Elementary Data Structures:

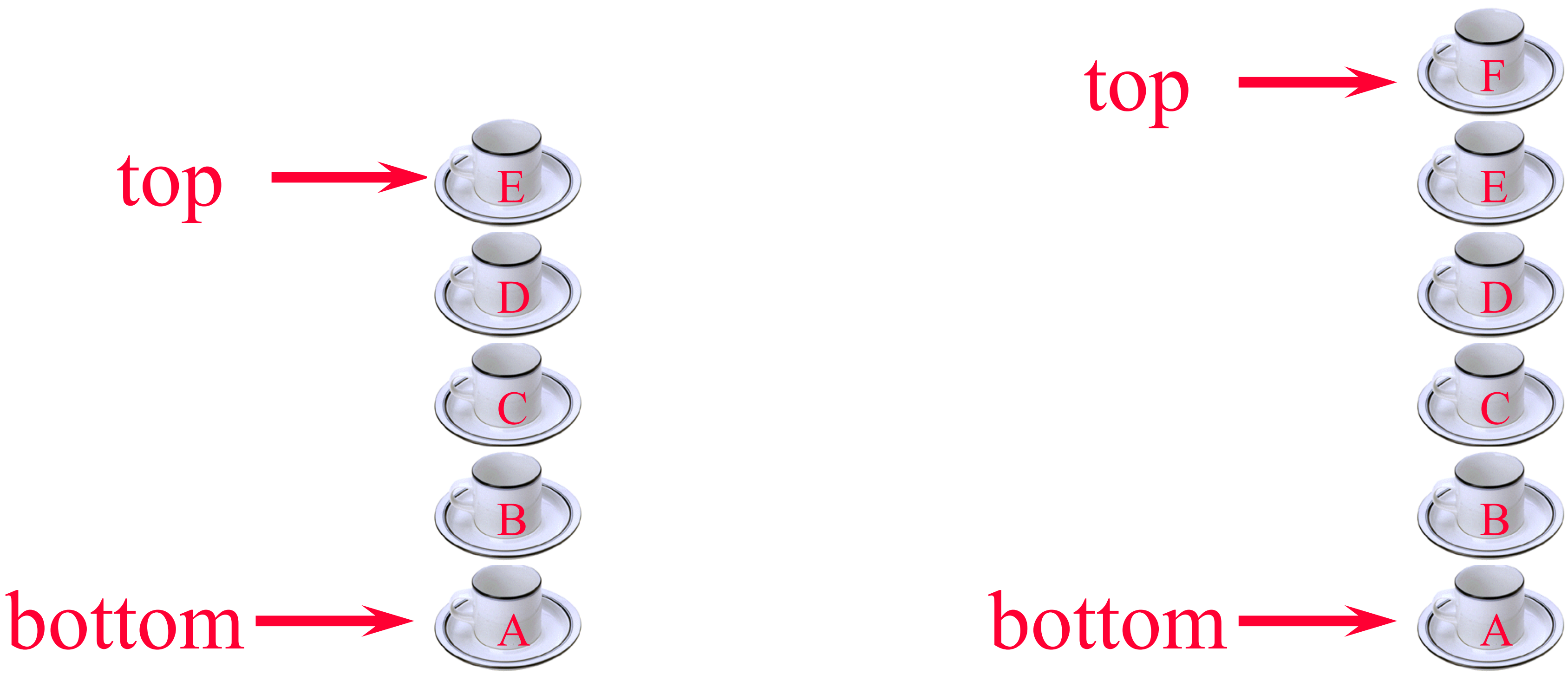
Stack, Queue, and Linked List

Stacks

- Linear list.
- One end is called top, other end is bottom.
- Additions to and removals from top only.
- Basic operations of stack
  - Pushing, popping etc.
- Stacks are less flexible

  - but are more efficient and easy to implement

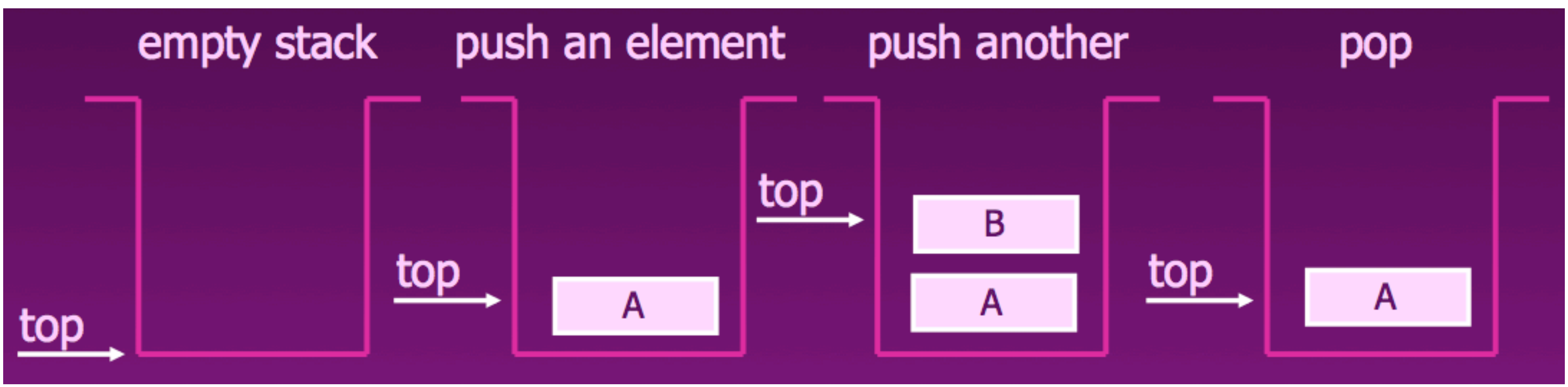
Stack Of Cups



- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO (last in first out) list.

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



Implementation of Stacks

- Any list implementation could be used to implement a stack
  - Arrays (static: the size of stack is given initially)
  - Linked lists (dynamic: never become full)
- How to use an array to implement a stack?

Array Implementation

- Need to declare an array size ahead of time
- Associated with each stack is TopOfStack
  - for an empty stack, set TopOfStack to -1
- Push (X): make the item argument
  - (1) Increment TopOfStack by 1.
  - (2) Set Stack[TopOfStack] = X
- Pop
  - (1) Set return value to Stack[TopOfStack]
  - (2) Decrement TopOfStack by 1
- These operations are very fast: O(1)

Push Stack

- `void Push(const double x);`
  - Push an element onto the stack
  - If the stack is full, print the error information.
  - Note `top` always represents the index of the top element. After pushing an element, increment `top`.

```
void Stack::Push(const double x) {
      if (IsFull())
            cout << "Error: the stack is full." << 
endl;
      else
            values[++top]     =     x;
}
```

Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don't forgot to decrement `top`

```
double Stack::Pop() {
      if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else {
            return values[top--];
      }
}
```

Stack Top

- `double Top()`
  - Return the top element of the stack
  - Unlike `Pop`, this function does not remove the top element

```
double Stack::Top() {
      if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else
            return values[top];
}
```

Practice Problem

- Write a function Pop2nd() for Stack that will pop the second element from the top.
- What is the complexity of Pop2nd()?

Stack Applications

- Stacks are a very common data structure
  - compilers
    - parsing data between delimiters (brackets)
  - operating systems
    - program stack
  - artificial intelligence
    - finding a path

Parentheses Matching

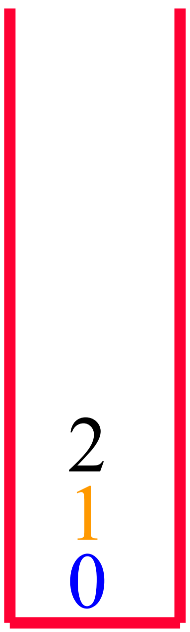
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

Parentheses Matching

- scan expression from left to right
- when a left parenthesis is encountered, push its position to the stack
- when a right parenthesis is encountered, pop matching position from stack

Example

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



Example

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

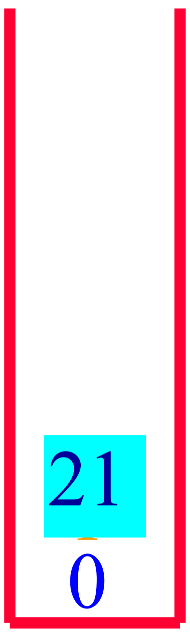
15

0

(2,6) (1,13)

Example

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



(2,6) (1,13) (15,19)

Example

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

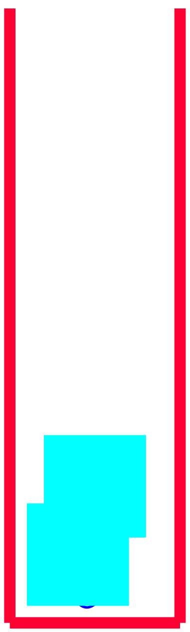
27

0

(2,6) (1,13) (15,19) (21,25)

Example

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



(2,6) (1,13) (15,19) (21,25)(27,31) (0,32)

- and so on

Practice Problem

- Show similar Stack operations for

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

- What will happen?

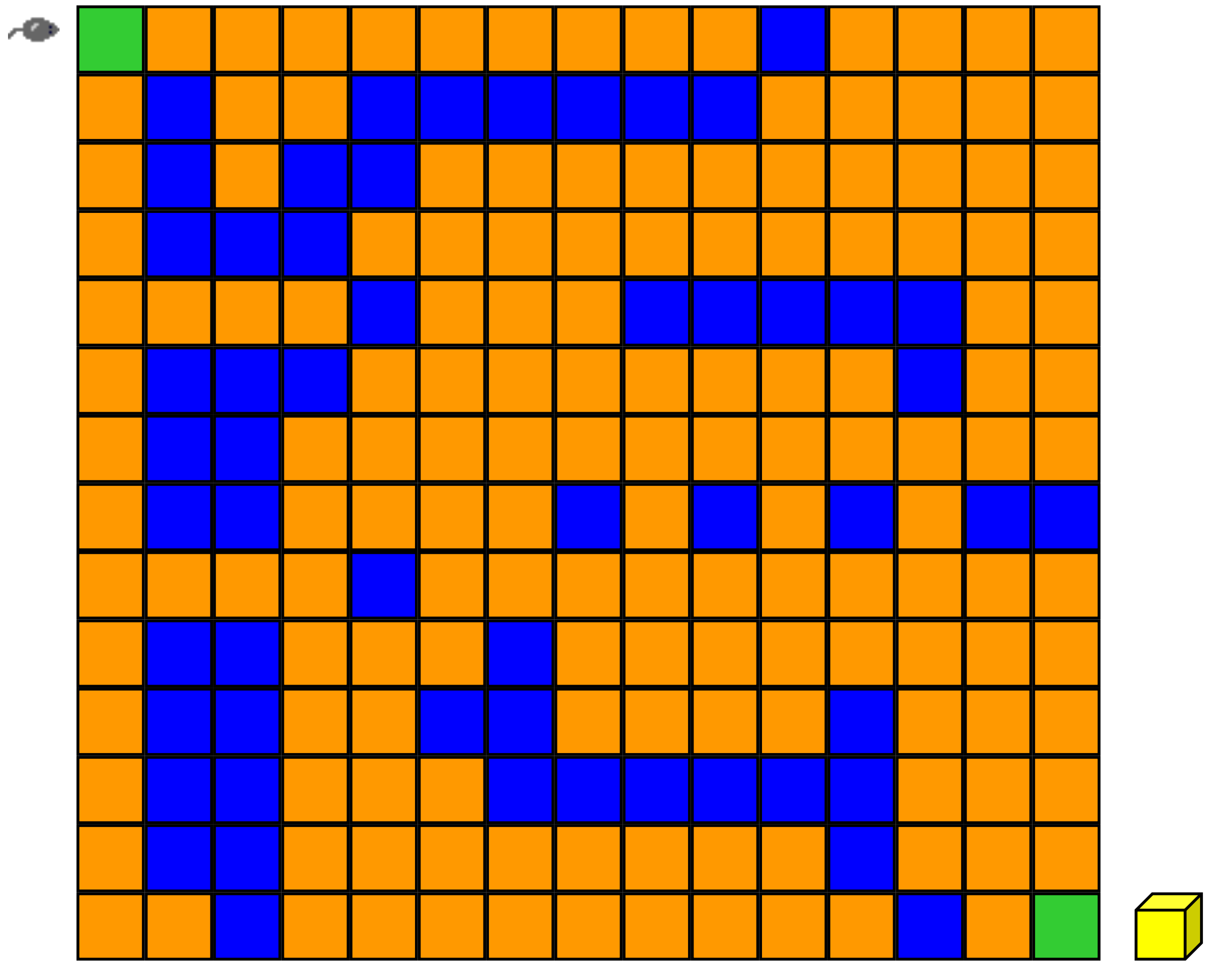
Practice Problem

- Show similar Stack operations for

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

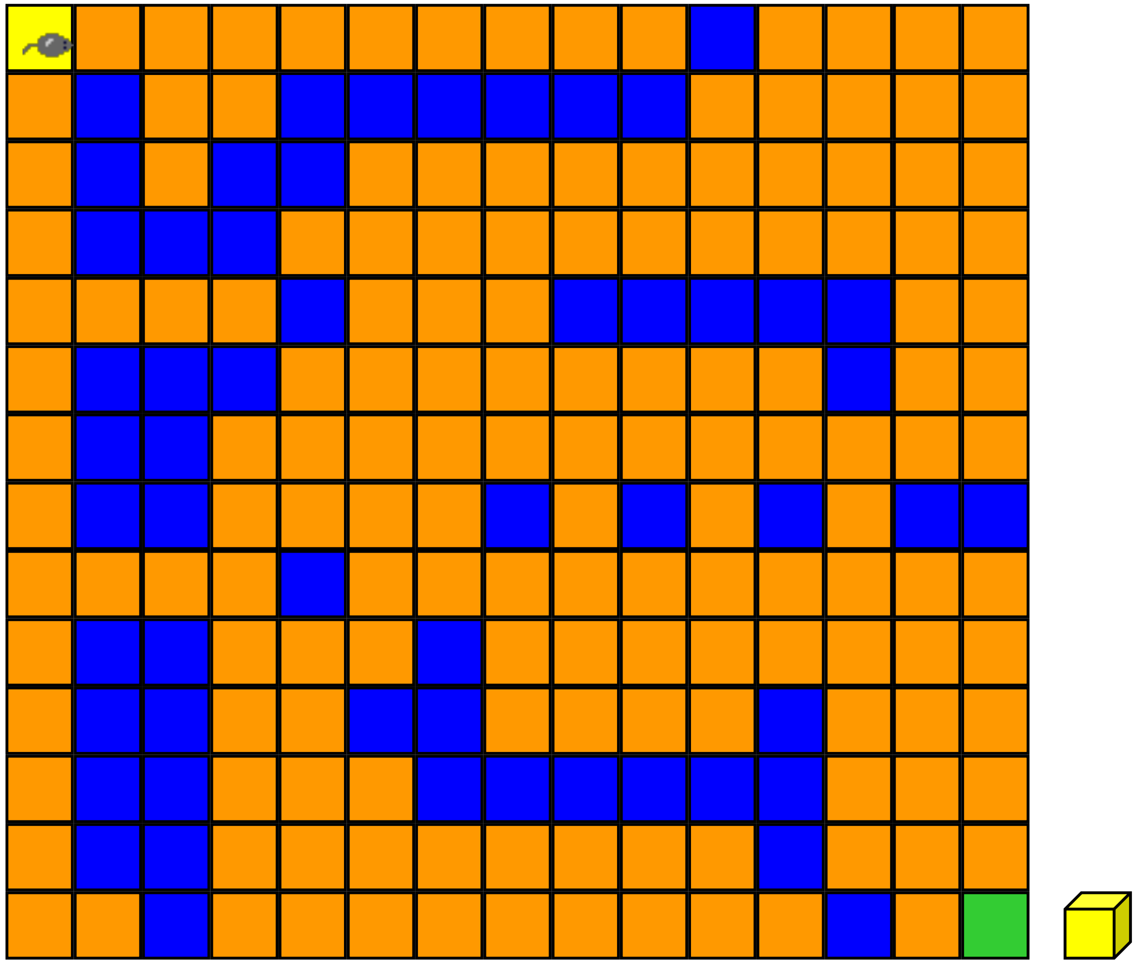
- What will happen?
  - For missing (, empty stack pop
  - For missing), statck remains non-empty at the end

Finding Path: Rat In A Maze



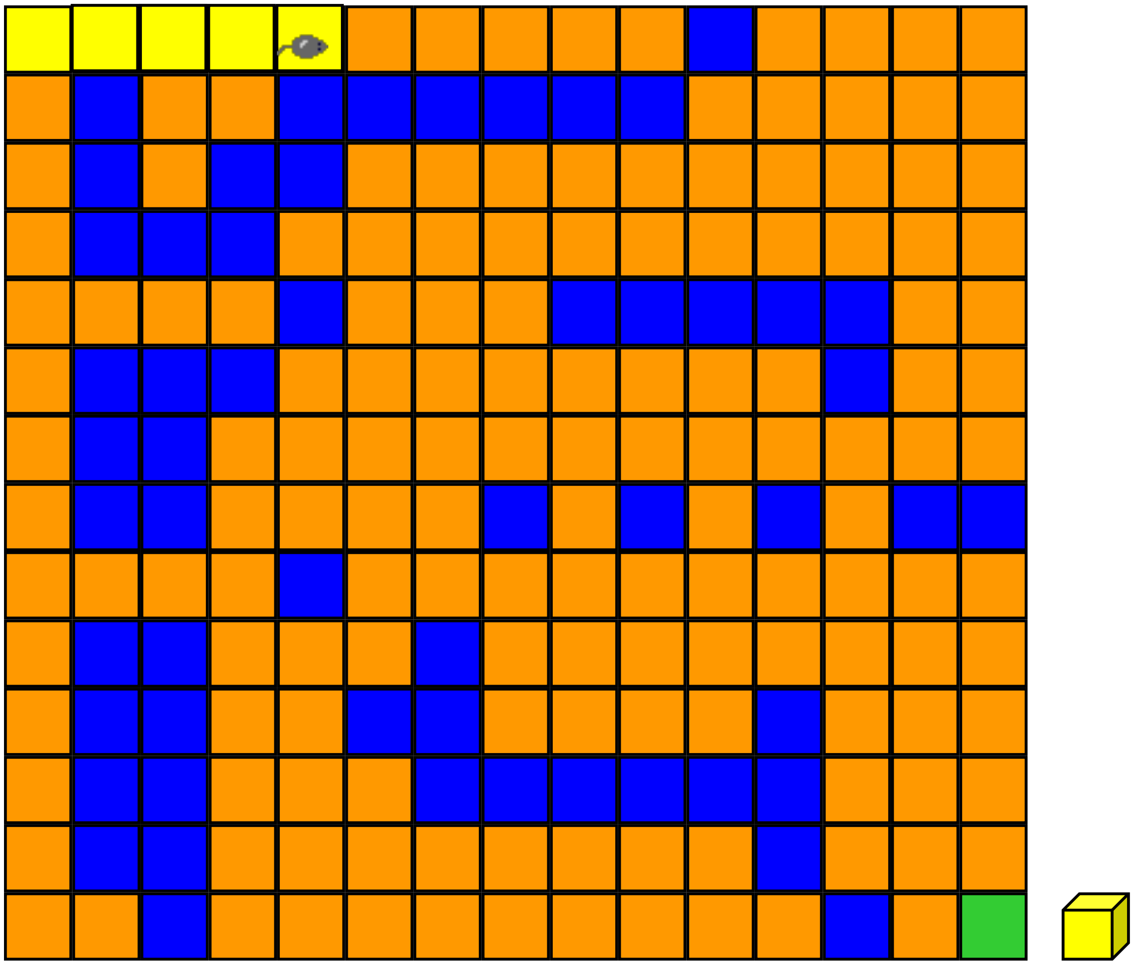
Orange and green squares are squares the rat can move to; blue squares cannot be moved to.

