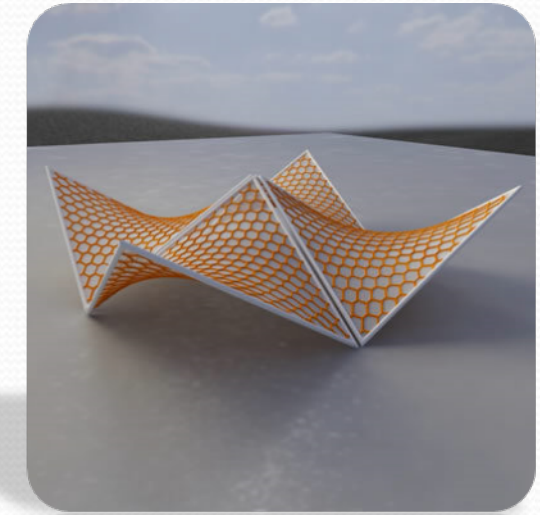
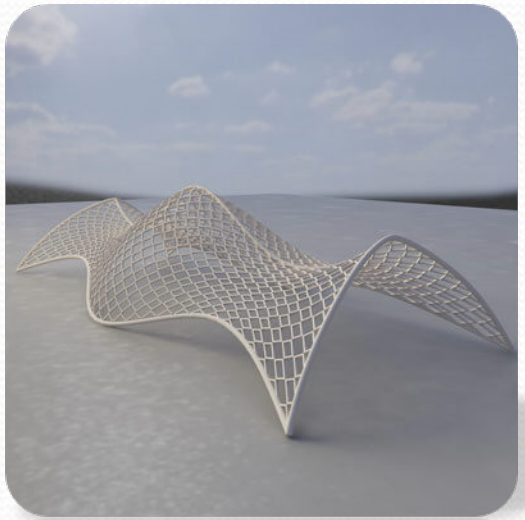
DSA – L10

Data structures

Mankind's progress is measured by the number of t n do without thinking !!

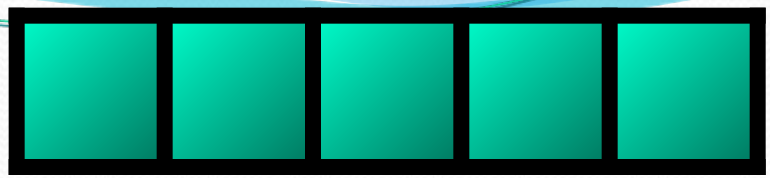




- Various implementations of sets

Suppose that we want to represent a set S of at most n numbers, each element of S is a positive integer no more than m. We would like to support the following operations on S for any given integer i with $1 \leq i \leq m$:

- *membership*: determining whether i belongs to S.
- *insertion*: inserting i into S.
- *deletion*: deleting i from S.

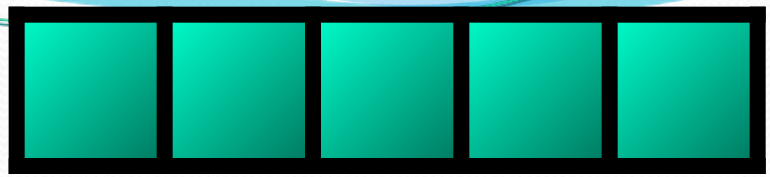


Attempt #1

Keep all the elements of S in an array $A[1 \ldots, n]$.

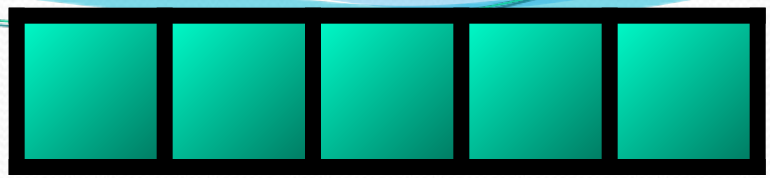
- Space: $\Theta(n)$.
- Time:
  - creation and initialization: $O(1)$.
  - membership: $O(n)$ (linear search).
  - insertion: $O(1)$.
  - deletion: $O(n)$.

Attempt #2



Keep all the elements of S in a sorted array $B[1 \ldots, n]$.

- Space: $\Theta(n)$.
- Time:
  - creation and initialization: $O(1)$.
  - membership: $O(\log n)$ (binary search).
  - insertion: $O(n)$.
  - deletion: $O(n)$.



Attempt #3

Keep all the elements of S in a binary array $C[1 \ldots, m]$. Specifically, for each $j = 1, \ldots, m$, we maintain the condition that $C[j] = 1$ if and only if S contains element j.

- Space: $\Theta(m)$.
- Time:
  - creation and initialization: $O(m)$.
  - membership: $O(1)$ (direct look-up).
  - insertion: $O(1)$.
  - deletion: $O(1)$.

An impressive feature

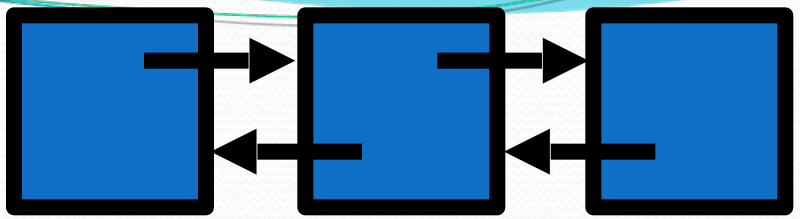
Insertion, deletion, and membership take only $O(1)$ time.

Disadvantages

(1) The expensive $\Theta(m)$ time for initialization.

(2) If $m = \omega(n)$, then the data structure seems to take more space than necessary.

Attempt #5

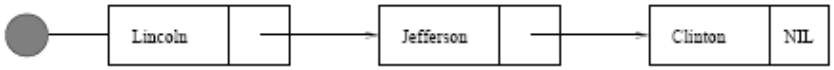


Keep all the elements of S in a linked list.

- Space: $\Theta(n)$.
- Time:
  - creation and initialization: $O(1)$.
  - membership: $O(n)$ (linear search)
  - insertion: $O(1)$.
  - deletion: $O(n)$.

Linked List Structures

```
typedef struct list {
        item_type item;
        struct list *next;
} list;
```



Searching a List

Searching in a linked list can be done iteratively or recursively.

```
list *search_list(list *l, item_type x)
{
      if (l == NULL) return(NULL);

      if (l->item == x)
            return(l);
      else
            return( search_list(l->next, x) );
}
```

Inserting into a List

Since we have no need to maintain the list in any particular order, we might as well insert each new item at the head.

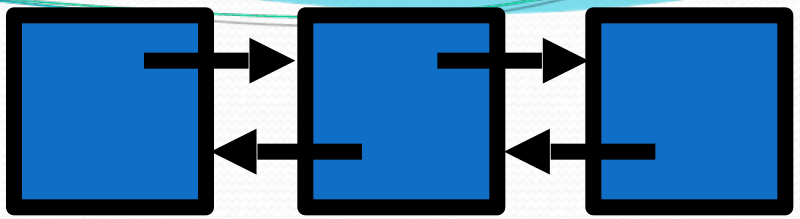
```
void insert_list(list **l, item_type x)
{
      list *p;

      p = malloc( sizeof(list) );
      p->item = x;
      p->next = *l;
      *l = p;
}
```

Note the `**l`, since the head element of the list changes.

Deleting from a List

```
delete_list(list **l, item_type x)
{
      list *p; (* item pointer *)
      list *last = NULL; (* predecessor pointer *)

      p = *l;
      while (p->item != x) { (* find item to delete *)
            last = p;
            p = p->next;
      }

      if (last == NULL) (* splice out of the list *)
            *l = p->next;
      else
            last->next = p->next;

      free(p); (* return memory used by the node *)
}
```

Attempt #5



Keep all the elements of S in a linked list.

- Space: $\Theta(n)$.
- Time:
  - creation and initialization: $O(1)$.
  - membership: $O(n)$ (linear search)
  - insertion: $O(1)$.
  - deletion: $O(n)$.

Terribly expensive for membership and deletion.

Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either *contiguous* or *linked* depending upon whether they are based on arrays or pointers:

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- Linked data structures are composed of multiple distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

Advantages of Arrays

An array is a structure of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.

Advantages of contiguously-allocated arrays include:

- Constant-time access given the index.
- Arrays consist purely of data, so no space is wasted with links or other formatting information.
- Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

Advantages of Linked Lists

The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are *simpler* than for contiguous (array) lists.
3. With large records, moving pointers is easier and faster than moving the items themselves.

Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

Stacks & Queues

- Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.
- A *stack* supports last-in, first-out operations: push and pop.
- A *queue* supports first-in, first-out operations: enqueue and dequeue.
- Lines in banks are based on queues, execution of function in a program is based on stacks.

