CS2040C Data Structures and Algorithms

Stack ADT

Outline

- Stack
  - Introduction
  - Specification
  - Implementations
    - Linked List
    - STL vector
  - Applications
    - Bracket Matching
    - Towers of Hanoi
    - Maze Exploration

Stack: A specialised list

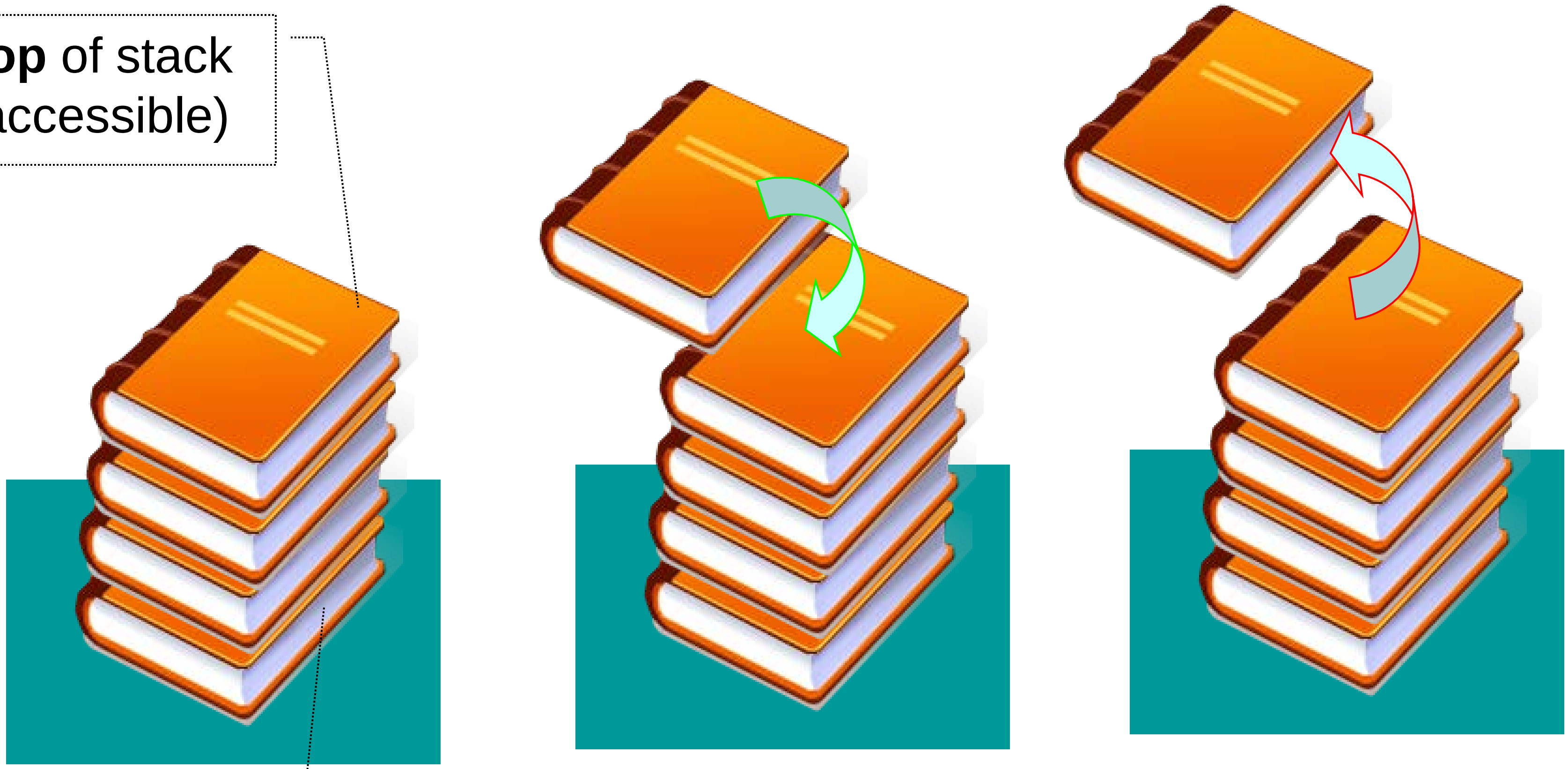
- List ADT allows user to manipulate (insert/retrieve/remove) item at **any position within the sequence of items**
- There are cases where we only want to consider a few specific positions only
  - E.g. only the first/last position
  - Can be considered as special cases of list
- Stack is one such example: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

What is a stack?

- 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

Top of stack (accessible)



Bottom of stack (inaccessible)

A **stack** of four books

Push a new book on top

Pop a book from top

Stack ADT: C++ Specification

```
template <typename T>
class Stack {
   public:
     Stack();

     bool isEmpty() const;
     int size() const;

     void push(const T& newItem) throw (SimpleException);

     void pop() throw (SimpleException);
     void pop(T& stackTop) throw (SimpleException);


     void getTop(T& stackTop) const
               throw (SimpleException);
private:
     //Implementation dependent
     //See subsequent implementation slides
};
```

Stack ADT is a template class just like List ADT

Note the 2 versions of **pop()**

Stack ADT: Implementations

- Many ways to implement Stack ADT, we will cover:
  - Linked List implementation:
    - Study the best way to make use of linked list
  - STL vector implementation:
    - Make use of STL container vector
- Learn how to weigh the pros and cons for each implementation

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 (underlying) data structure of an ADT
    - Can the internal data structure support what you need?
    - Is the internal data structure efficient in those operations?

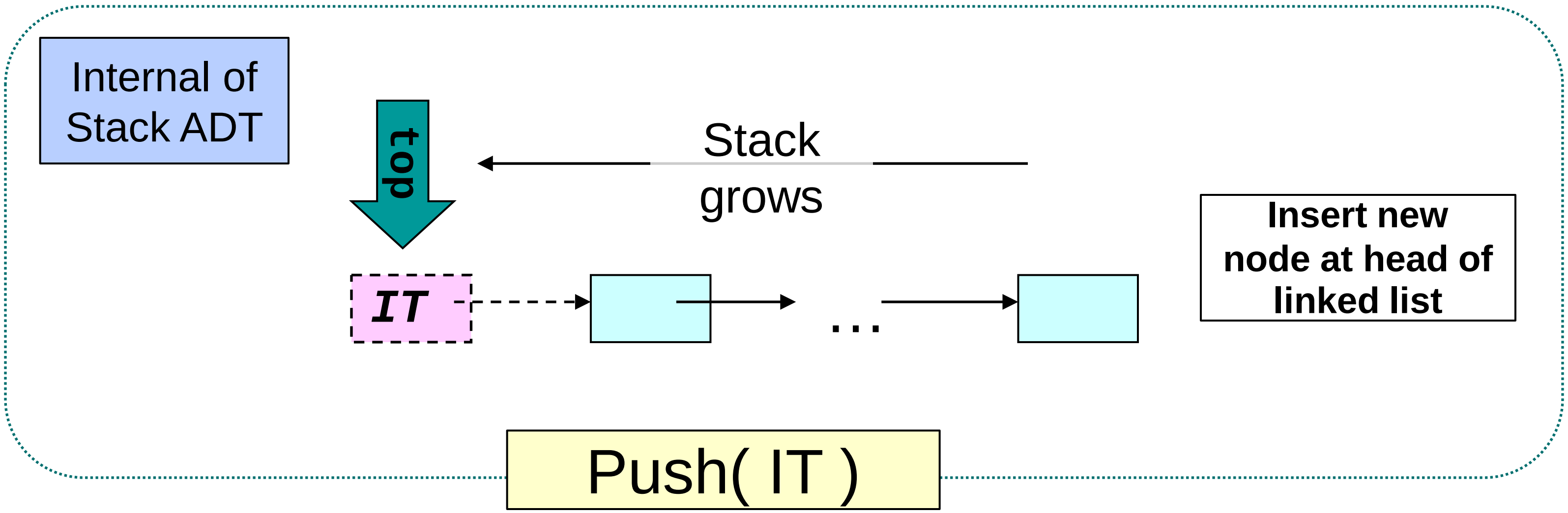
- Internal data structure like array, linked list etc are usually very flexible:
  - Make sure you use them in the best possible way

