Stack ADT

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Overview

- Stack
  - Introduction
  - Specification
  - Implementations
    - Python List (dynamic array)
    - Linked List
  - Applications
    - Tower of Hanoi
    - Bracket Matching
    - Maze Exploration

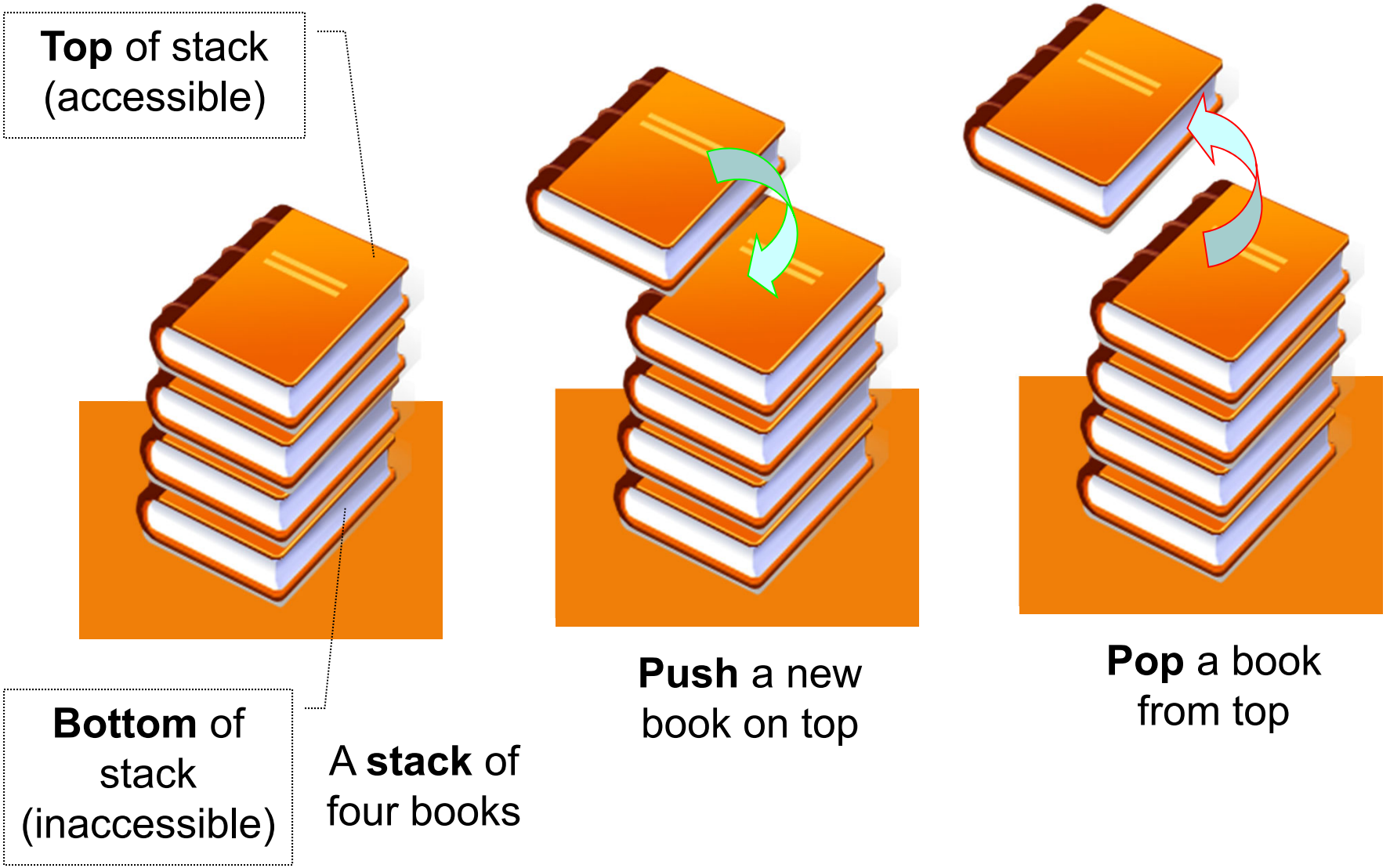
Stack: A specialized list

- List ADT allows user to manipulate (insert/retrieve/remove) item at **any position within the sequence of items**
- There are cases where we only want to consider a few specific positions only
  - E.g. only the first/last position
  - Can be considered as special cases of list
- **Stack** is one such example:
  - Only manipulation at the **last position** is allowed
- **Queue** (coming next) is another example

Stack: **Overview**

- Real life example:
  - A stack of books, A stack of plates, Etc
- It is easier to add/remove item to/from the **top of the stack**
- The latest item added is the first item you can get out from the stack
  - Known as **L**ast **I**n **F**irst **O**ut (**LIFO**) order
- Major Operations:
  - **Push** : Place item on top of the stack
  - **Pop** : Remove item from the top of the stack
- It is also common to provide:
  - **Top** : Take a look at the topmost item without removing it

Stack : Illustration



Stack ADT: **Python Specification**

```
#imports not shown
class StackBase(ABC):
    @abstractmethod
    def getTop(self):
        pass

    @abstractmethod
    def push(self, newItem):
        pass

    @abstractmethod
    def pop(self):
        pass
```

```
    @abstractmethod
    def size(self):
        pass

    @abstractmethod
    def isEmpty(self):
        pass
```

Question:

It is not meaningful to provide a ***`toString()`*** method in stack, why?

Stack ADT: Sample User Program

- Generate ***Fibonacci Sequence*** { 1, 1, 2, 3, 5, 8, … }
  - Basic idea: ***Fib*(N) = *Fib*(N-1) + *Fib*(N-2)**

```
def main():
    mystack = 

    mystack.push( 1 )
    mystack.push( 1 )

    for i in range(10):
        prev1 = mystack.getTop( )
        mystack.pop( )
        prev2 = mystack.getTop( )

        cur = prev1 + prev2;
        mystack.push( prev1 );
        mystack.push( cur );
        print(cur)
```

To be replaced by actual implementations of the `StackBase` class

We start off with a stack with 2 items, "1" and "1" on top

Stack ADT : **Implementations**

- Many ways to implement Stack ADT, we will cover:
  - **Python List**
    - Make use of the built-in Python List
    - Essentially a **dynamic array**
  - **Linked List implementation:**
    - Study the best way to make use of linked list
- Learn how to weight the pros and cons for each implementations

Stack ADT : **Design Consideration**

- How to choose appropriate implementation?
  - Concentrate on the major operations in ADT
  - Match with data structures you have learned
    - Pick one to be the internal data structure of an ADT
    - Can the internal data structure support what you need?
    - Is the internal data structure efficient in those operations?

