Lecture 7a
Stack ADT

A Last-In-First-Out Data Structure

Lecture Overview

- Stack
  - Introduction
  - Specification
  - Implementations
    - Linked List :O:O
    - STL vector :O
    - STL stack ☺
  - Applications
    - Bracket Matching
    - Infix to Postfix Conversion

Stack: A Specialized List

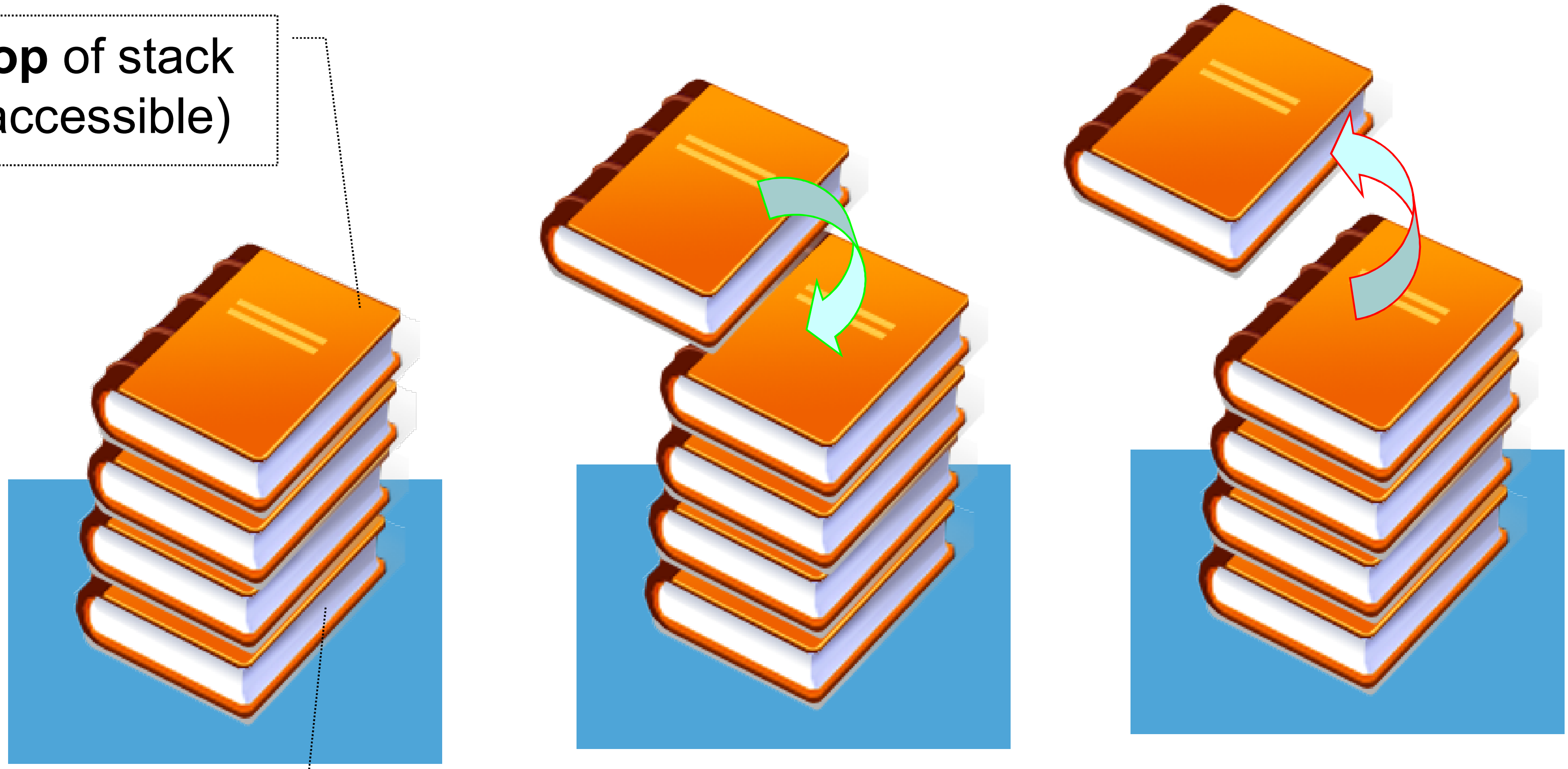
- List ADT (Lecture 6) allows user to manipulate (insert/retrieve/remove) item at **any position within the sequence of items**
- There are cases where we only want to consider a few specific positions only
  - e.g. only the first/last position
  - Can be considered as special cases of list
- **Stack** is one such example
  - Only manipulation at the **first (top) position** is allowed
- **Queue** (Lecture 7b) is another example
  - Only manipulation at the **first (head)** and **last (tail) position** are allowed

