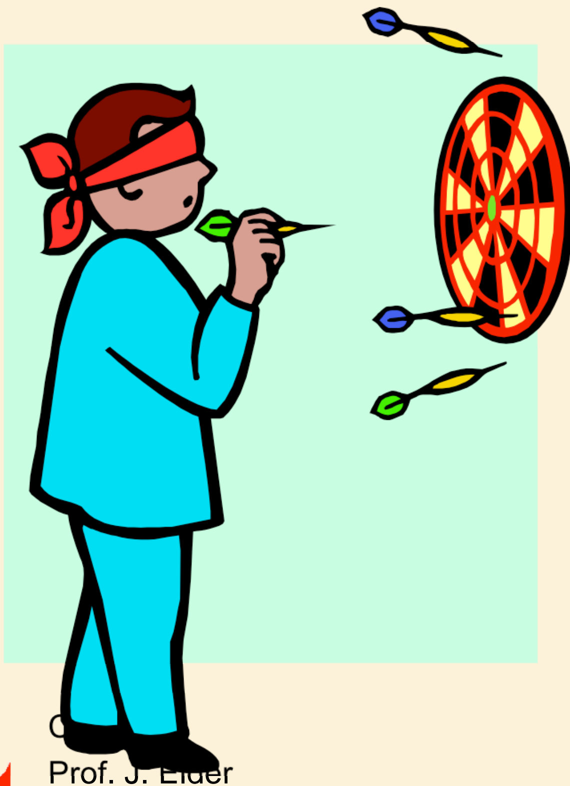
If key ≤ mid, then key is in left half.

If key > mid, then key is in right half.

```
BinarySearch(A[1..n], key)
<precondition>:  A[1..n] is sorted in non-decreasing order
<postcondition>: If key is in A[1..n], algorithm returns its location
p = 1, q = n
while q > p
    < loop-invariant>: If key is in A[1..n], then key is in A[p..q]
    mid = ⌊(p + q) / 2⌋
    if key ≤ A[mid]
        q = mid
    else
        p = mid + 1
    end
end
if key = A[p]
    return(p)
else
    return("Key not in list")
end
```

Simple, right?

- Although the concept is simple, binary search is notoriously easy to get wrong.
- Why is this?



The Devil in the Details

- The basic idea behind binary search is easy to grasp.
- It is then easy to write pseudocode that works for a 'typical' case.
- Unfortunately, it is equally easy to write pseudocode that fails on the *boundary conditions*.

The Devil in the Details

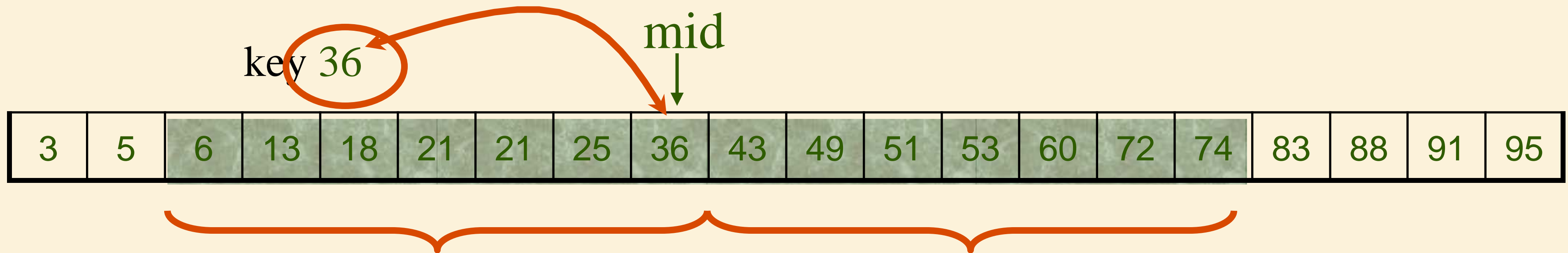
$$\begin{aligned}&\text{if } key \leq A[mid]\\&\quad q = mid\\&\text{else}\\&\quad p = mid+1\\&\text{end}\end{aligned}$$

or

$$\begin{aligned}&\text{if } key < A[mid]\\&\quad q = mid\\&\text{else}\\&\quad p = mid+1\\&\text{end}\end{aligned}$$

What condition will break the loop invariant?

The Devil in the Details



Code: key ≥ A[mid] → select right half

Bug!!