- Internal data structure like array, linked list etc are usually very flexible:
  - Make sure you use them in the best possible way

Python List is quite versatile

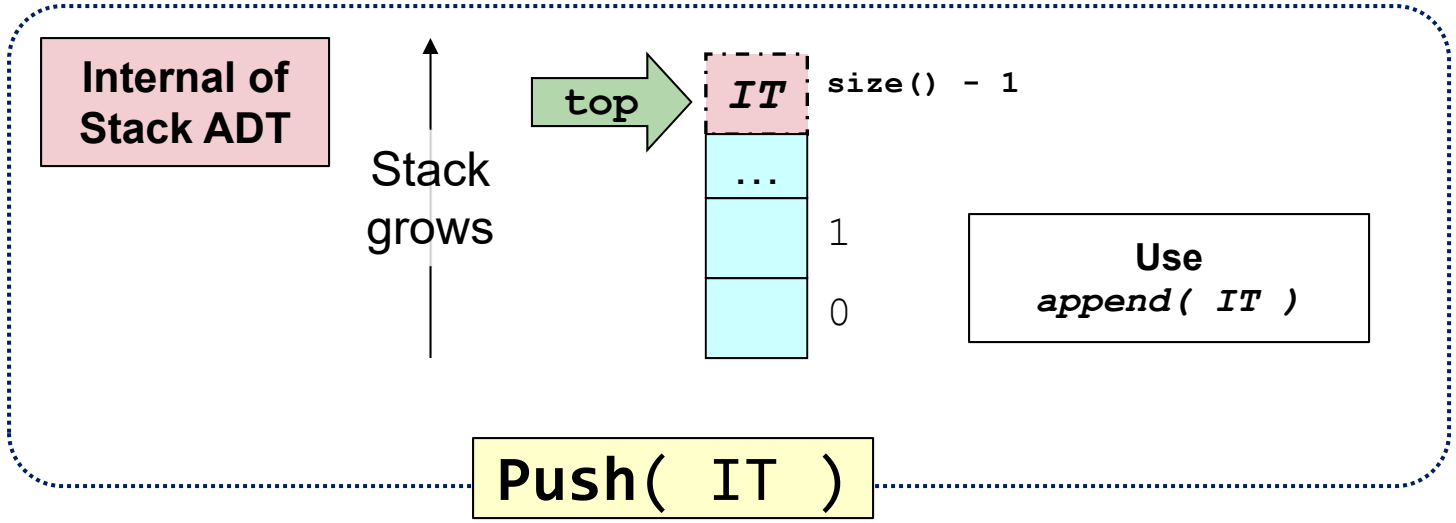
STACK ADT USING PYTHON LIST

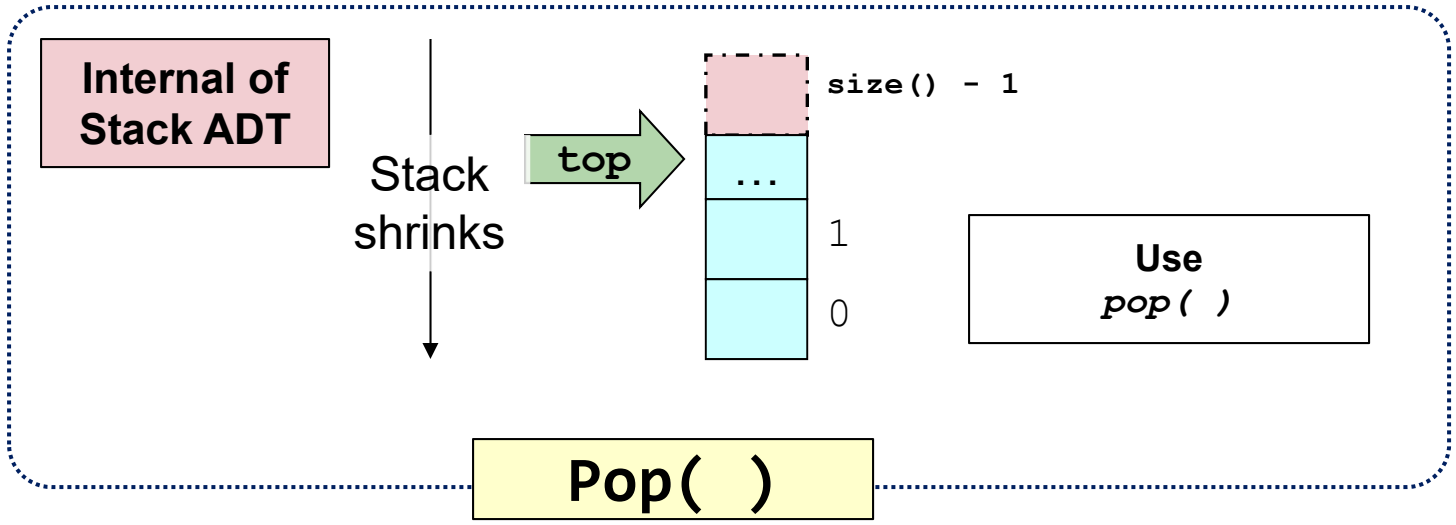
Stack ADT: Using Python List

- Python List has the following capabilities:
  - Add the last item
    - ***append()*** [Very efficient]
  - Remove item from any location
    - Many ways: ***pop(index)***, ***remove(item)***
    - Only pop() last item is efficient (why?)

- What Stack ADT needs:
  - Add/Remove from **top of stack**
    - **No manipulation of other locations**
  - Hence, to make the best use of Python List:
    - Use the **back of list** as the **top of stack**

Using Python List as Stack: Illustration





`StackList`: Implementation

```
class StackList(StackBase):

    def __init__(self):
        self._items = []

    def getTop(self):
        if not self.isEmpty():
            return self._items[-1]
            #negative index = count from the end of list
        else:
                return None

    def push(self, newItem):
        self._items.append(newItem)
```

`StackList`: Implementation

```
def pop(self):
    if not self.isEmpty():
        self._items.pop() #last item is removed
        return True
    else:
        return False

def size(self):
    return len(self._items)

def isEmpty(self):
    return len(self._items) == 0
```

- Very straightforward as **Python List** provides all the required operation with good time complexity

Python List: Comments

- Python List is designed with versatility in mind:
  - Especially to "double up" as simple **stack** / **queue**
- We provide the ***Fibonacci*** program written directly with Python List (but used as a "stack") to highlight the differences
- Contrast with Stack ADT to understand how ADT wraps around and "protect" the property of stack

`Fibonacci`: Using Python List

```
def main():
    mystack = []
    #Initialize the first two fibonacci numbers
    mystack.append( 1 )
    mystack.append( 1 )

    for i in range(10):
        prev1 = mystack[-1]
        mystack.pop()
        prev2 = mystack[-1]

        cur = prev1 + prev2
        mystack.append( prev1 )
        mystack.append( cur )
        print(cur)
```

Question:

What are the potential issues of using list directly as stack?