What is a Stack

- Real life examples
  - 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(T newItem);
  void top(T& stackTop) const;
  void pop();

private:
  // Implementation dependant
  // See subsequent implementation slides
};
```

Stack ADT is a template class (our previous List ADT in Lecture 6 can also be made as template class)

New C++ feature: const means this function should not modify anything, i.e. a 'getter' function, your compiler will check it

Stack ADT: Implementations

- Many ways to implement Stack ADT, we will see
  - Linked List implementation
    - Study the best way to make use of linked list
    - Will go through this in detail
  - STL vector implementation
    - Make use of STL container vector
    - Just a quick digress
  - Or just use STL stack ☺

- Learn how to weight 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?

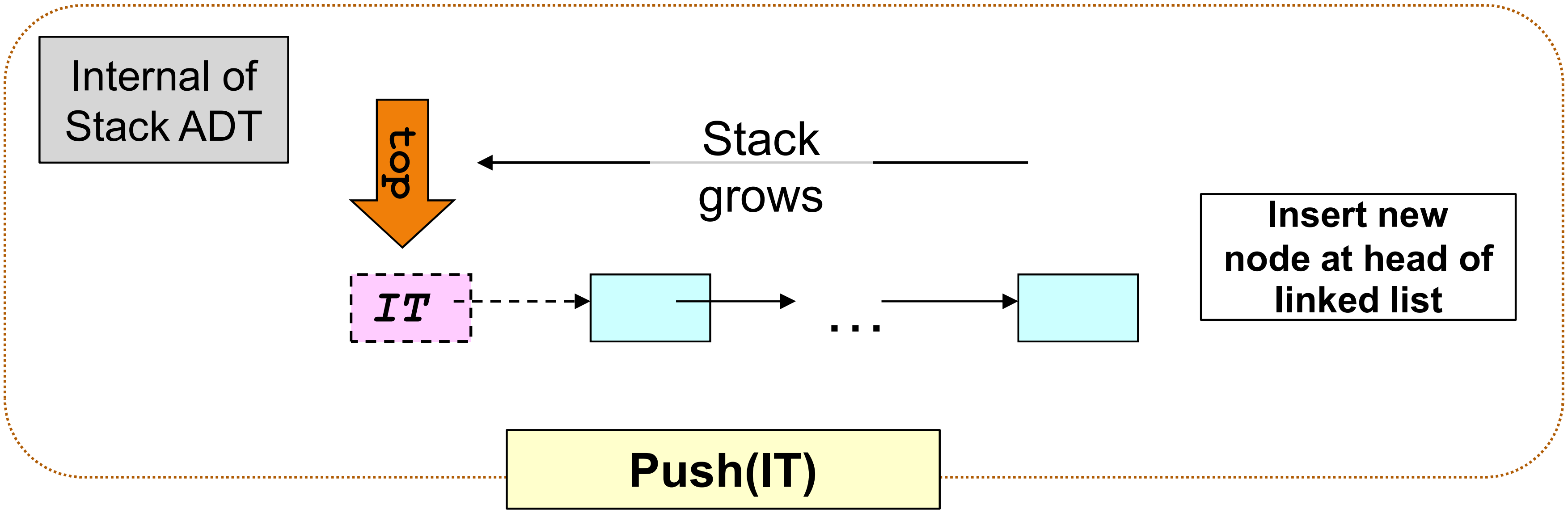
- Internal data structure like array, linked list, etc. are usually very flexible