Bus Stop Queue

Bus Stop Queue

Bus Stop Queue

Bus Stop Queue

Bus Stop Queue

front rear

Bus Stop Queue

front

rear

Bus Stop Queue

front

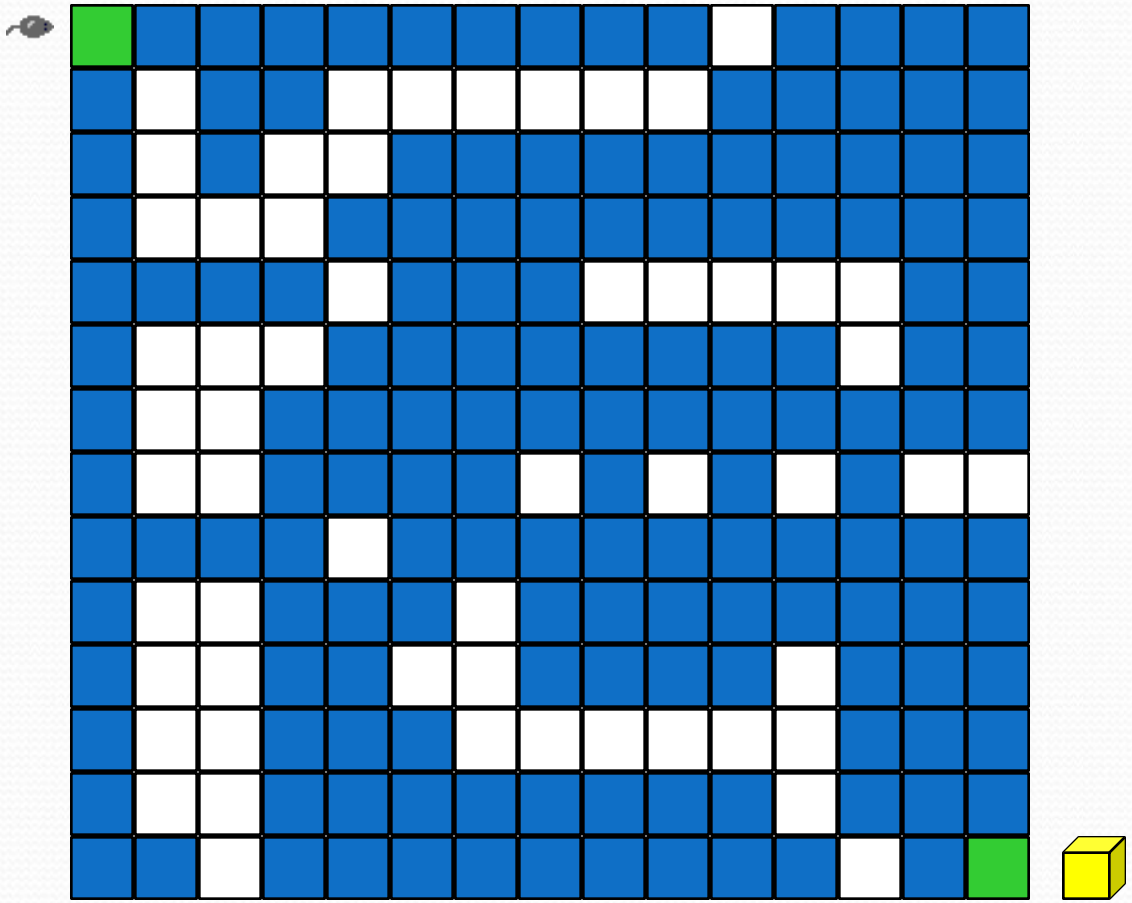
rear

Bus Stop Queue

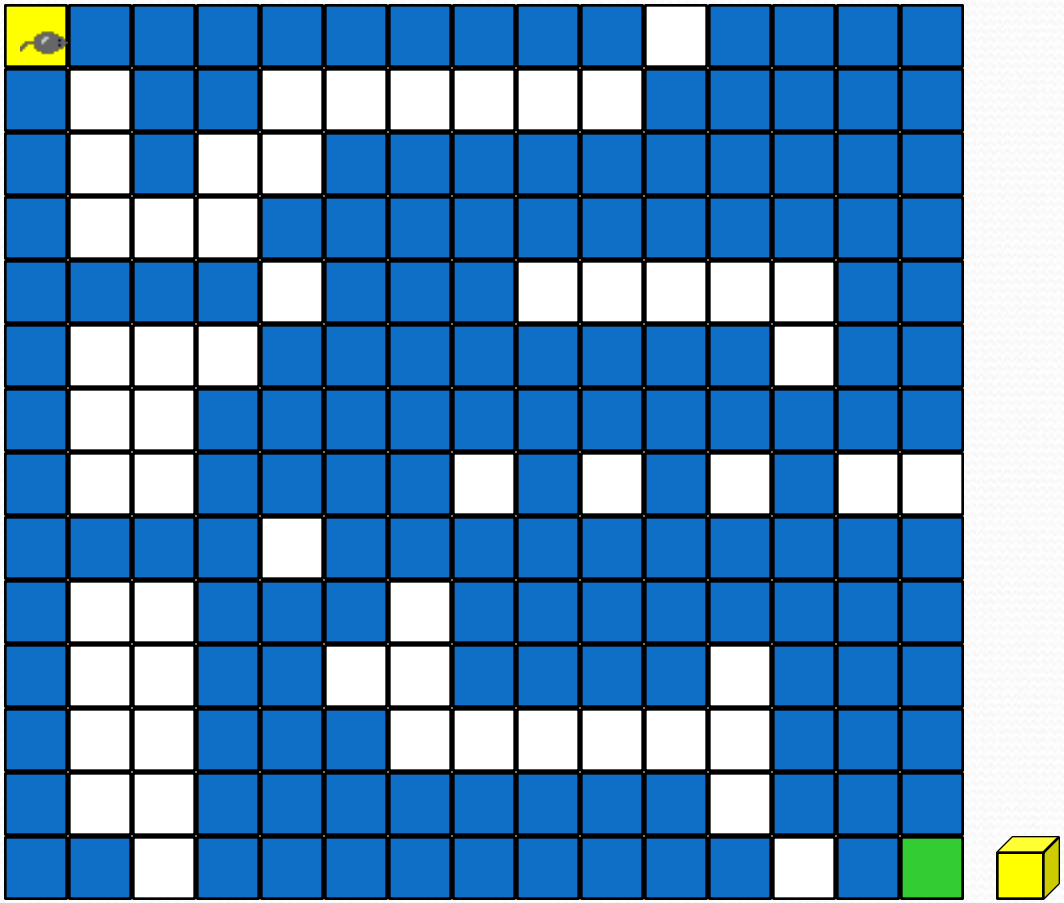
front

rear