Rat In A Maze



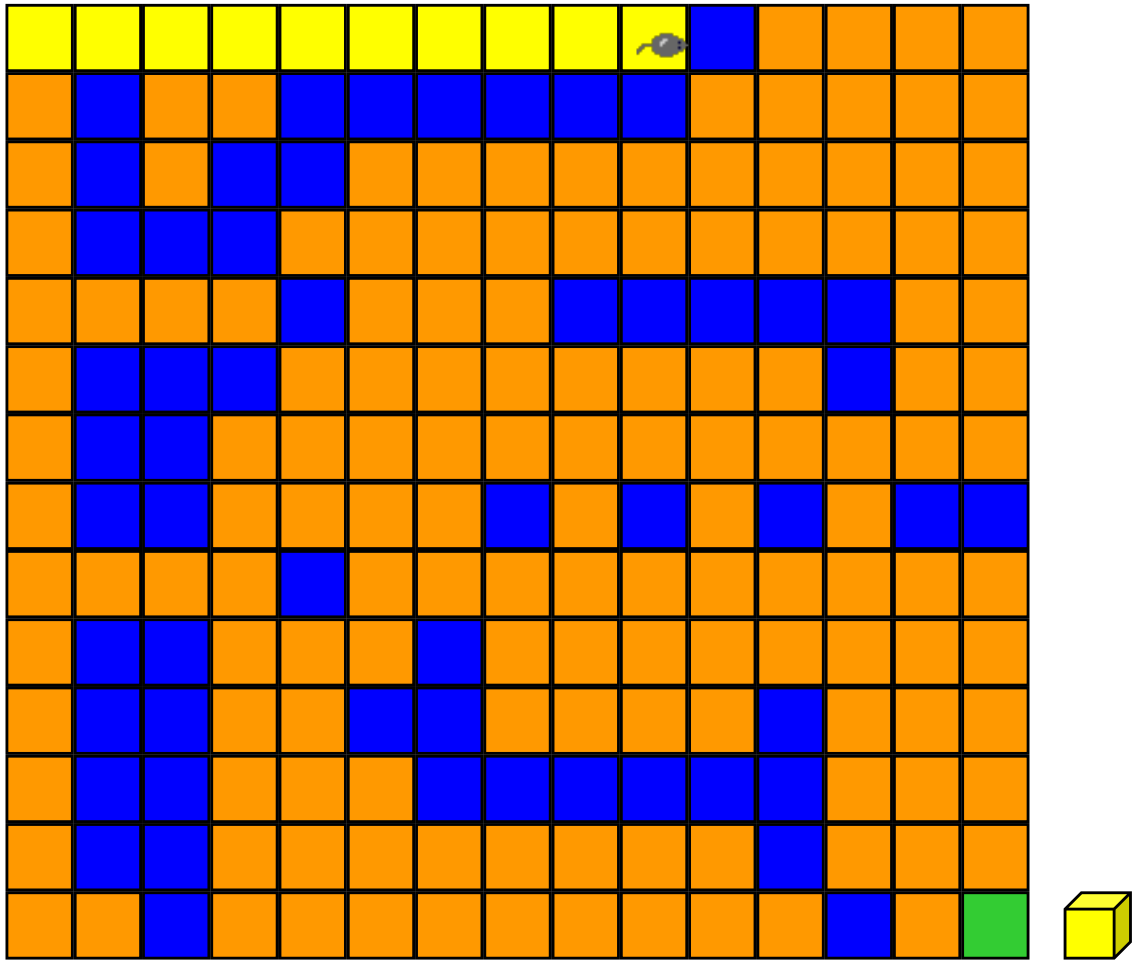
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



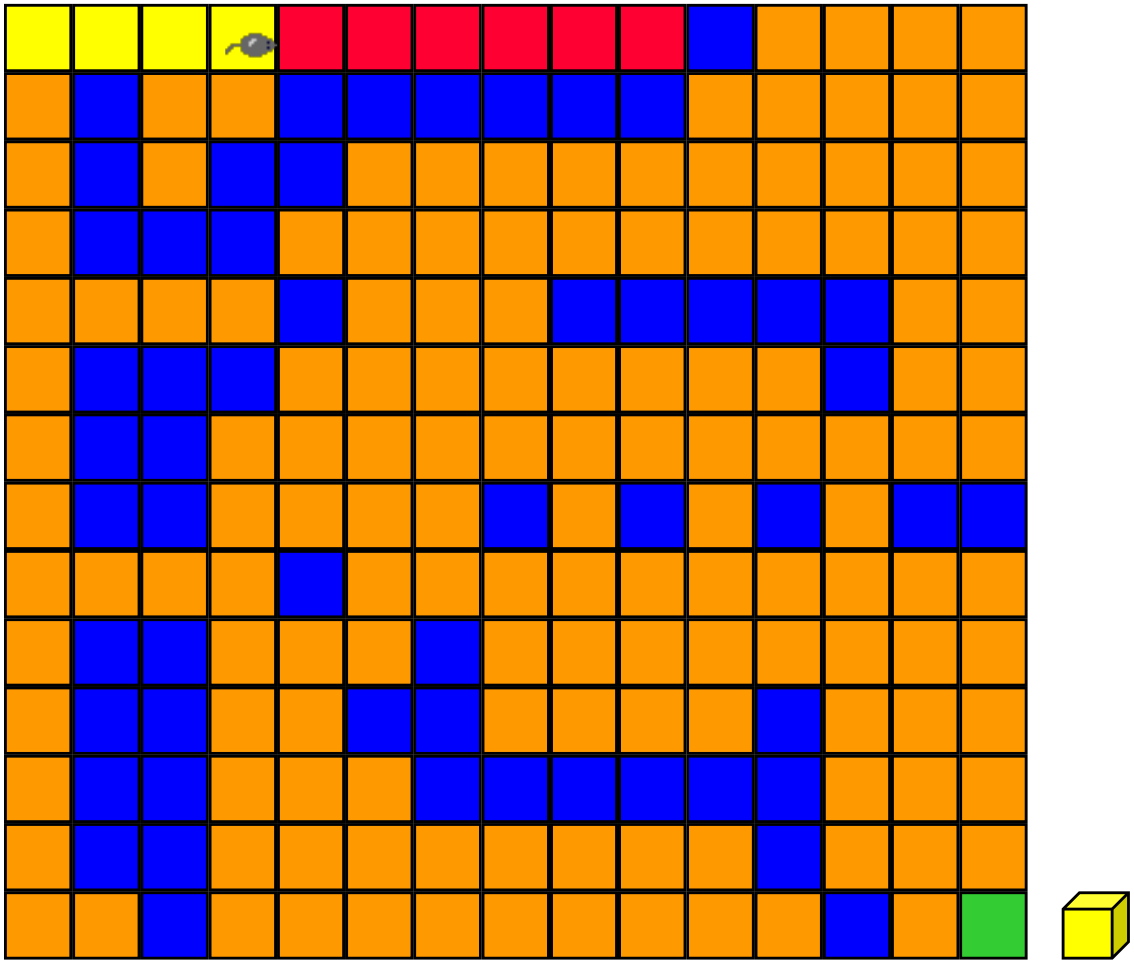
- Move order is: right, down, left, up
- Block positions to avoid revisit.

Rat In A Maze



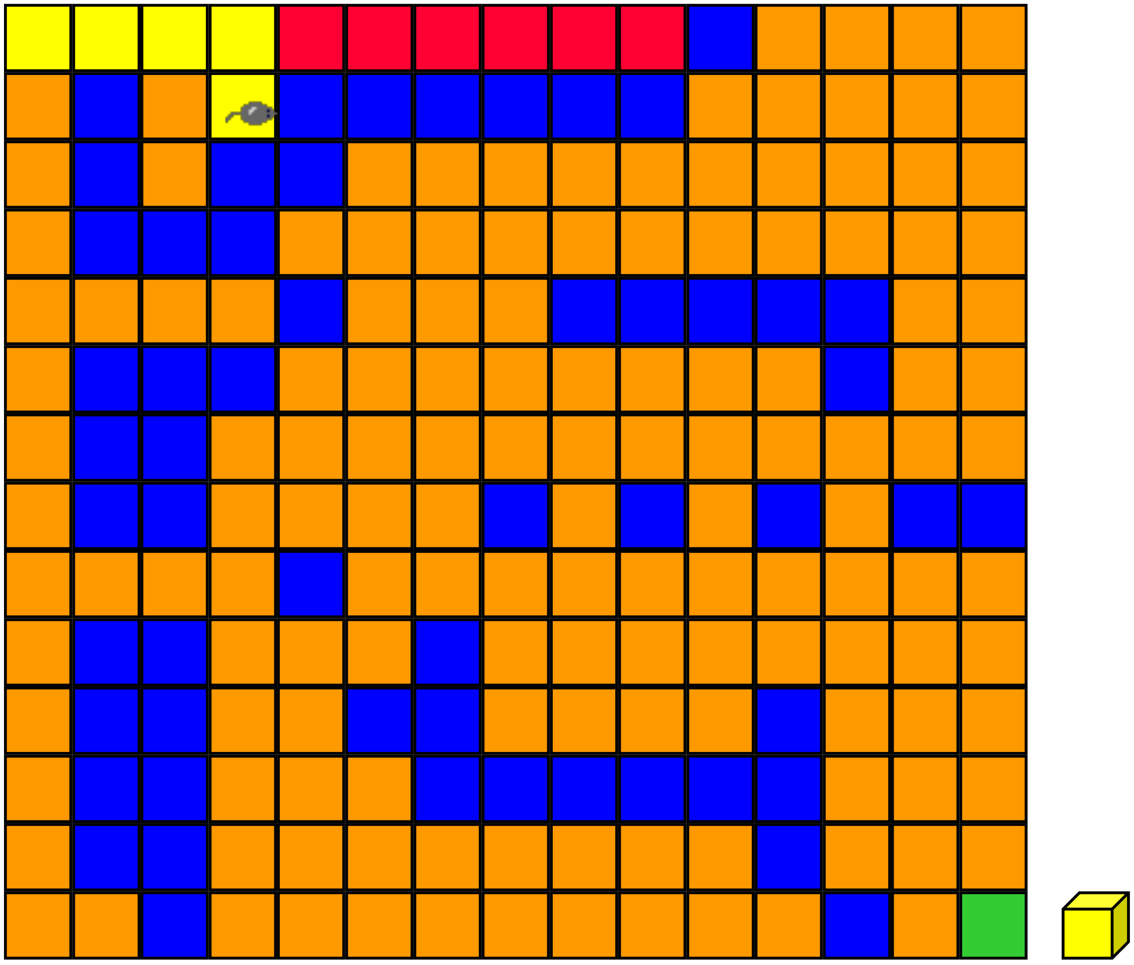
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



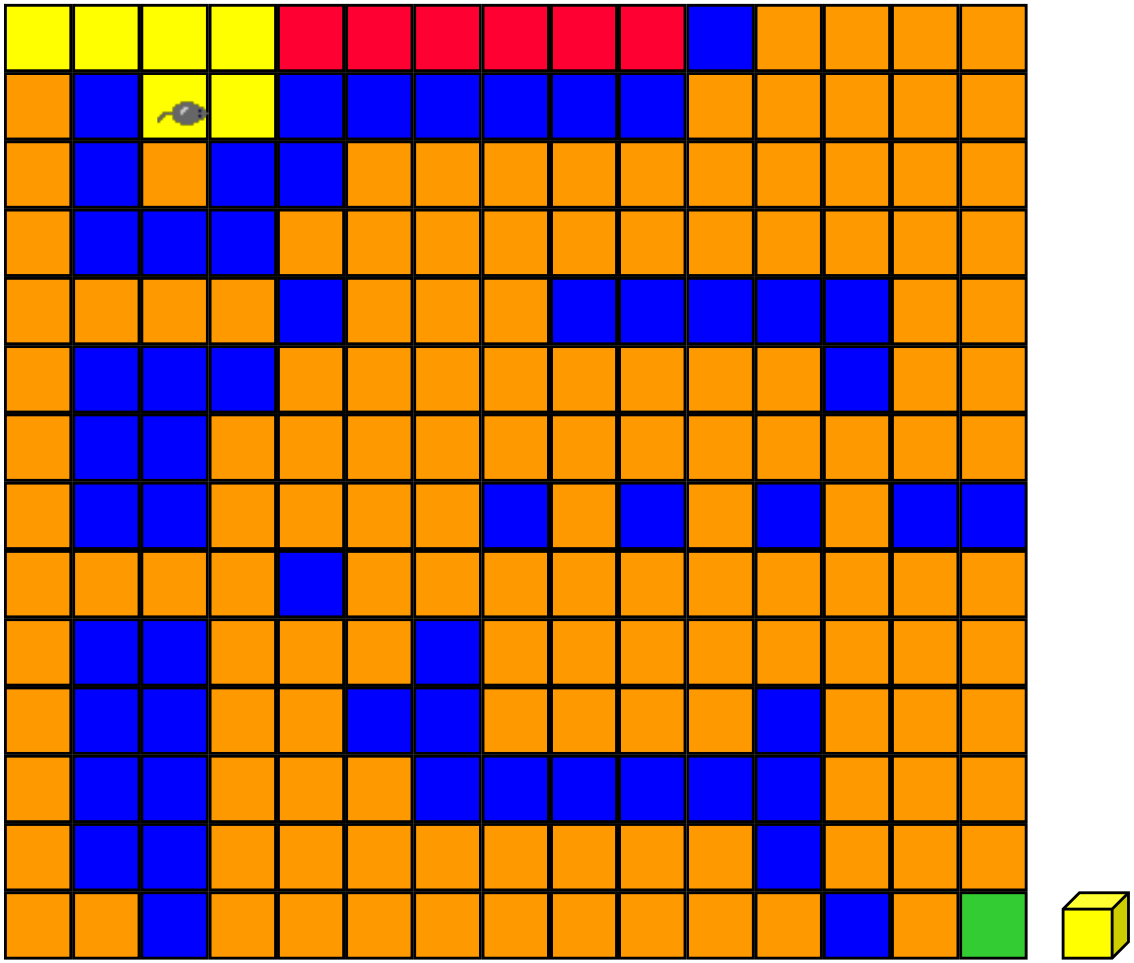
- Move down.

Rat In A Maze



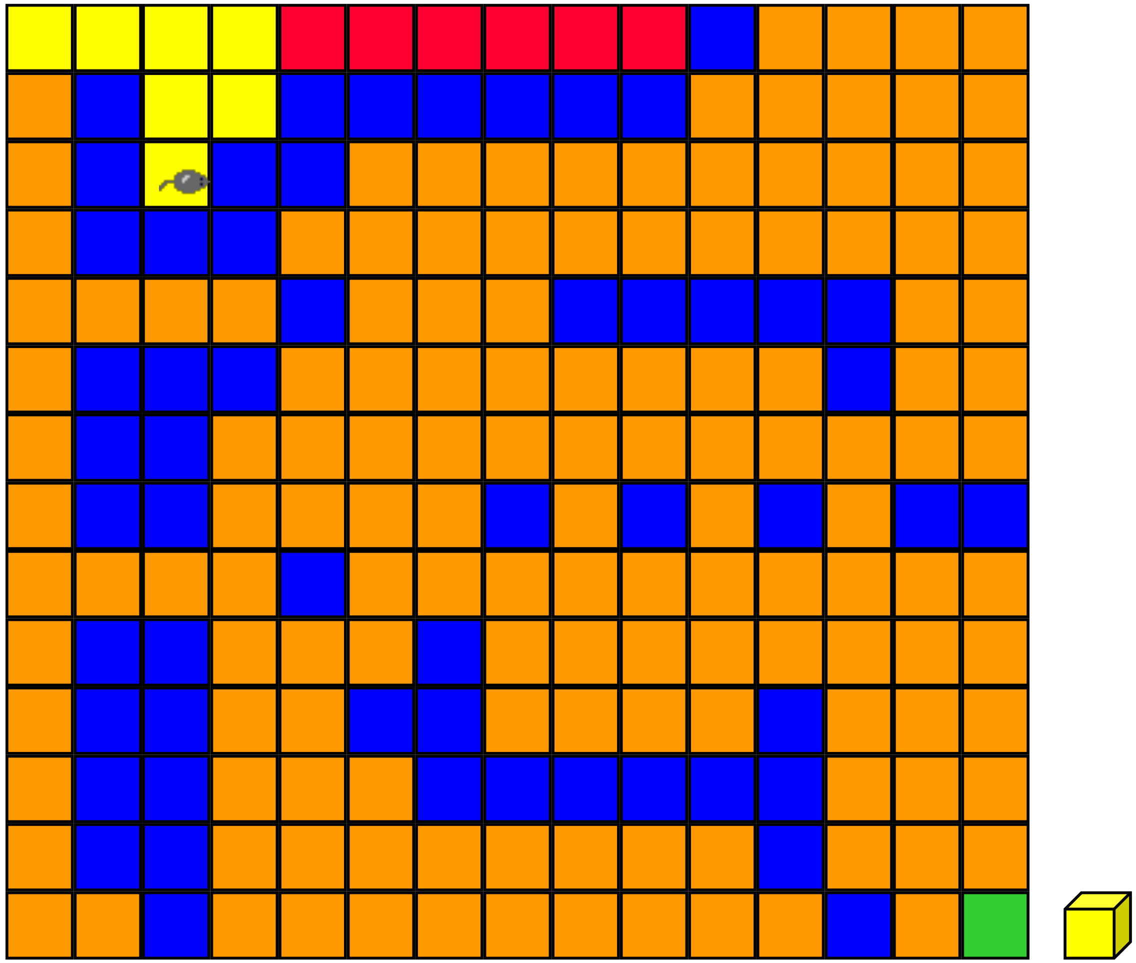
- Move left.

Rat In A Maze



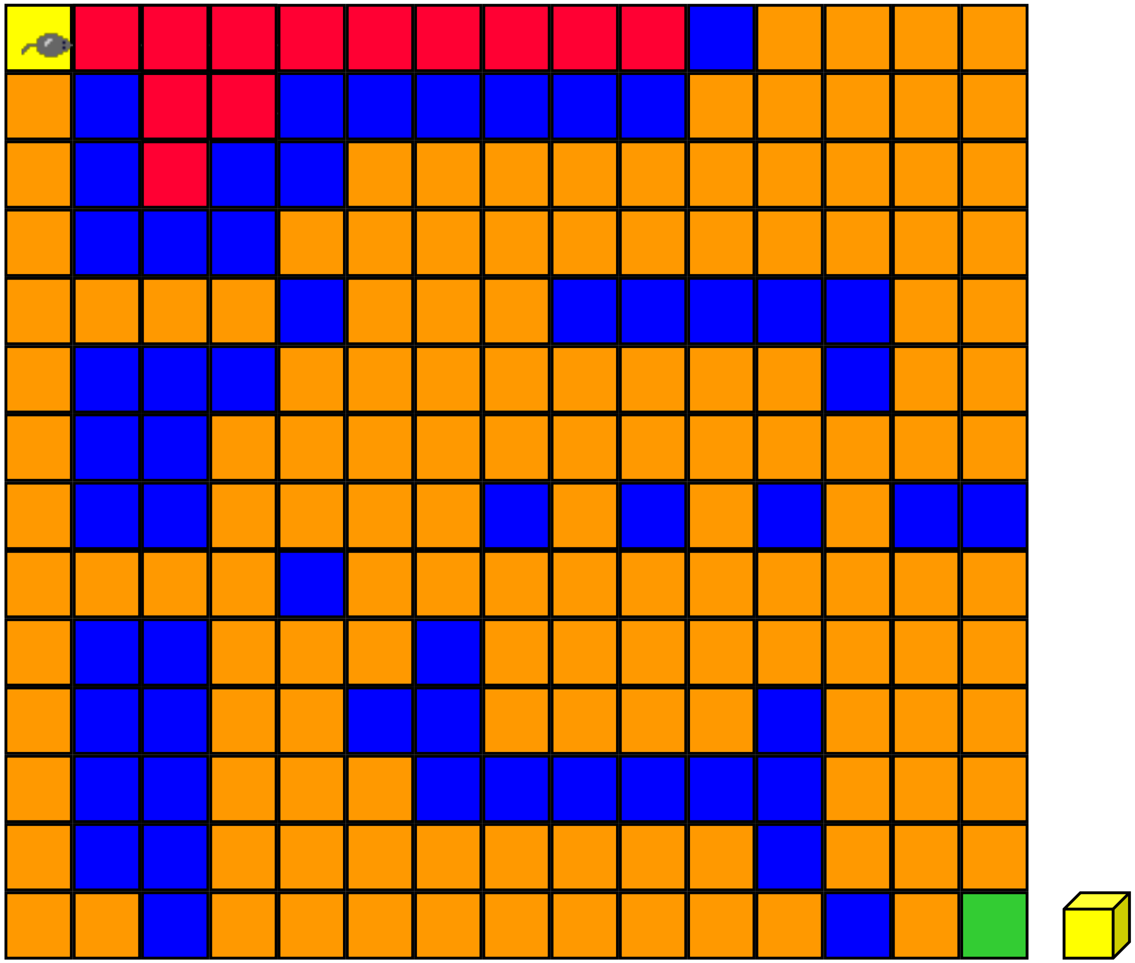
- Move down.