  - Make sure you use them in the best possible way

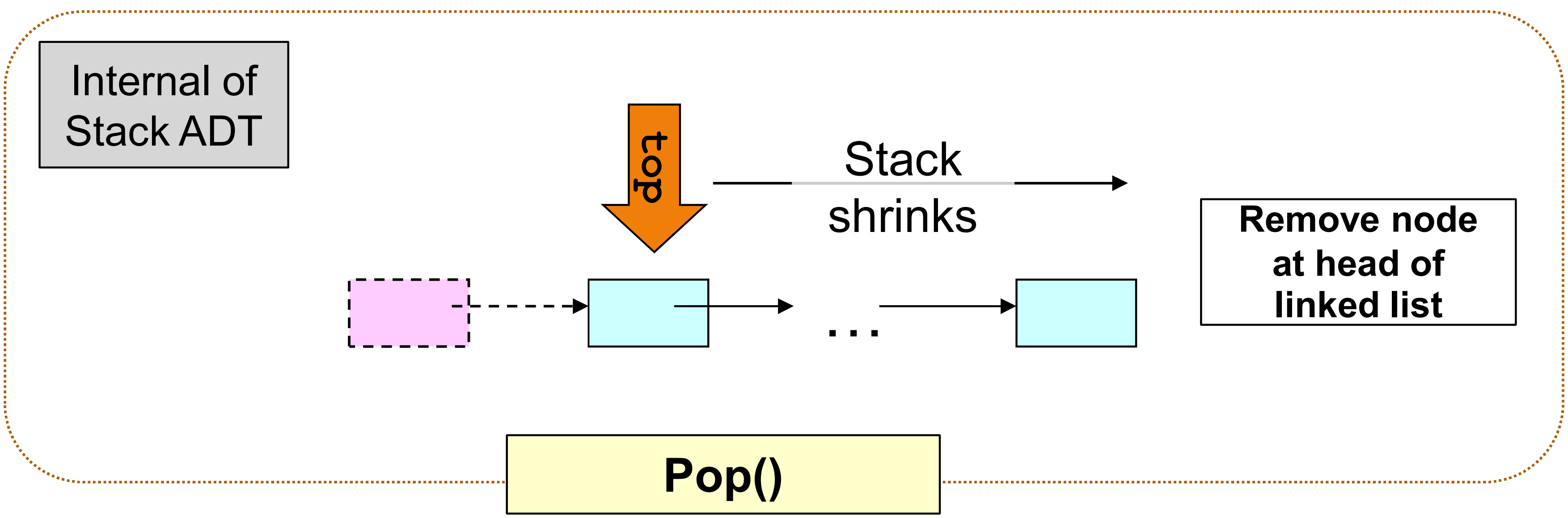
Stack ADT using Linked List

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?**
  - Will Doubly Linked List, Circular Linked List, or Tailed Linked List help for Stack ADT implementation?

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);
  void getTop(T& stackTop) const;
  void pop();

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

Yes, we reuse List ADT from L6, but our L6 code is not on template class so we violate the OOP rule ☹

StackLL.h

Implement Stack ADT (Linked List): 1/3

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

template <typename T>
class StackLL {
public:
  StackLL() : _size(0), _head(NULL) {}

  ~StackLL() {
    while (!isEmpty())
      pop();
  }

  bool isEmpty() const {
    return _size == 0; // try modify something here,
  }                    // you'll get compile error

  int size() const {
    return _size;
  }
```