Rat In A Maze

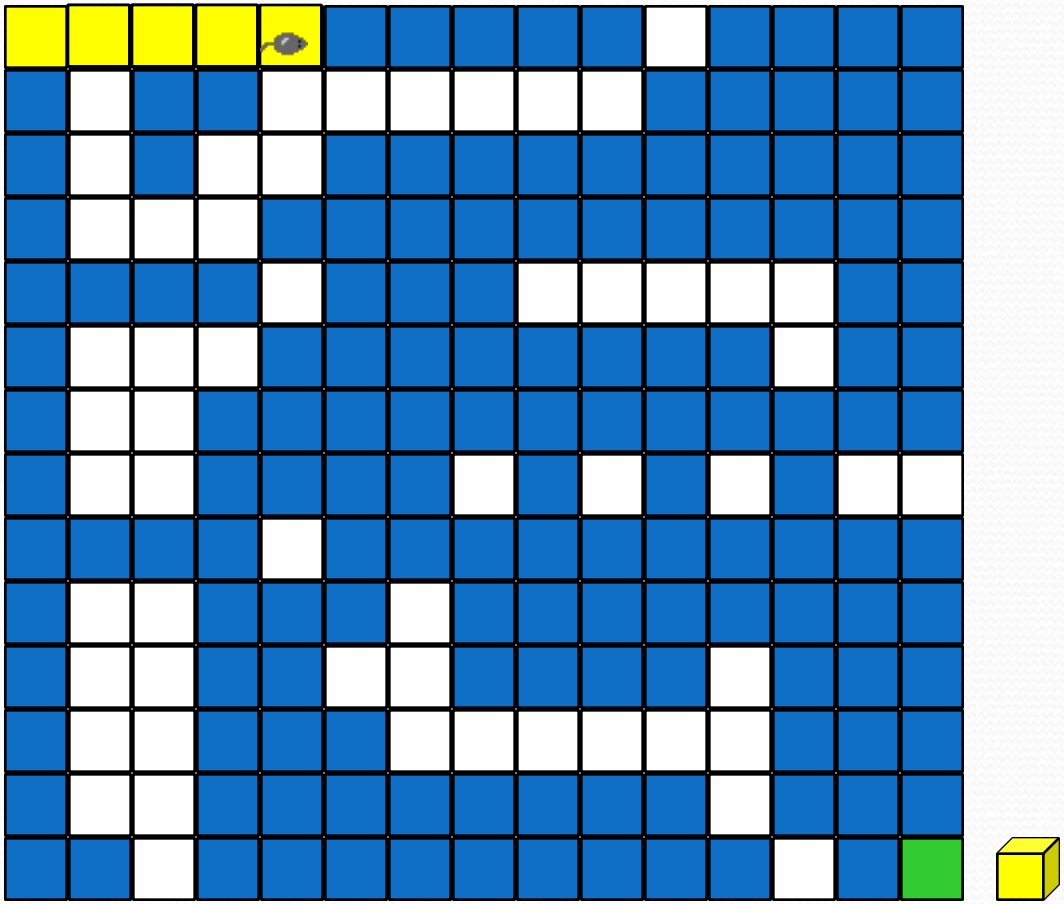


Rat In A Maze



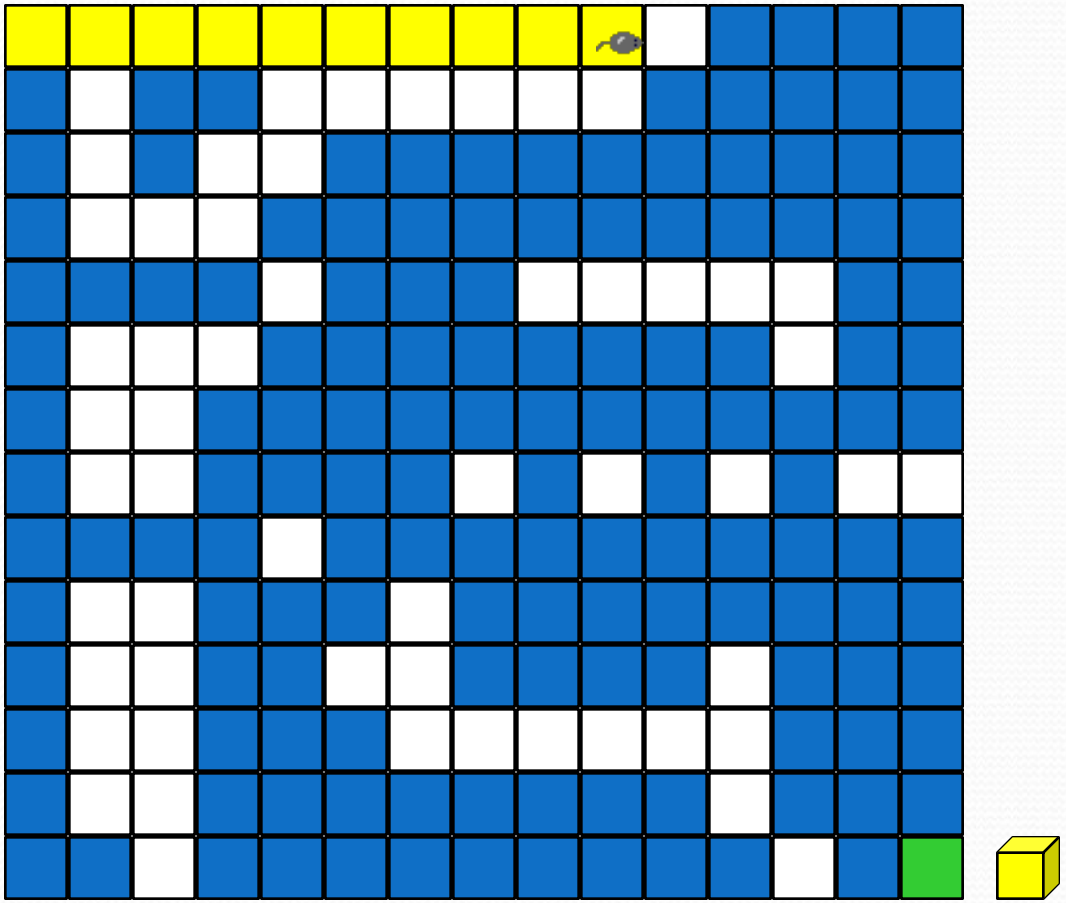
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



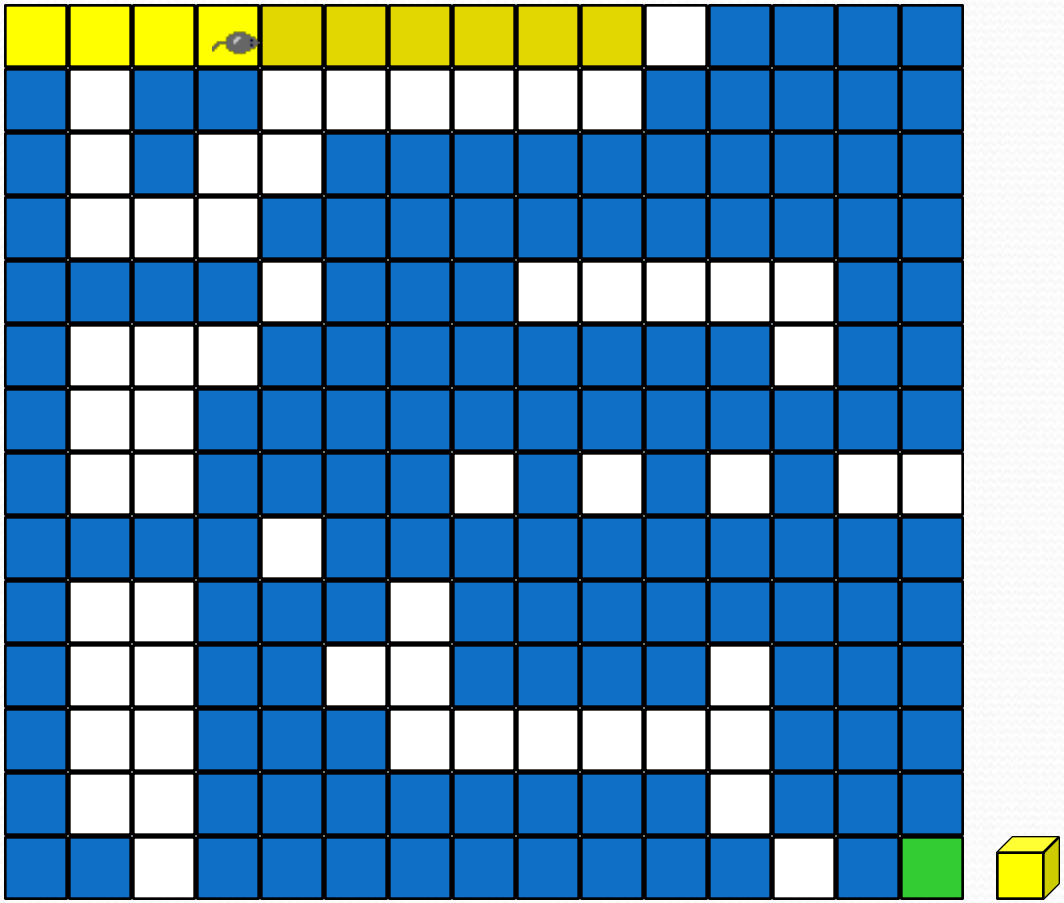
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



