Lecture 22: Stack and Queue

Class page: https://github.com/tsung-wei-huang/cs1410-40

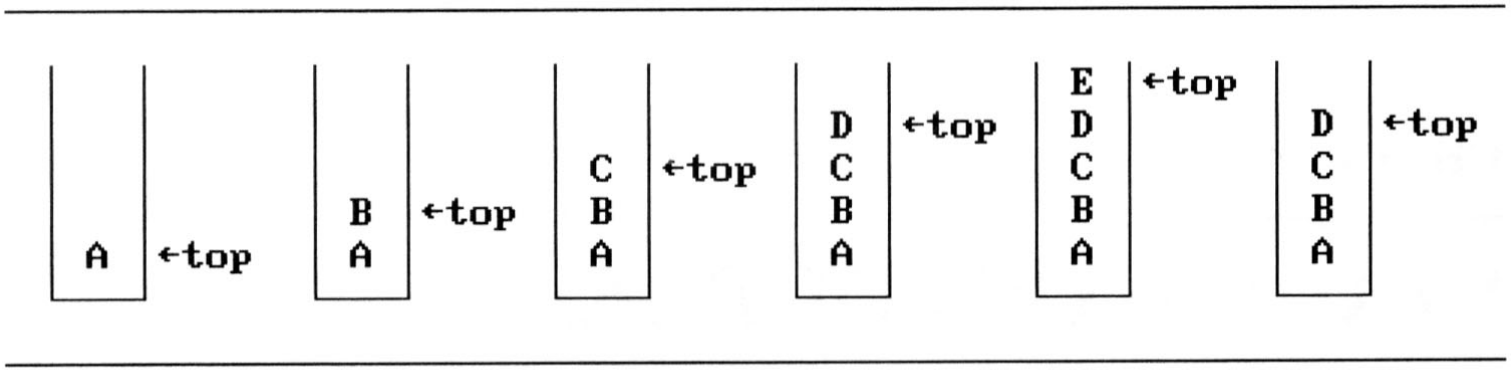
Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Announcement

- **Final Exam starts on 11/30 and ends on 23:59 PM 12/6**
  - Take-home exam, same as midterm
  - Cover all topics
    - 50% concept questions
    - 50% programming questions
  - Free to discuss with your friends and use internet resources
  - Never copy solutions
- **We do not have any more labs**
- **We will still have lectures on 11/30 and 12/2**