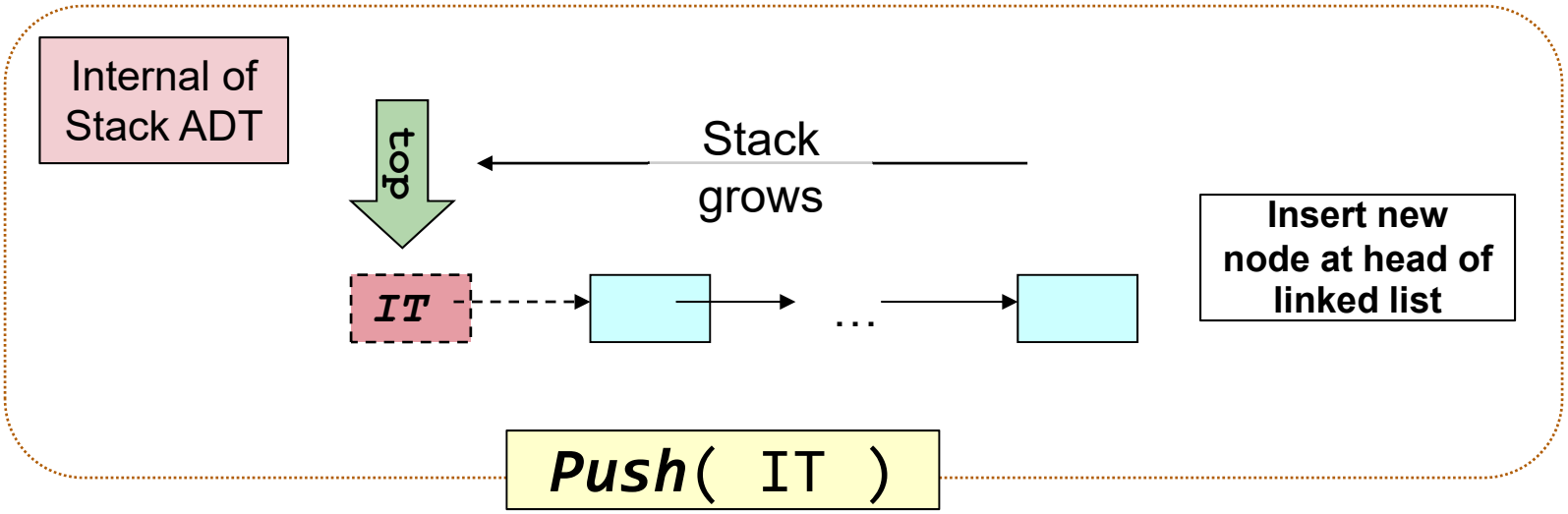
Not linked list again!

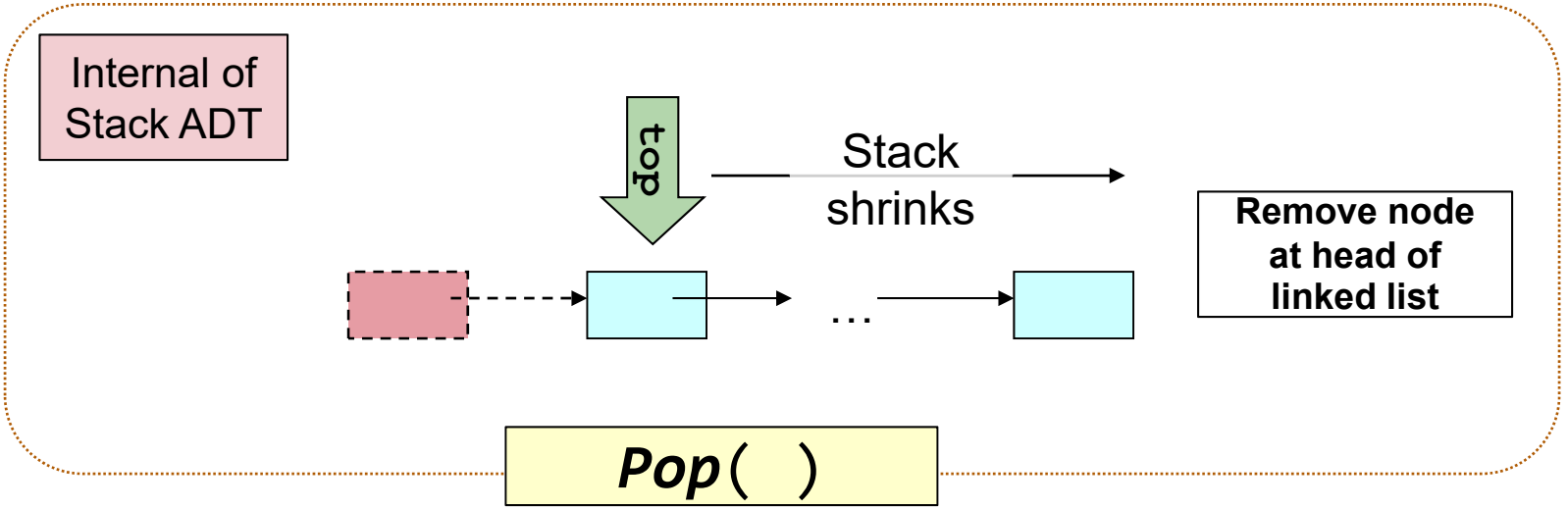
STACK ADT USING LINKED LIST

Stack ADT: Using Linked List

- Characteristics of singly linked list:
  - Efficient manipulation of 1st Node:
    - Has a **`head`** pointer directly pointing to it
    - No need to traverse the list
  - Manipulation of other locations is possible:
    - Need to first traverse the list, less efficient
- Hence, best way to use singly linked list:
  - Use 1st Node as the top of stack
- Question:
  - How would you use **other variations of linked list?**

Stack ADT: Using Linked List (Illustration)





`StackLinkedList`: Implementation

```
class StackLinkedList(StackBase):

    def __init__(self):
        self._head = None
        self._size = 0

    def getTop(self):
        if not self.isEmpty():
            return self._head.item
        else:
            return None

    def push(self, newItem):
        newPtr = SinglyNode(newItem)

        newPtr.next = self._head
        self._head = newPtr

        self._size += 1
        return True
```

Reuse the **SinglyNode** linked list node from the List ADT

As we only insert at head position. General insertion code not needed.

`StackLinkedList.py`

`StackLinkedList`: Implementation

```
    def pop(self):
        if not self.isEmpty():
            self._head = self._head.next
            self._size -= 1
            return True
        else:
            return False

    def size(self):
        return self._size

    def isEmpty(self):
        return self._size == 0
```

Similarly, only need to consider removal from head position in ***`pop`***`()`

LIFO? Is it good for anything?

STACK APPLICATIONS

Stack Applications

- Many useful applications for stack, we cover:
  a. **Tower of Hanoi**
  b. **Bracket Matching**
  c. **Maze Exploration**

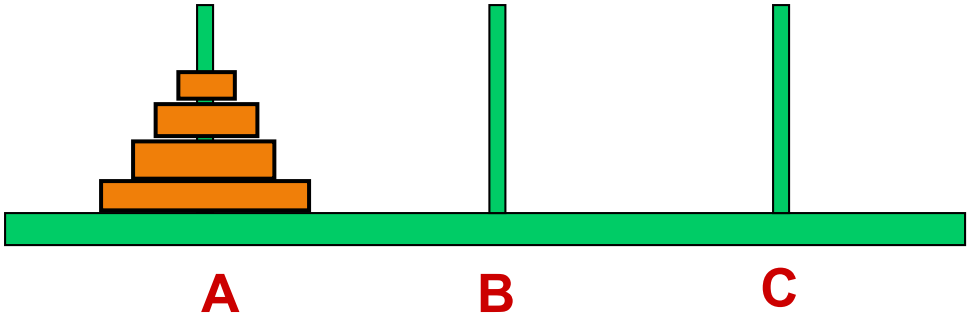
- More "Computer Science" inclined examples:
  - **Base-N number conversion**
  - **Postfix evaluation**
  - **Infix to postfix conversion**