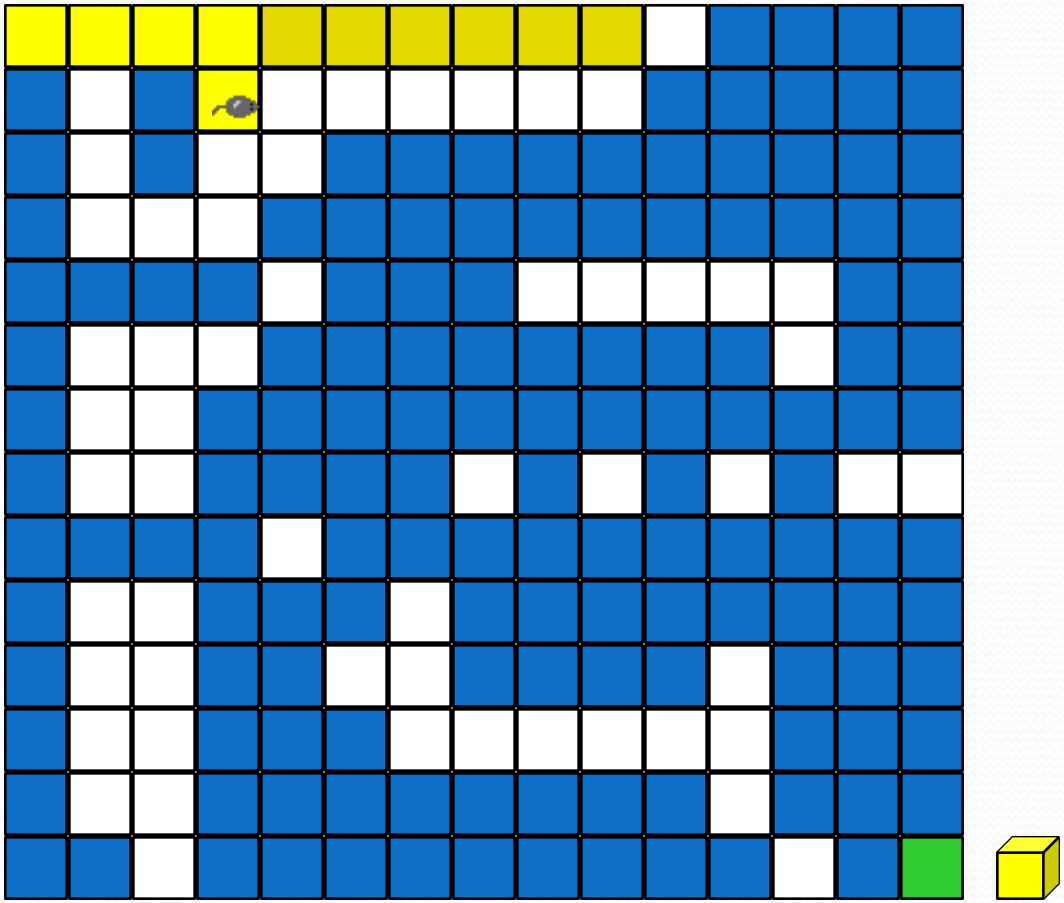
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



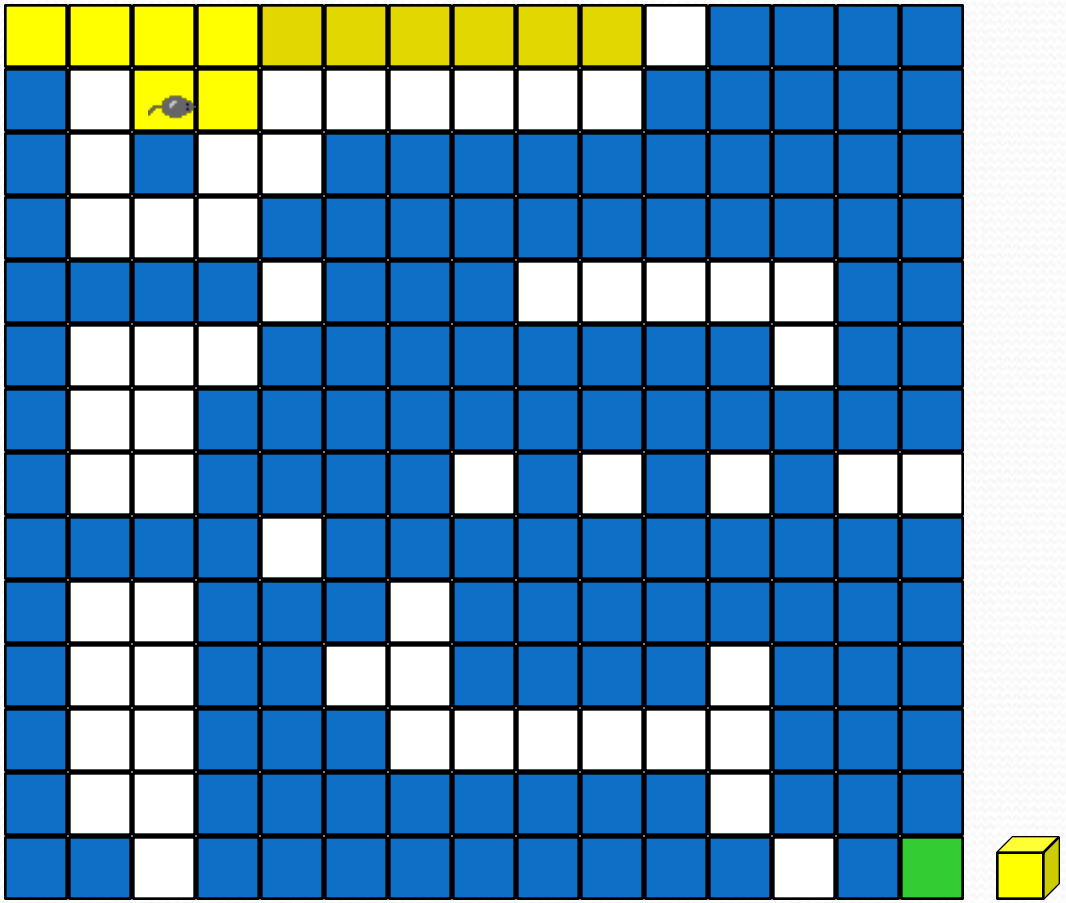
- Move down.

Rat In A Maze



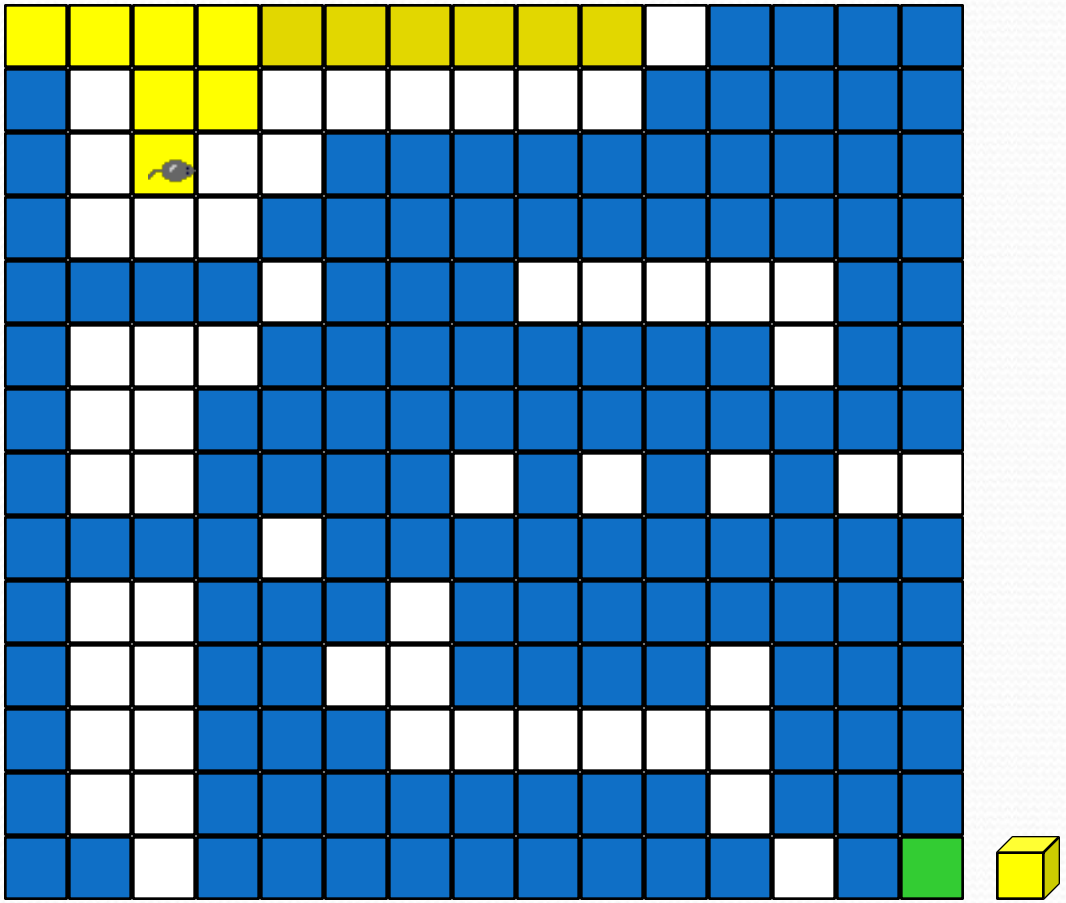
- Move left.