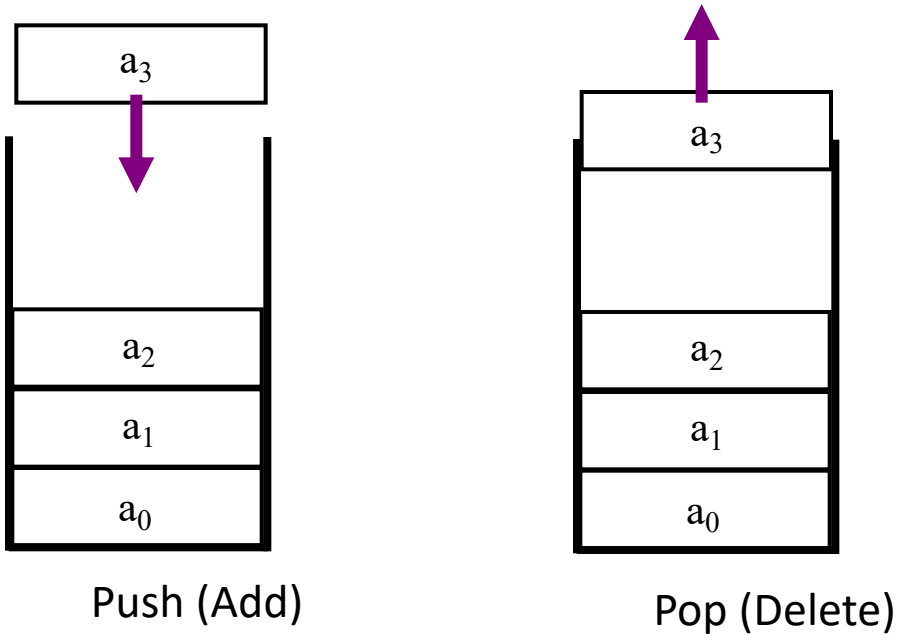
Stack

- A stack is an ordered list in which insertions and deletions are made at one end called the *top*
  - Support **push** and **pop** operations and **top** query
- A stack is also known as a *Last-In-First-Out* (*LIFO*) list
- If we add the elements *A*, *B*, *C*, *D*, *E* to the stack, in that order, then *E* is the first element we delete from the stack



Visualization of Stack

- **Main Subroutine**
  - Push
  - Pop
  - Top



std::stack

- C++ Standard Template Library (STL) stack
  - https://en.cppreference.com/w/cpp/container/stack

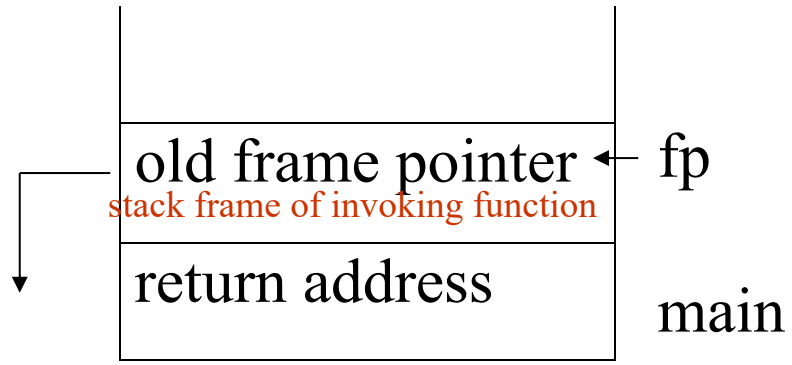
```
/* stack example */
#include <iostream>
#include <stack>

int main()
{
    std::stack<int> stk;
    stk.push(1);
    stk.push(2);
    std::cout<<stk.top();

    /* clear the stack */
    while(!stk.empty())
      stk.pop();
}
```

Application: Recursion Stack Frame

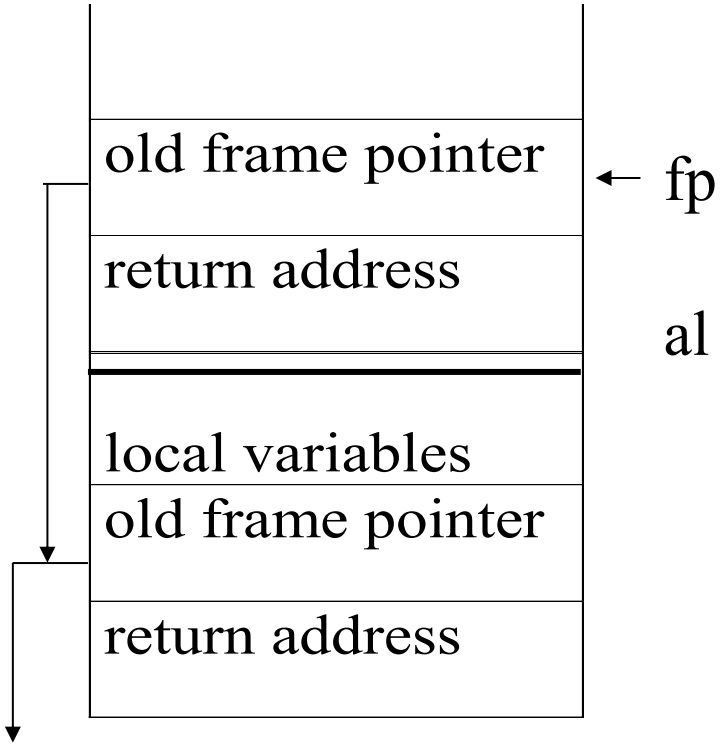
stack frame of recursive function call

(activation record)