Rat In A Maze



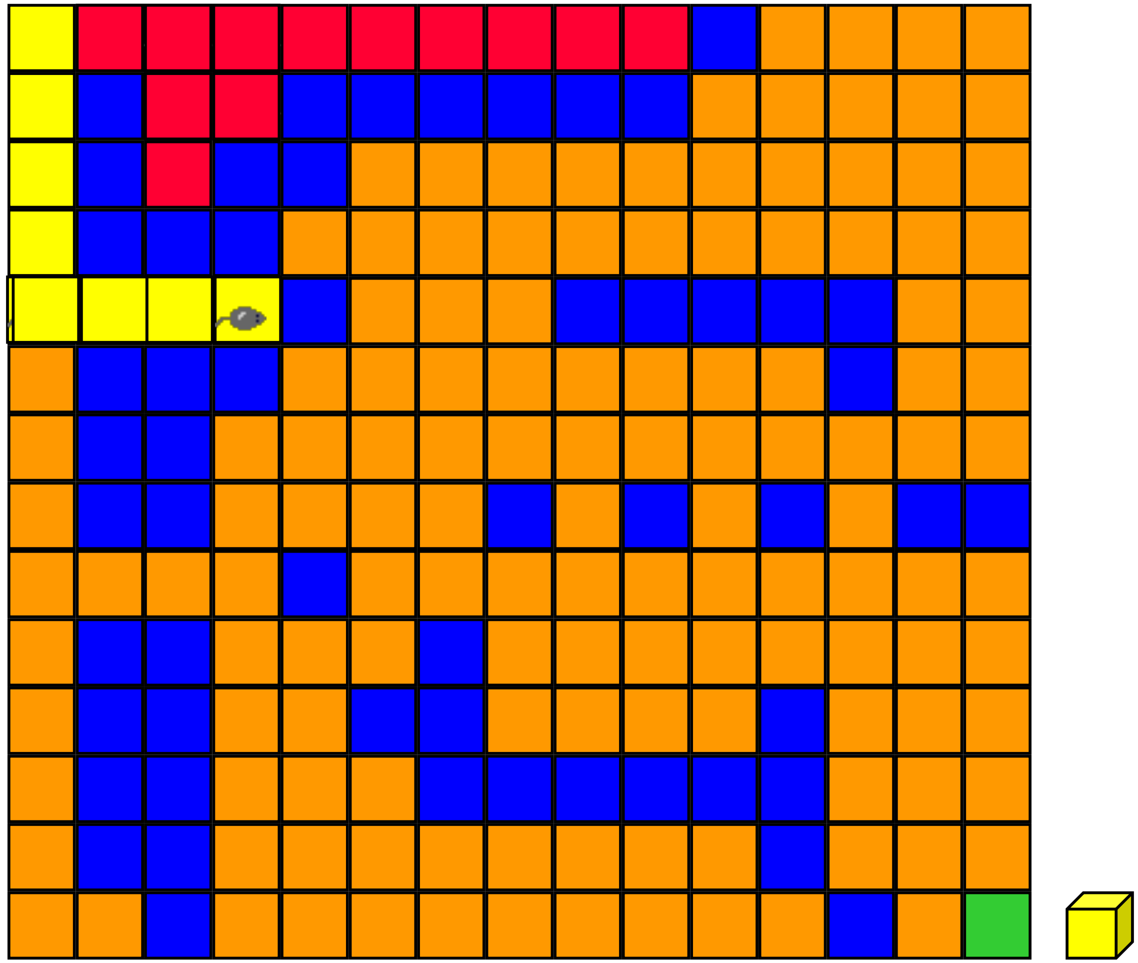
- Move backward until we reach a square from which a forward move is possible.

Rat In A Maze



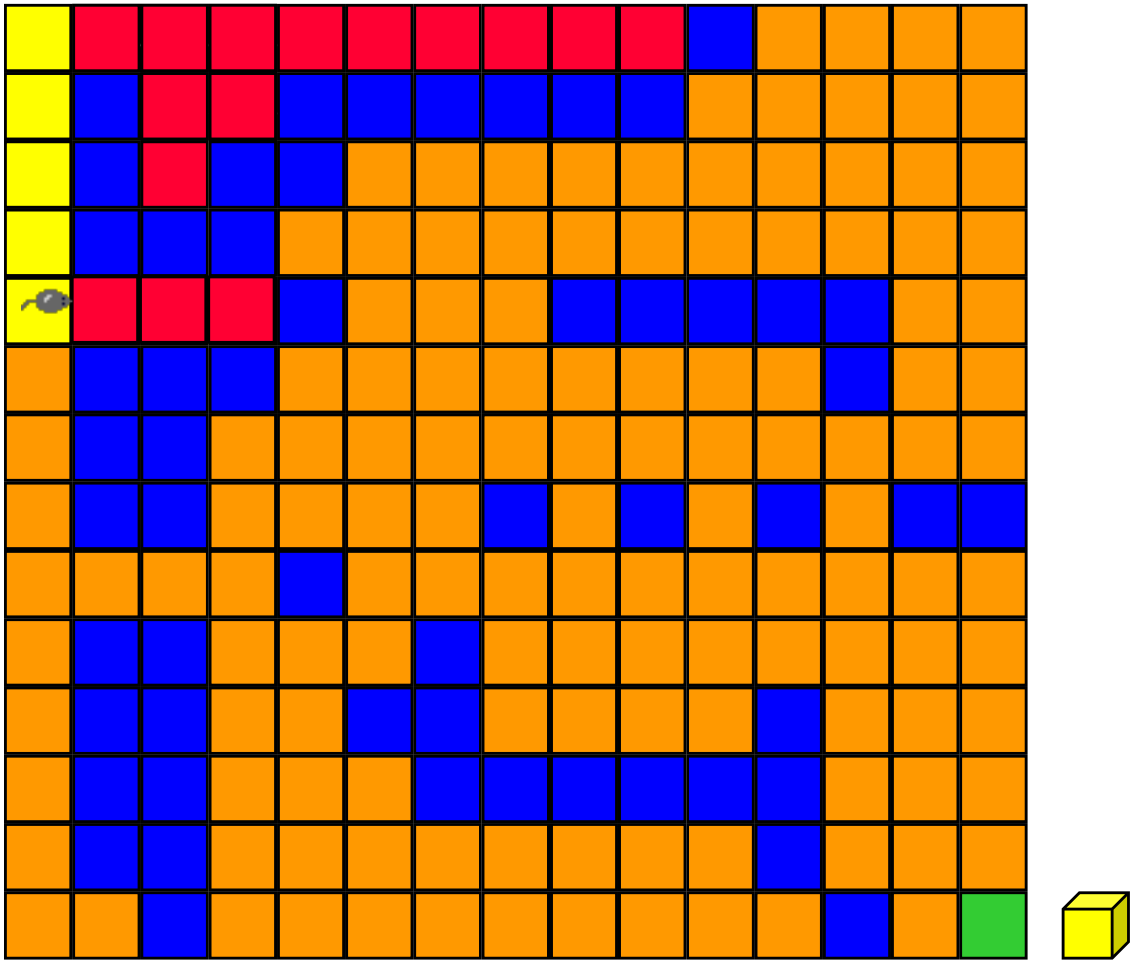
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

Rat In A Maze



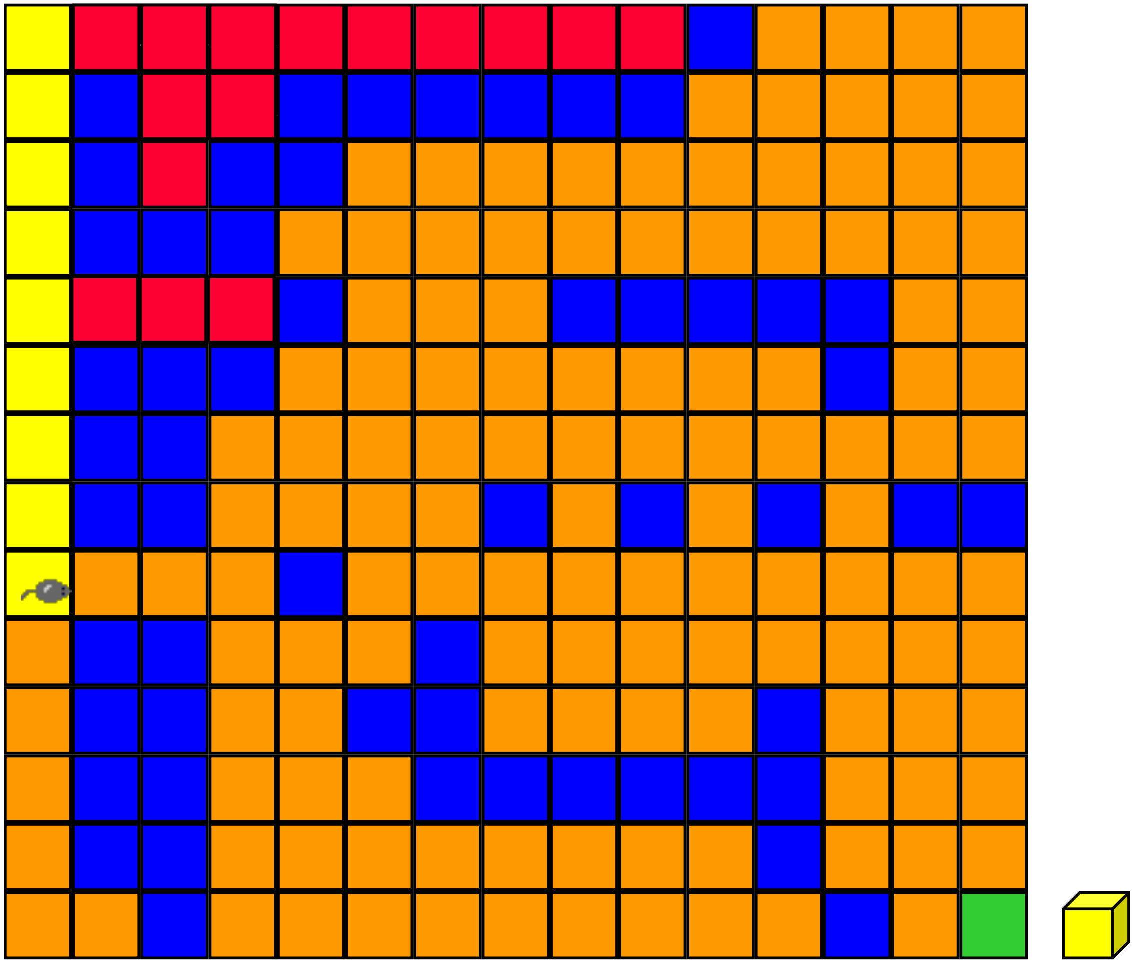
- Move right.
- Backtrack.

Rat In A Maze



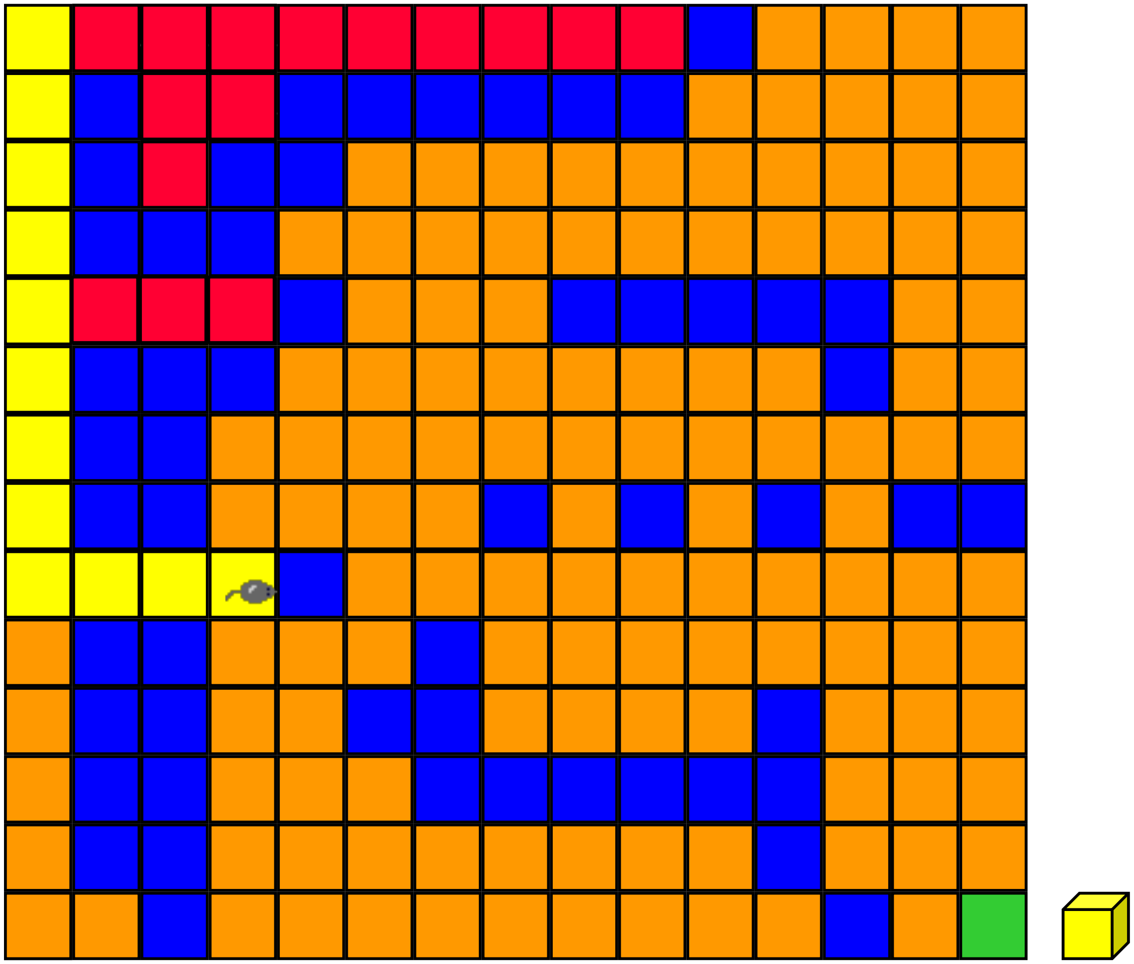
- Move downward.

Rat In A Maze



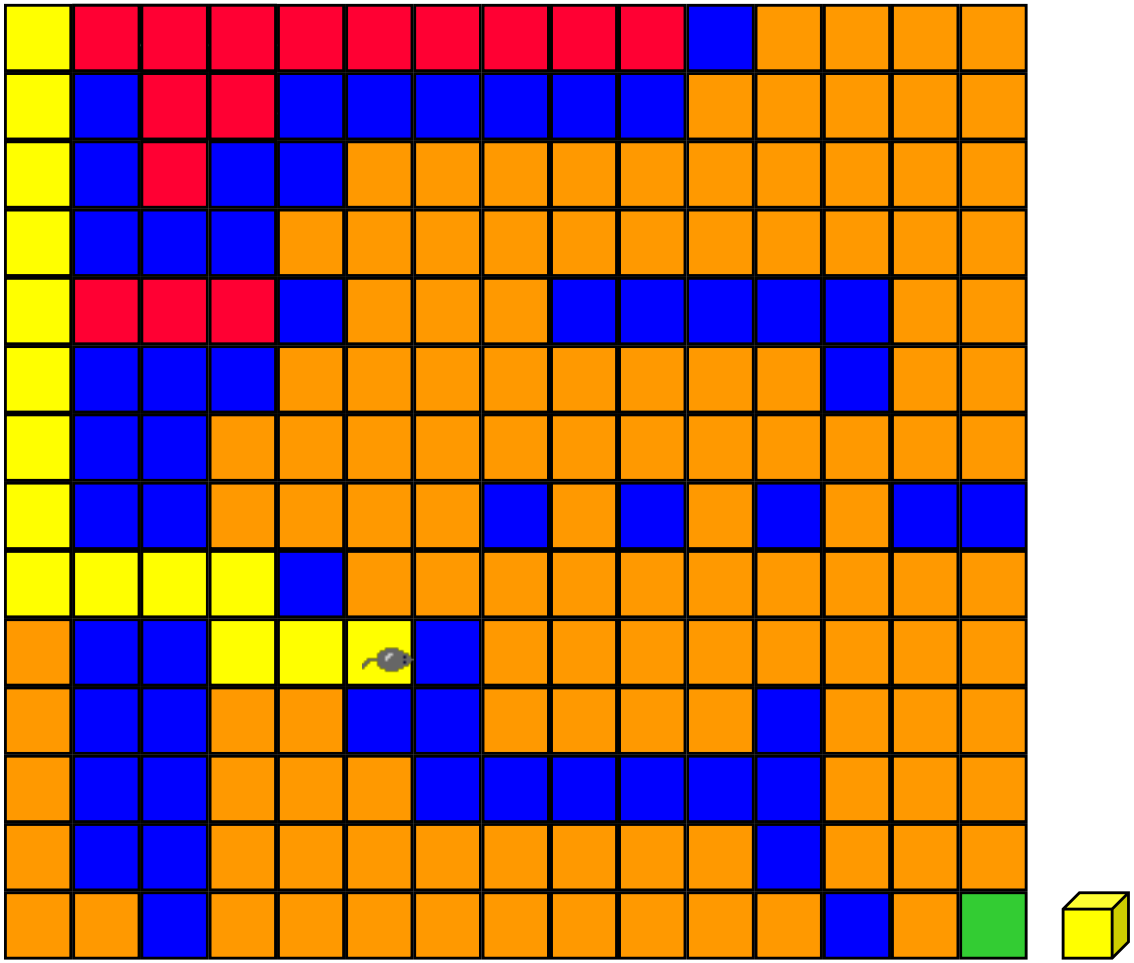
- Move right.