Rat In A Maze



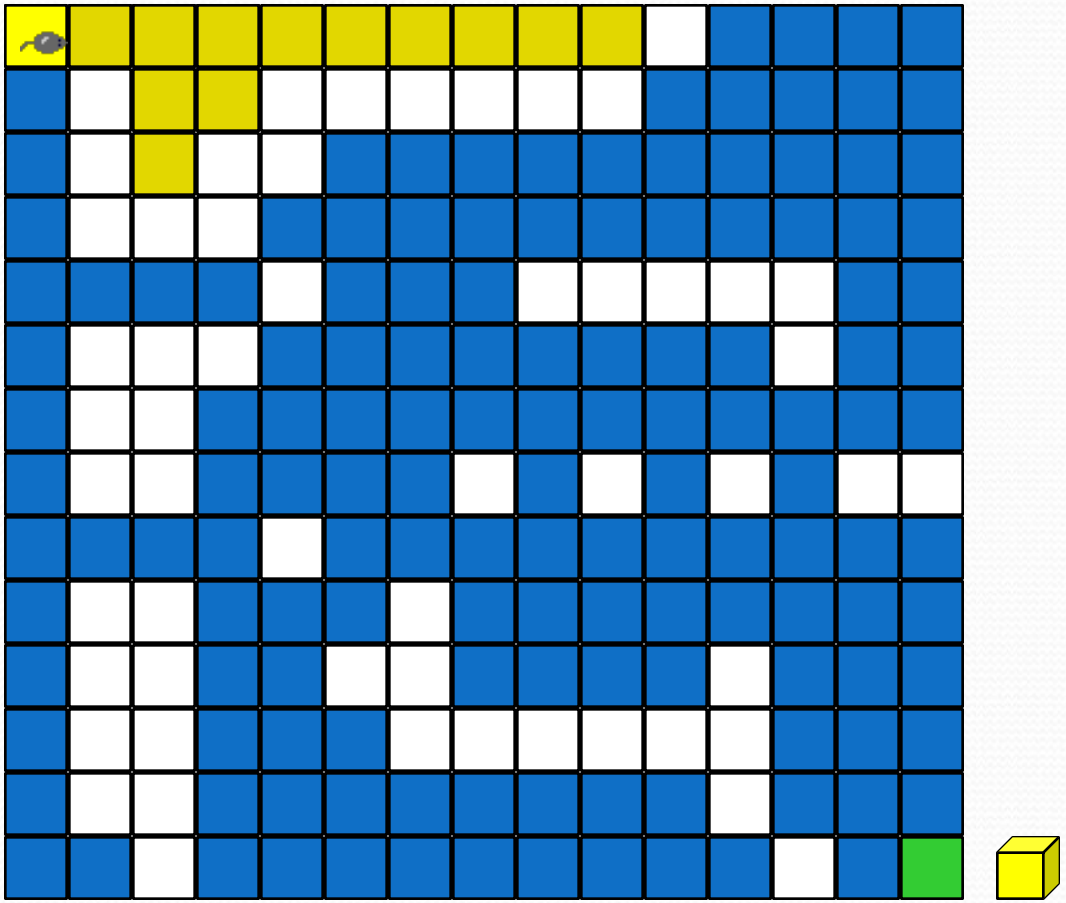
- Move down.

Rat In A Maze



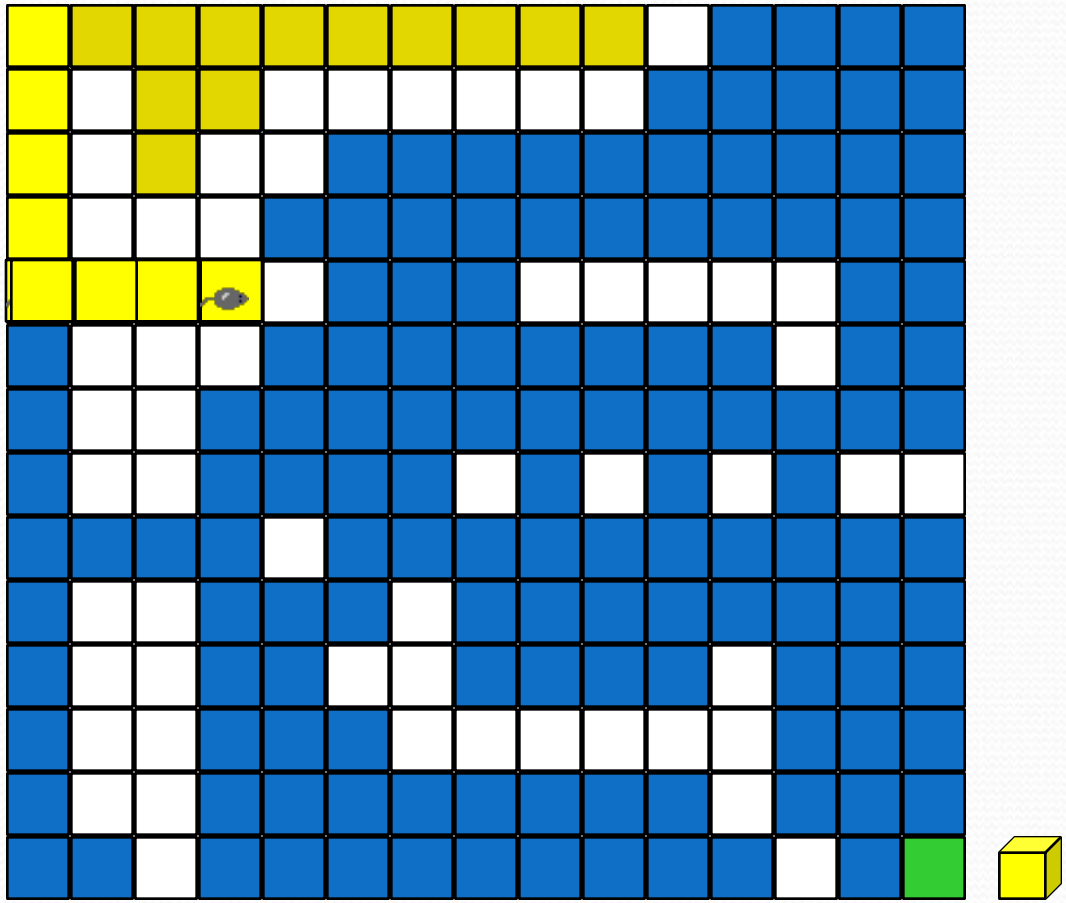
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



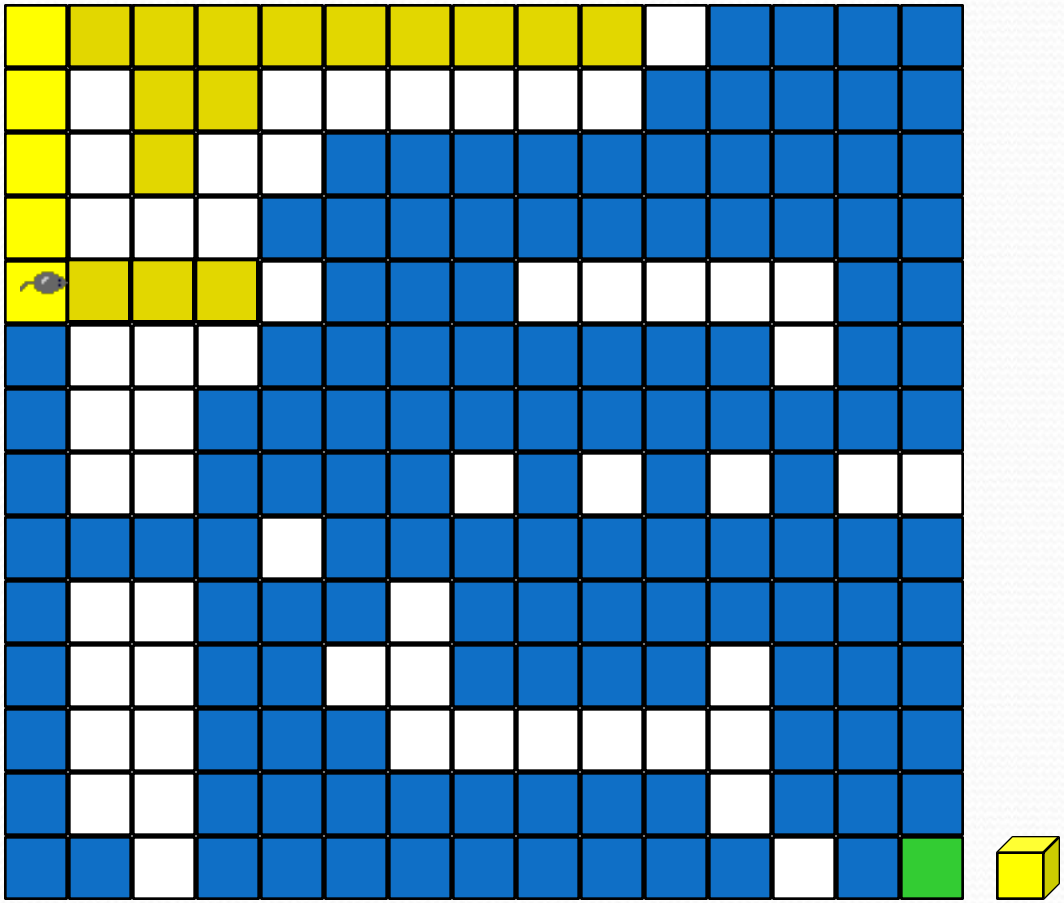
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