Stack Application Example One

TOWER OF HANOI

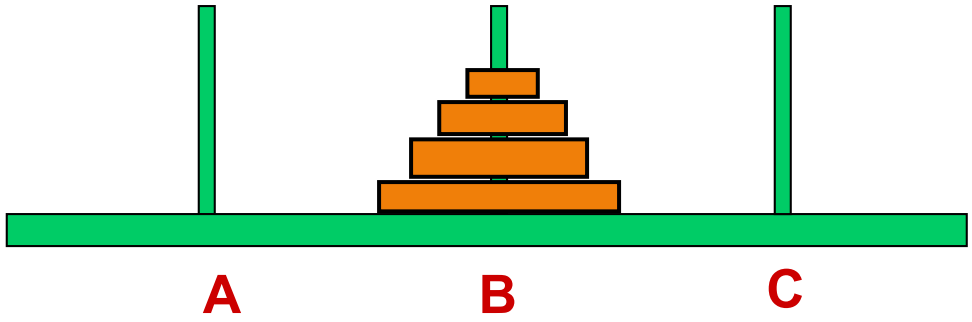
Tower of Hanoi : Description

initial state



- How do we move all the disks from pole "**A**" to pole "**B**", using pole "**C**" as temporary storage
  - One disk at a time
  - Disk must rest on top of larger disk

final state



Each pole is a stack…

- We are not writing a program to ***solve the puzzle automatically***
  - Coming soon…. ☺
- Just a simple program to let user **play the puzzle:**
  - Keep track of the discs
  - Check movement
  - Display the current state
  - etc
- Since we can only
  - Remove the topmost disc from a pole, then
  - Place the disc on top of other pole
- Clearly, **each pole is a stack**

Tower of Hanoi : Implementation

- Use a **List of 3 Stacks** to represent the 3 poles

```
class TowerOfHanoi():

    def __init__(self, nDiscs):
        self._poles = [StackList() for i in range(3)]
        self._nDiscs = nDiscs

        for i in range(nDiscs,0,-1):
            self._poles[0].push(i)
```

Initialize the first pole to have a number of discs. The number represent the disc diameter.

Tower of Hanoi : **Implementation**

A simple function to check and perform a disc movement

```
    def move(self, src, dst):
        if src == dst:
            return True #pretend we have moved

        if src < 0 or src > 2 or dst < 0 or dst > 2:
            return False    #illegal pole number

        if self._poles[src].isEmpty():
            return False        #no disc at the source pole

        srcDisc = self._poles[src].getTop()

        if (not self._poles[dst].isEmpty()) and \
            (srcDisc > self._poles[dst].getTop()) :
            return False        #destination pole has a larger disc

        #all checks passed, we can move safely
        self._poles[dst].push( srcDisc )
        self._poles[src].pop()
```

Tower of Hanoi : **Implementation**

```
    def _displayPole(self, pole):
        copy = StackList()
        while not pole.isEmpty():
            disc = pole.getTop()
            print(disc)
            pole.pop()
            copy.push(disc)

        while not copy.isEmpty():
            pole.push(copy.getTop())
            copy.pop()

    def display(self):
        for i in range(3):
            self._displayPole(self._poles[i])
            print("Pole "+ chr(ord('A')+i))
            print("------")
```

A straightforward implementation to show the discs on each pole. Functional but not pretty ☺

Try to write a better looking display function?

Tower of Hanoi : **Main Function**

```
def main():
    toh = TowerOfHanoi(3)

    toh.display()

    srcPole = int(input("src pole [0,1,2; -1 dst exit]:"))
    dstPole = int(input("dst pole [0,1,2; -1 dst exit]:"))
    while srcPole >= 0 and dstPole >= 0:
        if toh.move(srcPole, dstPole):
            print("Move Ok!")
        else:
            print("Illegal Move!")

        toh.display()
        srcPole = int(input("src pole [0,1,2; -1 dst exit]:"))
        dstPole = int(input("dst pole [0,1,2; -1 dst exit]:"))
```

A simple program to allow a human player to play the puzzle

Stack Application Example Two

BRACKET MATCHING

Bracket Matching : Description

- Mathematical expression can get quite convoluted:
  - E.g. { [x+2(i-4!)]^e+4π/5*(φ-7.28) ……

- We are interested in checking whether all brackets are matched correctly **(** with **)**, **[** with **]** and **{** with **}**

- Bracket matching is equally useful for checking programming code

Bracket Matching : Pseudo-Code

1. Go through the input string character by character
   - Non-bracket characters
     - Ignore
   - Open bracket { , [or (
     - Push into stack
   - Close bracket },] or)
     - Pop from stack and check
     - If the stack top bracket does not agree with the closing bracket, complain and exit
     - Else continue
2. If the stack is not empty after we read through the whole string
   - The input is wrong also

Bracket Matching : Implementation (1)

```
def matchBracket( input ):
    openBrackets = ['(','[','{']
    closeBrackets = [')',']','}']
    bracketStack = StackList()

    for cur in input:
        if cur in openBrackets: #cur is an open bracket
            idx = openBrackets.index(cur)
            bracketStack.push(closeBrackets[idx])

        elif cur in closeBrackets:
            if bracketStack.isEmpty() or \
                bracketStack.getTop() != cur:
                return False
            else:
                bracketStack.pop()

    return bracketStack.isEmpty()
```

Why do we push the opposite bracket into the stack?

What is the meaning of these conditions?

Shouldn't we just return **True**?

Bracket Matching : Implementation (2)

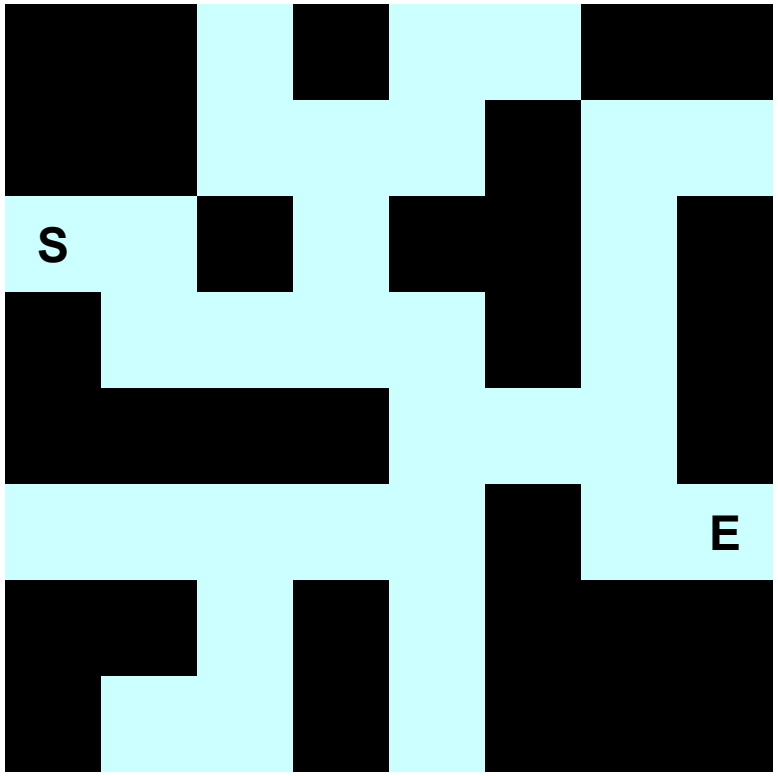
```
def main():

    userInput = input("Enter a math expression:")

    if matchBracket(userInput):
        print("Bracket matched!")
    else:
        print("Bracket NOT matched!")
```