The Devil in the Details

$$\begin{aligned}&\text{if } key \le A[mid]\\&\quad q = mid\\&\text{else}\\&\quad p = mid + 1\\&\text{end}\end{aligned}$$

OK

$$\begin{aligned}&\text{if } key < A[mid]\\&\quad q = mid - 1\\&\text{else}\\&\quad p = mid\\&\text{end}\end{aligned}$$

OK

$$\begin{aligned}&\text{if } key < A[mid]\\&\quad q = mid\\&\text{else}\\&\quad p = mid + 1\\&\text{end}\end{aligned}$$

Not OK!!

The Devil in the Details

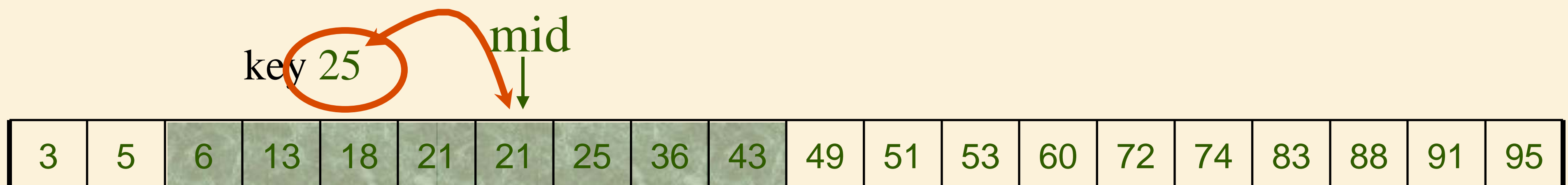
$$\text{mid} = \left\lfloor \frac{p+q}{2} \right\rfloor$$ or $$\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$$

key 25

Shouldn't matter, right?

Select $\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$

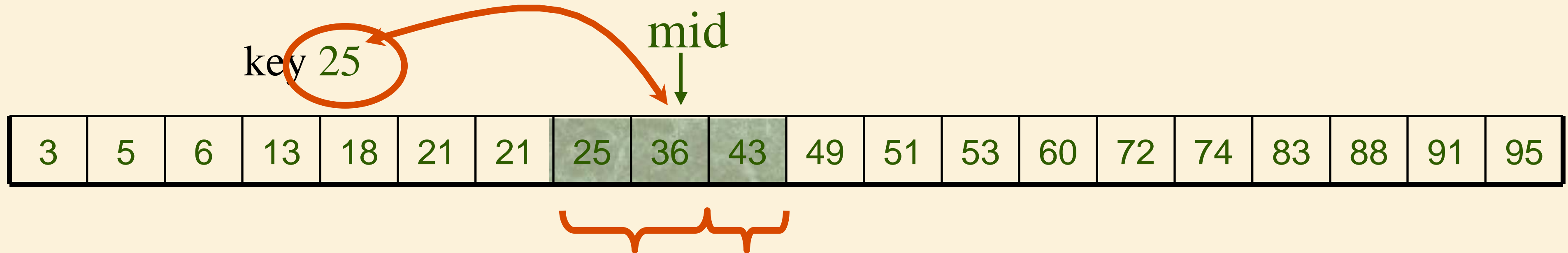
The Devil in the Details



If key ≤ mid, then key is in left half.

If key > mid, then key is in right half.

The Devil in the Details



If key $\leq$ mid, then key is in left half.

If key $>$ mid, then key is in right half.

The Devil in the Details

- Another bug!

No progress toward goal: Loops Forever!

key 25

mid

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

If key $\leq$ mid, then key is in left half.

If key > mid, then key is in right half.

The Devil in the Details

$$\text{mid} = \left\lfloor \frac{p+q}{2} \right\rfloor$$
if $key \le A[mid]$
  $q = mid$
else
  $p = mid + 1$
end

OK

$$\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$$
if $key < A[mid]$
  $q = mid - 1$
else
  $p = mid$
end

OK

$$\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$$
if $key \le A[mid]$
  $q = mid$
else
  $p = mid + 1$
end

Not OK!!

How Many Possible Algorithms?

$$mid = \left\lfloor \frac{p+q}{2} \right\rfloor$$ ← or $mid = \left\lceil \frac{p+q}{2} \right\rceil$?

$if\ key \le A[mid]$ ← or $if\ key < A[mid]$?