Rat In A Maze



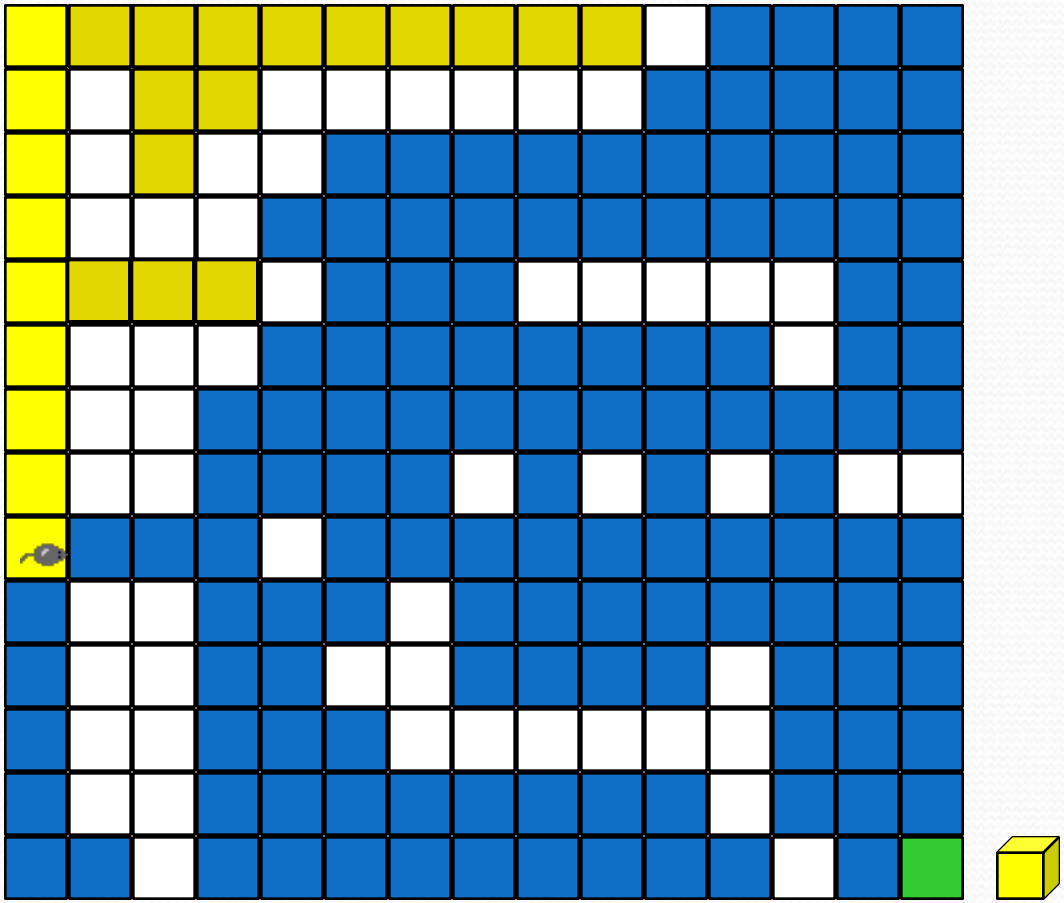
- Move right.
- Backtrack.

Rat In A Maze



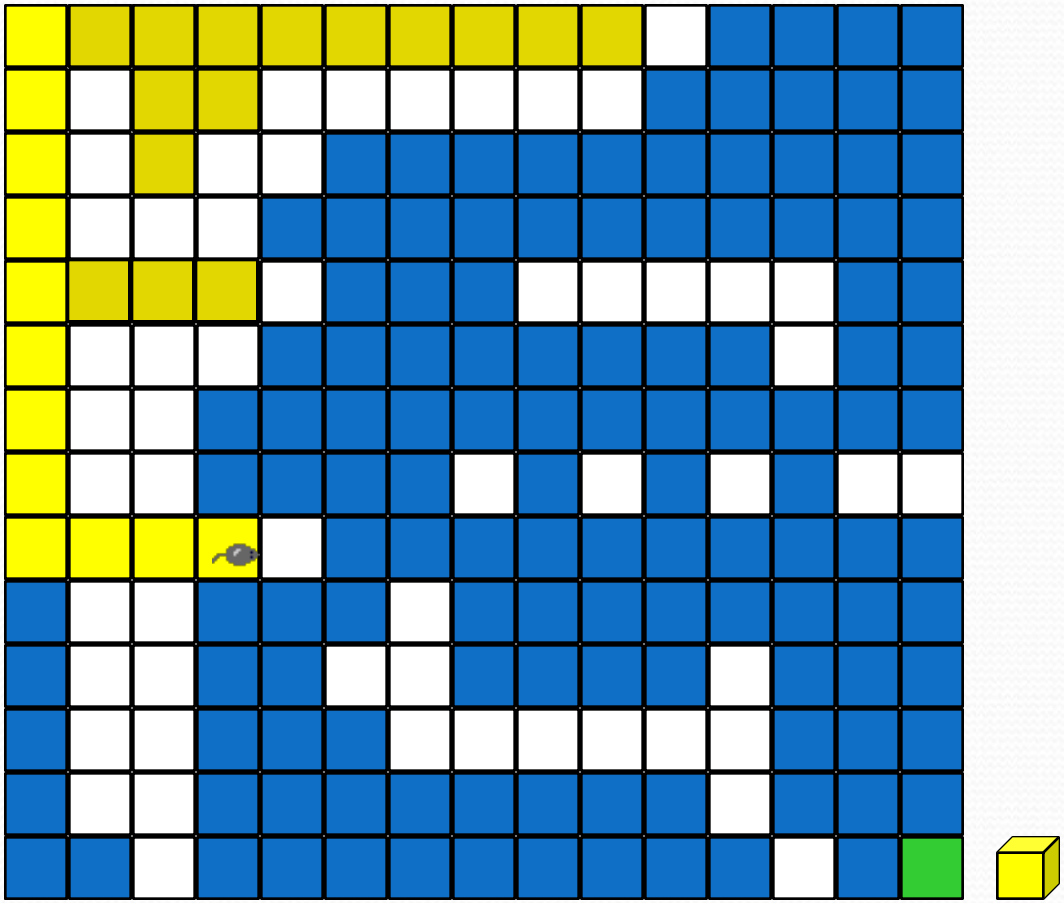
- Move downward.

Rat In A Maze



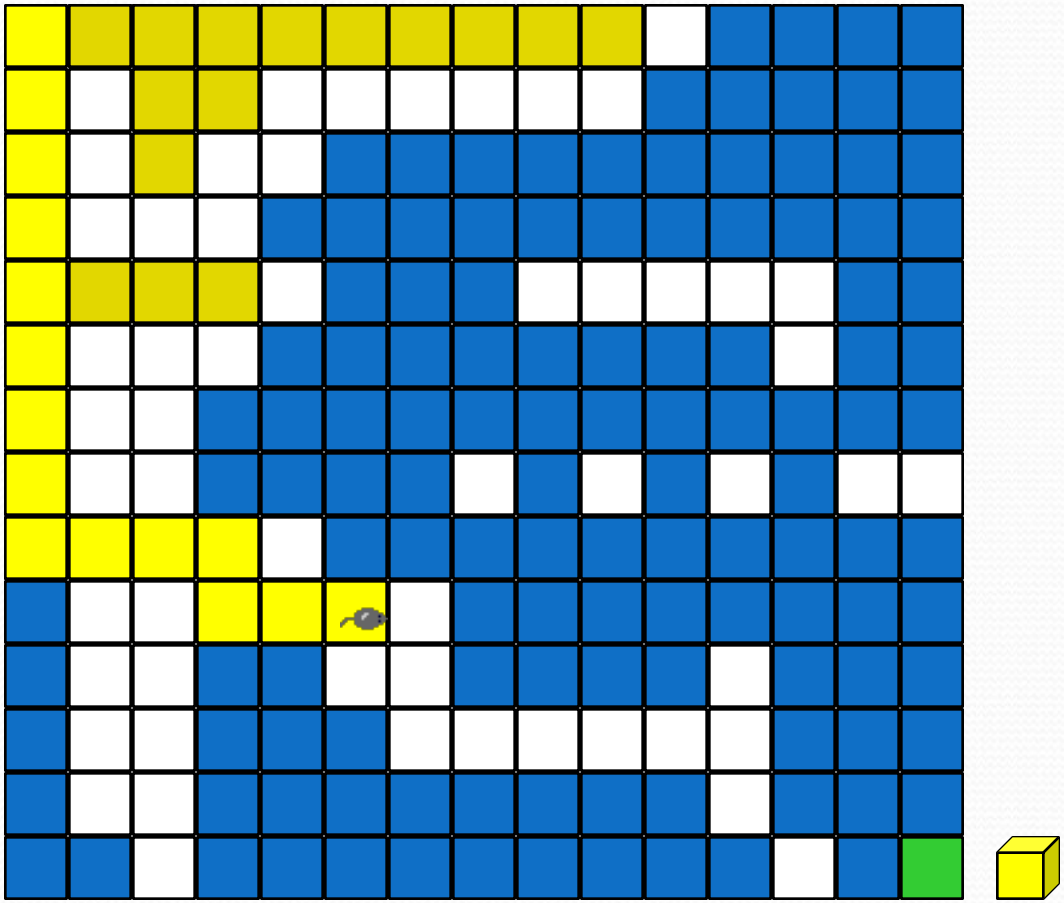
- Move right.