fp: a pointer to current stack frame



system stack before a1 is invoked

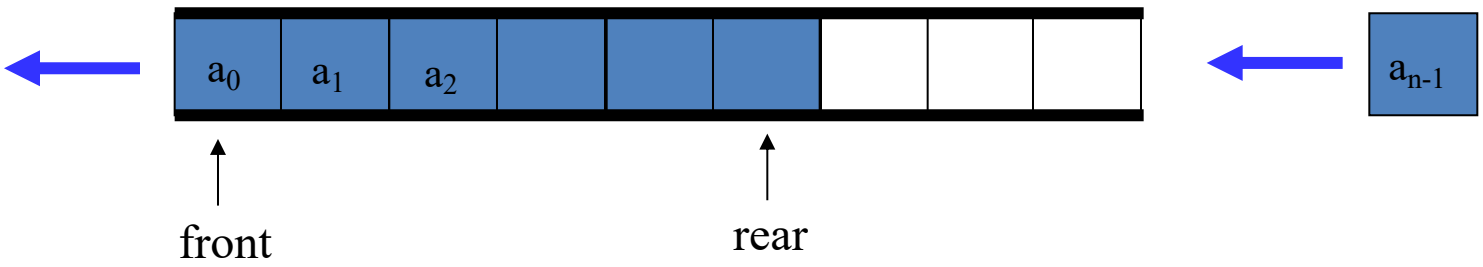


system stack after a1 is invoked

All recursive algorithm can be rewritten iteratively using either flat for-loop or stack

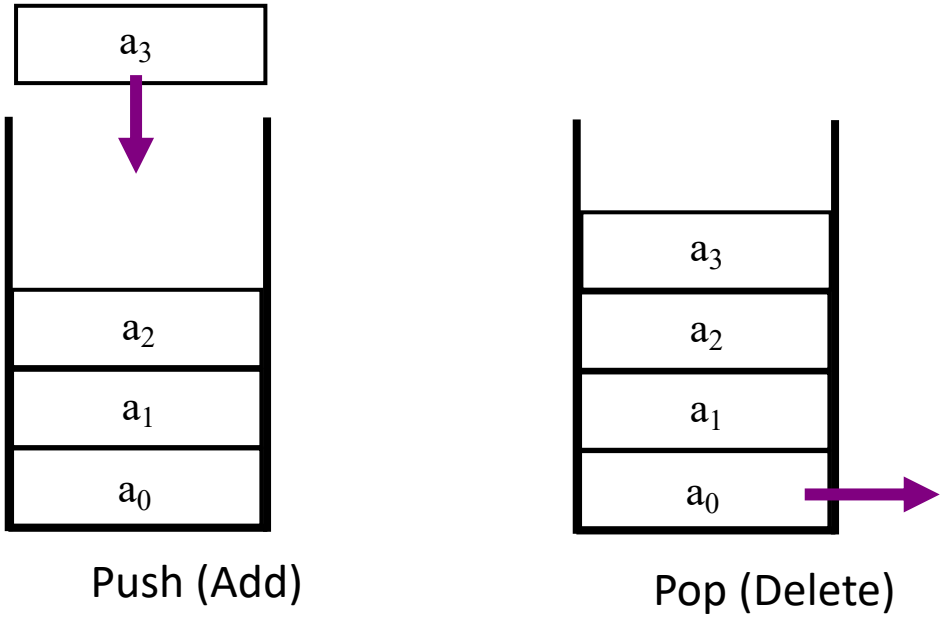
Queue

- A queue is an ordered list in which insertions and deletions are made at one end called the rear and front
  - Support **push** and **pop** operations and **front** query
- A queue is also known as a *First-In-First-Out* (*LIFO*) list
- If we add the elements *A*, *B*, *C*, *D*, *E* to the queue, in that order, then *A* is the first element we delete from the queue