Rat In A Maze



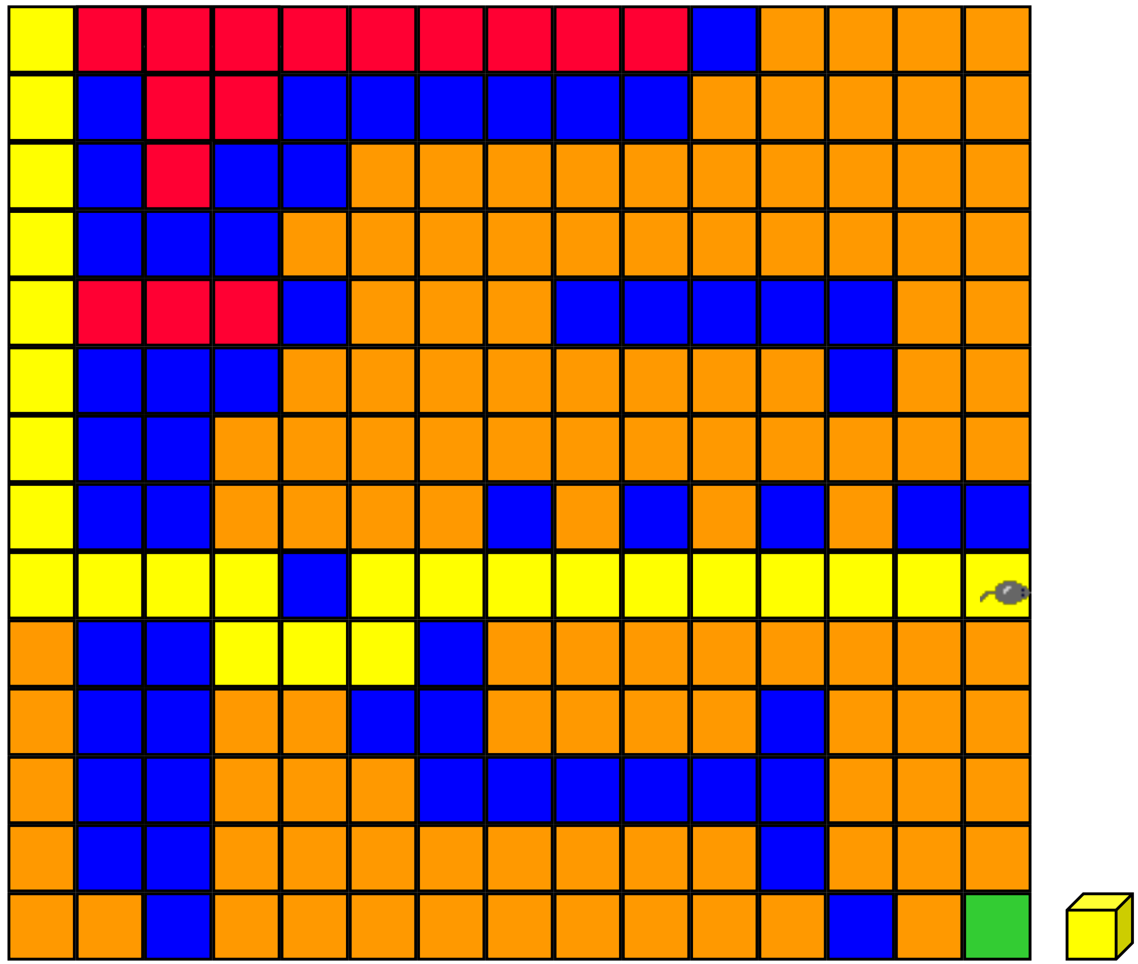
- Move one down and then right.

Rat In A Maze



- Move one up and then right.

Rat In A Maze



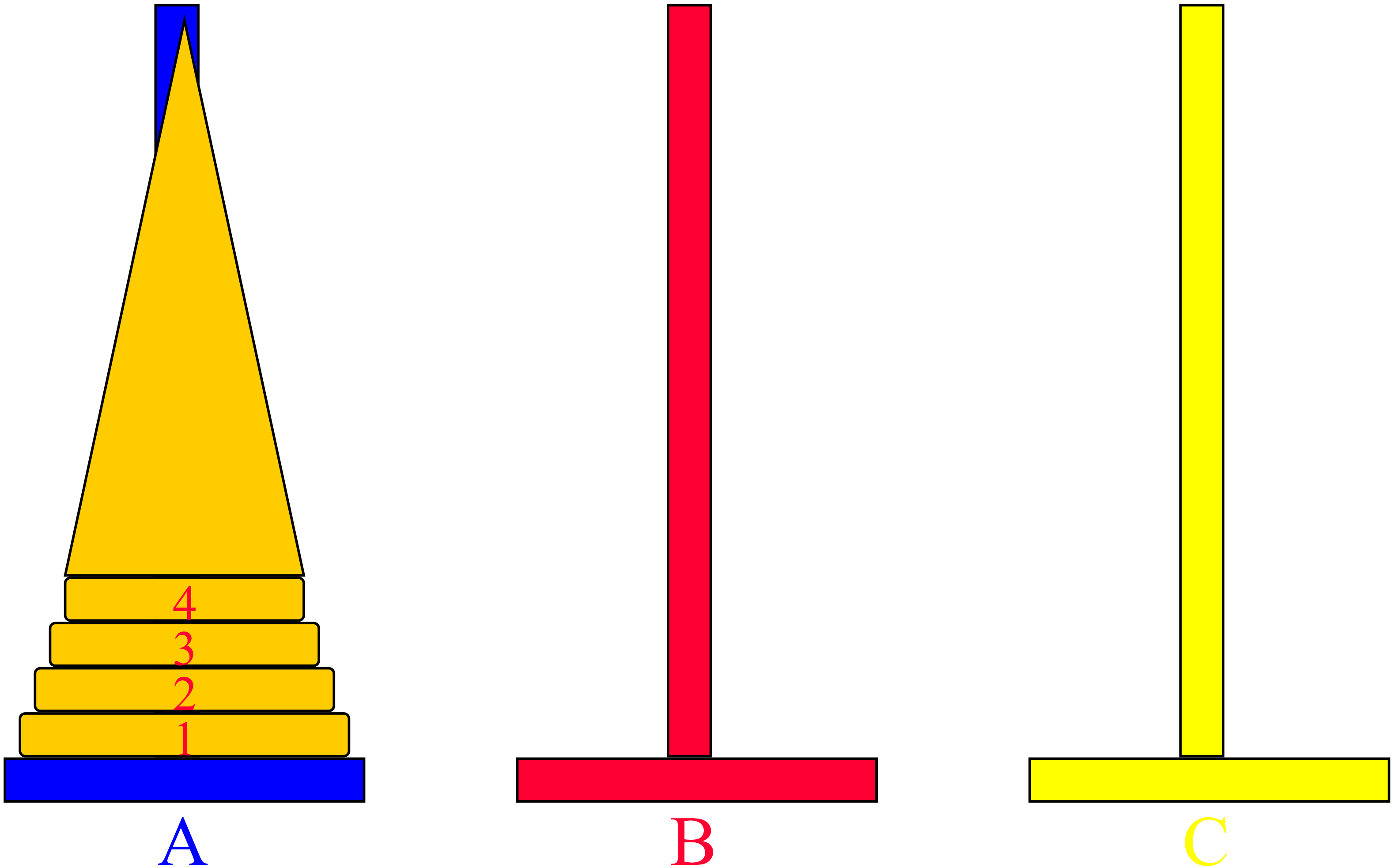
- Move down to exit and eat cheese.
- Path from maze entry to current position operates as a stack.

Method Invocation and Return

```
public void a()
{ …; b(); …}
public void b()
{ …; c(); …}
public void c()
{ …; d(); …}
public void d()
{ …; e(); …}
public void e()
{ …; c(); …}
```

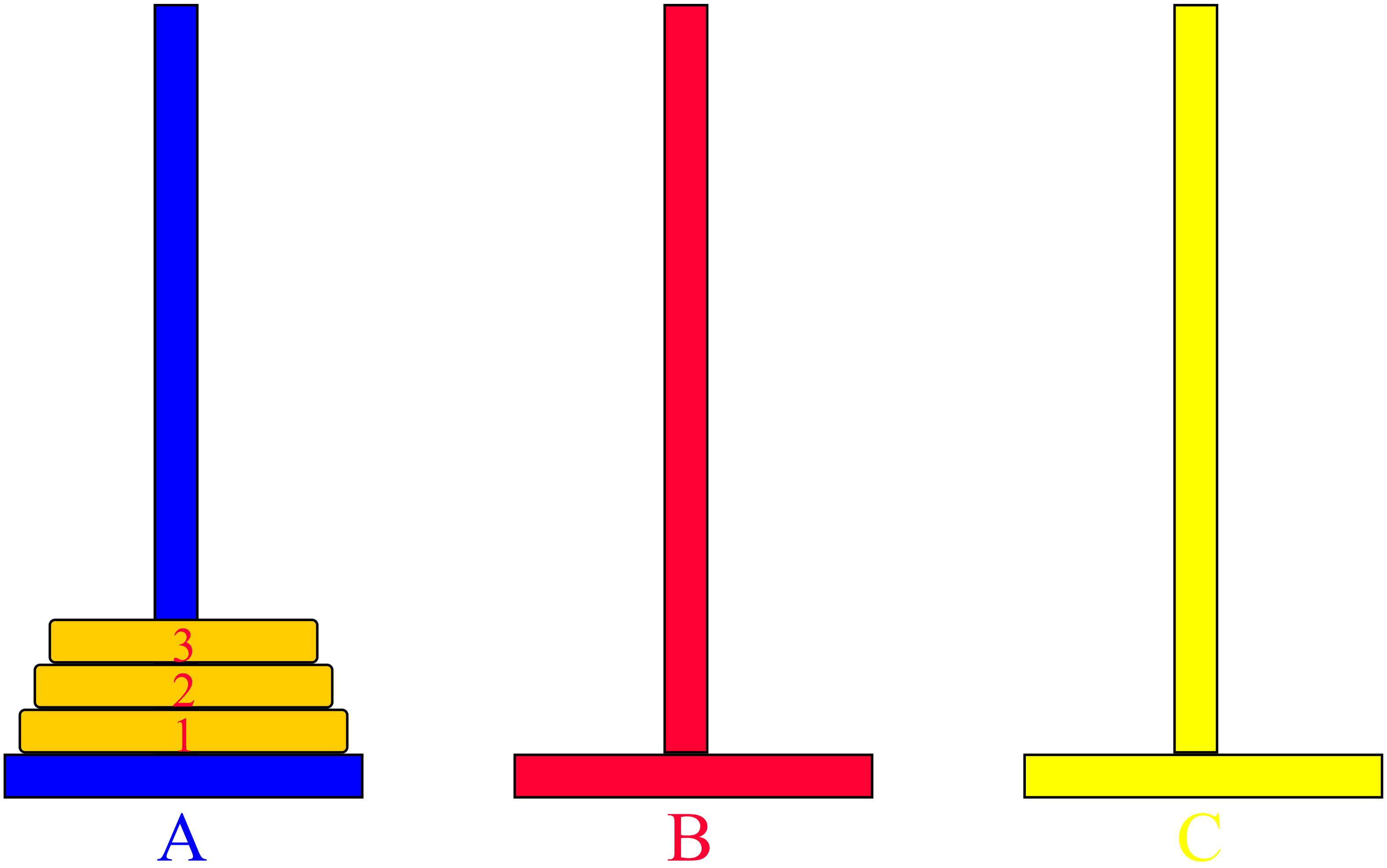
return address in d()
return address in c()
return address in e()
return address in d()
return address in c()
return address in b()
return address in a()

Towers Of Hanoi/Brahma



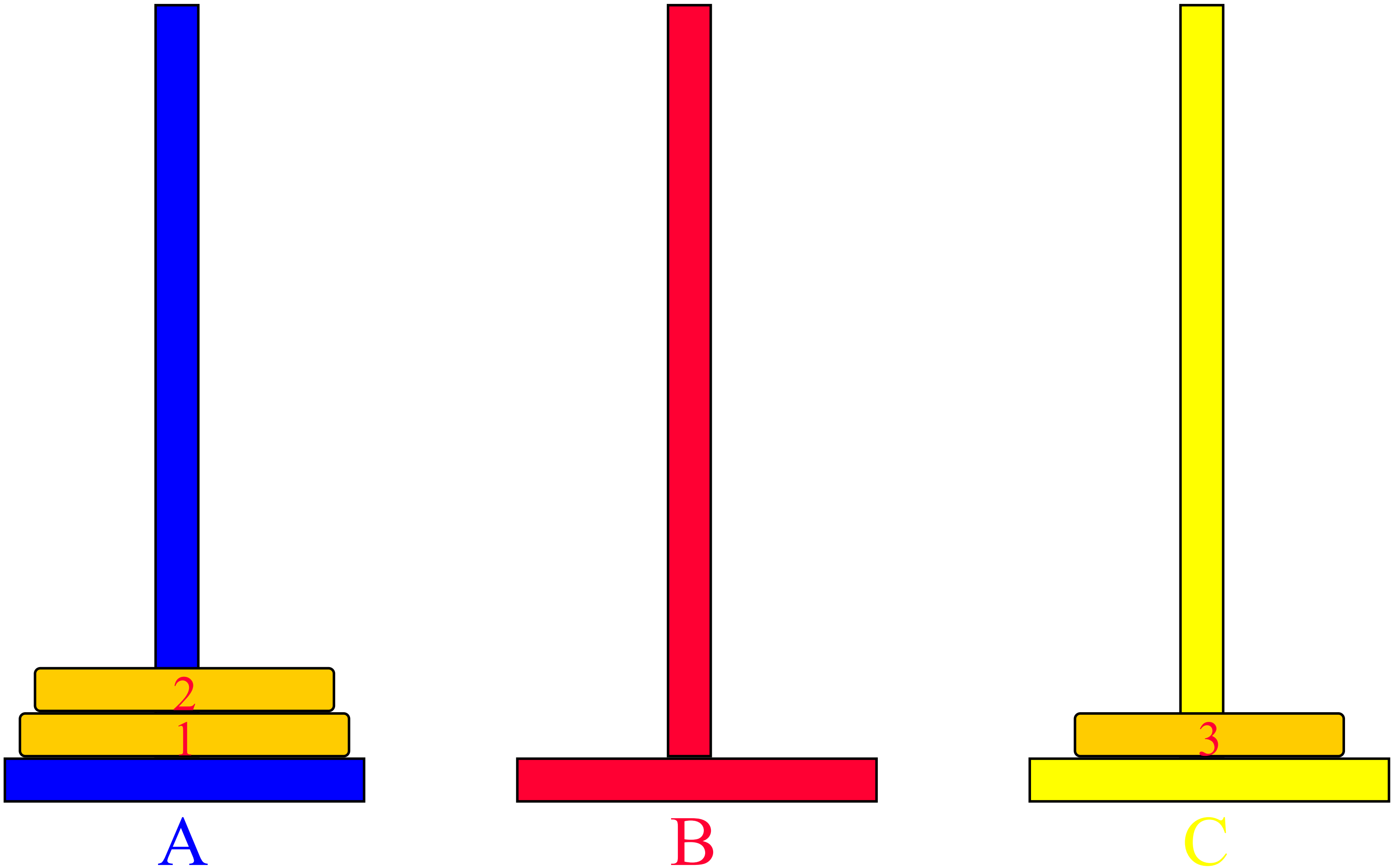
- 64 gold disks to be moved from tower A to tower C
- each tower operates as a stack
- cannot place big disk on top of a smaller one

Towers Of Hanoi/Brahma



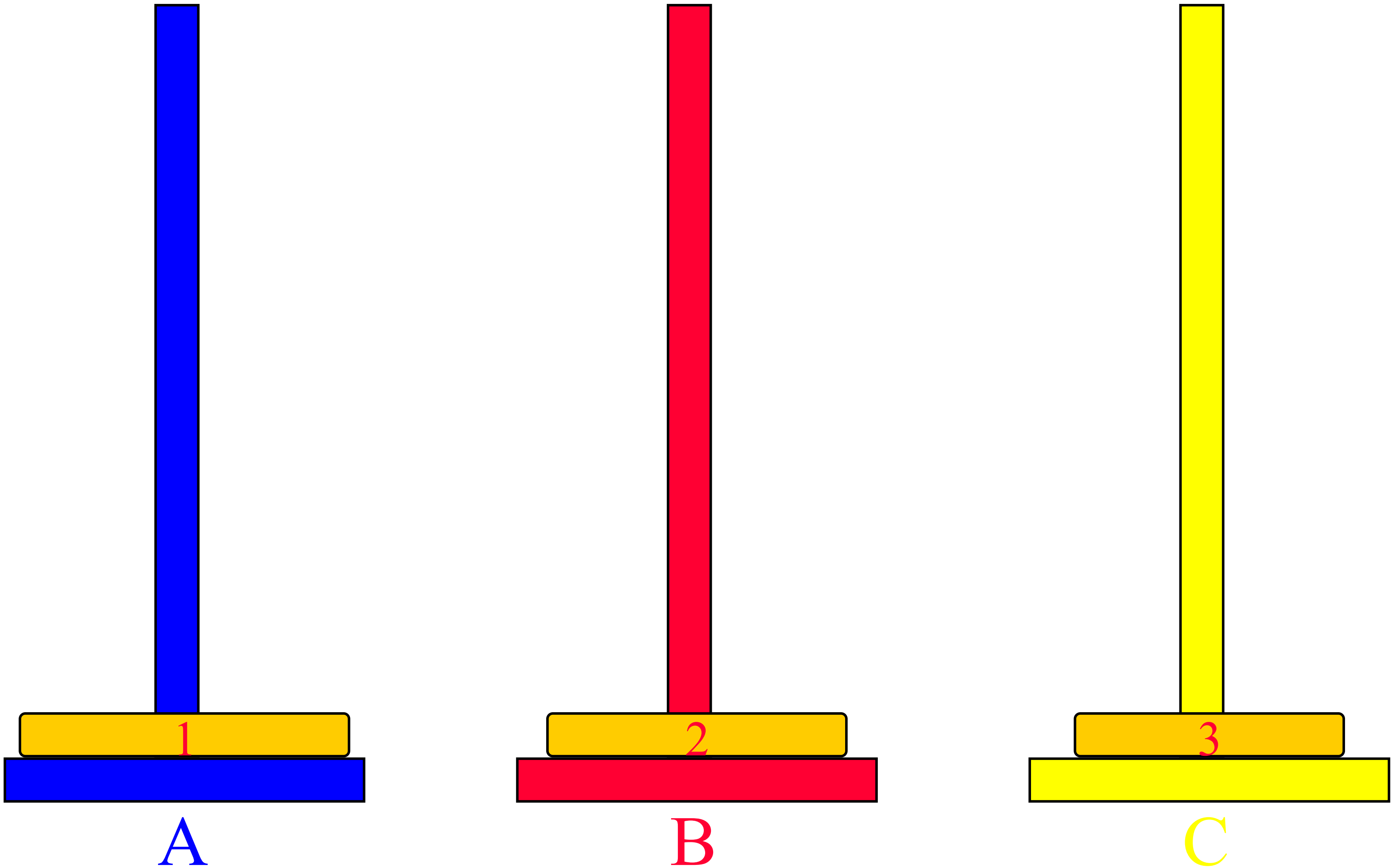
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



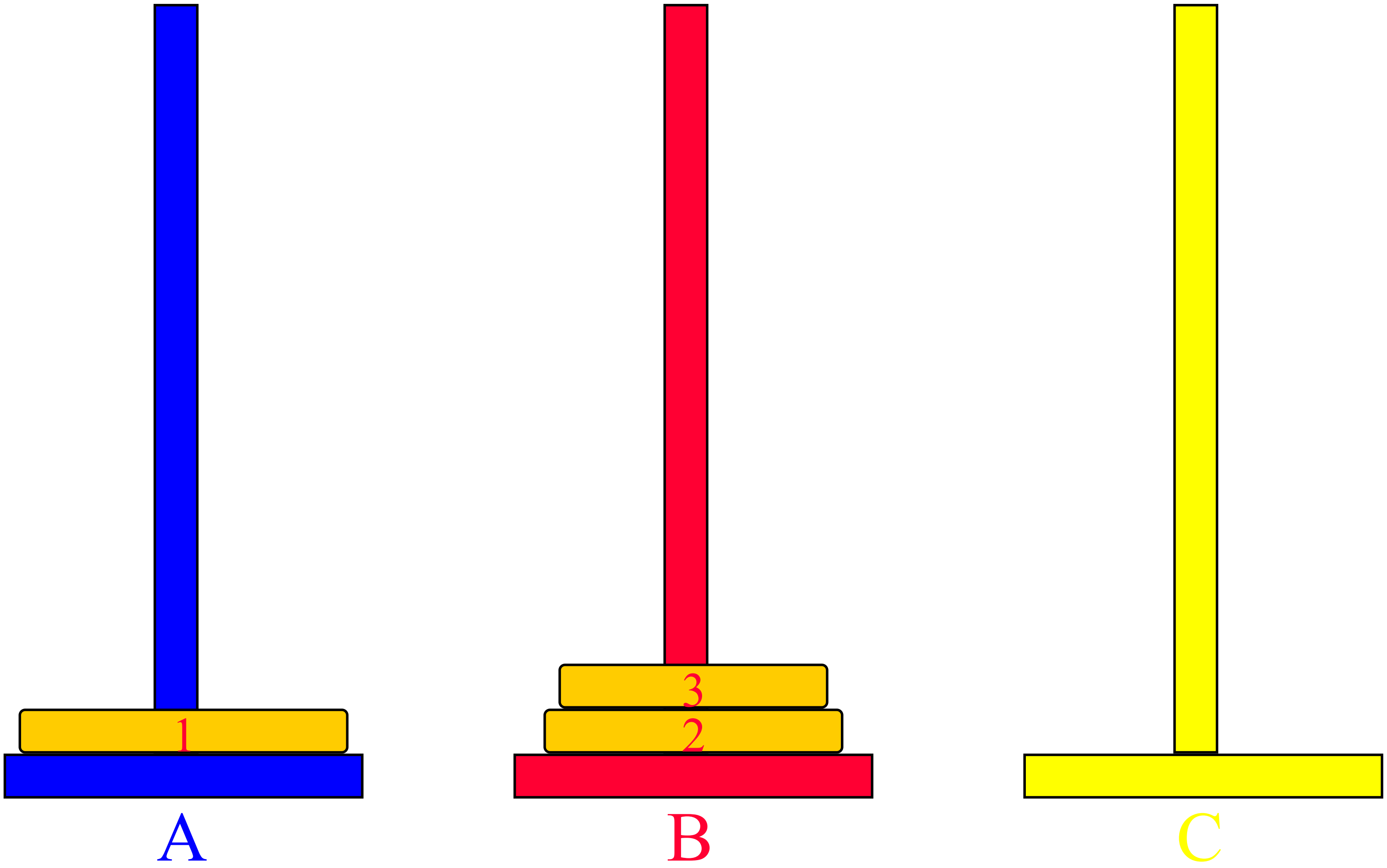
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