Rat In A Maze



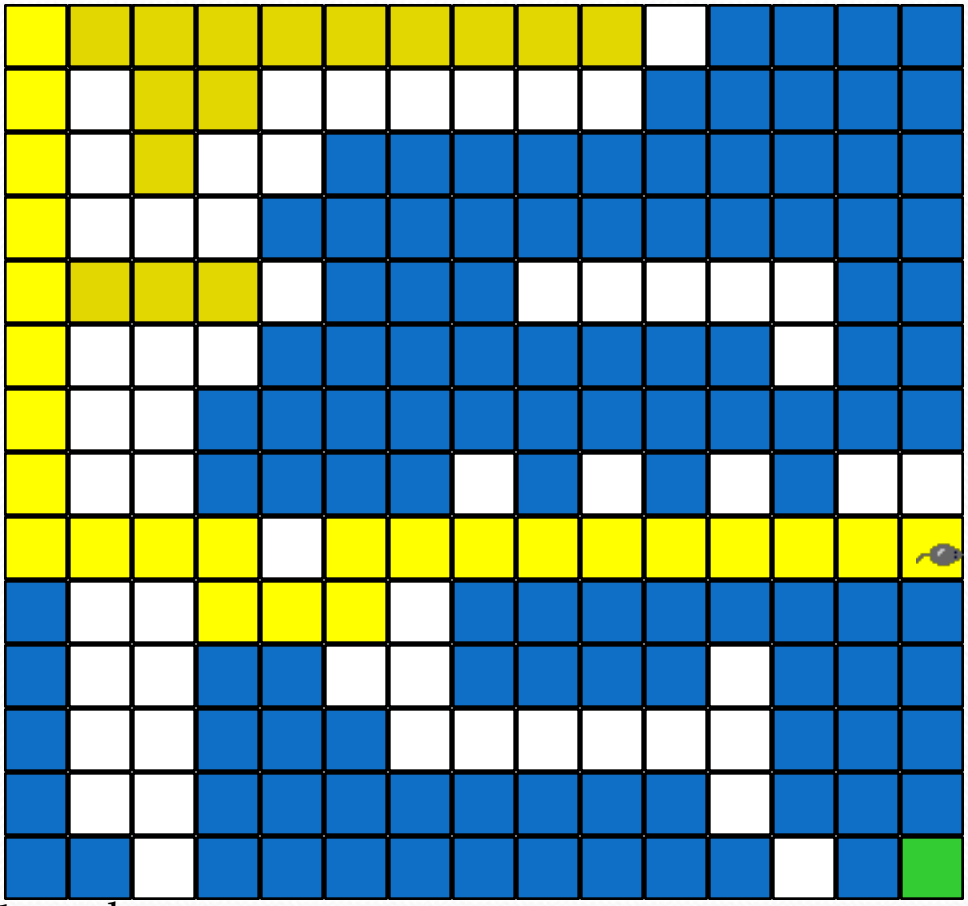
- Move one down and then right.

Rat In A Maze



- Move one up and then right.

Rat In A Maze



- Move down to exit and eat cheese.
- Path from maze entry to current position operates as a stack.

Abstract Data Types

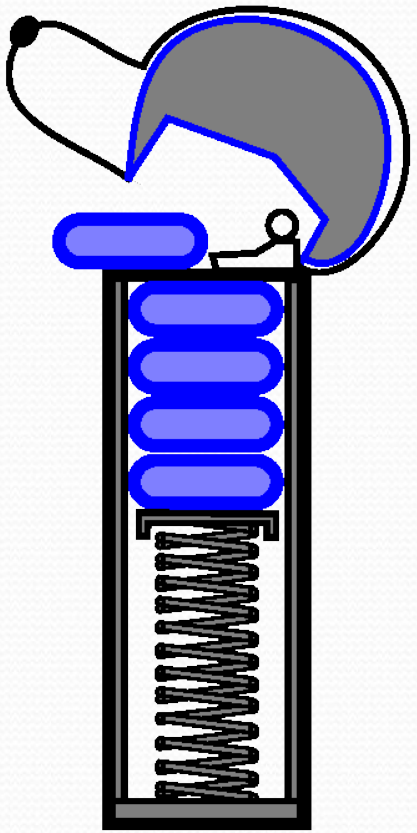
- Why do we need to talk about ADTs in A&DS course?
  - They serve as *specifications* of *requirements* for the building blocks of solutions to algorithmic problems
  - Provides a language to talk on a higher level of abstraction
  - ADTs encapsulate *data structures* and algorithms that *implement* them

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Stacks (2)

- A PEZ ® dispenser as an analogy:



Stacks(3)

- A stack is an ADT that supports three main methods:
  - **push(S:***ADT,* **o:***element***):***ADT* - Inserts object *o* onto top of stack *S*
  - **pop(S:***ADT***):***ADT* - Removes the top object of stack *S*; if the stack is empty an error occurs
  - **top(S:***ADT***):***element* – Returns the top object of the stack, without removing it; if the stack is empty an error occurs

Stacks(4)

- The following support methods should also be defined:
  - **size(S:***ADT***):***integer* - Returns the number of objects in stack *S*
  - **isEmpty(S:***ADT***):** *boolean* - Indicates if stack *S* is empty
- Axioms
  - **Pop(Push(S, v)) = S**
  - **Top(Push(S, v)) = v**

An Array Implementation

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

An Array Implementation (2)

Pseudo code

```
Algorithm size()
return t+1

Algorithm isEmpty()
return (t<0)

Algorithm top()
if isEmpty() then
    return Error
return S[t]
```

```
Algorithm push(o)
if size()==N then
    return Error
t=t+1
S[t]=o

Algorithm pop()
if isEmpty() then
    return Error
return S[t]

S[t]=null
t=t-1
```

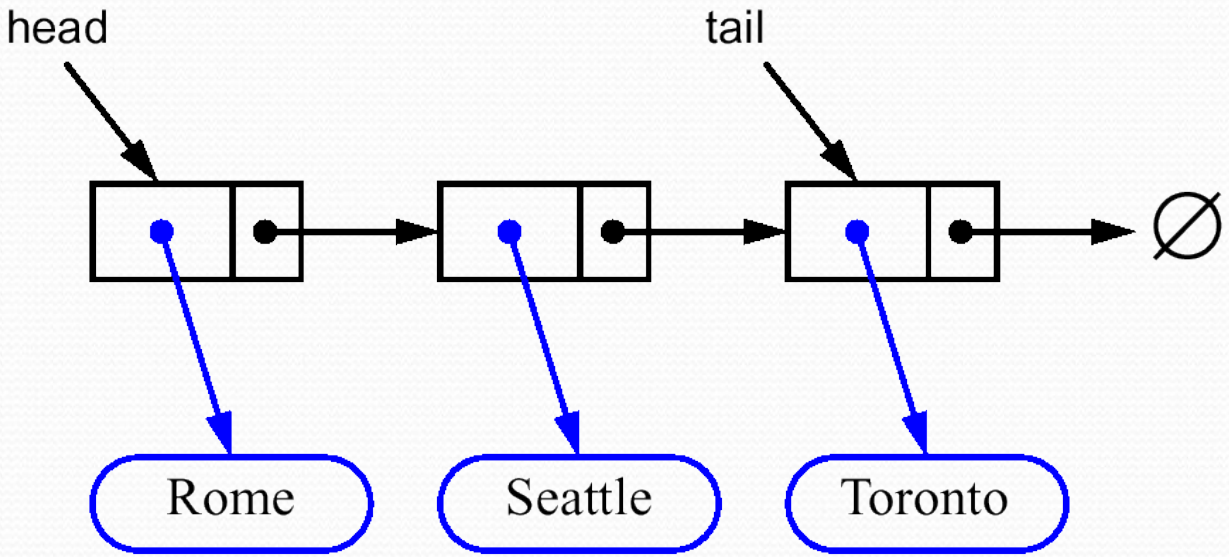
An Array Implementation (3)

- The array implementation is simple and efficient (methods performed in O(1)).