Make use of own methods to clear up the nodes

StackLL.h, expanded

Implement Stack ADT (Linked List): 2/3

```
void push(T newItem) {
   ListNode* newPtr = new ListNode;
   newPtr->item = newItem;
   newPtr->next = _head;
   _head = newPtr;
   _size++;
 }

 void top(T& stackTop) const {
   if (isEmpty())
     throw string("Stack is empty on top()");
   else {
     stackTop = _head->item;
   }
 }
```

As we only insert at head position. General insertion code not needed. But yes, we could have just use ListLL code from L6

New C++ feature: Exception handling. We can throw RTE

StackLL.h, expanded

Implement Stack ADT (Linked List): 3/3

```
 void pop() {
    if (isEmpty())
      throw string("Stack is empty on pop()");
    else {
      ListNode* cur;
      cur = _head;
      _head = _head->next;
      delete cur;
      cur = NULL;
      _size--;
    }
  }

private:
  struct ListNode {
    T item;
    ListNode* next;
  };
  ListNode* _head;
  int _size;
};
```

As we only remove from head position. General removal code not needed. But yes, we could have just use ListLL code from L6

`StackLL.h, expanded`

Stack ADT using STL vector

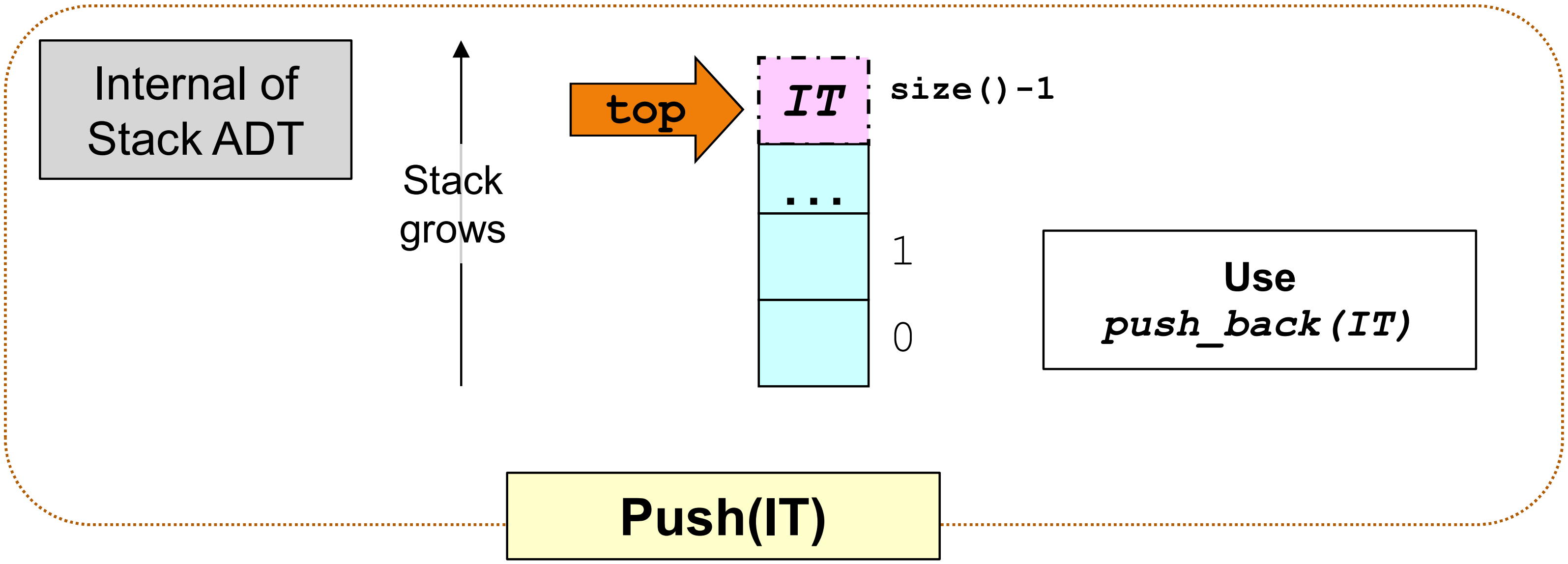
STL vector can be used to implement Stack ADT too