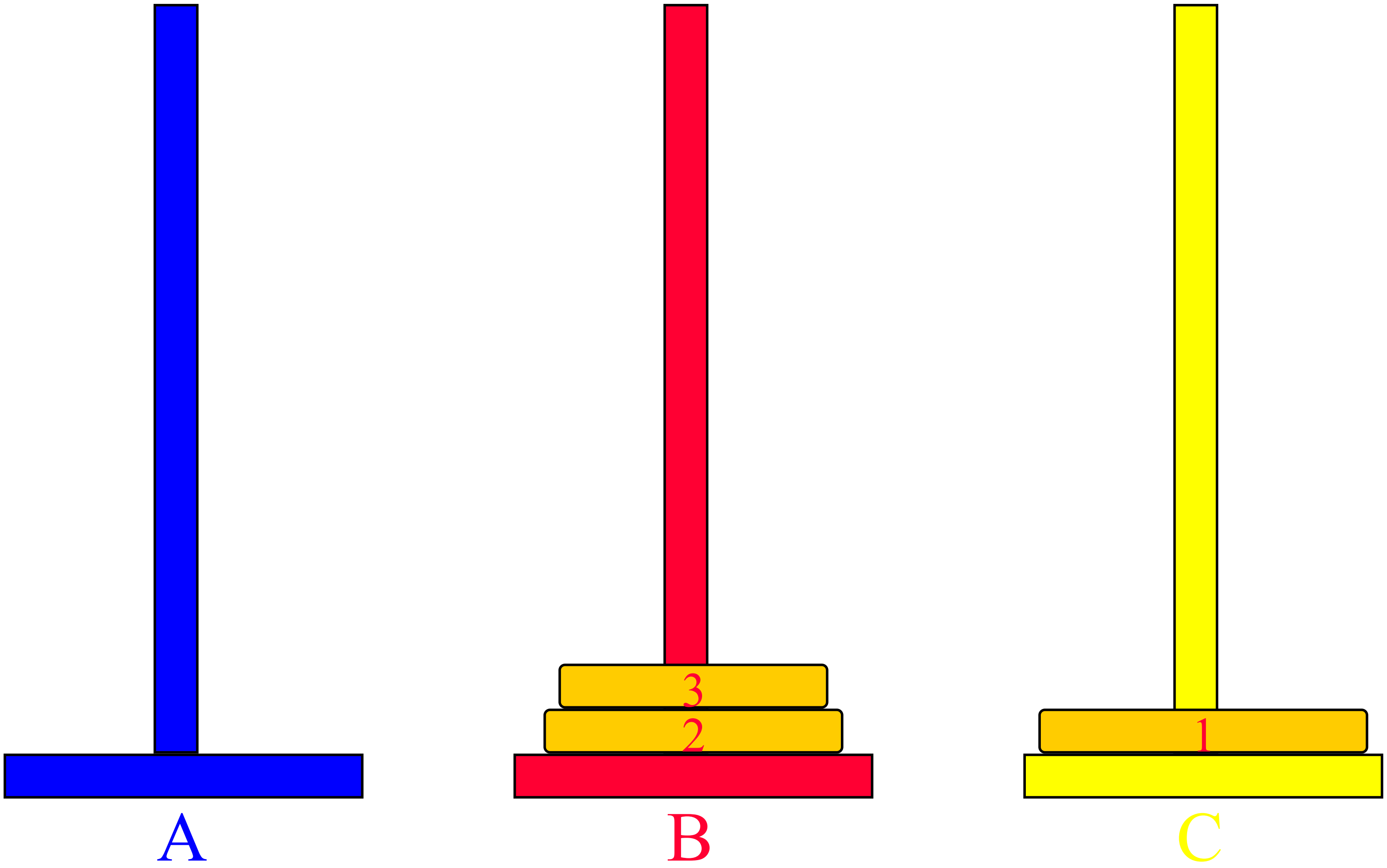
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



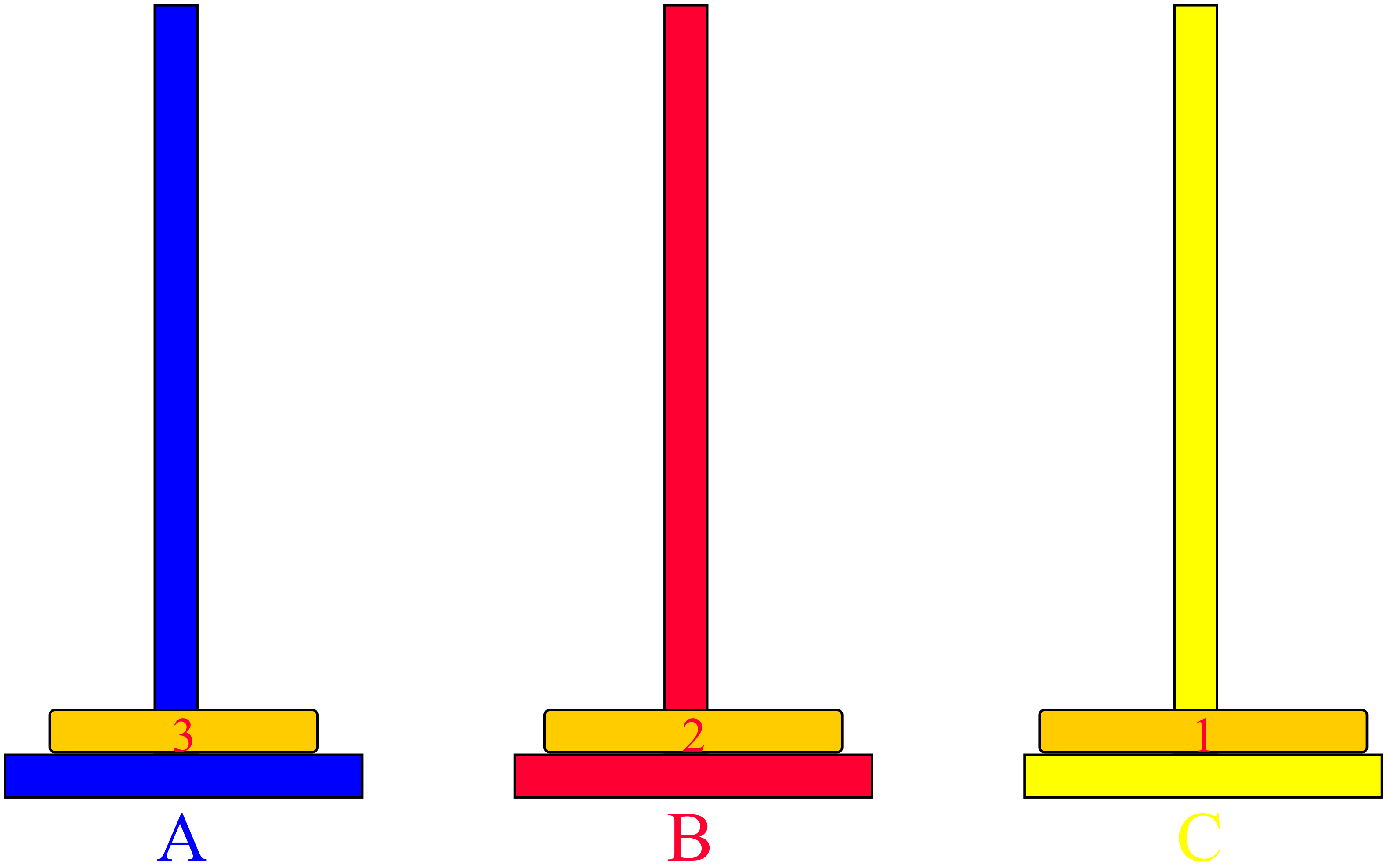
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



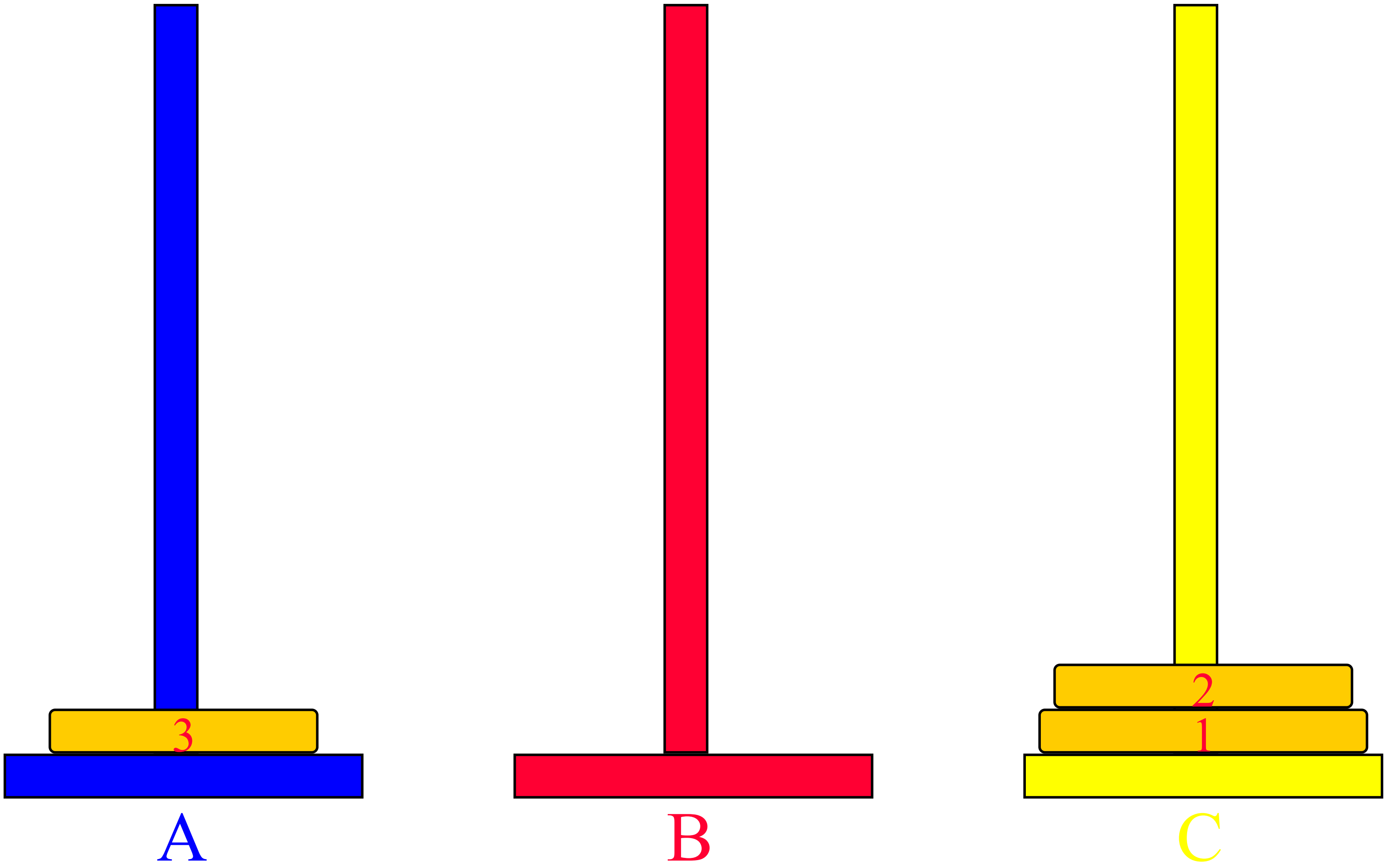
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



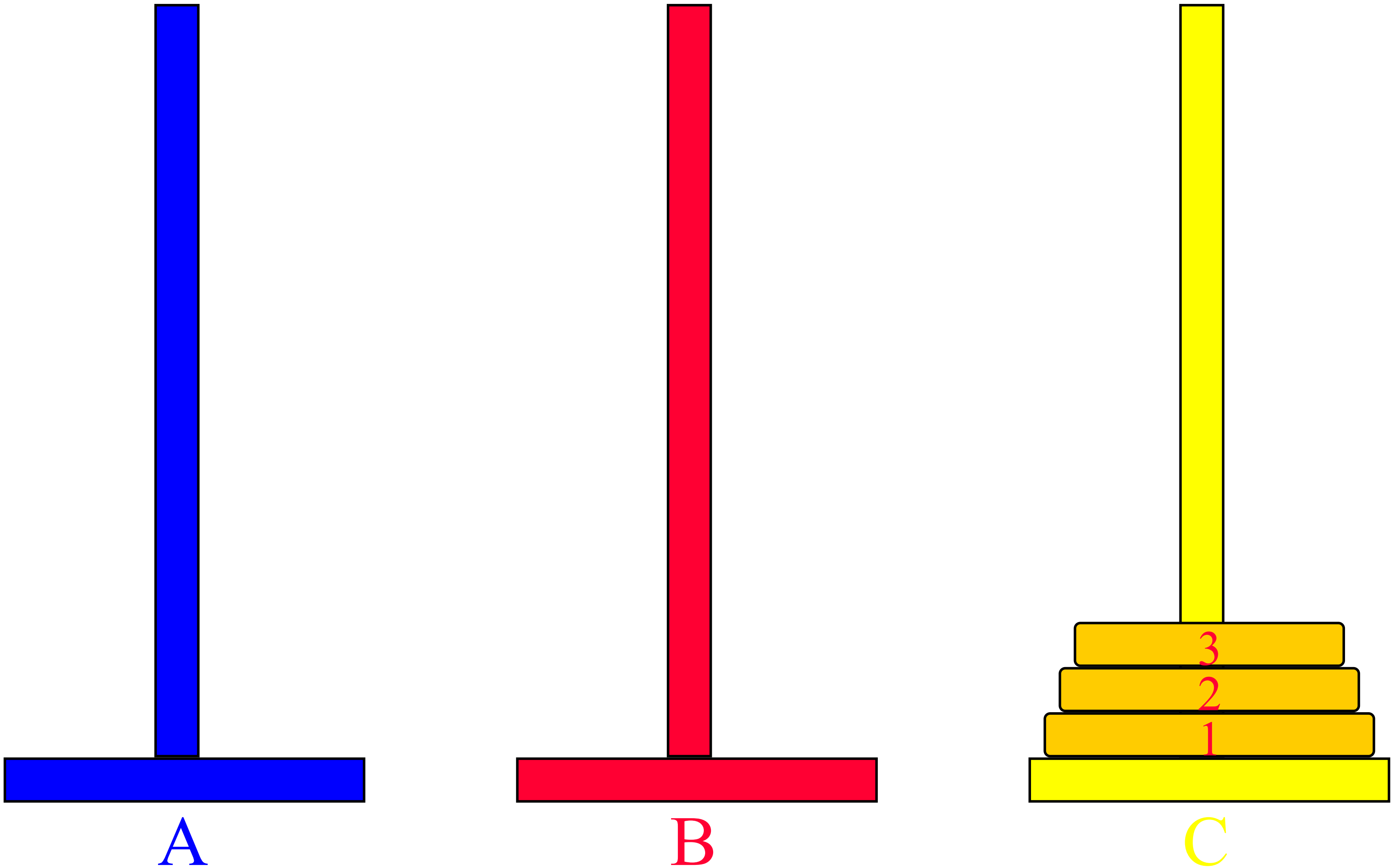
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



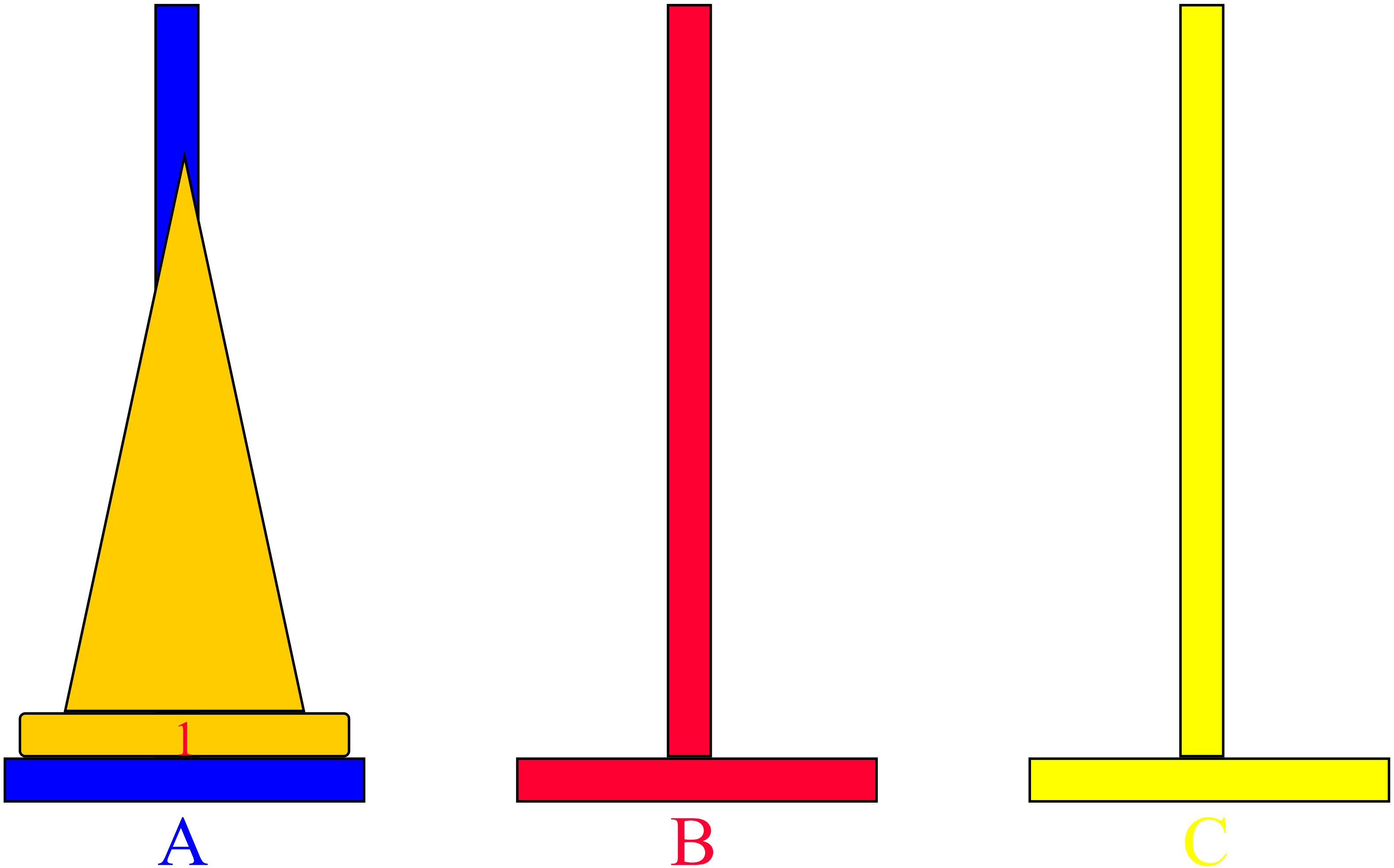
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi/Brahma