$\quad q = mid$

else

$\quad p = mid + 1$

end

or

$\quad q = mid - 1$

else

$\quad p = mid$

end

Alternative Algorithm: Less Efficient but More Clear

```
BinarySearch(A[1..n], key)
<precondition>:  A[1..n] is sorted in non-decreasing order
<postcondition>: If key is in A[1..n], algorithm returns its location
p = 1, q = n
while q > p
    < loop-invariant>: If key is in A[1..n], then key is in A[p..q]
    mid = ⌊(p+q)/2⌋
    if key = A[mid]
        return(mid)
    elseif key < A[mid]
        q = mid - 1
    else
        p = mid + 1
    end
end
if key = A[p]
    return(p)
else
    return("Key not in list")
end
```

Still $\Theta(\log n)$, but with slightly larger constant.

Moral

- Use the loop invariant method to think about algorithms.
- Be careful with your definitions.
- Be sure that the loop invariant is always maintained.
- Be sure progress is always made.
- Having checked the 'typical' cases, pay particular attention to boundary conditions and the end game.
- Sometimes it is worth paying a little in efficiency for clear, correct code.

Card Trick

- A volunteer, please.

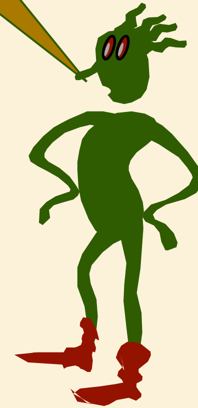
Loop Invariants for Iterative Algorithms

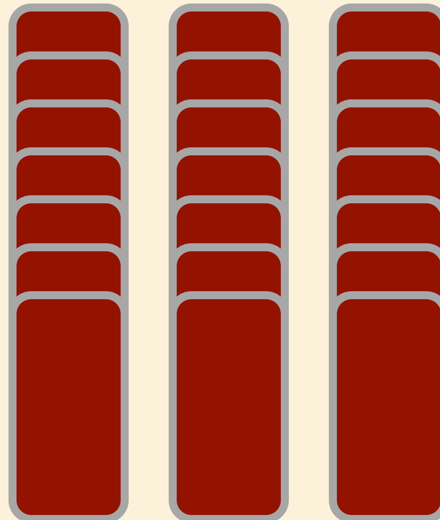
A Third

Search Example:

A Card Trick

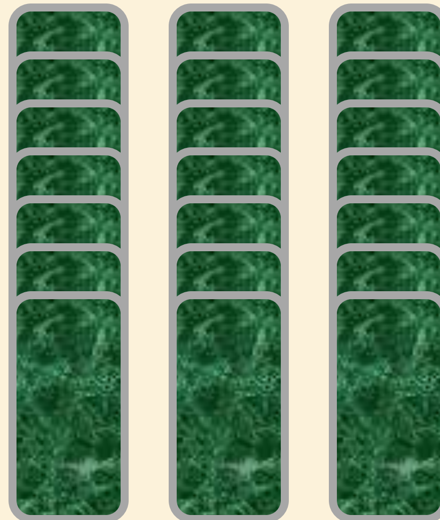
Pick a Card

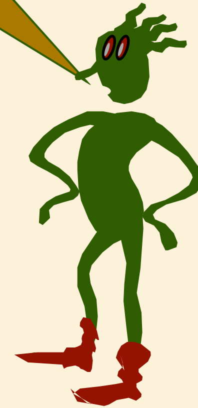




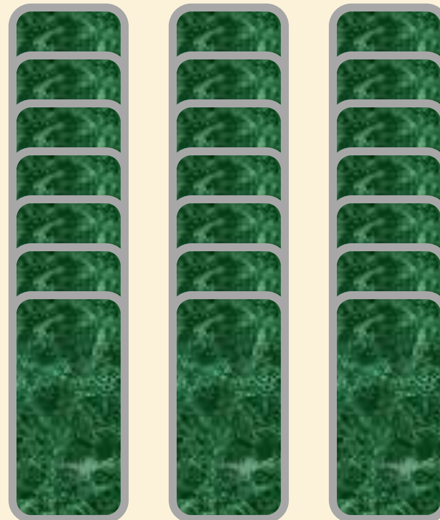
Done

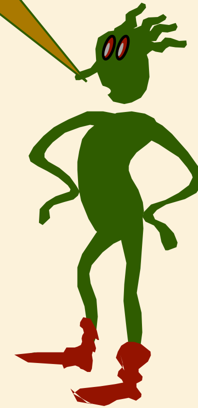
Loop Invariant:
The selected card is one of these.