Visualization of Queue

- **Main Subroutine**
  - Push
  - Pop
  - front



Push (Add)

Pop (Delete)

std::queue

- C++ Standard Template Library (STL) queue
  - https://en.cppreference.com/w/cpp/container/queue

```
#include <iostream>
#include <queue>
int main()
{
    std::queue<int> que;
    que.push(1);
    que.push(2);
    std::cout<<que.front();

    /* clear the queue */
    while(!que.empty())
      que.pop();
}
```

Example: Parenthesis Problem

- Given a string of characters '(', ')', '{', '}', '[' and ’]’
- Goal: Determine if the input string is valid.
  - An input string is valid if:
  - Open brackets must be closed by the same type of brackets.
  - Open brackets must be closed in the correct order.
  - Note that an empty string is also considered valid.

()	valid
()[]{}	valid
(]	invalid
([)]	invalid
{[]}	valid

Applications

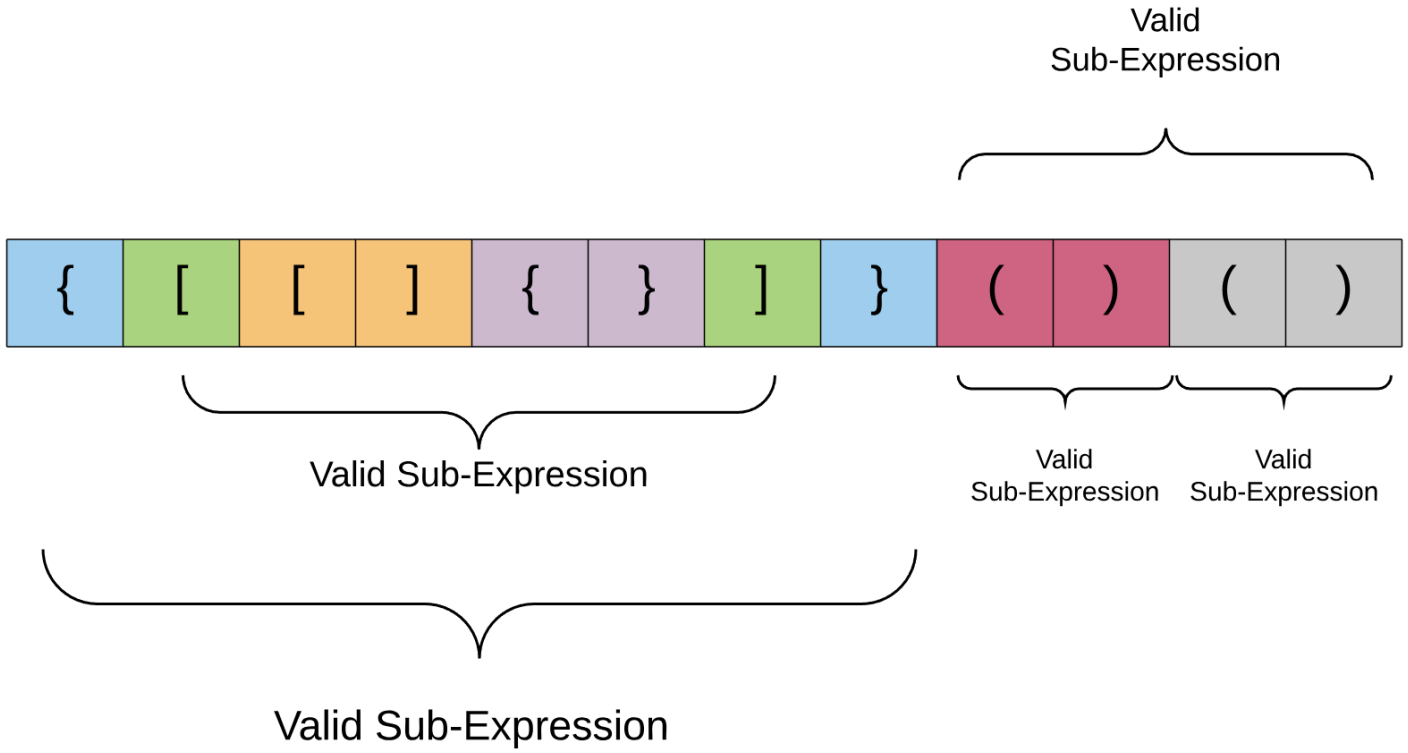
- **A fundamental routine in language compiler**
  - Need to parse a valid mathematical expressions
    - (3+2)*4*((9-6)/6)
    - (double)(1)/(2+7)
  - Need to parse a valid code block
    - int main () { return 0; }
    - void function() {}
    - auto lambda = [] () { my_work(); }
- **Also a very frequently asked question in interview …**