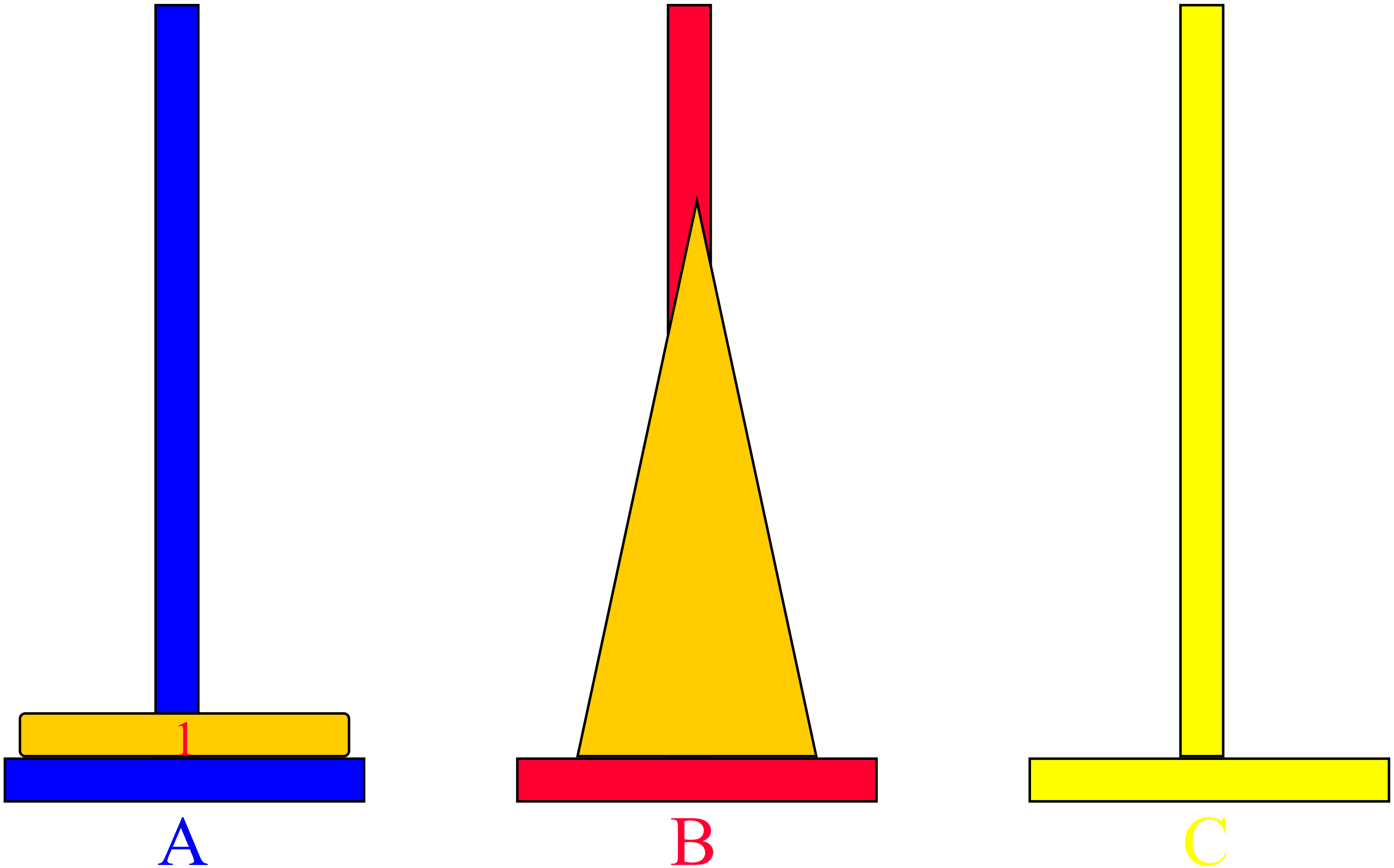
- 3-disk Towers Of Hanoi/Brahma
- 7 disk moves

Recursive Solution



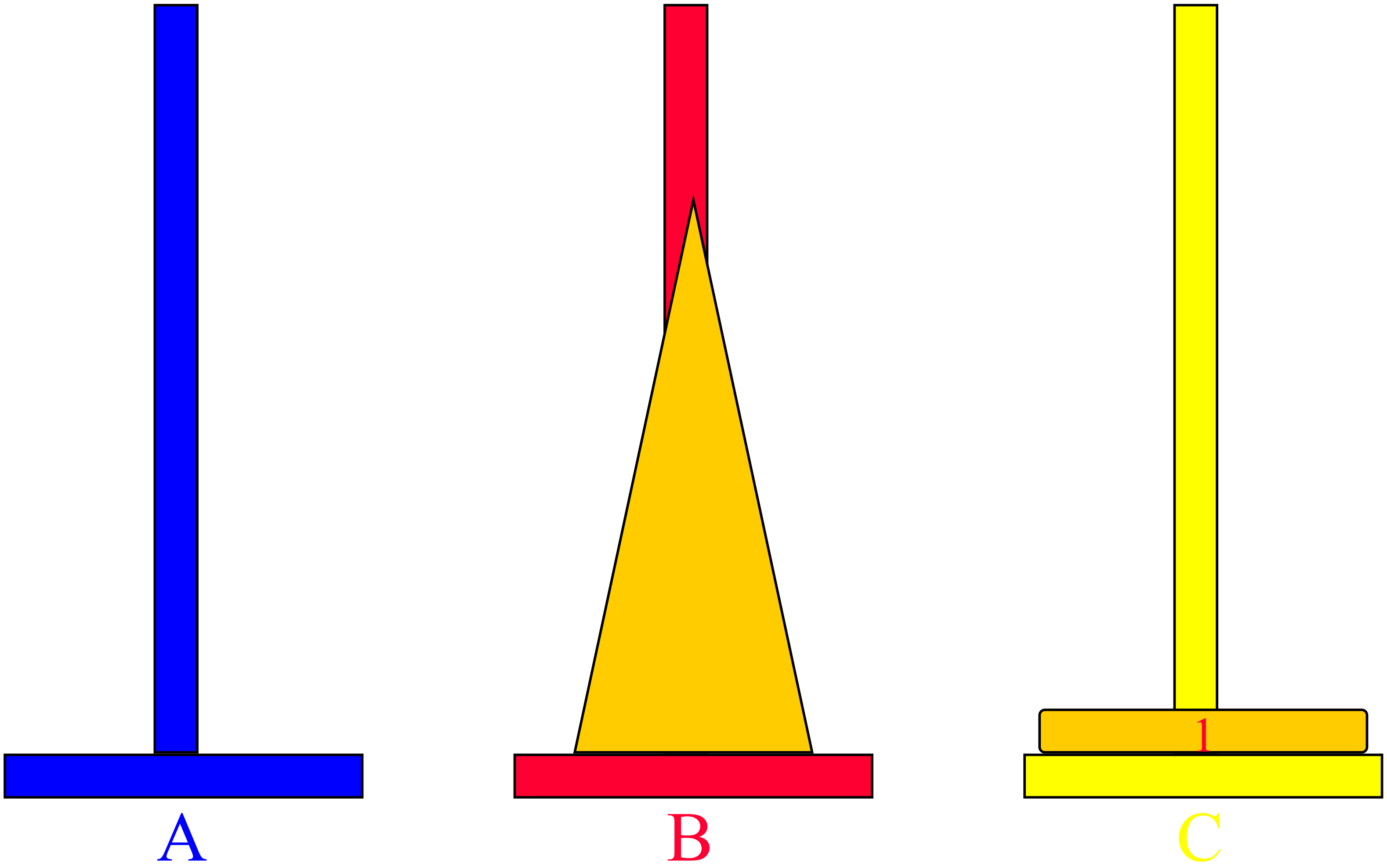
- n > 0 gold disks to be moved from A to C using B
- move top n-1 disks from A to B using C

Recursive Solution



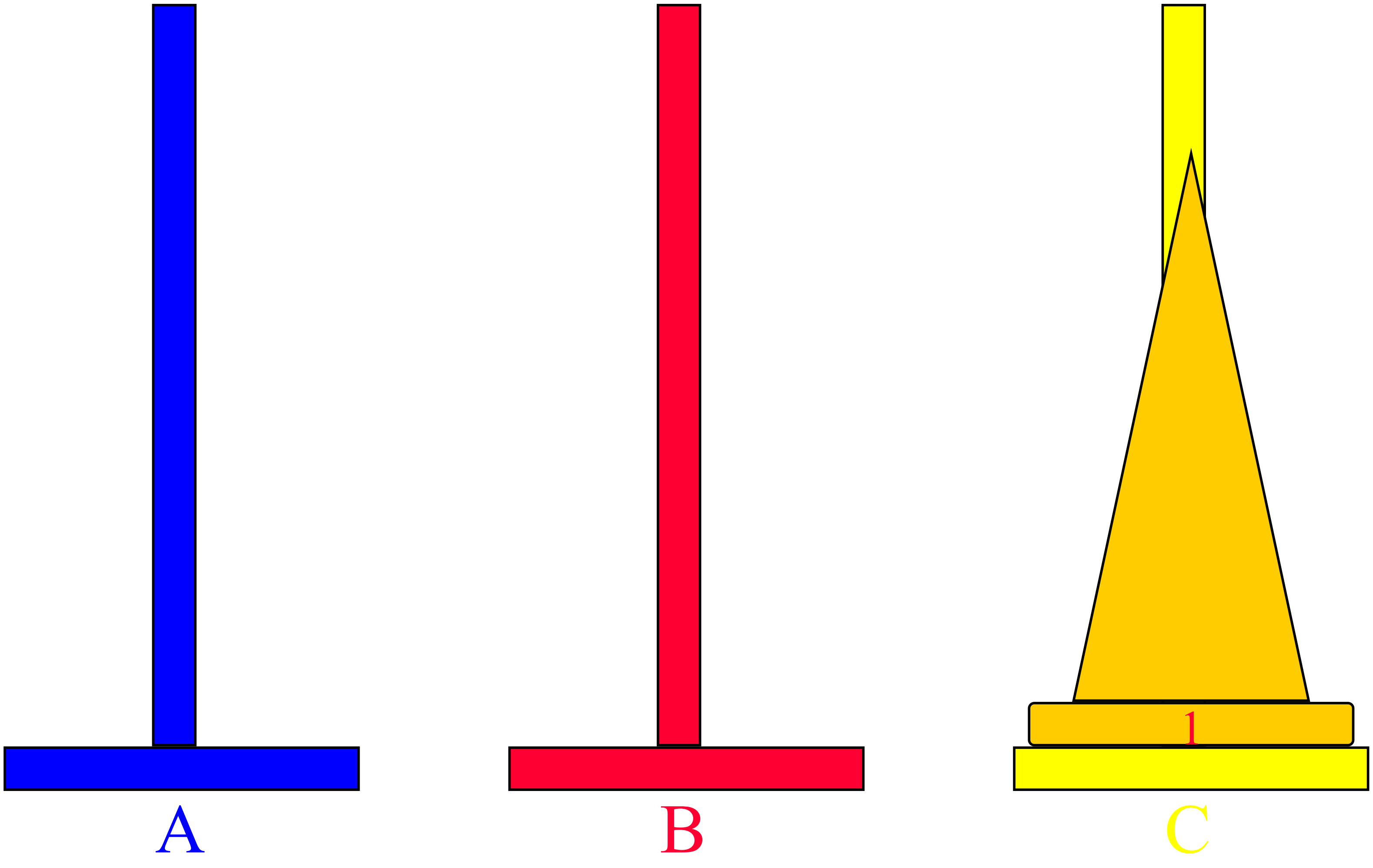
- move top disk from A to C

Recursive Solution



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

Recursive Solution



- moves(n) = 0 when n = 0
- moves(n) = 2*moves(n-1) + 1 = 2^n-1 when n > 0

Code

void main() { moves(n, 'A', 'C', 'B'); }

- Void moves(int n, char A, char C, char B)
- {
-   if (n == 1)
-   {
-     printf(“A → C”); return;
-   }
-   moves(n - 1, A, B, C); // move top n-1 from A to B
-   printf(“A → C”); // move the remaining 1 from A to C
-   moves(n - 1, B, C, A); // move all n-1 from B to C
- }

Example (n=1, n=2)

N=1: moves(1, A, C, B)

A → C;   Done!

N=2: moves(2, A, C, B)

moves(1, A, B, C);

A → C;

moves (1, B, C, A);

Example (n=1, n=2)

N=1: moves(1, A, C, B)

A → C; Done!

N=2: moves(2, A, C, B)

moves(1, A, B, C); A → B;

A → C; A → C;

moves (1, B, C, A); B → C;

Example (n=2, n=3)

n=2: moves(2, A, C, B)

A → B

A → C

B → C

N=3: moves(3, A, C, B)

moves(2, A, B, C);

A → C;

moves (2, B, C, A);

Example

n=2: moves(2, A, C, B)

A → B

A → C

B → C

N=3: moves(3, A, C, B)

	A → C
moves(2, A, B, C);	A → B
	C → B
A → C;	A → C
moves (2, B, C, A);	moves (2, B, C, A);

Example

n=2: moves(2, A, C, B)

A → B

A → C

B → C

N=3: moves(3, A, C, B)

	A → C	A → C
moves(2, A, B, C);	A → B	A → B
	C → B	C → B
A → C;	A → C	A → C
		B → A
moves (2, B, C, A);	moves (2, B, C, A);	B → C
		A → C

Towers Of Hanoi/Brahma

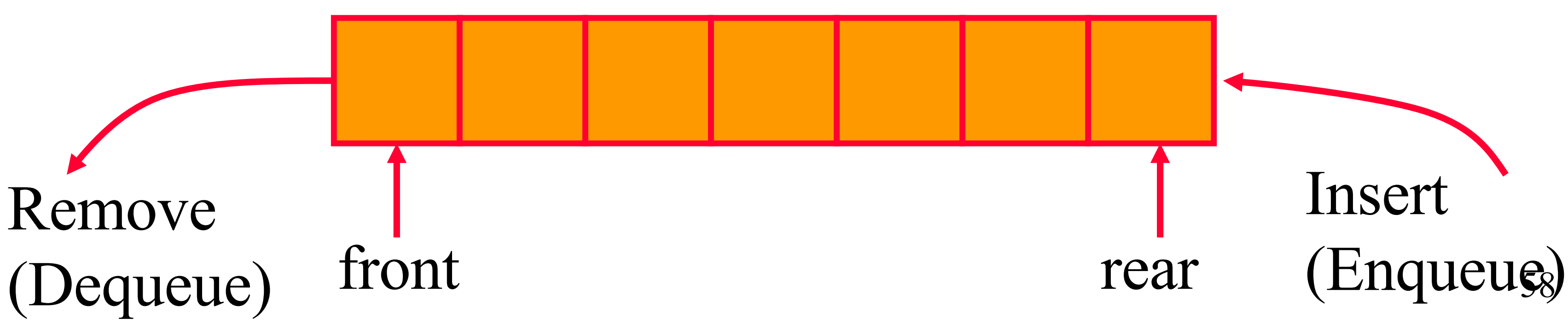
- moves(64) = $1.8 * 10^{19}$ (approximately)
- Performing 10^9 moves/second, a computer would take about 570 years to complete.
- At 1 disk move/min, the monks will take about $3.4 * 10^{13}$ years.

Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.
- Like bus stop queue, ticket counter queue.
- First In, First Out (FIFO)

Enqueue and Dequeue

- Primary queue operations: Enqueue and Dequeue
- Like check-out lines in a store, a queue has a front and a rear.
- Enqueue
  - Insert an element at the rear of the queue
- Dequeue
  - Remove an element from the front of the queue

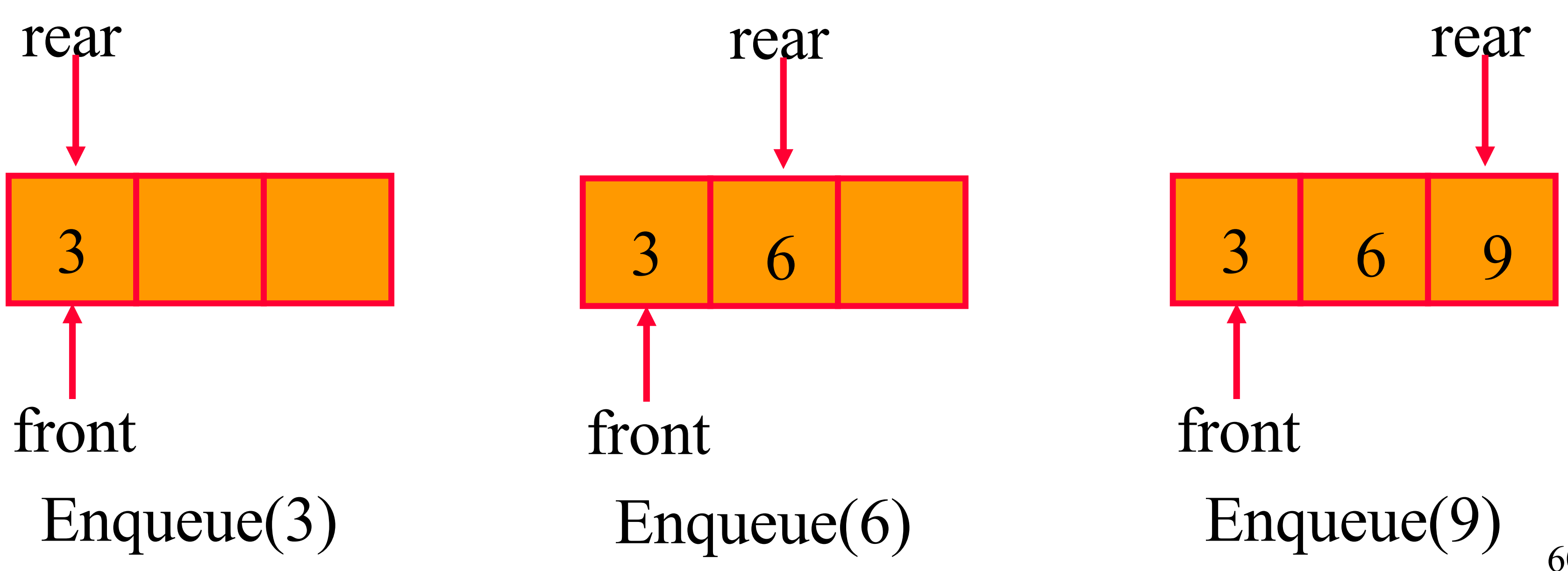


Implementation of Queue

- Just as stacks can be implemented as arrays or linked lists, so with queues.
- Dynamic queues have the same advantages over static queues as dynamic stacks have over static stacks

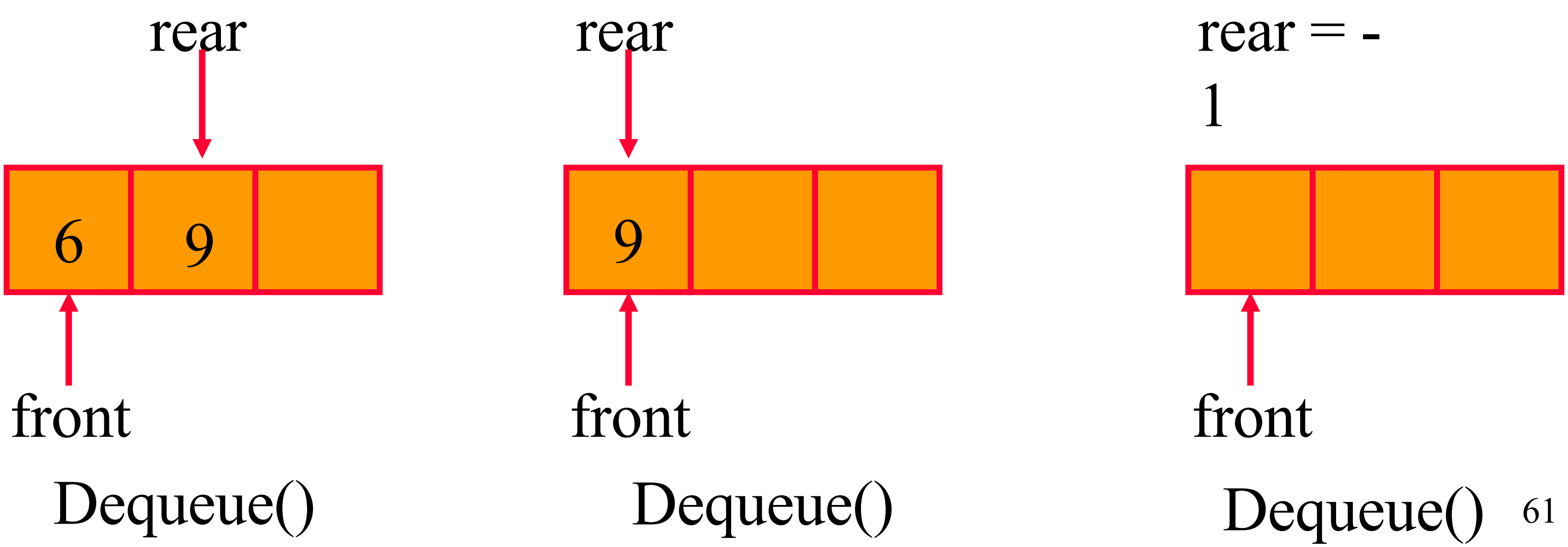
Queue Implementation of Array

- There are several different algorithms to implement Enqueue and Dequeue
- Naïve way
  - When enqueuing, the <u>front index</u> is always fixed and the <u>rear index</u> moves forward in the array.