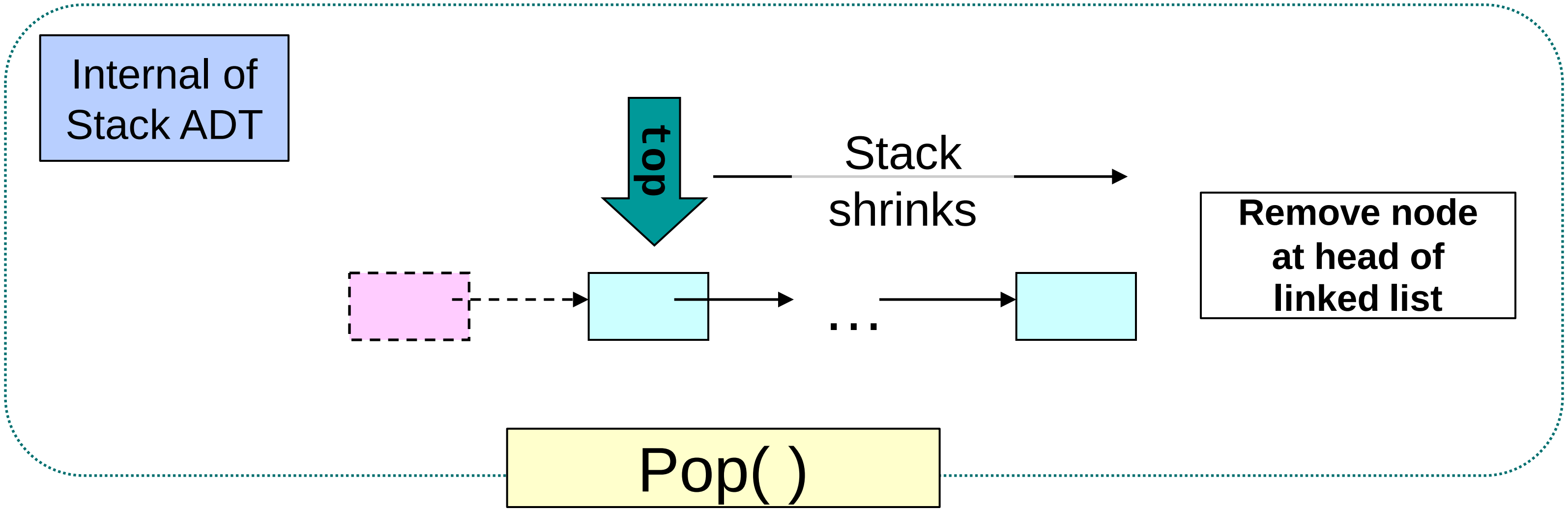
Stack ADT using Linked List

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)





Stack ADT (Linked List): C++ Specification

```
template <typename T>
class Stack {
  public:
    Stack();
    ~Stack()

    bool isEmpty() const;
    int size() const;

    void push(const T& newItem) throw (SimpleException);

    void pop() throw (SimpleException);
    void pop(T& stackTop) throw (SimpleException);


    void getTop(T& stackTop) const
                    throw (SimpleException);


private:
    struct ListNode {
        T item;
        ListNode* next;
    };

    ListNode* _head;
    int _size;
};
```

Need destructor as we allocate memory dynamically

Methods from slide 6. No change.

Similar to Linked List implementation of List ADT

StackP.h

Implement Stack ADT (Linked List): 1/3

```
template<typename T>
Stack<T>::Stack()
: _size(0), _head(NULL) { }

template<typename T>
Stack<T>::~Stack()
{
    while (!isEmpty())
        pop();
}

template<typename T>
bool Stack<T>::isEmpty() const
{
    return _size == 0;
}

template<typename T>
int Stack<T>::size() const
{
    return _size;
}
```

Make use of own methods to clear up the nodes

StackP.cpp

Implement Stack ADT (Linked List): 2/3

```
template<typename T>
void Stack<T>::pop(T& stackTop)
    throw (SimpleException)
{
    ListNode* cur;

    if ( isEmpty() )
        throw SimpleException("Stack is empty on pop()");
    else {

        stackTop = _head->item;

        cur = _head;
        _head = _head->next;

        delete cur;
        cur = NULL;
        _size-- ;

    }
}
```

As we only remove from head position. General removal code not needed.

StackP.cpp

Implement Stack ADT (Linked List): 3/3

```
template<typename T>
void Stack<T>::push(const T& newItem)
    throw (SimpleException)
{
    ListNode* newPtr = new ListNode;

    newPtr->item = newItem;
    newPtr->next = _head;
    _head = newPtr;
    _size++;

}

template<typename T>
void Stack<T>::getTop(T& stackTop) const
    throw (SimpleException)
{
    if ( isEmpty() )
        throw SimpleException("Stack is empty on getTop()");
    else
        stackTop = _head->item;
}
```