So, by how?

()	valid
()[]{}	valid
(]	invalid
([)]	invalid
{[]}	valid
((((((((()))))))))	valid
()()()()	valid
((((((((	invalid
(((())))	valid
[](){(}}	invalid

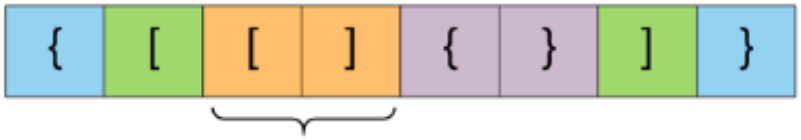
Property

- **A valid expression must imply:**
  - All subexpressions are valid



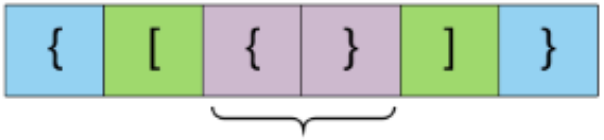
Recursive Validness

- Remove yellow pair



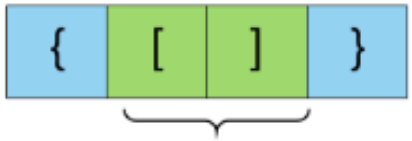
Recursive Validness

- Remove purple pair



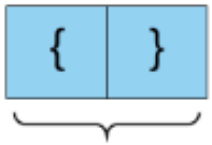
Recursive Validness

- Remove green pair



Recursive Validness

- Every subexpression is valid

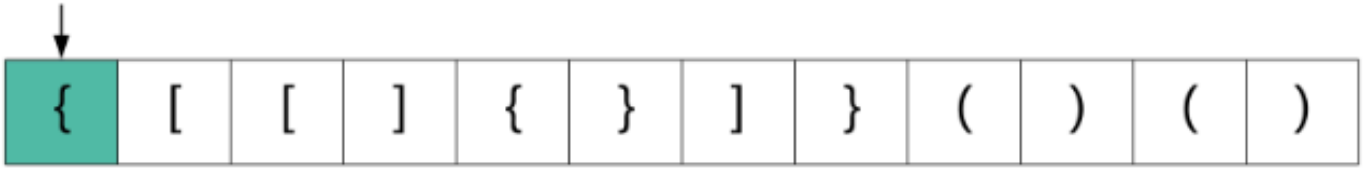


All recursive algorithm can be rewritten iteratively using either flat for-loop or stack