- A simple main program to test the `matchBracket()` function

Stack Application Example Three

MAZE EXPLORATION

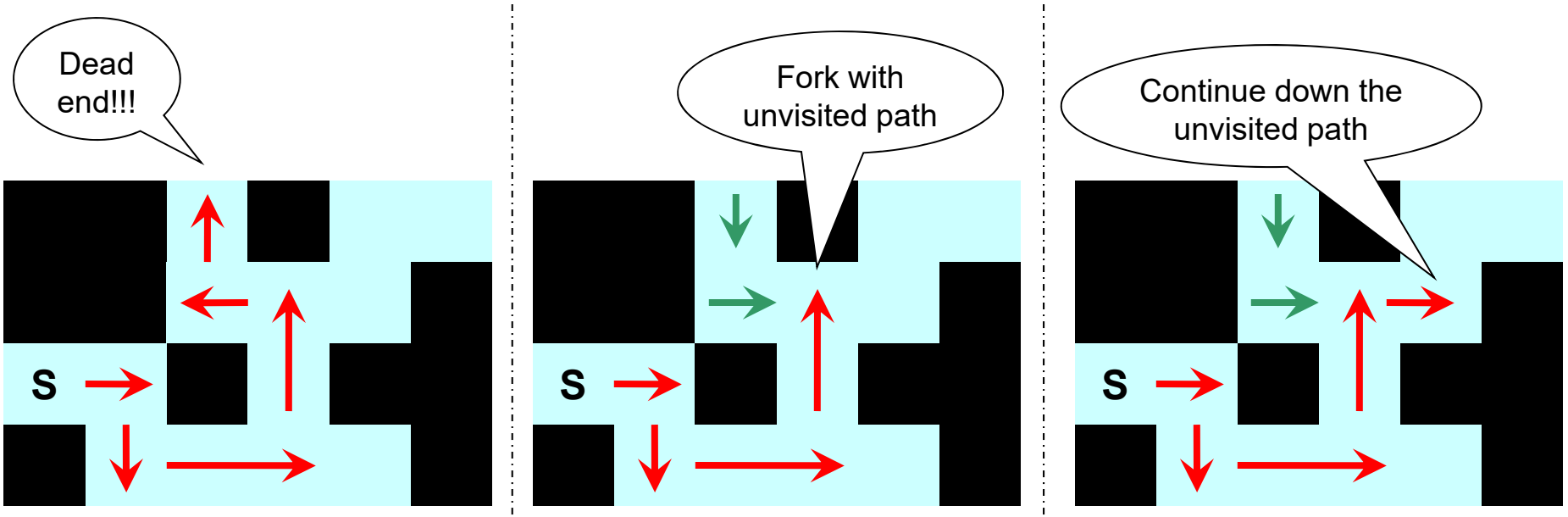
Exploring Maze: **D**escription



- How to define an algorithm that **always** get you from **S** to **E** (as long as there is a path)?
  - What should you do when you reached a dead end?

Exploring Maze: **Basic Ideas**

- When we reached a dead end
  - Always restart from **S** is usually not a good idea
- Instead, we retrace our steps until:
  - the most recent fork which has an unvisited path
  - take the unvisited path and continue exploration



Exploring Maze: **Some design issues**

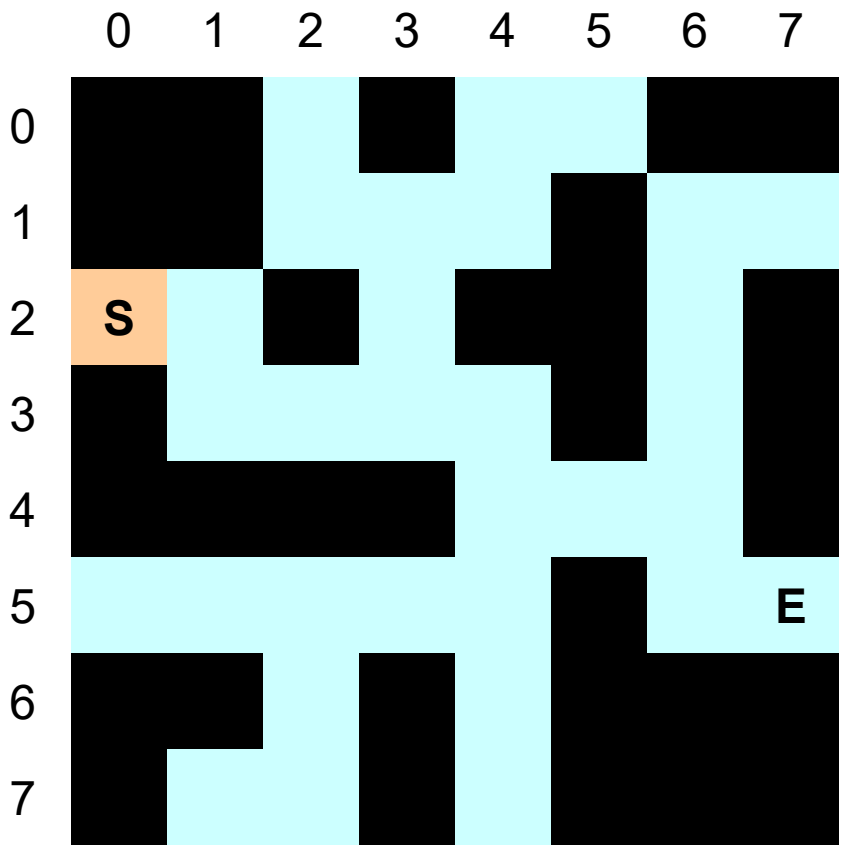
- The maze is represented as a **N x N 2D array**
  - Each square has a unique `(row, column)` coordinate
- There are 4 directions of movement from a square:
  - `{Up, Left, Down, Right}`

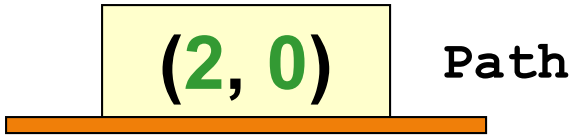
- Each square will know about:
  - Which direction is ***unvisited***
  - Assume a method `getUnvisitedDir()` is implemented for this purpose
- When a square has multiple unvisited exits:
  - We visit them in the order of the description above
- The path traveled is kept as a **stack of coordinates**

Exploring Maze: **Pseudo Code**

```
1. Path = empty
2. done = false      //are we are the end yet?
3. Path.push(coordinate of S)
4. While (Path is not empty && not done)
   i. CurSq = Path.top( )    //where are we now?
   ii. NewDir = CurSq.getUnvisitedDir( )
   iii. If ( NewDir == None)  //dead end!
        Path.pop( )           //move back one square
   iv. Else                   //there is an exit
      a) NewSq = CurSq.move( NewDir )
      b) Path.push( coordinate of NewSq )
      c) If (NewSq == E)      //Yes! We reached the end!
            done = true
```