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

StackP.cpp

Stack ADT using STL vector

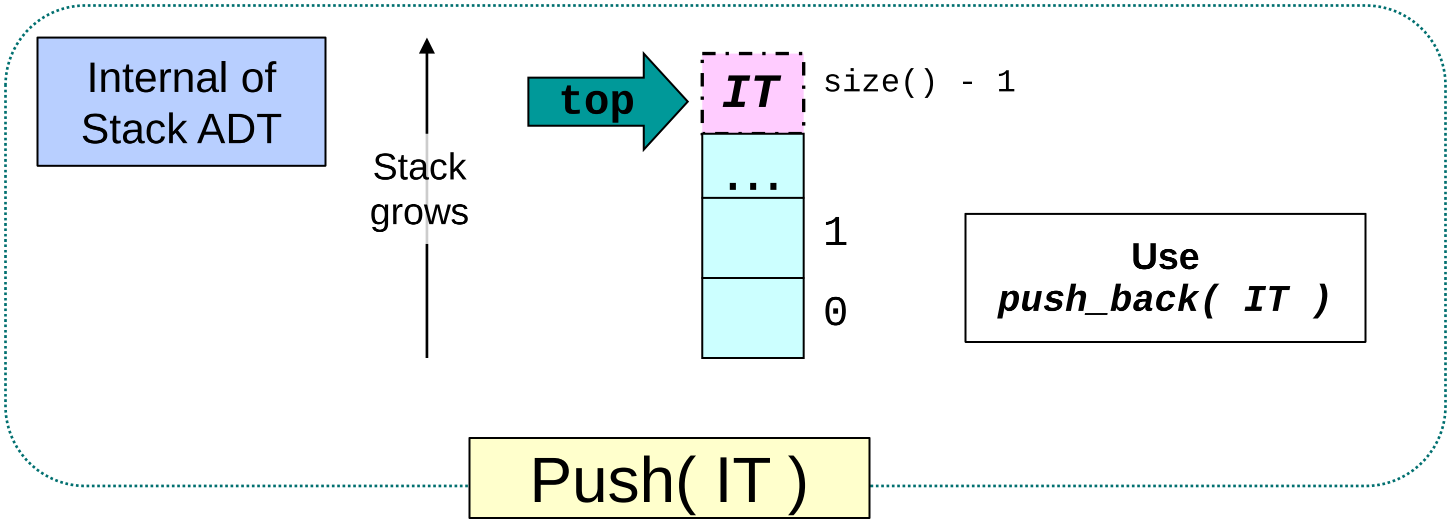
STL Vector can simplify your code

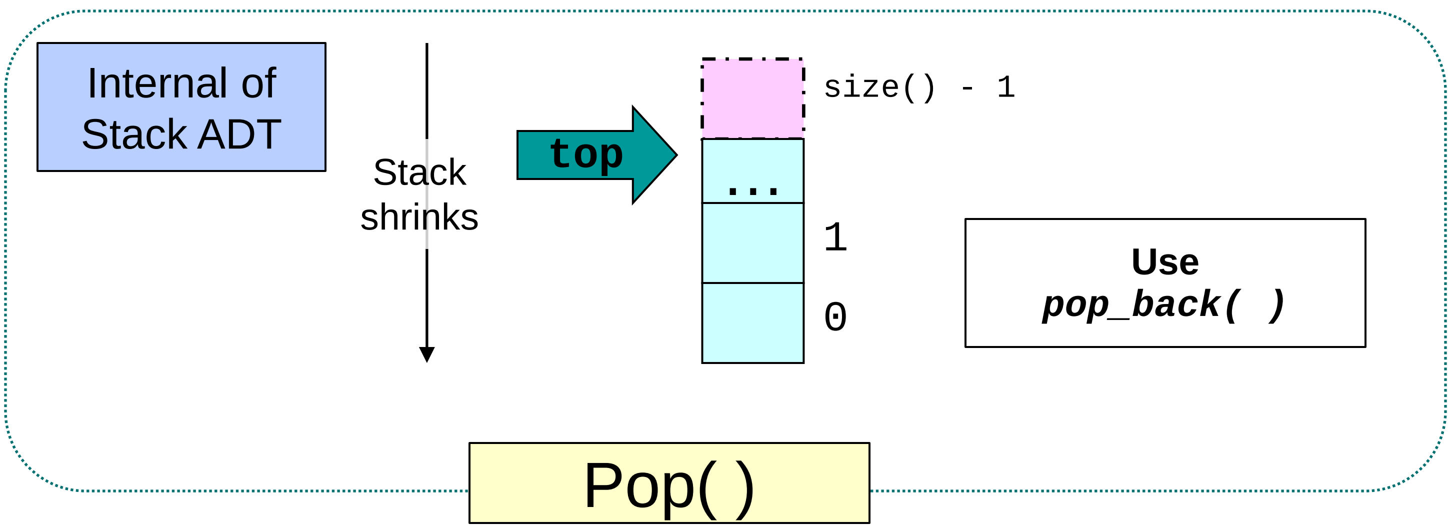
Stack ADT: Using STL Vector

- STL Vector has the following capabilities:
  - Add/remove the last item
    - ***push_back()*** and ***pop_back()***
    - Very efficient
  - Use iterator to add/remove item from any location
    - Not efficient
    - Quite cumbersome (need to setup and move iterator)

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

Stack ADT: Using STL Vector (Illustration)





Stack ADT (STL vector) : C++ Specification

```
#include <vector>
using namespace std;

template <typename T>
class Stack {
   public:
     Stack();

     bool isEmpty() const;
     int size() const;

     void push(const T& newItem) throw (SimpleException);

     void pop() throw (SimpleException);
     void pop(T& stackTop) throw (SimpleException);


     void getTop(T& stackTop) const
                          throw (SimpleException);


private:
      vector<T> _items;
};
```

We need STL Vector.

Methods from slide 6. No change.

The only private declaration.

StackV.h

Implement Stack ADT (STL vector) : 1/2

```
template<typename T>
Stack<T>::Stack()
{ }

template<typename T>
bool Stack<T>::isEmpty() const
{
    return _items.empty();
}

template<typename T>
int Stack<T>::size() const
{
    return _items.size();
}

template<typename T>
void Stack<T>::push( const T& newItem )
    throw (SimpleException)
{
    _items.push_back(newItem);
}
```

No need to initialize anything.

We use **methods from vector** class to help us

StackV.cpp

Implement Stack ADT (STL vector) : 2/2

```
template<typename T>
void Stack<T>::getTop(T& stackTop) const
    throw (SimpleException)
{
    if ( _items.empty() )
        throw SimpleException("Stack is empty on getTop()");
    else
        stackTop = _items.back();
}

template<typename T>
void Stack<T>::pop(T& stackTop)
    throw (SimpleException)
{
    if ( _items.empty() )
        throw SimpleException("Stack is empty on pop()");
    else {
        stackTop = _items.back();
        _items.pop_back();
    }
}
```

Code for the other version of **`pop()`** is not shown as it is very similar.

`StackV.cpp`

STL Stack

STL has a built-in stack ADT

STL Stack: Specification

```
template <typename T>
class stack {
public:
      bool empty() const;
      size_type size() const;
      T& top();
      void push(const T& t);
      void pop();
};
```

- Very close to our own specification ☺
- See example which highlights the differences
  - Especially the `top()` method

STL Stack: Example Usage

```
#include <stack>
#include <iostream>
using namespace std;

int main()
{
  stack<int> s;

   s.push(5);
   int &j = s.top();

   s.push(3);

   j++;
   cout << "top: " << s.top() << endl;

   s.pop();
   cout << "After pop, top: " << s.top() << endl;
     ... ...
}
```

Output:

```
top: 3
After pop, top: 6
```

Stack Applications

LIFO? Is it good for anything?

Stack Applications

- Many useful applications for stack:
  - Bracket Matching
  - Towers of Hanoi
  - Maze Exploration
- More "Computer Science" inclined examples (for your own reading):
  - Base-N number conversion
  - Postfix evaluation
  - Infix to postfix conversion

Stack Application 1

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

- 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 stack is empty or the stack top bracket does not agree with the closing bracket, complain and exit
    - Else continue
- If the stack is not empty after we read through the whole string
  - The input is incorrect

Bracket Matching: Implementation (1)

```
bool check_bracket( string input )
{
     stack<char> sc;
     char current;
     bool ok = true;

     for (unsigned int pos = 0;
                              ok && pos < input.size(); pos++){
        current = input[pos];
        switch (current){
            case '{':
                sc.push('}'); //Question: Why are we pushing the
                break;        //         closing bracket here??
            case '[':
                sc.push(']');
                break;
            case '(':
                sc.push(')');
                break;
            ... ...
```

Bracket Matching: Implementation (2)