Queue Implementation of Array

- Naïve way
  - When enqueuing, the front index is always fixed and the rear index moves forward in the array.
  - When dequeuing, the element at the front the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Queue Implementation of Array

- Better way
  - When an item is enqueued, make the <u>rear index</u> move forward.
  - When an item is dequeued, the <u>front index</u> moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed).

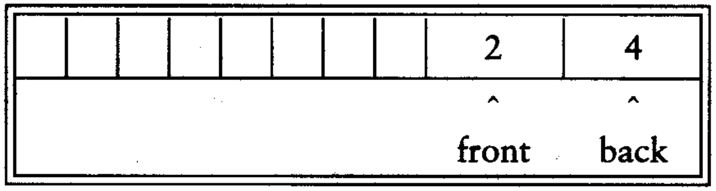
(front) XXXXOOOOO (rear) [X: inserted item, O: empty position]
OXXXXOOOO (after 1 dequeue, and 1 enqueue)
OOXXXXXOO (after another dequeue, and 2 enqueues)
OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

The problem here is that the rear index cannot move beyond the last element in the array.

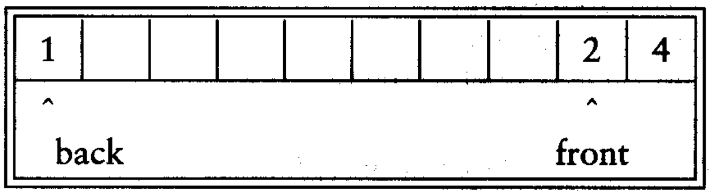
Implementation using Circular Array

- Using a circular array
- When an element moves past the end of a circular array, it wraps around to the beginning, e.g.
  - OOOOO7963 → 4OOOO7963 (after Enqueue(4))
  - After Enqueue(4), the rear index moves from 3 to 4.

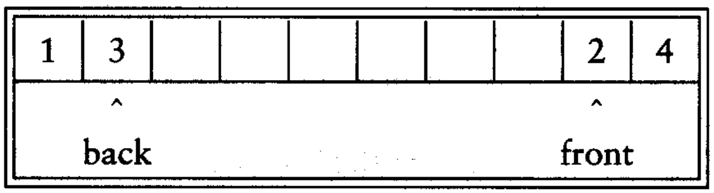
Initial State



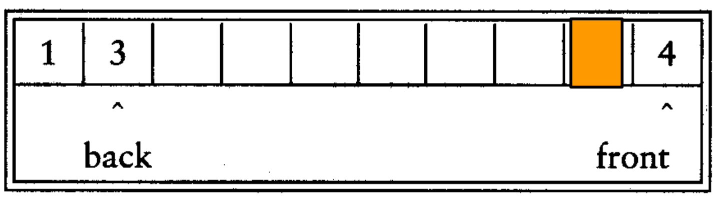
After enqueue(1)



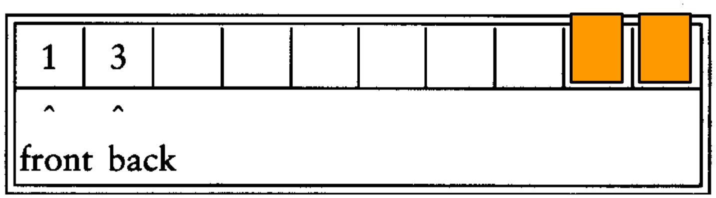
After enqueue(3)



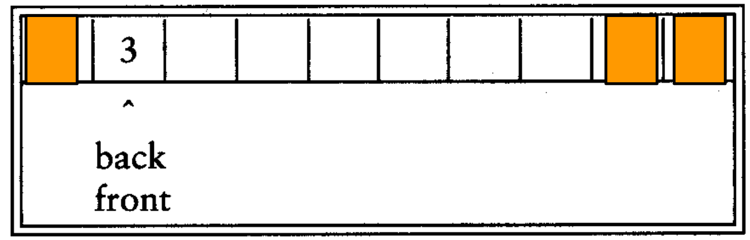
After dequeue, Which Returns 2



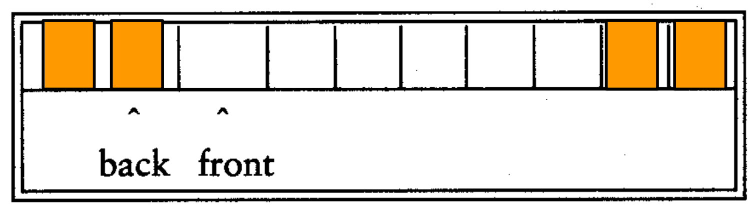
After dequeue, Which Returns 4



After dequeue, Which Returns 1



After dequeue, Which Returns 3
and Makes the Queue Empty



Enqueue

```
bool Queue::Enqueue(double x) {
      if (IsFull()) {
            cout << "Error: the queue is full." << endl;
            return false;
      }
      else {
            // calculate the new rear position (circular)
            rear              = (rear + 1) % maxSize;
            // insert new item
            values[rear]      = x;
            // update counter
            counter++;
            return true;
      }
}
```

Dequeue

```
bool Queue::Dequeue(double & x) {
      if (IsEmpty()) {
            cout << "Error: the queue is empty." << endl;
            return false;
      }
      else {
            // retrieve the front item
            x           = values[front];
            // move front
            front = (front + 1) % maxSize;
            // update counter
            counter--;
            return true;
      }
}
```

Practice Problems

1. Display a stack in reverse order using the help of another stack.
2. Implement a Stack using two Queues.
3. Implement a Queue using two Stacks.
4. Describe a stack data structure that supports ‘push’ and ‘pop’ and ‘find minimum’ operations.
5. Write an efficient way to sort the numbers in a stack.

Implement a Stack using two Queues.

```
Class stack {
    queue q1;
    queue q2;

    public:
    void push(int t)
    {
        q1.enqueue(t);
    }
..........
```

Implement a Stack using two Queues.

```
int pop()
  {
    int t;
    while (!q1.empty()) {
      t = q1.front();
      q1.dequeue();
      if (!q1.empty()) q2.enqueue(t);
    }
    while (!q2.empty()) {
      int x = q2.front();
      q2.dequeue();
      q1.enqueue(x);
    }
    return t;
  } ………
```

Implement a Stack using two Queues.

```
bool empty()
    {
        return q1.empty();
    }
};
```

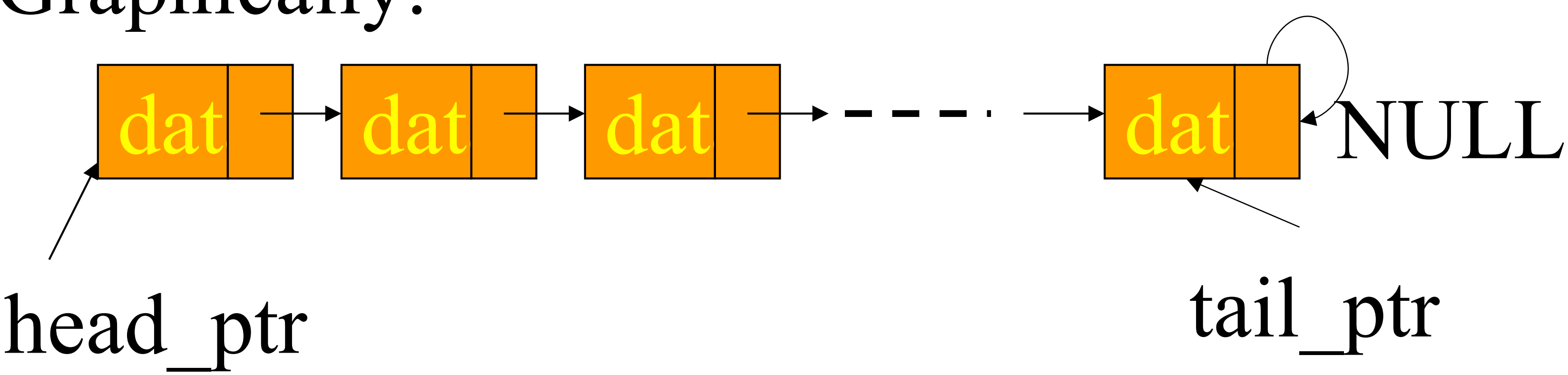
Complexity: Push O(1). Pop O(n).

How can you make Pop in O(1) and Push in O(n?)

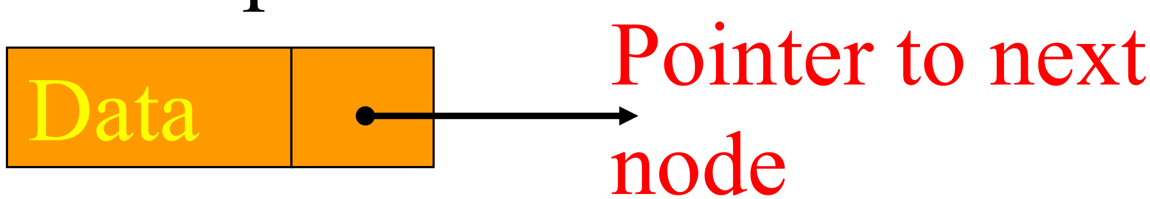
Linked List

Definition of Linked Lists

- A linked list is a sequence of items (objects) where every item is linked to the next.
- Graphically:



- Each node has 2 parts



Definition Details

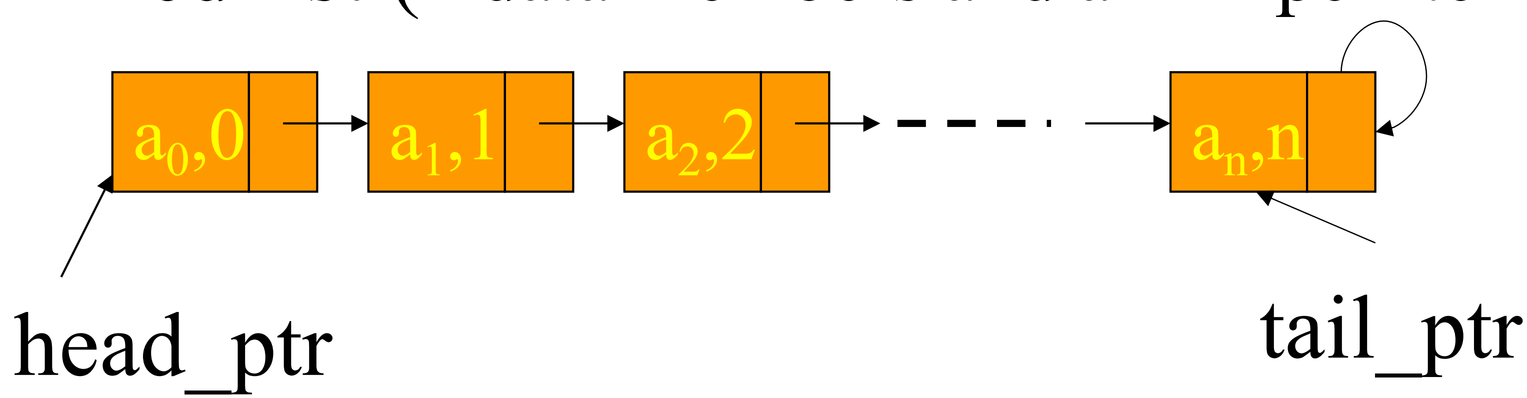
- Each item has a data part (one or more data members), and a link that points to the next item
- One natural way to implement the link is as a pointer; that is, the link is the address of the next item in the list
- It makes good sense to view each item as an object, that is, as an instance of a class.
- We call that class: Node
- The last item does not point to anything. We set its link member to **NULL**. This is denoted graphically by a self-loop.

Advantages of Linked Lists over Arrays and `vectors`

- A linked list can easily grow or shrink in size.
- Insertion and deletion of nodes is quicker with linked lists than with vectors.

Examples of Linked Lists
(A Polynomial)

- A polynomial of degree n is the function $P_n(x)=a_0+a_1x+a_2x^2+\ldots+a_nx^n$. The a_i' s are called the coefficients of the polynomial
- The polynomial can be represented by a linked list (2 data members and a link per item):



Operations on Linked Lists

- **Insert** a new item
  - At the head of the list, or
  - At the tail of the list, or
  - Inside the list, in some designated position
- **Search** for an item in the list
  - The item can be specified by position, or by some value
- **Delete** an item from the list
  - Search for and locate the item, then remove the item, and finally adjust the surrounding pointers
- **size**( );
- **isEmpty**( )

Implementation of search( )

- Node *List::search(**int** x){

```
        Node * currentPtr = getHead( );
        while (currentPtr != NULL){
                if (currentPtr->getData( ) == x)
                        return currentPtr;
                else
                        currentPtr = currentPtr->getNext();
        }
        return NULL;          // Now x is not, so return NULL
};
```

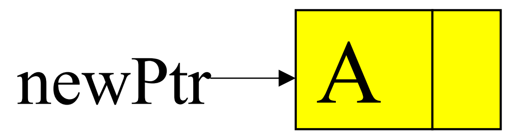
Problem 1: Write a recursive version of search().

Problem 2: Write a recursive version of display().

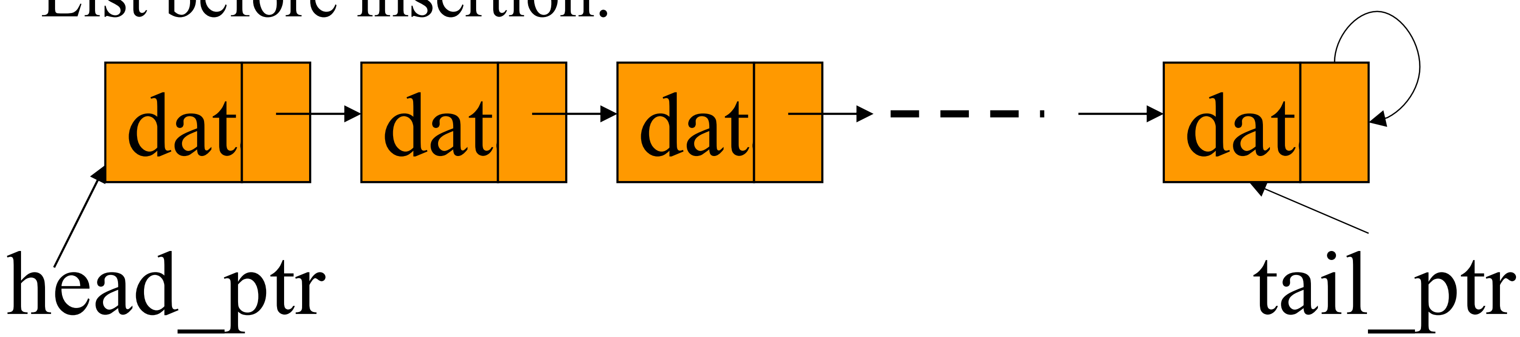
Problem 2: Write a recursive version to display a linked list in reverse order.

Insert– At the Head

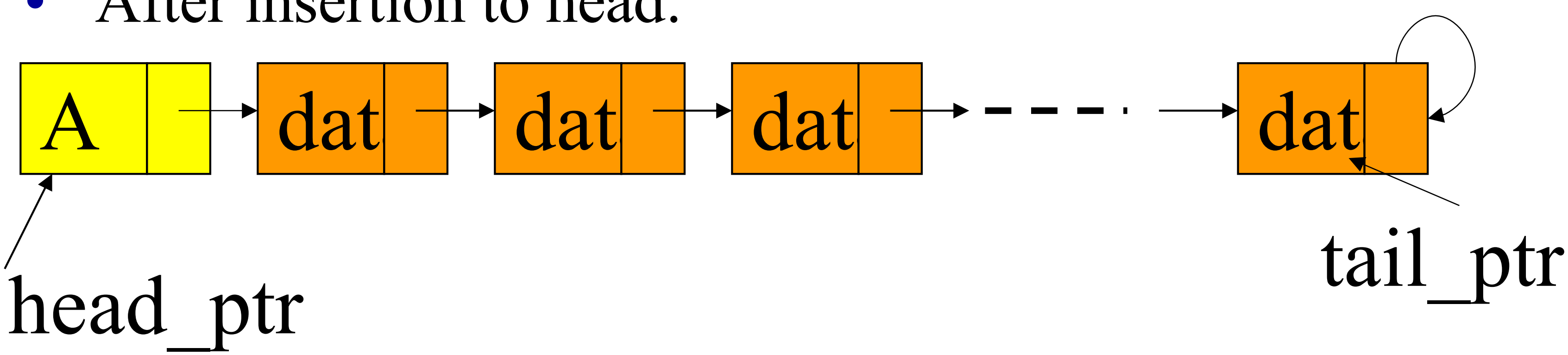
- Insert a new data A. Call **new:**



List before insertion:



- After insertion to head:



•The link value in the new item = old head_ptr

•The new value of head_ptr = newPtr

Implementation of insertHead( )

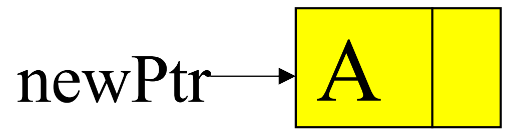
- **void** List::insertHead(**int** x){

```
        Node * newHead = new Node(x);
        newHead ->setNext(head_ptr);

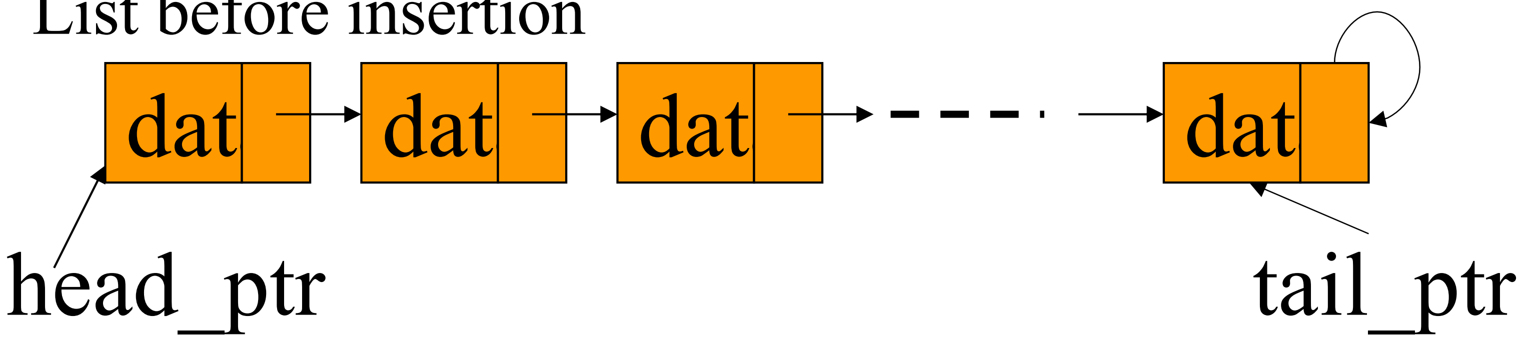
        head_ptr= newHead;
        if (tail_ptr == NULL) // only one item in list
            tail_ptr = head_ptr;
        numOfItems++;
    };
```