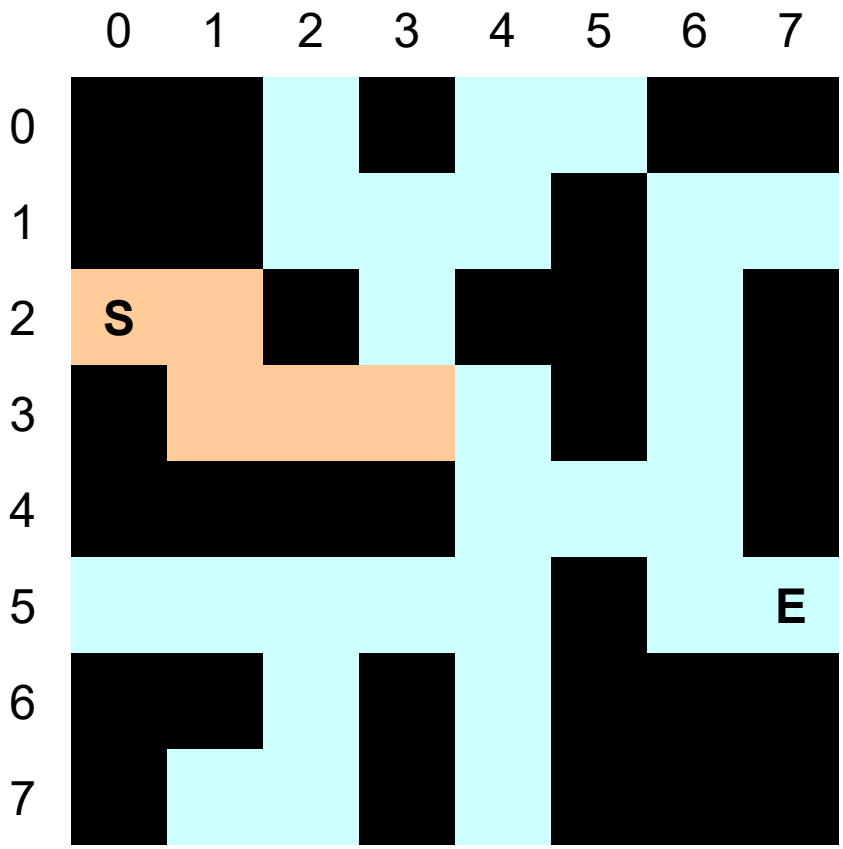
Exploring Maze: **Test Run …. (1)**

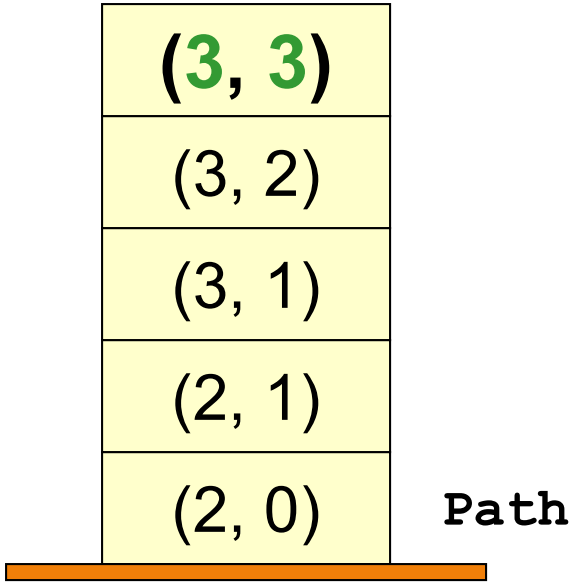




- Just started at **(2, 0)**

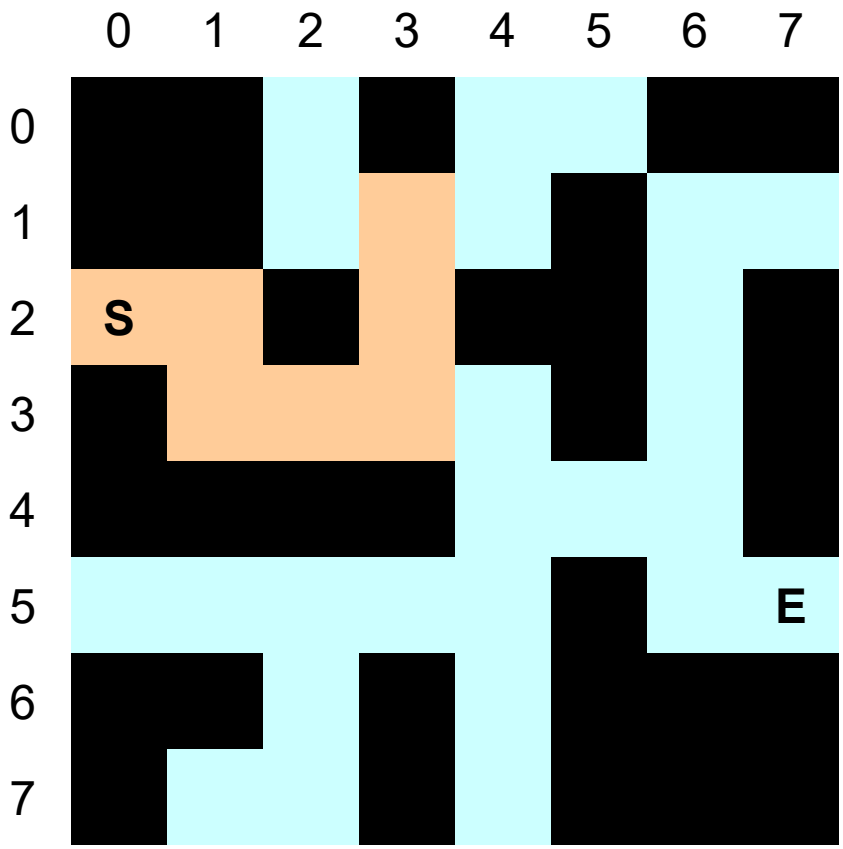
Exploring Maze: **Test Run** ….(2)

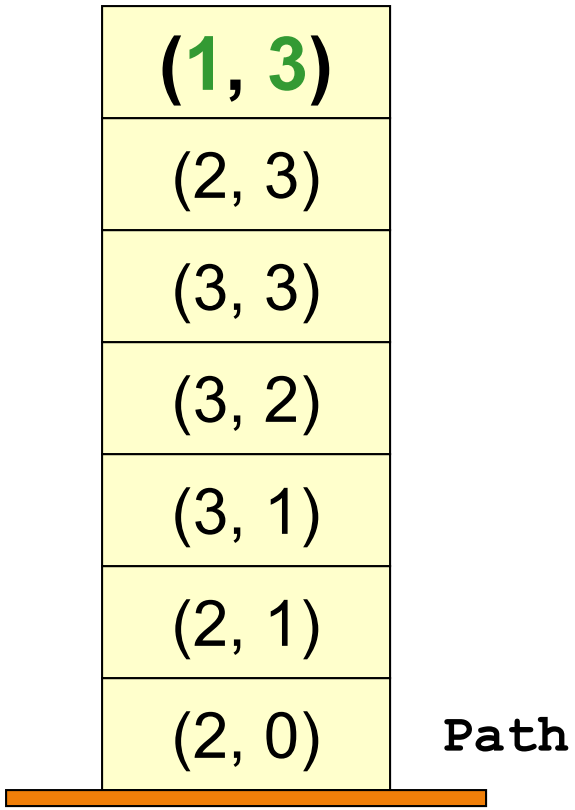




- (3, 3) is the first square with multiple exits
  - As stated, we will first try to go Up