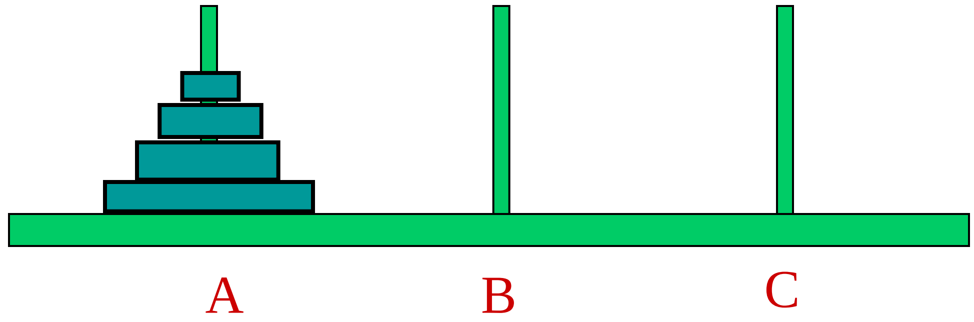
```
            case '}':
            case ']':
            case ')':
                if (sc.empty())      //missing open bracket
                   ok = false;
                else {
                     if (sc.top() == current)//matched!
                        sc.pop();
                     else                   //mismatched!
                         ok = false;
                }
                break;
        }
    }

    if (sc.empty() && ok){      //make sure no leftover
       return true;
    } else
       return false;
}
```

Stack Application 2

Towers of Hanoi

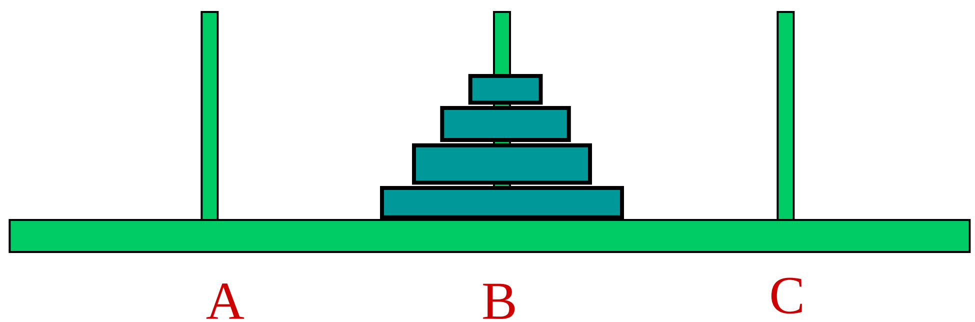
Towers of Hanoi : Description

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
- Just want a simple program to allow a user to play the puzzle instead
  - 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

Towers of Hanoi: Header File

```
#include <stack>

using namespace std;

class ToH
{
public:
    ToH(unsigned int nDiscs = 10);

    bool move(int from, int to);

    void display();

private:
    stack<int> _poles[3];
    unsigned int _nDiscs;

};
```

ToH.h

Towers of Hanoi: Implementation 1/2

```
ToH::ToH(unsigned int nDiscs)
{
    _nDiscs = nDiscs;

    for (int i = _nDiscs; i > 0; i--)
        _poles[0].push(i);
}

ToH::display()
{
   //code not shown, left as a challenge
}
```

ToH.cpp

Towers of Hanoi: Implementation 2/2

```
bool ToH::move(int from, int to)
{
     if ( from == to )
        return true;     //pretend we have moved

     if ( from < 0 || from > 2 || to < 0 || to > 2 )
        return false;

     if ( _poles[from].empty() )
        return false;

     if ( (!_poles[to].empty()) &&
              ( _poles[from].top() > _poles[to].top()) )
        return false;

     _poles[to].push( _poles[from].top() );
     _poles[from].pop();

     return true;
}
```

ToH.cpp

Towers of Hanoi: Sample Usage

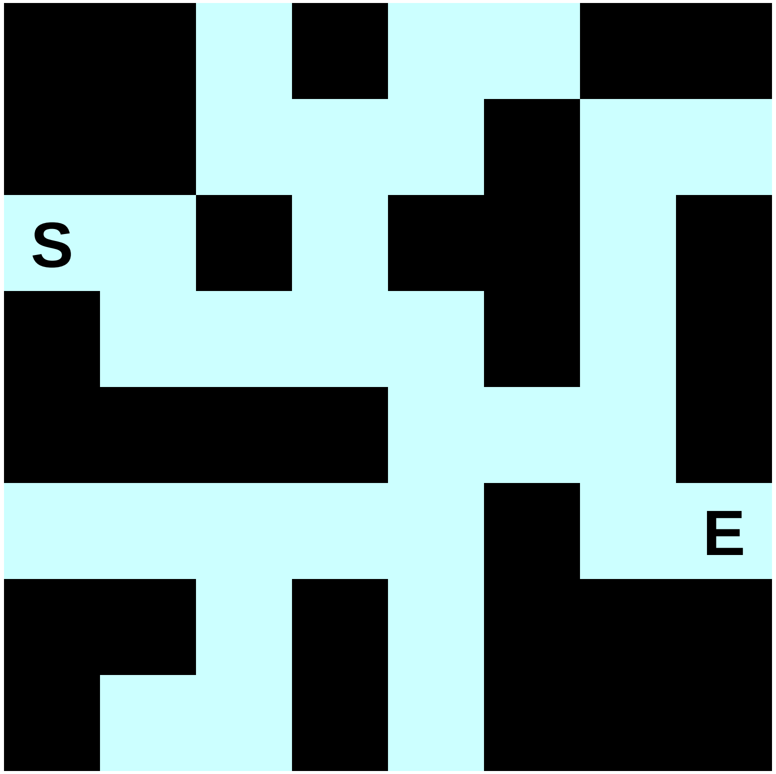
A simple program to allow a human player to play the puzzle.

```
... ...
int main()
{
    ToH t(3);
    int from, to;

    t.display();
    do {
        cout << "Move from: ";
        cin >> from;
        cout << "To: ";
        cin >> to;

        if (from != -1 && to != -1){
            if (t.move(from, to)){
               cout << "Move ok!" << endl;
            } else
               cout << "Cannot move!!" << endl;
        }
        t.display();
    } while (from != -1 && to != -1);
    ... ...
}
```