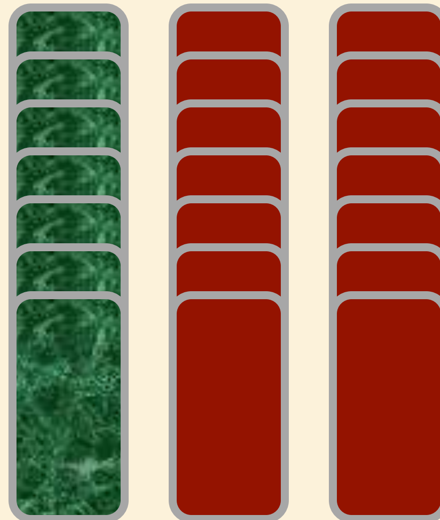
Which column?

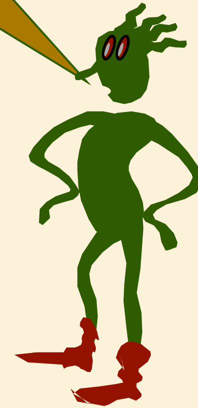




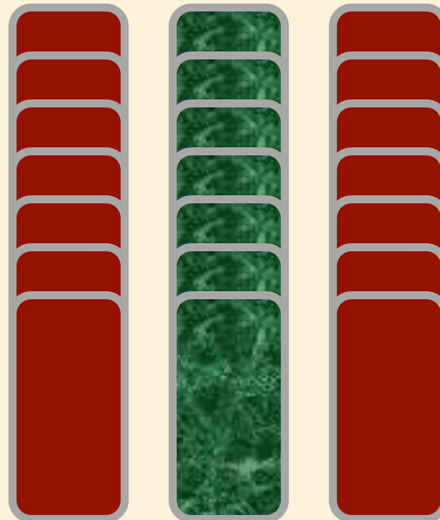
left

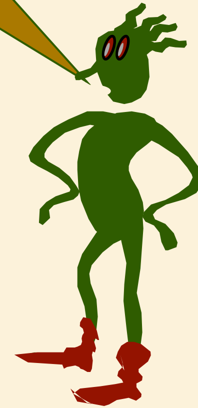
Loop Invariant:
The selected card is one of these.