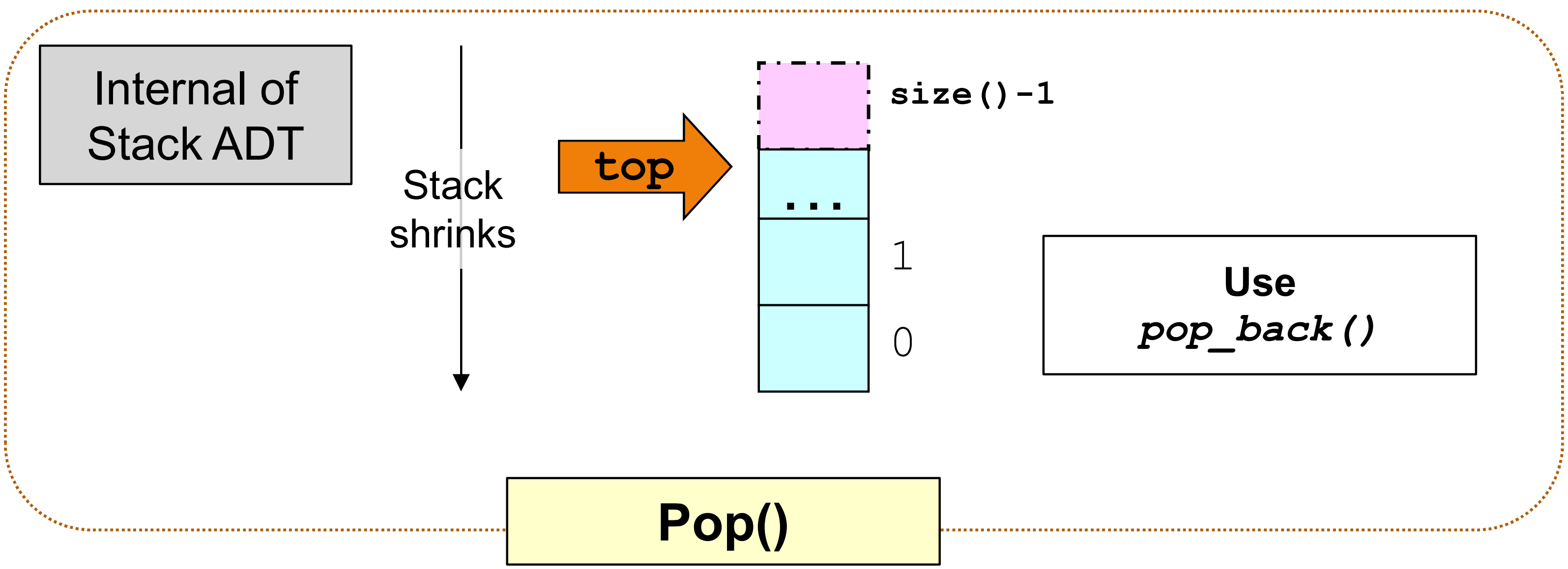
Stack ADT: Using STL vector

- STL vector has the following capabilities
  - Add/remove the last item
    - ***push_back()*** and ***pop_back()***
    - Very efficient, later you will know that this is **O(1)**
  - Use iterator to add/remove item from any location
    - Not efficient
    - Quite cumbersome (need to set up 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 <string>
#include <vector>
using namespace std;

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

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

   void push(T newItem);
   void pop();
   void top(T& stackTop) const;

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

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

template <typename T>
class StackV {
public:
  StackV() {} // no need to do anything
  bool isEmpty() const { return _items.empty(); }
  int size() const { return _items.size(); }
  void push(T newItem) { _items.push_back(newItem); }

  void top(T& stackTop) const {
    if (isEmpty())
      throw string("Stack is empty on top()");
    else
      stackTop = _items.back();
  }
```

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

`StackV.h, expanded`

Implement Stack ADT (STL vector): 2/2

```
  void pop() {
    if (isEmpty())
      throw string("Stack is empty on pop()");
    else
      _items.pop_back();
  }

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

StackV.h, expanded

STL stack

- STL has a built-in stack ADT
- Just use this whenever you need to use Stack ADT
  - http://en.cppreference.com/w/cpp/container/stack

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 ☺
- One difference in top() method