Stack Application 3

Exploring a Maze

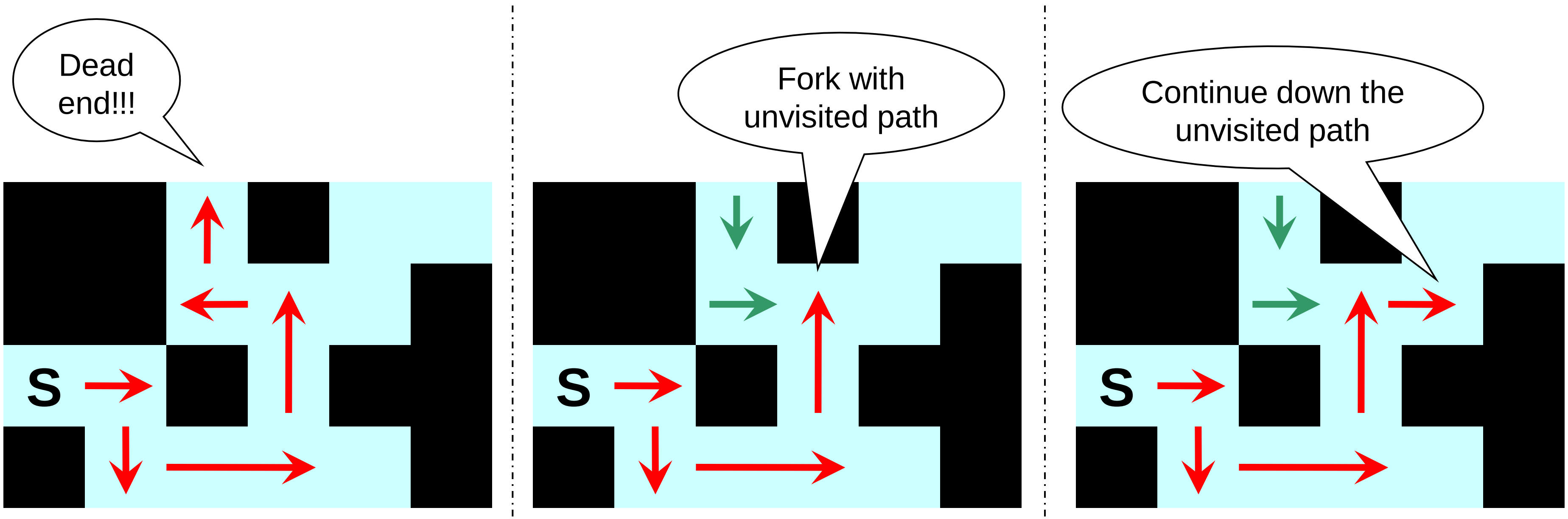
Exploring Maze: Description



- 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 reach a dead end?

Exploring Maze: Some ideas

- When we reach a dead end
  - Restarting 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 an N x N 2d array
  - Each square has a unique **`(row, column)`** coordinate
- Let's encode the directions of movement as an enumeration:

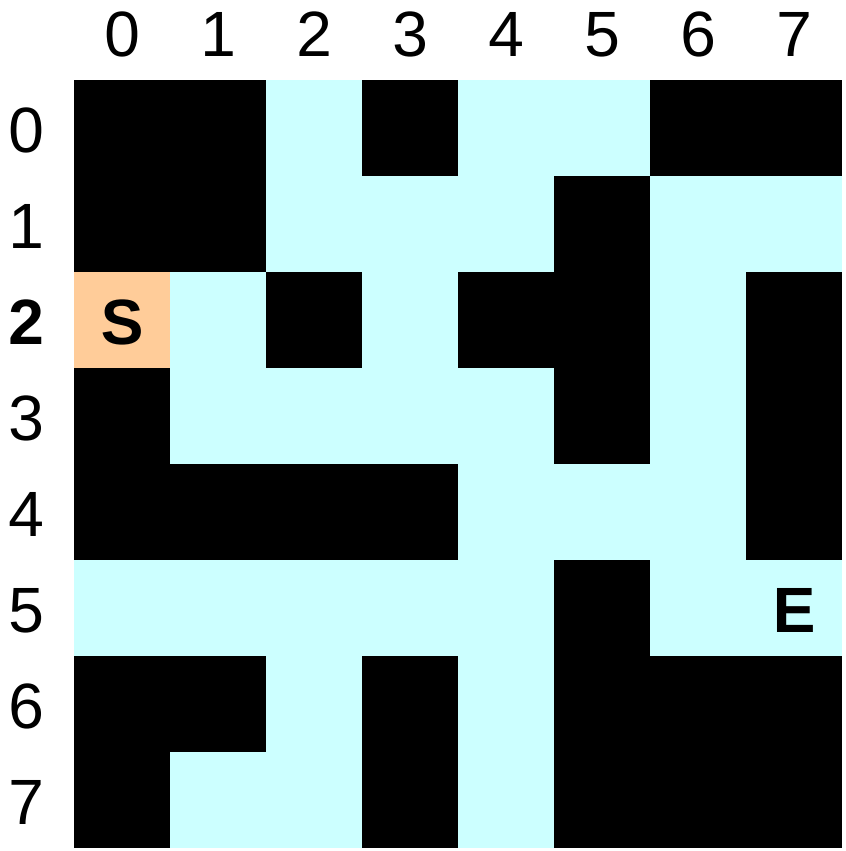
```
enum Direction {Up, Left, Down, Right, NoDir};
```

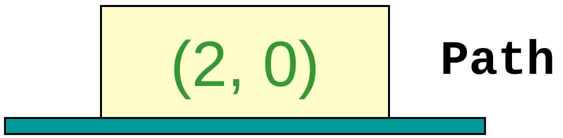
- Each square will know about:
  - Which direction is unvisited
  - Assume a method **`getUnvisitedDir()`** is implemented for this purpose
- When a square has multiple unvisited exits:
  - We visit them in the order of the **`enum`** above
- The path travelled is kept as a **stack of coordinates**

Exploring Maze: Pseudo Code

```
1.  Path = empty
2.  done = false      //are we at the end yet?
3.  Path.push(coordinate of S)
4.  while (Path is not empty && not done)
      CurSq = Path.top( )   //where are we now?
      NewDir = CurSq.getUnvisitedDir( )
      if (NewDir == NoDir)        //dead end!
         Path.pop( )              //move back one square
      else                        //there is an exit
         NewSq = CurSq.move(NewDir)
         Path.push(coordinate of NewSq)
         if (NewSq == E)          //Yes! We reached the end!
            done = true
```

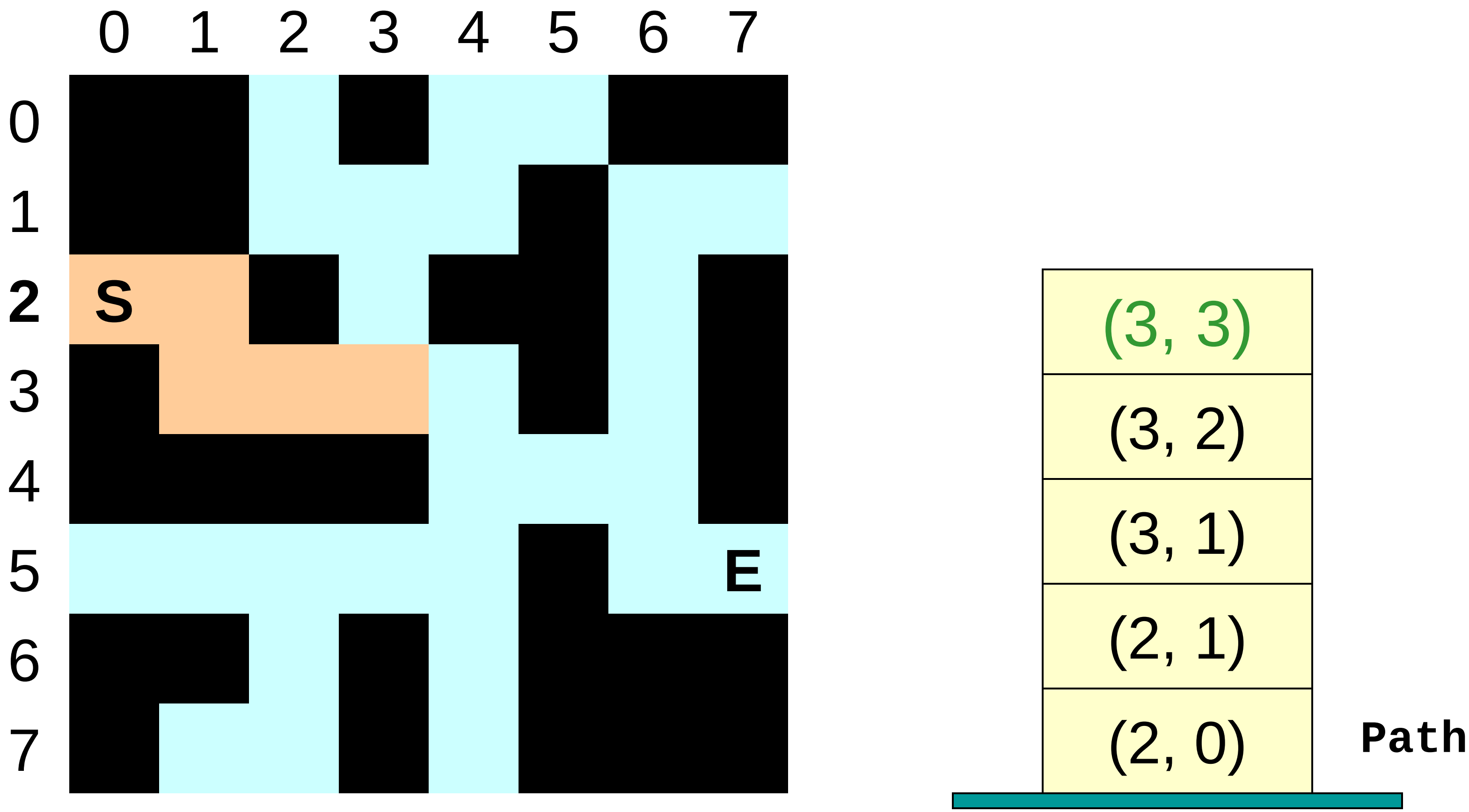
Exploring Maze: Test Run …. (1)