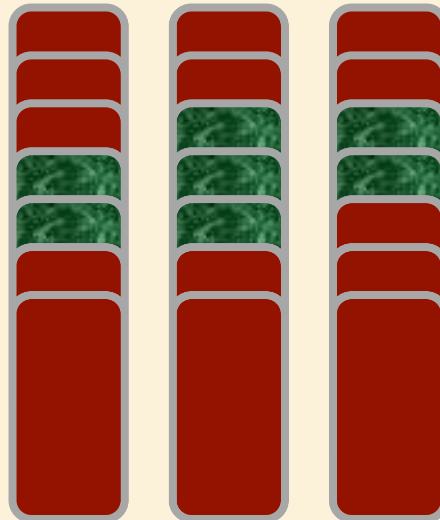


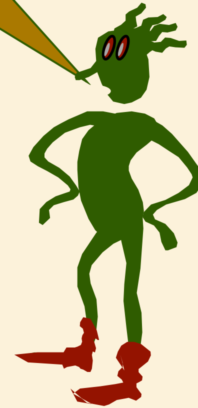
Selected column is placed in the middle



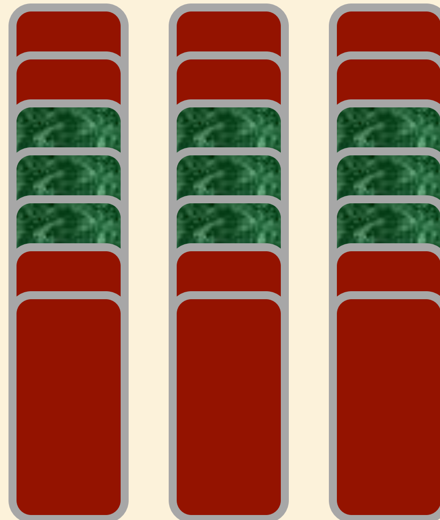


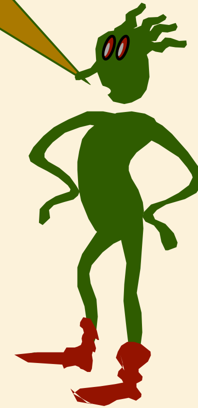
I will rearrange the cards

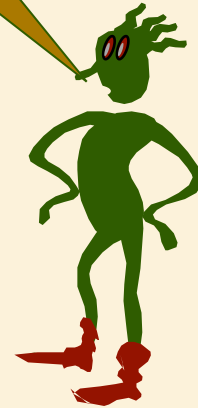


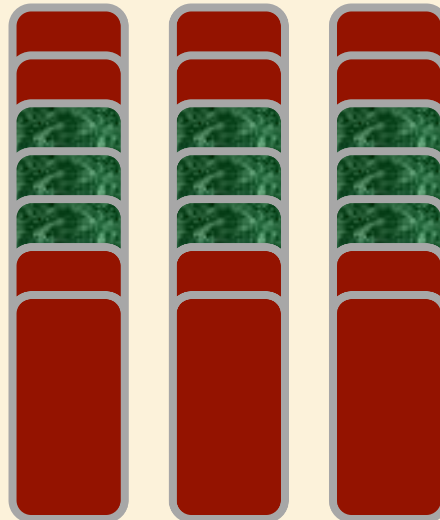


Relax Loop Invariant:
I will remember the same about each column.

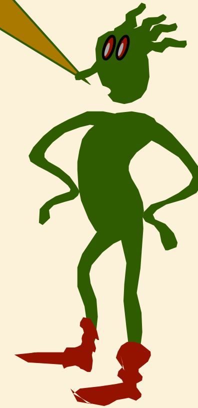


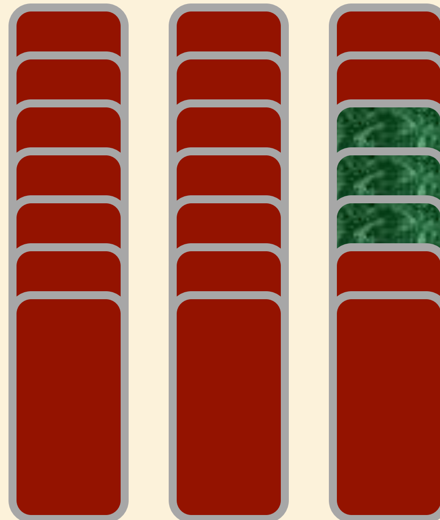




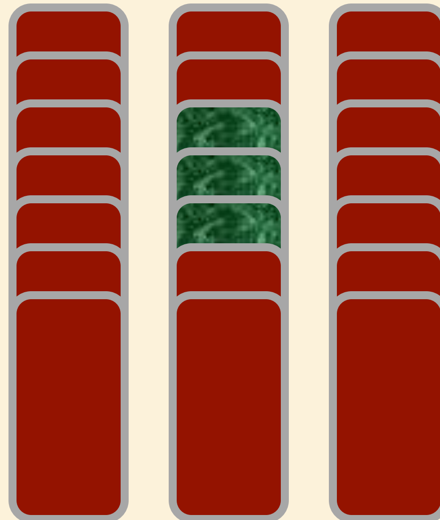


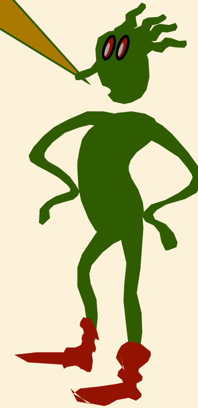
right


Loop Invariant:
The selected card is one of these.