STL stack: Example Usage

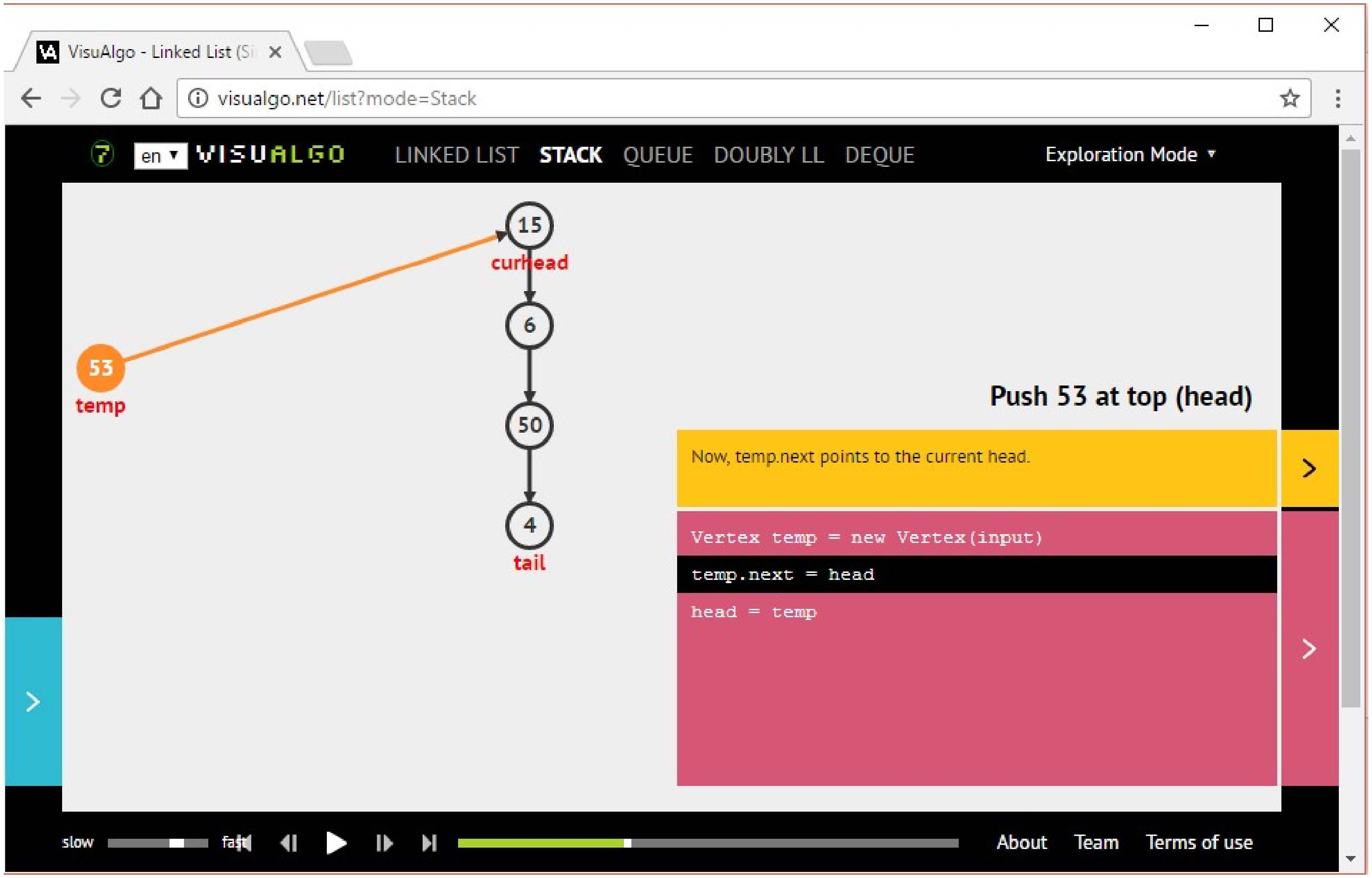
```
//#include "StackLL.h"
//#include "StackV.h"
#include <stack>
#include <iostream>
using namespace std;

int main() {
  //StackLL<int> s;
  //StackV<int> s;
  stack<int> s;
  int t;

  s.push(5);
  s.push(3);
  //s.top(t);
  t = s.top();
  cout << "top: " << t << ", size: " << s.size() << endl;

  s.pop();

  //s.top(t);
  t = s.top();
  cout << "After pop, top: " << t << ", size: " << s.size() << endl;

  s.pop(); // now the stack is empty
  cout << "size: " << s.size() << endl;
  //s.pop(); // will get RTE as stack is empty by now

  return 0;
}
```

Output:

```
top: 3, size: 2
After pop, top: 5, size: 1
size: 0
```

VisuAlgo

- http://visualgo.net/list?mode=Stack
- I use Single Linked List



Stack Applications

Stack Applications

- Many useful applications for stack
  - Bracket Matching
  - Calling a function
    - Before the call, the state of computation is saved on the stack so that we will know where to resume
- We may cover this 2 after we discuss recursion
  - Tower of Hanoi
  - Maze Exploration
- More "computer science" inclined examples
  - 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, i.e. (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 character
     - 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
2. If the stack is not empty after we read through the whole string
   - The input is wrong also