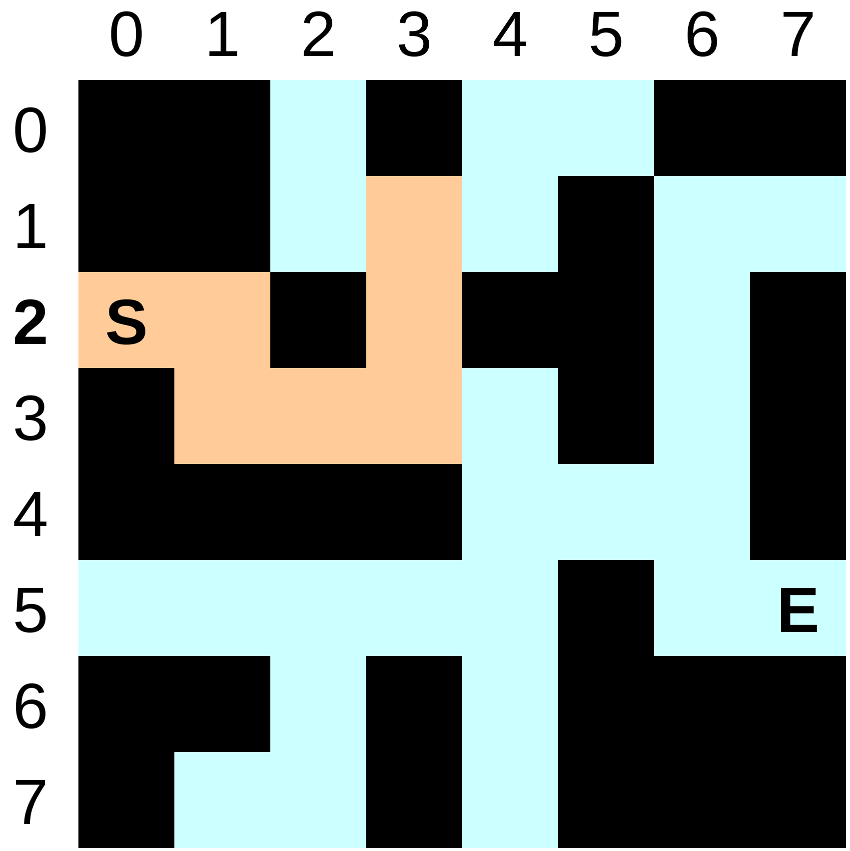
- Just started at (2, 0)

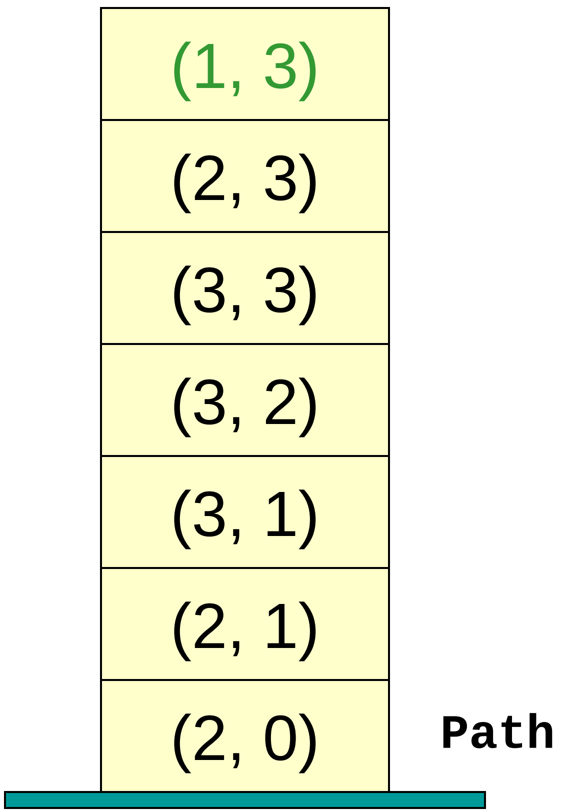
Exploring Maze: Test Run ….(2)



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

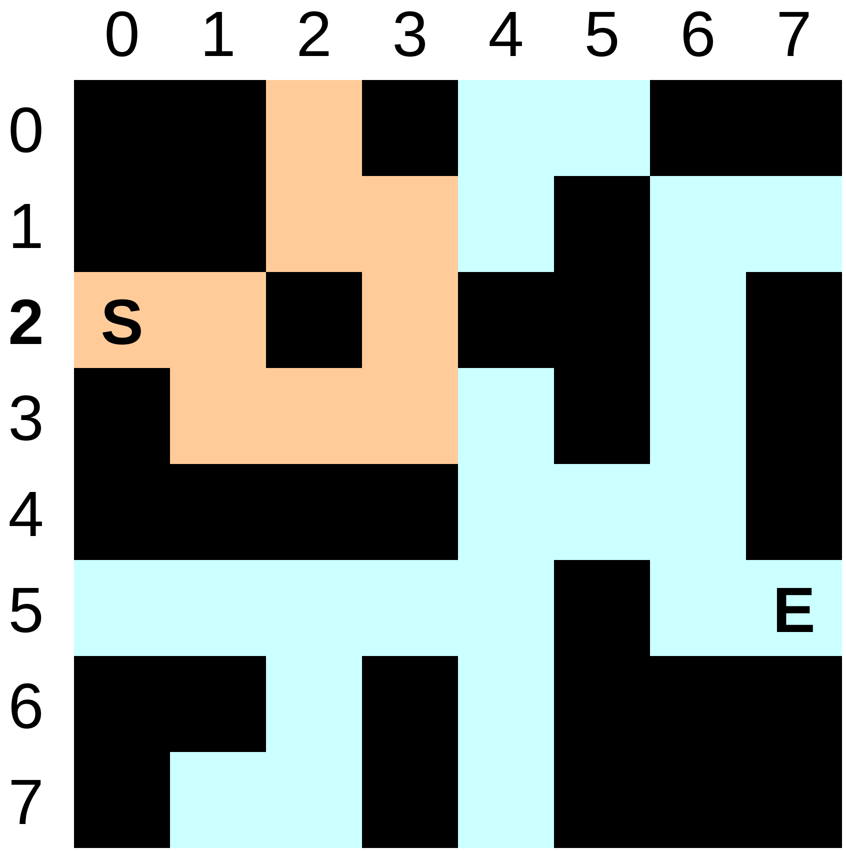
Exploring Maze: Test Run ….(3)