- There is an upper bound, *N*, on the size of the stack. The arbitrary value *N* may be too small for a given application, or a waste of memory.

Singly Linked List

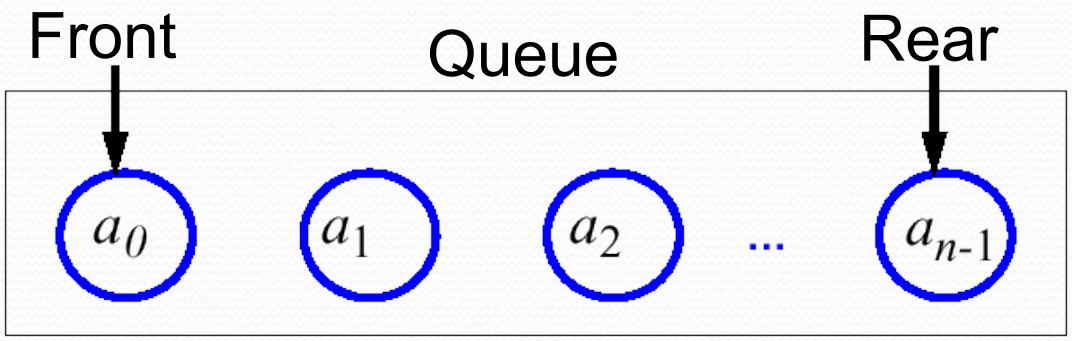
- Nodes *(data, pointer)* connected in a chain by links



- the head or the tail of the list could serve as the top of the stack

Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues (2)

- The **queue** supports three fundamental methods:
  - **Enqueue(S:***ADT*, **o:***element*)**:***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT*)**:***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT*)**:***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues (3)

- These support methods should also be defined:
  - **New():***ADT – Creates an empty queue*
  - **Size(S:***ADT***):***integer*
  - **IsEmpty(S:***ADT***):***boolean*

- Axioms:
  - **Front(Enqueue(New(), v))** = **v**
  - **Dequeque(Enqueue(New(), v)) = New()**
  - **Front(Enqueue(Enqueue(Q, w), v)) = Front(Enqueue(Q, w))**
  - **Dequeue(Enqueue(Enqueue(Q, w), v)) = Enqueue(Dequeue(Enqueue(Q, w)), v)**

An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size N is specified.
- The queue consists of an N-element array Q and two integer variables:
  - f, index of the front element (head – for dequeue)
  - r, index of the element after the rear one (tail – for enqueue)

An Array Implementation (2)

- "wrapped around" configuration

- what does *f*=*r* mean?

An Array Implementation (3)

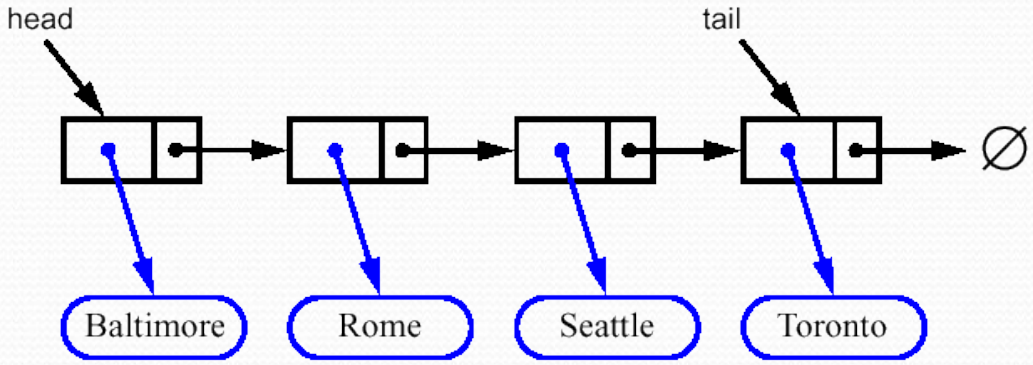
- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
    return Error
return Q[f]
```

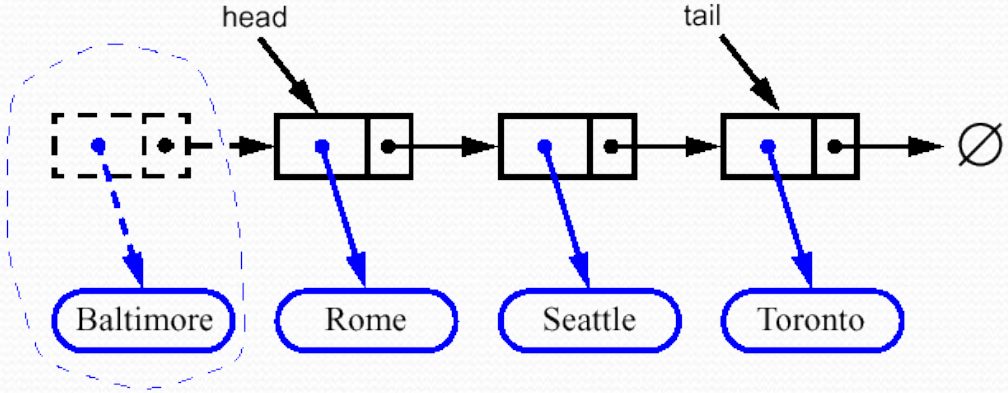
```
Algorithm dequeue()
if isEmpty() then
    return Error
Q[f]=null
f=(f+1)modN
```

```
Algorithm enqueue(o)
if size = N - 1 then
    return Error
Q[r]=o
r=(r +1)modN
```

Linked List Implementation

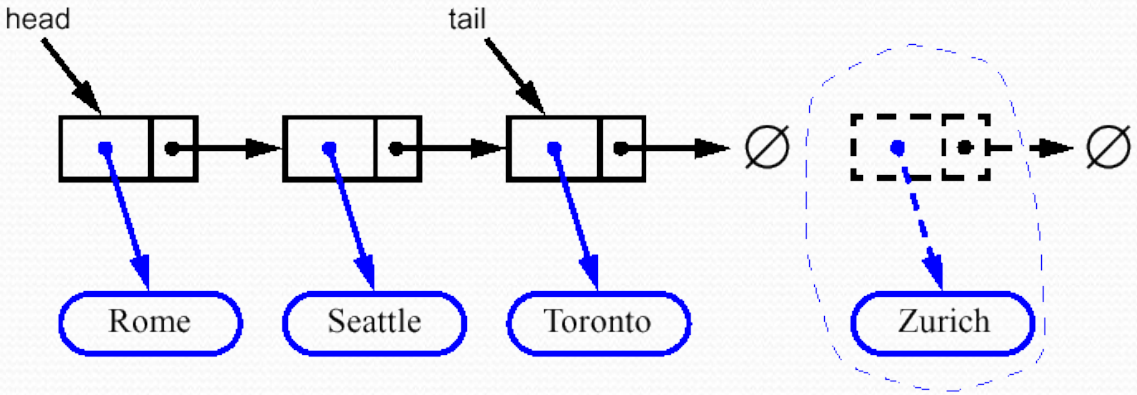


- Dequeue - advance head reference

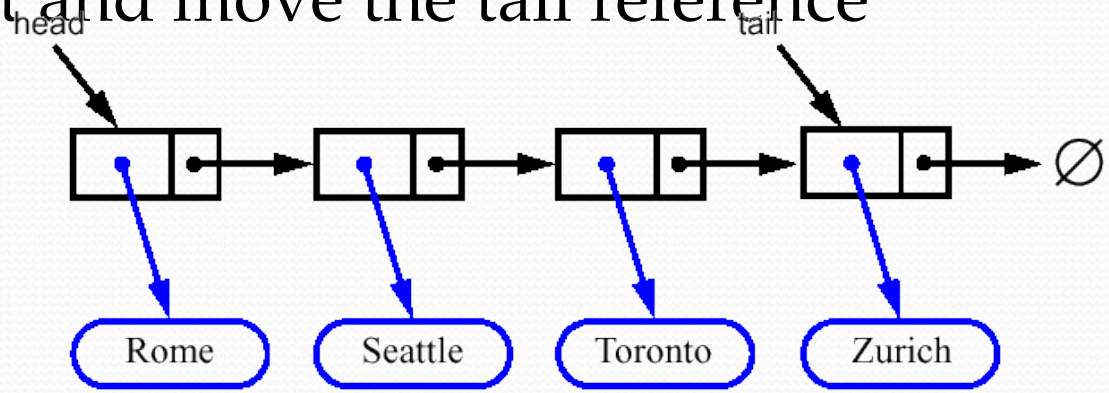


Linked List Implementation (2)

- Enqueue - create a new node at the tail



- chain it and move the tail reference



Thank You!!