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 left over
    return true;
  else
    return false;
}
```

Stack Application 2

Arithmetic Expression –

Evaluating Postfix Expression

Infix to Postfix Conversion

Application 2: Arithmetic Expression

- Terms
  - Expression: a = b + c * d
  - Operands: a, b, c, d
  - Operators: =, +, –, *, /, %

- **Precedence rules**: Operators have priorities over one another as indicated in a table (which can be found in most books)
  - Example: *, / have higher precedence over +, –.
  - For operators at the same precedence (e.g. * and /), we associate them from left to right

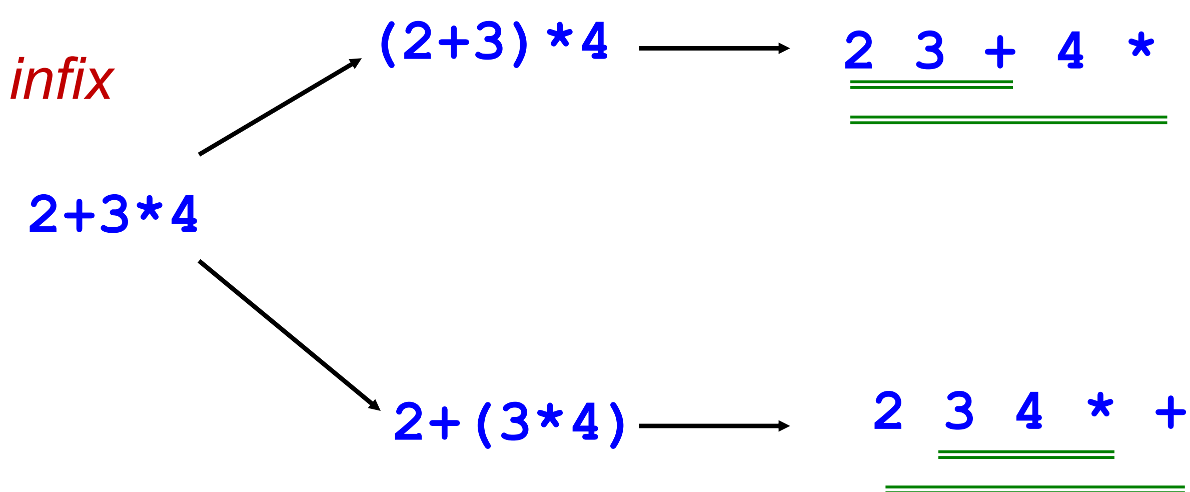
Application 2: Arithmetic Expression

Infix	- operand1	operator	operand2
Prefix	- operator	operand1	operand2
Postfix	- operand1	operand2	operator

Ambiguous, need () or precedence rules

Unique interpretation

postfix

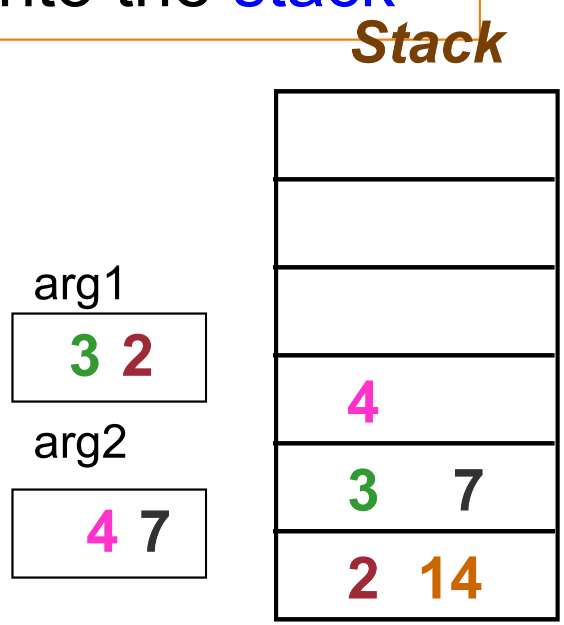


Algorithm: Calculating Postfix Expression with Stack

Create an empty stack
for each item of the expression,
    if it is an operand,
        push it on the stack
    if it is an operator,
        pop arguments from stack;
        perform the operation;
        push the result onto the stack

2 * (3 + 4) → 2 3 4 + *

2 s.push(2)
3 s.push(3)
4 s.push(4)
\+ arg2 = s.pop ()
arg1 = s.pop ()
s.push (arg1 + arg2)
* arg2 = s.pop ()
arg1 = s.pop ()
s.push (arg1 * arg2)



Algorithm: Converting Infix to Postfix