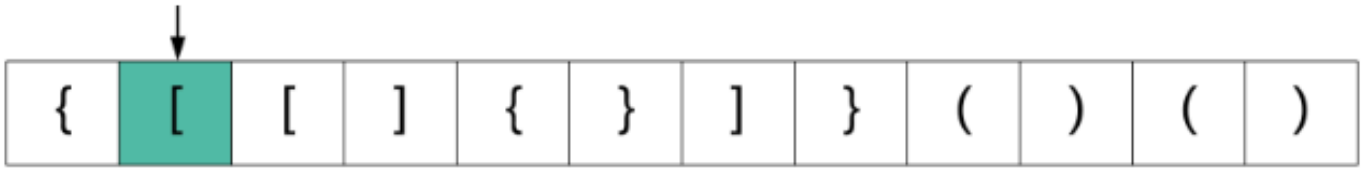
Algorithm

1. Initialize a stack S.
2. Process each bracket of the expression one at a time.
3. If we encounter an opening bracket, we simply push it onto the stack. This means we will process it later, let us simply move onto the **sub-expression** ahead.
4. If we encounter a closing bracket, then we check the element on top of the stack. If the element at the top of the stack is an opening bracket of the same type, then we pop it off the stack and continue processing. Else, this implies an invalid expression.
5. In the end, if we are left with a stack still having elements, then this implies an invalid expression.