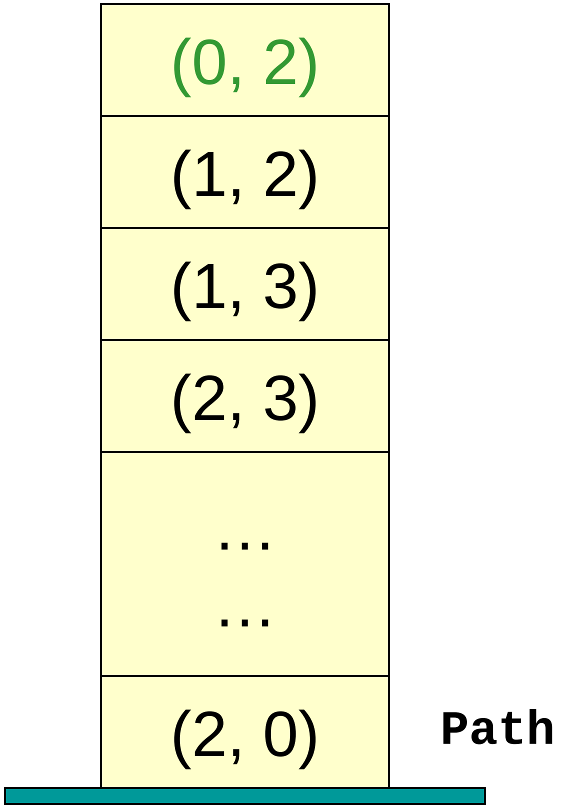




- Multiple exits at (1, 3)
  - go Up is impossible, so go Left is the 2nd choice

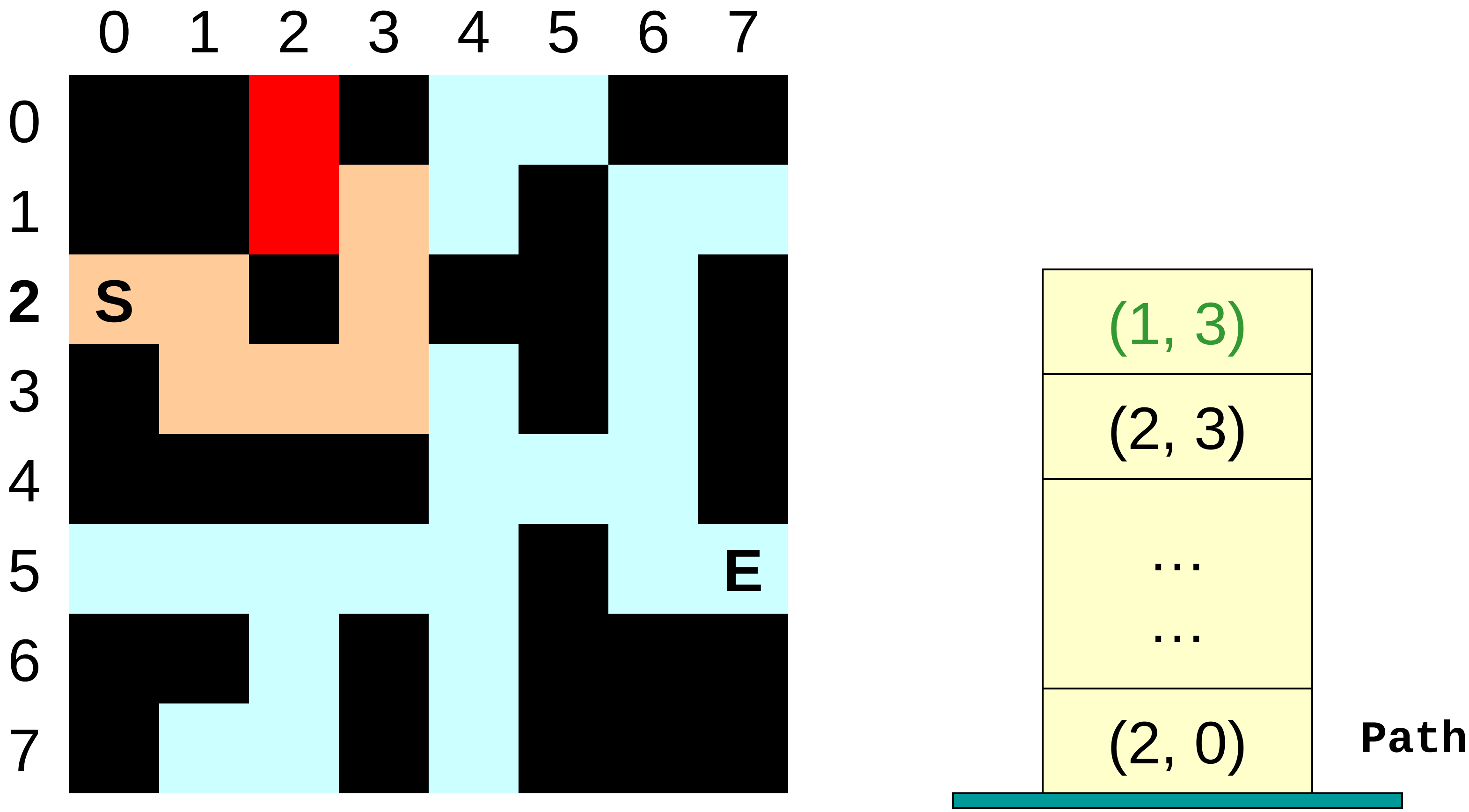
Exploring Maze: Test Run ….(4)