```
String postfixExp = "";
for (each character ch in the infix expression) {
  switch (ch) {
    case operand: postfixExp = postfixExp + ch; break;
    case '(':  stack.push(ch); break;
    case ')':  while ( stack.peek() != '(' )
                    postfixExp = postfixExp + stack.pop();
                    stack.pop(); break;     // remove '('
    case operator:
         while ( !stack.empty() && stack.peek() != '(' &&
               precedence(ch) <= precedence(stack.peek()) )
              postfixExp = postfixExp + stack.pop();
              stack.push(ch); break;
  } // end switch
} // end for
while  ( !stack.empty() )
   postfixExp = postfixExp + stack.pop();
```

Algorithm: Converting Infix to Postfix

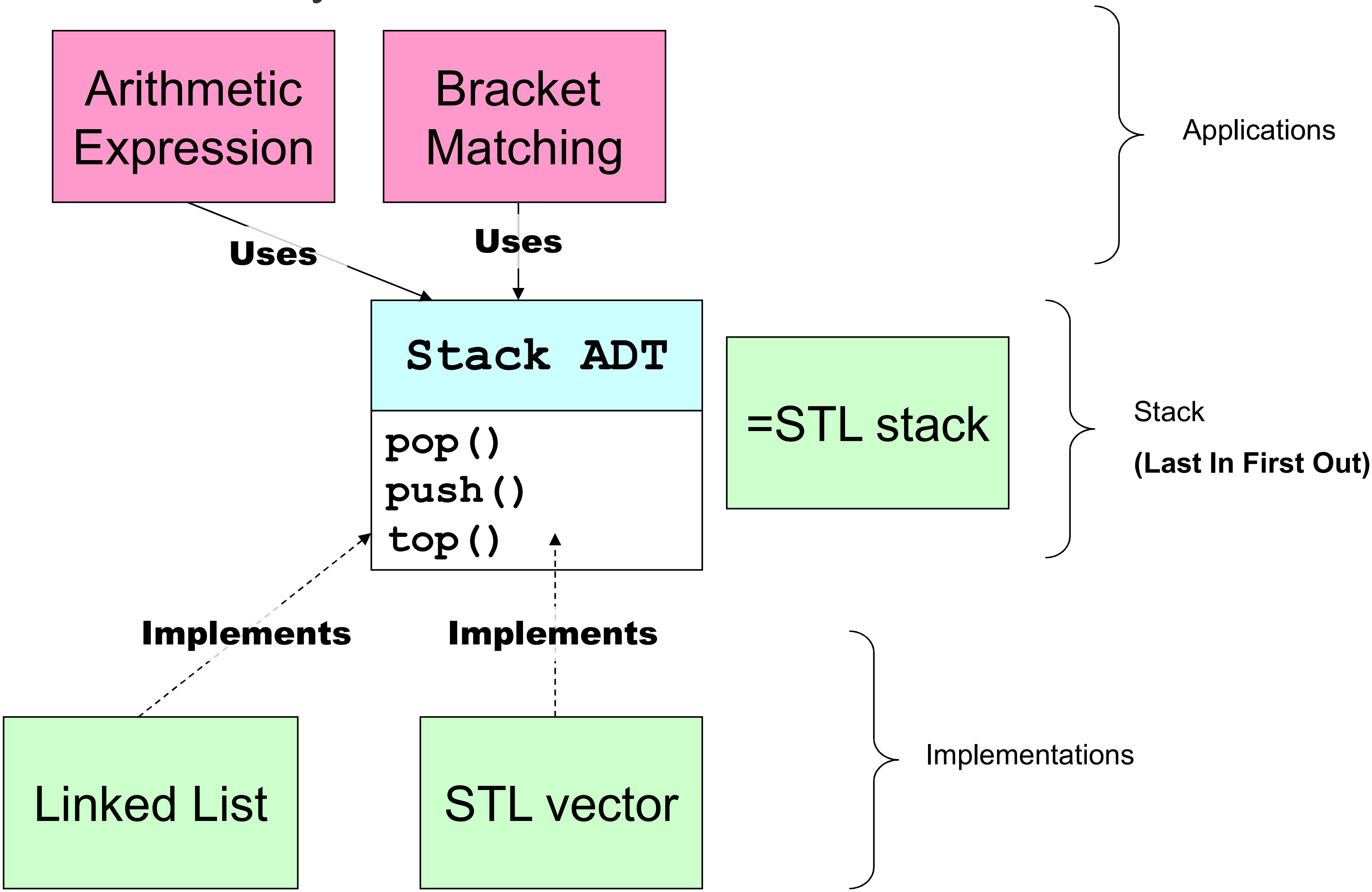
Example: a – (b + c * d) / e

ch	Stack	postfixExp
a		a
–	–	a
(	– (	a
b	– (	a b
+	– (+	a b
c	– (+	a b c
*	– (+ *	a b c
d	– (+ *	a b c d
)	– (+	a b c d *
	– (	a b c d * +
	–	a b c d * +
/	– /	a b c d * +
e	– /	a b c d * + e
		a b c d * + e / –

Move operators from stack to postfixExp until '('

Copy remaining operators from stack to postfixExp

Summary


Arithmetic Expression
Bracket Matching
Uses
Uses
Applications
Stack ADT
pop()
push()
top()
=STL stack
Stack
(Last In First Out)
Implements
Implements
Linked List
STL vector
Implementations