Insert – at the Tail

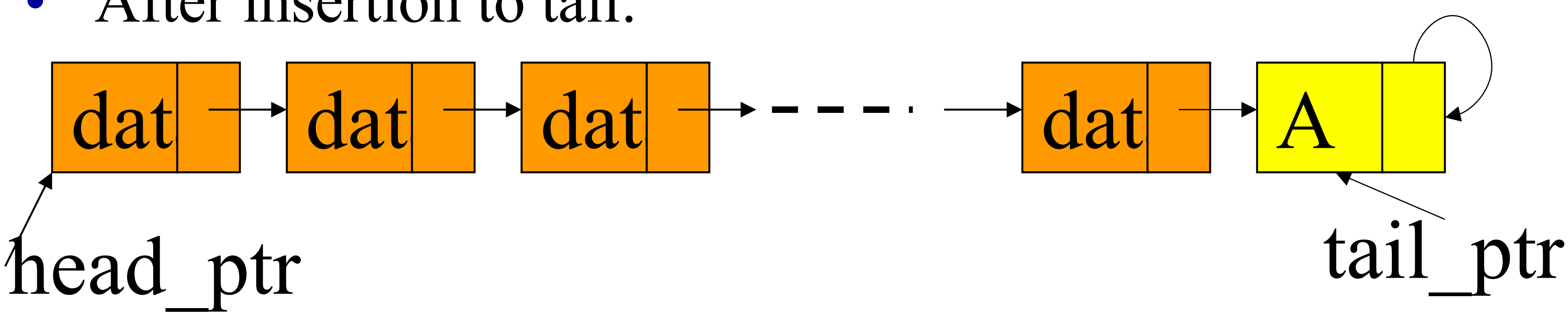
- Insert a new data A. Call **new:**



List before insertion



- After insertion to tail:



•The link value in the new item = NULL

•The link value of the old last item = newPtr

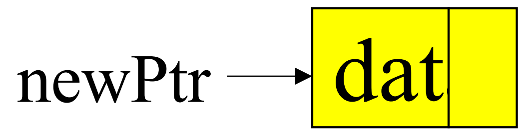
Implementation of insertTail( )

- **void** List::insertTail(**int** x){

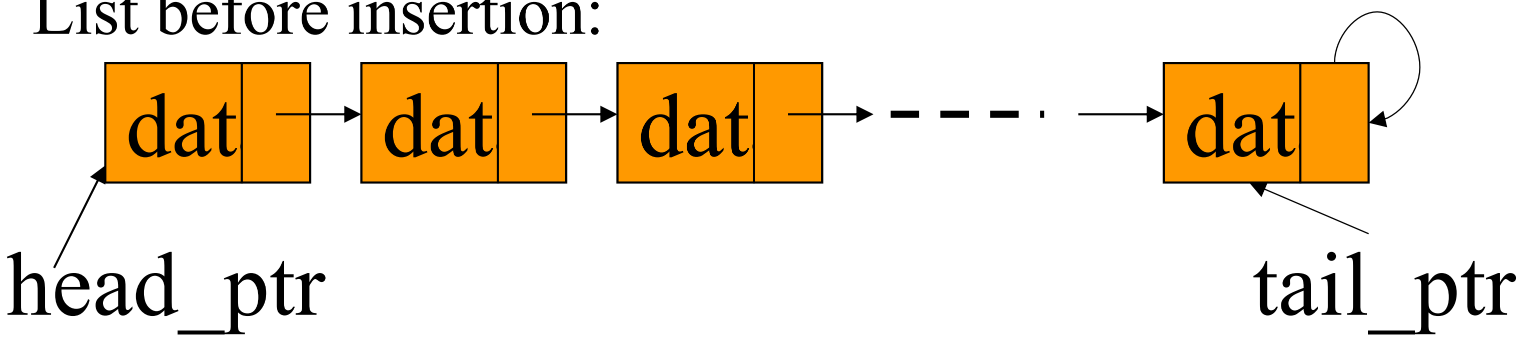
```
    if (isEmpty())
        insertHead(x);
    else{
        Node * newTail = new Node(x);
        tail_ptr->setNext(newTail);
        tail_ptr = newTail;  numOfItems++;
    }
};
```

Insert – inside the List

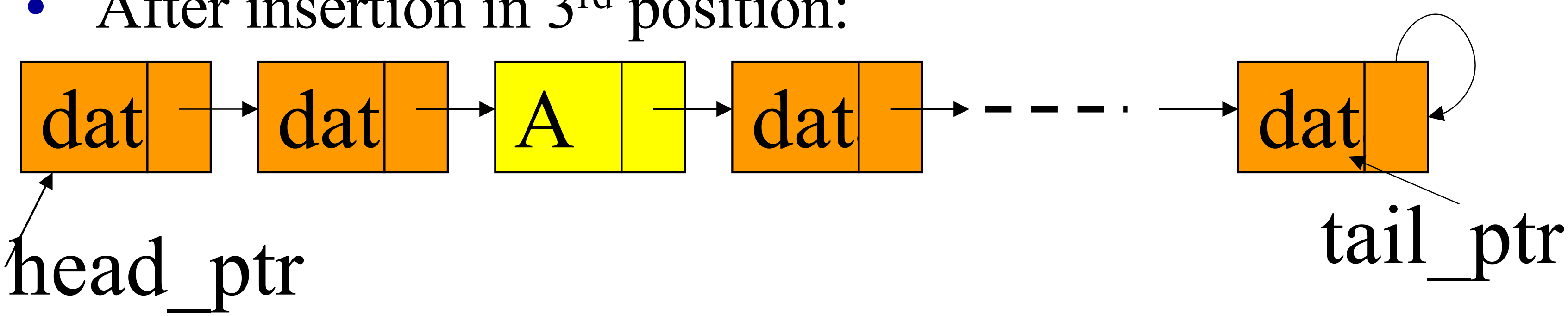
- Insert a new data A. Call **new:**



List before insertion:



- After insertion in 3rd position:



- The link-value in the new item = link-value of 2nd item
- The new link-value of 2nd item = newPtr

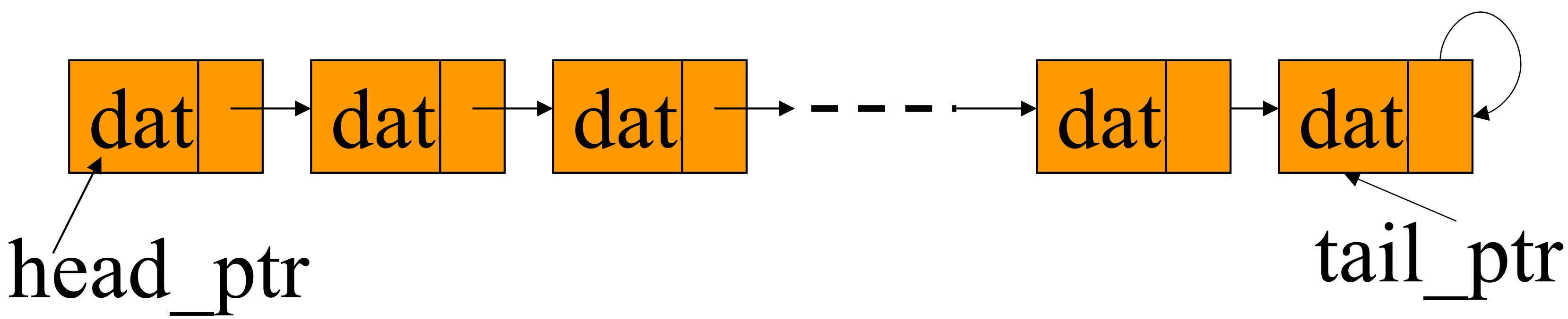
Implementation of insert( )

- // inserts item x after the item pointed to by p
- void List::insert(Node *p, int x){

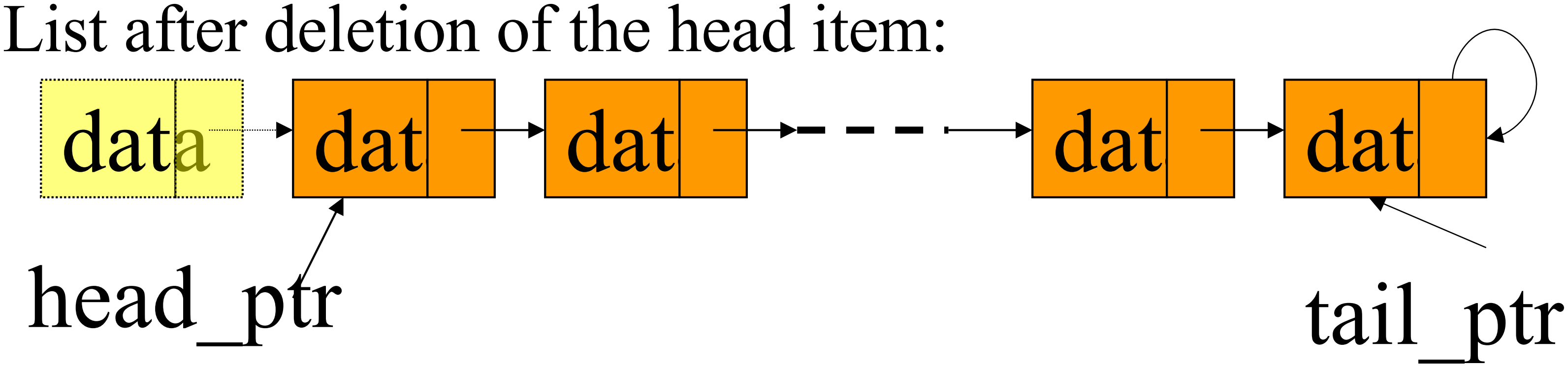
```
        Node *currentPtr = head_ptr;
        while(currentPtr !=NULL && currentPtr != p)
                currentPtr = currentPtr->getNext();
        if (currentPtr != NULL ) { // p is found
            Node *newNd=new Node(x);
             newNd ->setNext(p->getNext());
            p->setNext(newNd);
            numOfItems++;
        }
};
```

Delete – the Head Item

- List before deletion:



- List after deletion of the head item:



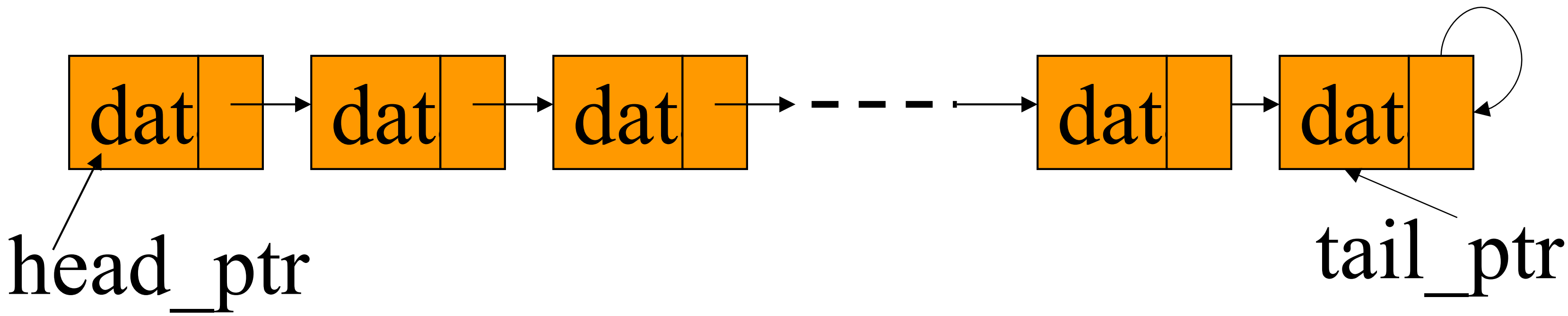
- The new value of head_ptr = link-value of the old head item
- The old head item is deleted and its memory returned

Implementation of removeHead( )

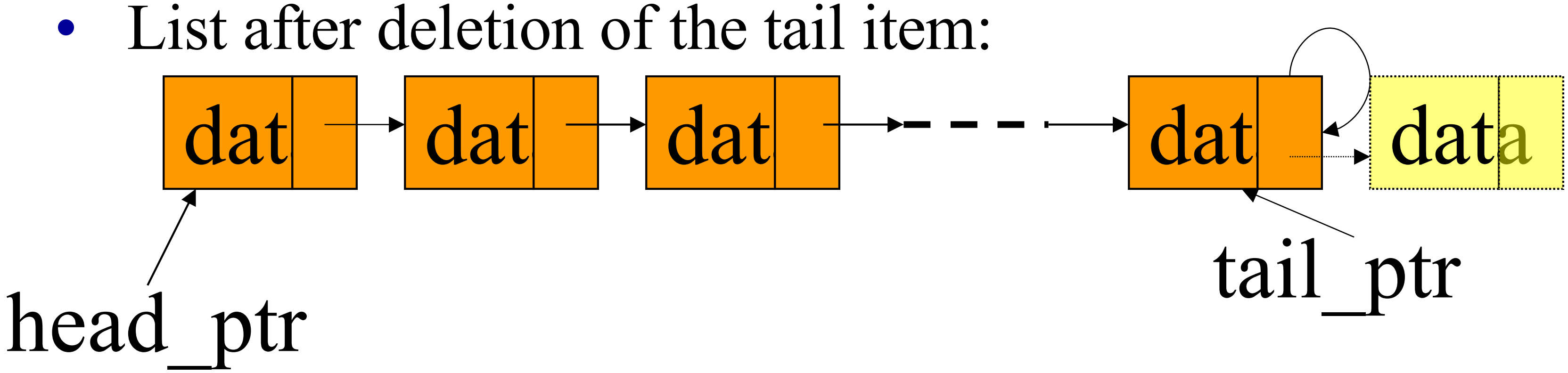
```
void List::removeHead( ){
    if (numOfItems == 0)
        return;
    Node * currentPtr = getHead( );
    head_ptr=head_ptr->getNext( );
    delete currentPtr;
    numOfItems--;
};
```

Delete – the Tail Item

- List before deletion:



- List after deletion of the tail item:



•New value of tail_ptr = link-value of the 3rd from last item
•New link-value of new last item = **NULL**.

Implementation of itemAt( )

- Node *List::itemAt(int position){

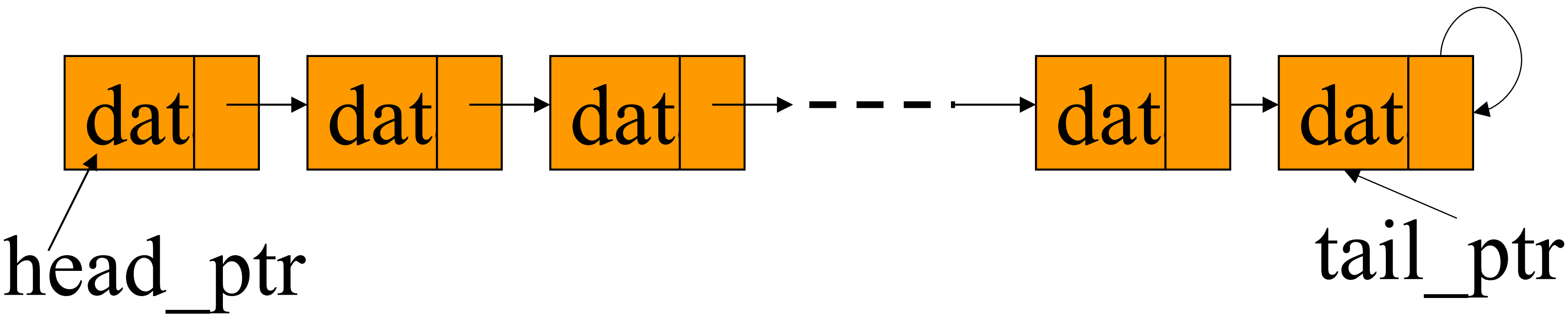
```
    if (position<0 || position>=numOfItems)
        return NULL;
    Node * currentPtr = getHead( );
    for(int k=0; k != position; k++)
        currentPtr = currentPtr -> getNext( );
    return currentPtr;
};
```

Implementation of removeTail( )

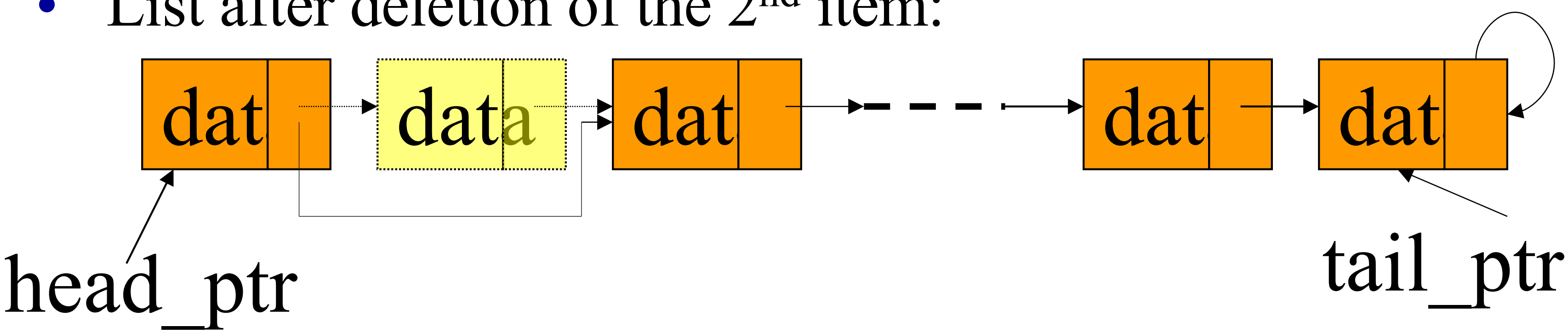
```
void List::removeTail( ){
        if (numOfItems == 0)
                return;
        if (head_ptr == tail_ptr){
                head_ptr=NULL; tail_ptr= NULL;
                numOfItems=0; return;
         }
        Node * beforeLast = itemAt(numOfItems-2);
        beforeLast->setNext(NULL); // beforeLast becomes last
        delete tail_ptr;   // deletes the last object
        tail_ptr=beforeLast;
        numOfItems--;
 };
```

Delete – an inside Item

- List before deletion:



- List after deletion of the 2nd item:



•New link-value of the item located before the deleted one = the link-value of the deleted item

Implementation of remove( )

- **void** List::remove(**int** x){ //delete node having x

    if (numOfItems == 0) return;

    if (head_ptr==tail_ptr && head_ptr->getData()==x){

        head_ptr=**NULL**; tail_ptr= **NULL**; numOfItems=0; **return**; }

  };

Implementation of remove( )

- ```
  void List::remove(int x){ //delete node having x
          if (numOfItems == 0)  return;
          if (head_ptr==tail_ptr && head_ptr->getData()==x){
              head_ptr=NULL; tail_ptr= NULL; numOfItems=0; return; }
          Node * beforePtr=head_ptr;  // beforePtr trails currentPtr
          Node * currentPtr=head_ptr->getNext();
          Node * tail = getTail();
          while (currentPtr != tail)
              if (currentPtr->getData( ) == x){ // x is found. Do the bypass
                  beforePtr->setNext(currentPtr->getNext());
                  delete currentPtr;     numOfItems--;   }
              else {  // x is not found yet. Forward beforePtr & currentPtr.
                  beforePtr = currentPtr;
                  currentPtr = currentPtr->getNext();   }
  };
  ```
  ```

Time of the Operations

- Time to search() is O(L) where L is the relative location of the desired item in the List. In the worst case. The time is O(n). In the average case it is O(N/2)=O(n).
- Time for remove() is dominated by the time for search, and is thus O(n).
- Time for insert at head or at tail is O(1).
- Time for insert at other positions is dominated by search time, and thus O(n).

Practice Problems

- Write code to reverse a singly linked list.
- Write code to sort a singly linked list.
- Write code to destroy a single linked list.
- Detect and remove a cycle in a singly linked list.
- Determine the mid-point of a singly linked list without using 2 separate passes or counter.
- Implement a stack using linked list.
- Implement a queue using a linked list.

Doubly linked list: each node has two pointers: to next and to previous node