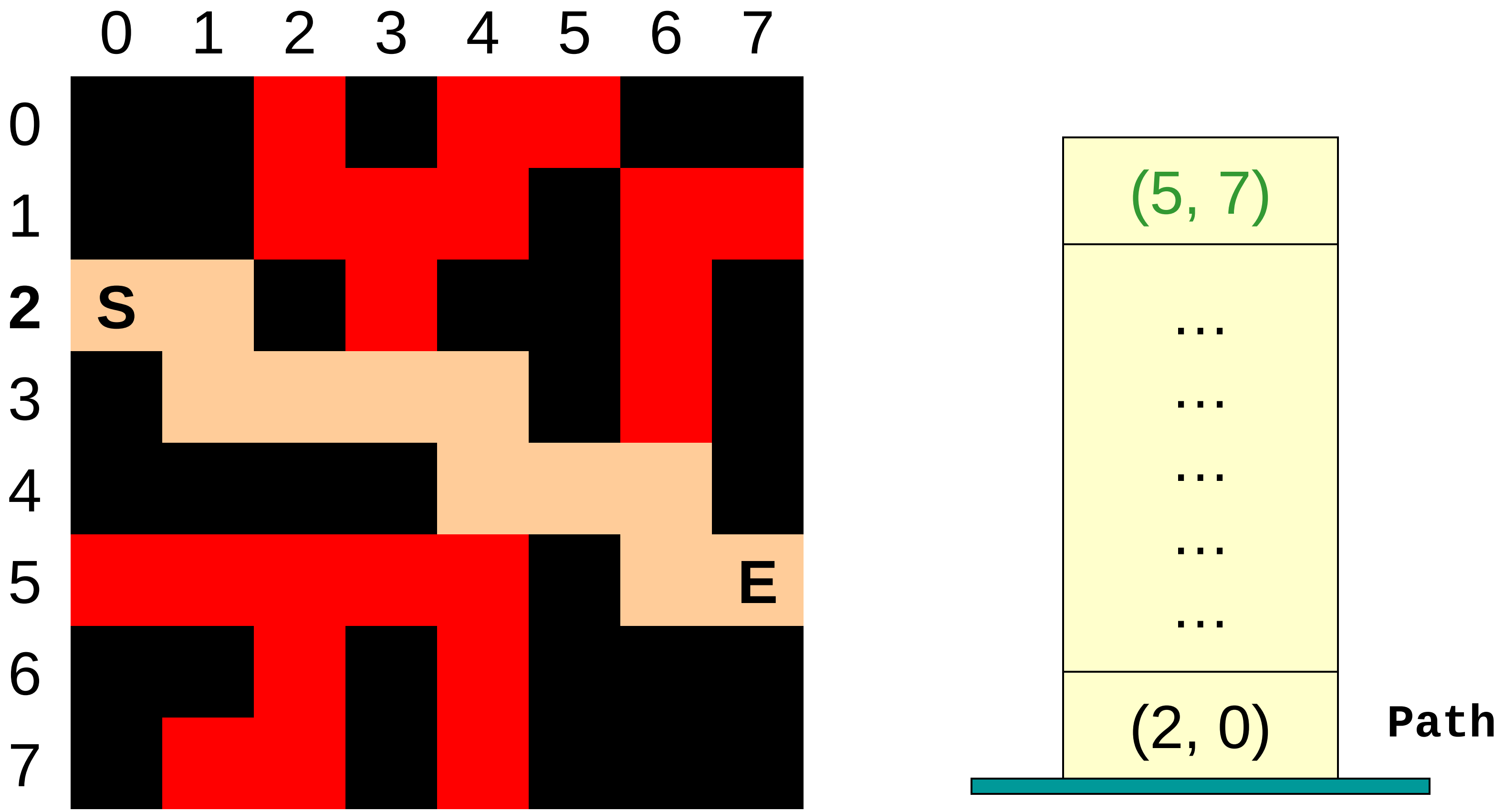
- No exits from (0, 2)
  - Back trace: pop until a square with unvisited exits

Exploring Maze: Test Run ….(5)



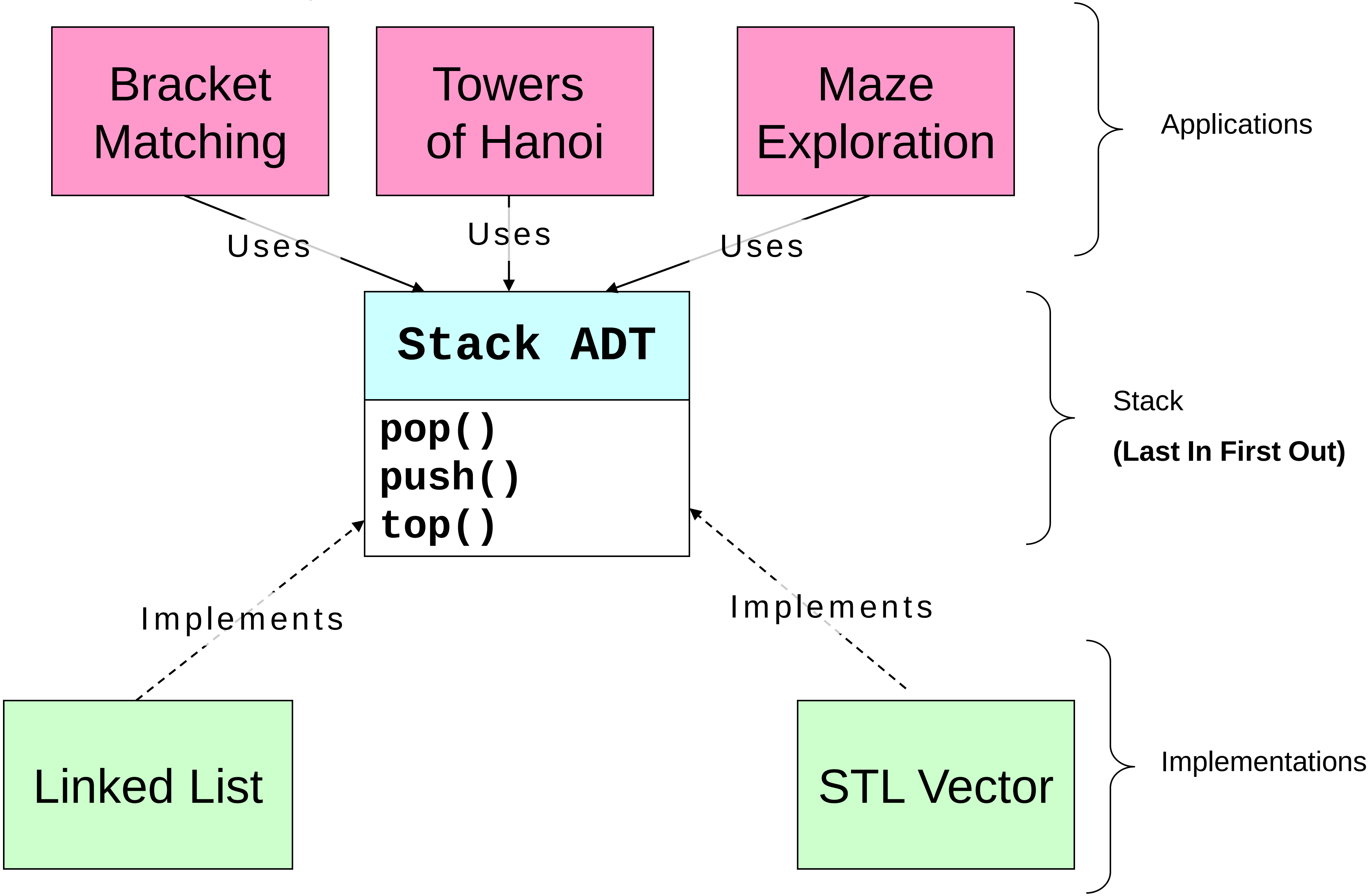
- Back to (1,3) after several pops
  - Up, Left, Down all impossible, going Right to (1,4)

Exploring Maze: Test Run (much later)



Note: algorithm travelled the whole maze to find the exit

Summary


Bracket Matching
Towers of Hanoi
Maze Exploration
Applications
Uses
Uses
Uses
Stack ADT
pop()
push()
top()
Stack
(Last In First Out)
Implements
Implements
Linked List
STL Vector
Implementations