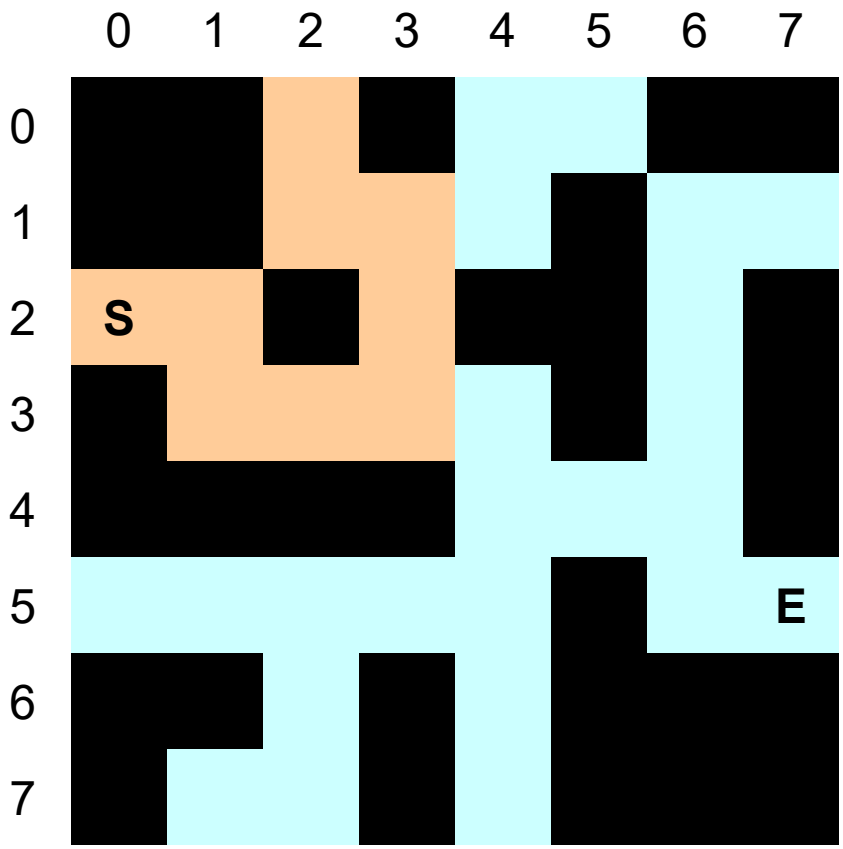
Exploring Maze: **Test Run** ….(3)

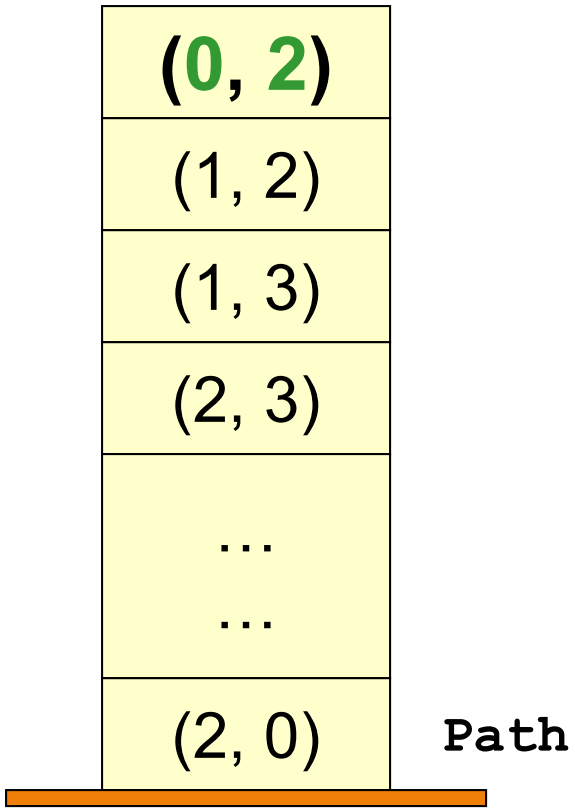




- Multiple exits at (1, 3)
  - go Up is impossible, so go Left is the 2nd choice

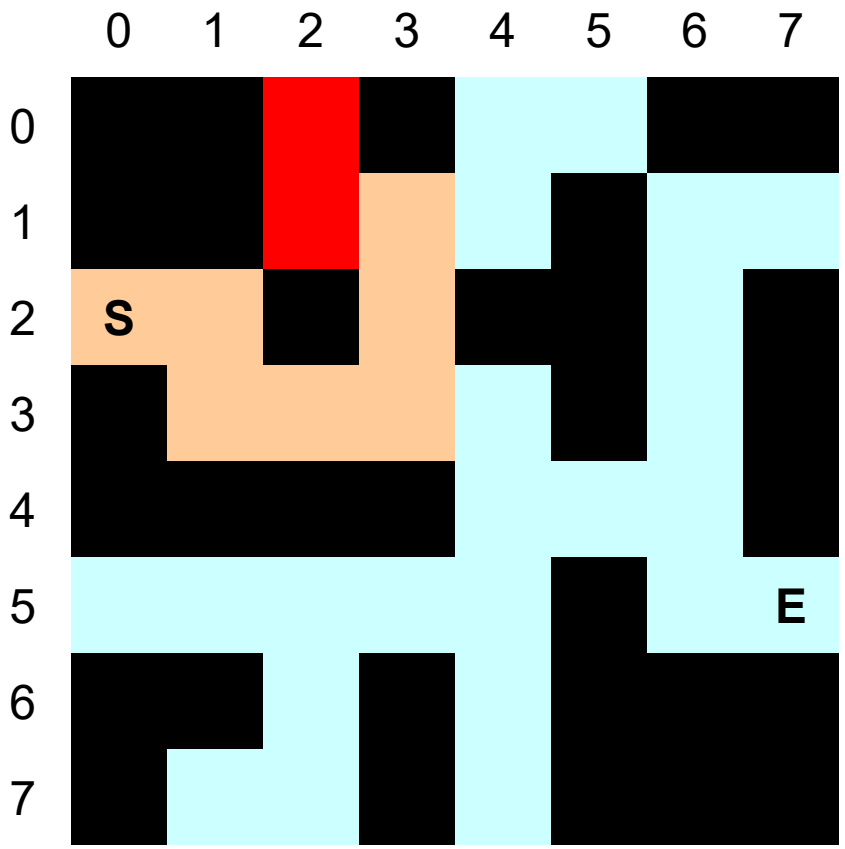
Exploring Maze: **Test Run** ….(4)

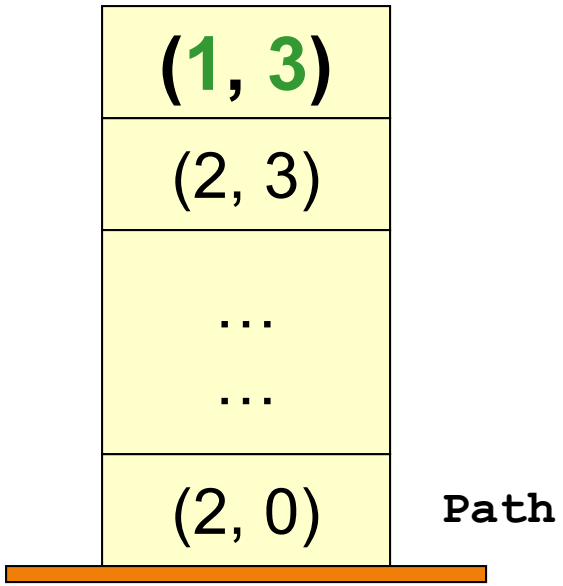




- No exits from **(0, 2)**
  - Back trace: pop until a square with unvisited exits