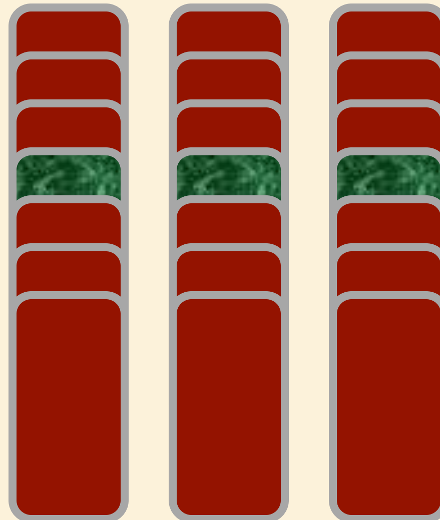


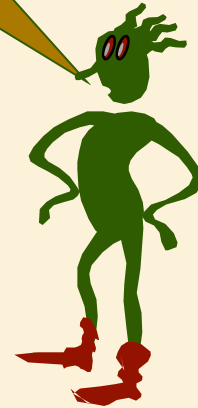
Selected column is placed in the middle

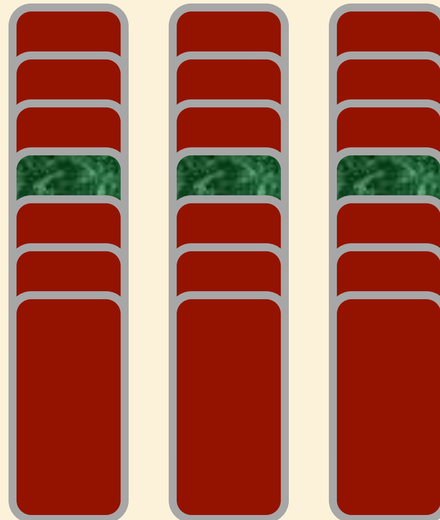


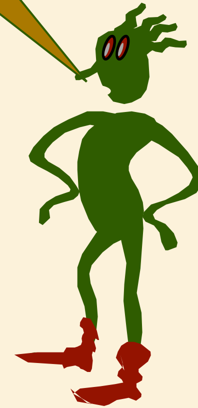


I will rearrange the cards





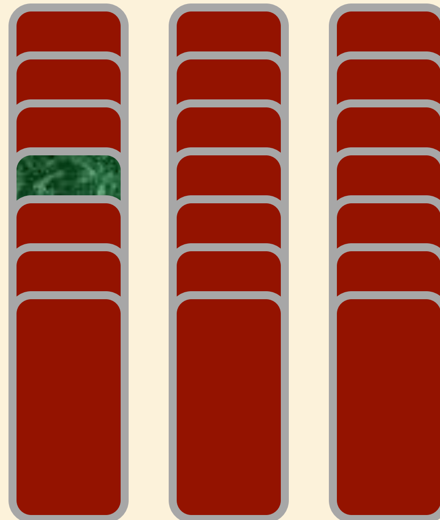


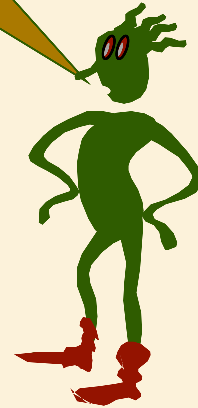


Which column?

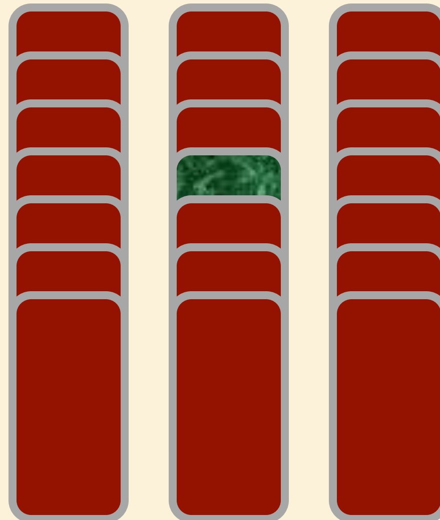
left

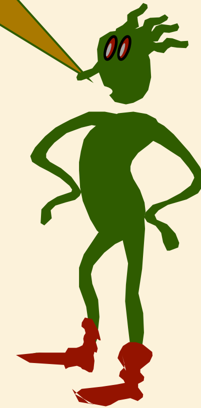
Loop Invariant:
The selected card is one of these.

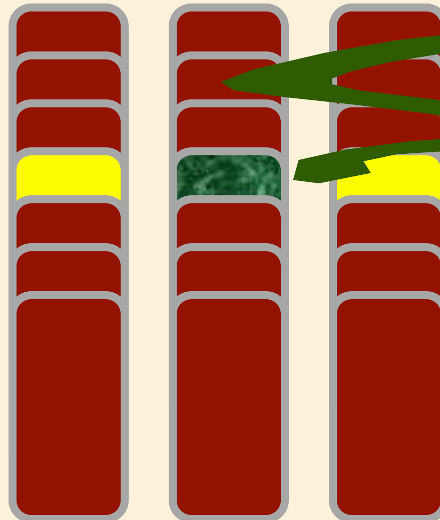


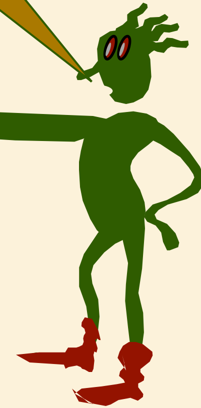


Selected column is placed in the middle









Here is your card.

Wow!

Ternary Search

- Loop Invariant: selected card in central subset of cards

  Size of subset = $\lceil n/3^{i-1} \rceil$

  where

  $n =$ total number of cards

  $i =$ iteration index

- How many iterations are required to guarantee success?