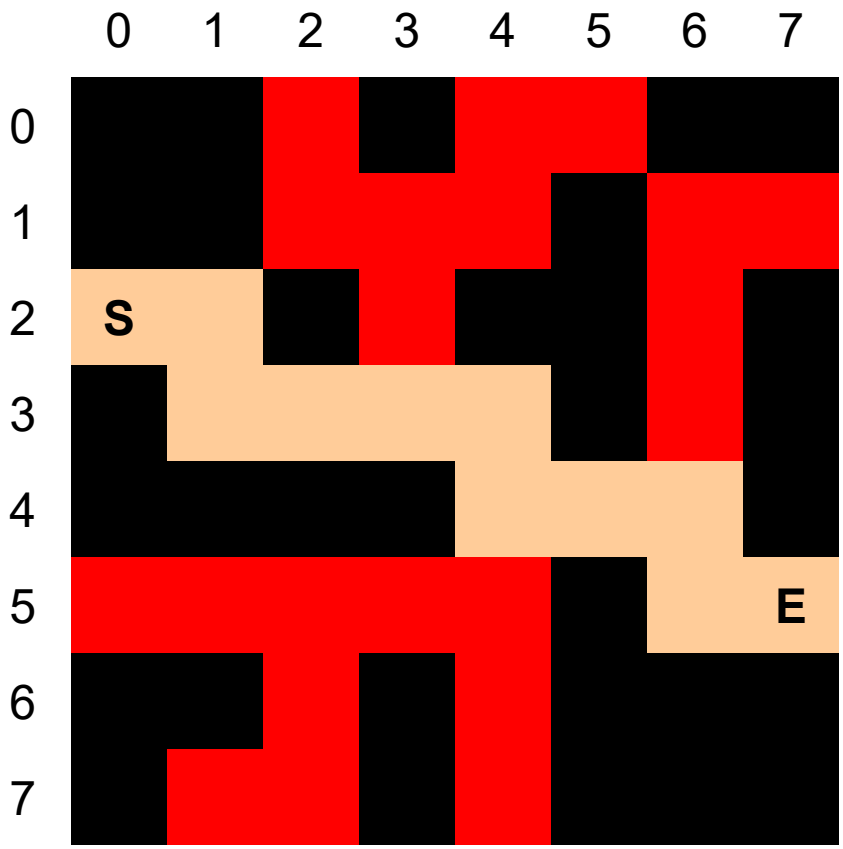
Exploring Maze: **Test Run** ….(5)

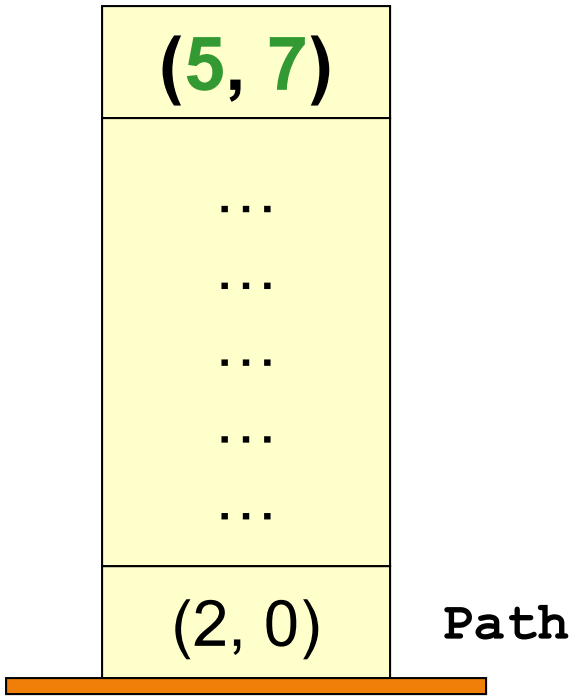




- Back to **(1,3)** after several pops
  - Up, Left, Down all impossible, going Right to **(1,4)**

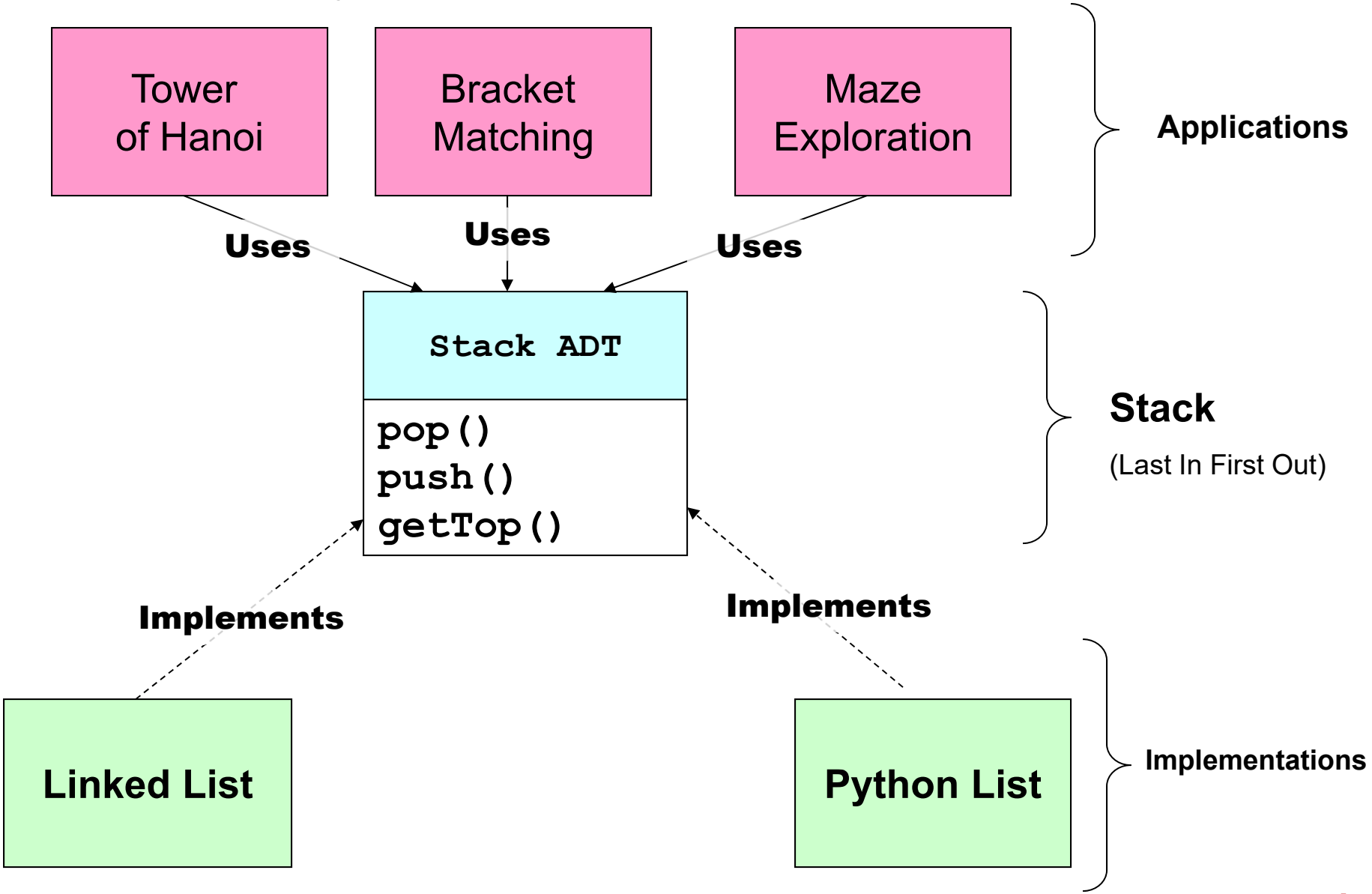
Exploring Maze: **Test Run (much later)**





- Poor Guy/Girl ☹
  - Traveled the whole maze to find the exit….
  - Can we refine the algorithm that **always perform better?**

Summary


Tower of Hanoi
Bracket Matching
Maze Exploration
Applications
Uses
Uses
Uses
Stack ADT
pop()
push()
getTop()
Stack
(Last In First Out)
Implements
Implements
Linked List
Python List
Implementations

END