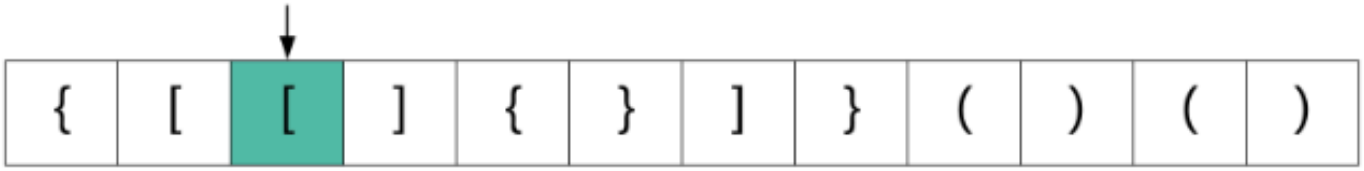
Illustration



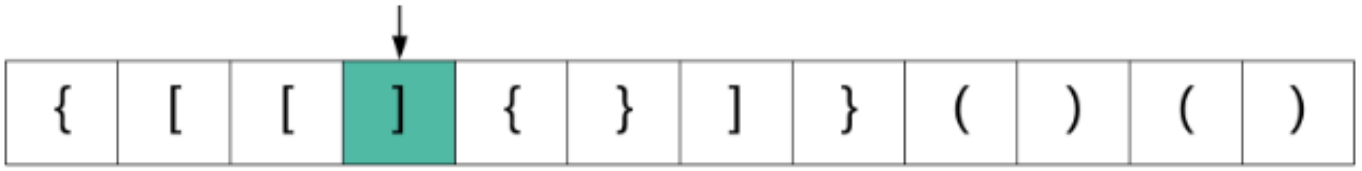
Illustration

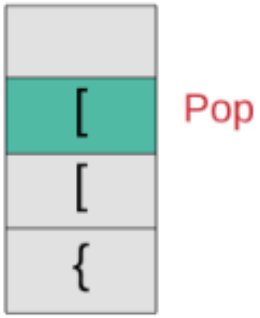


Illustration

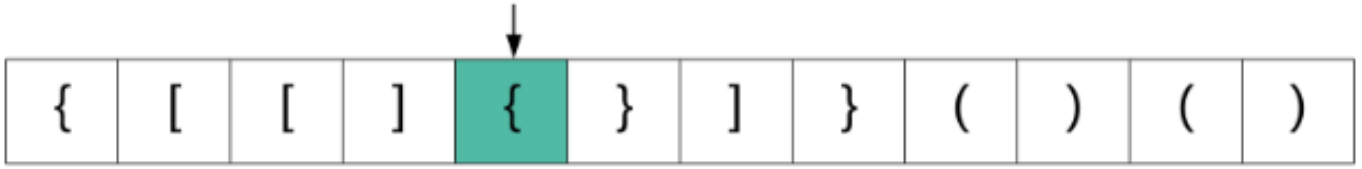


Illustration

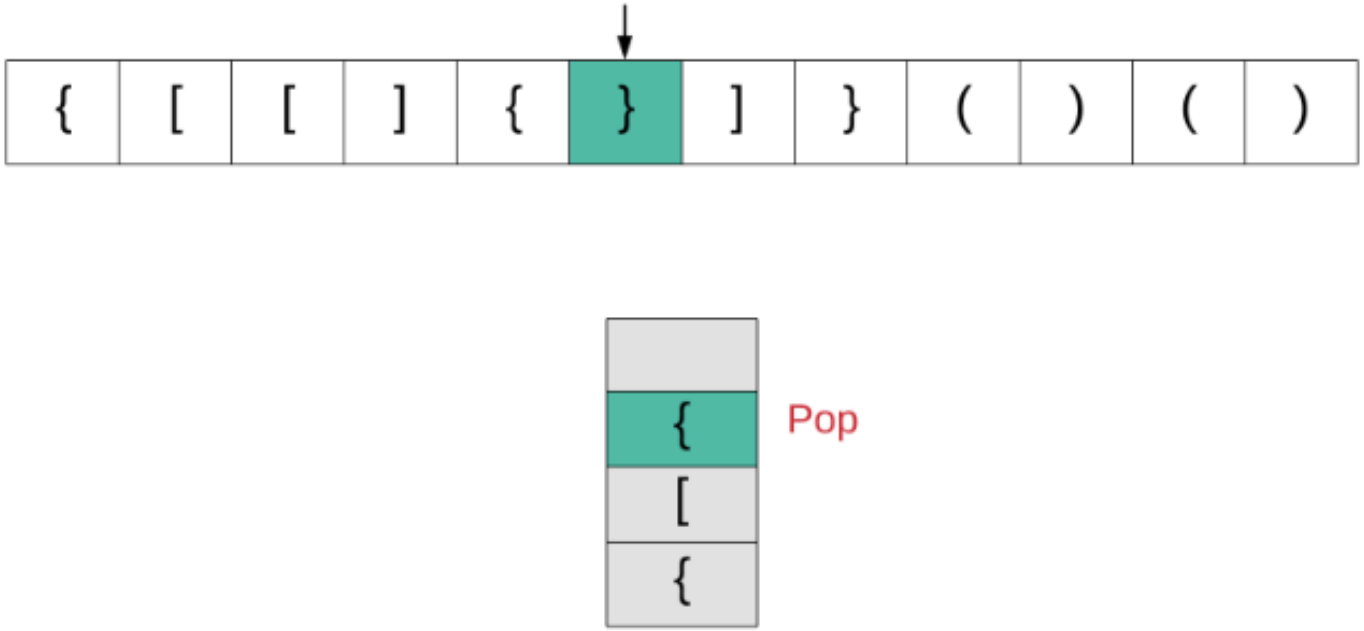




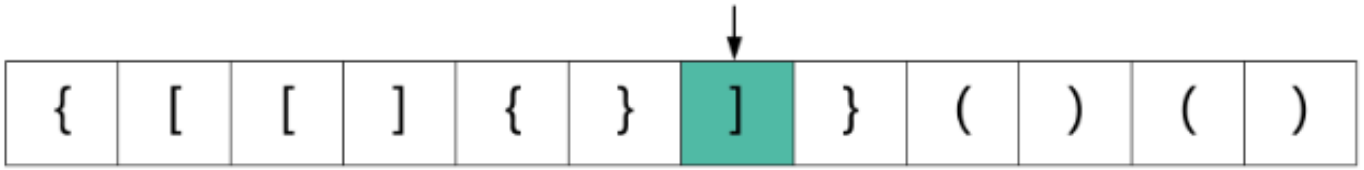
Illustration

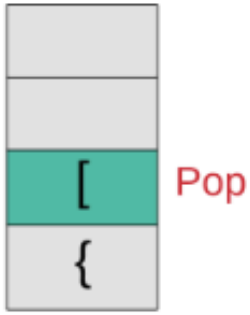


Illustration

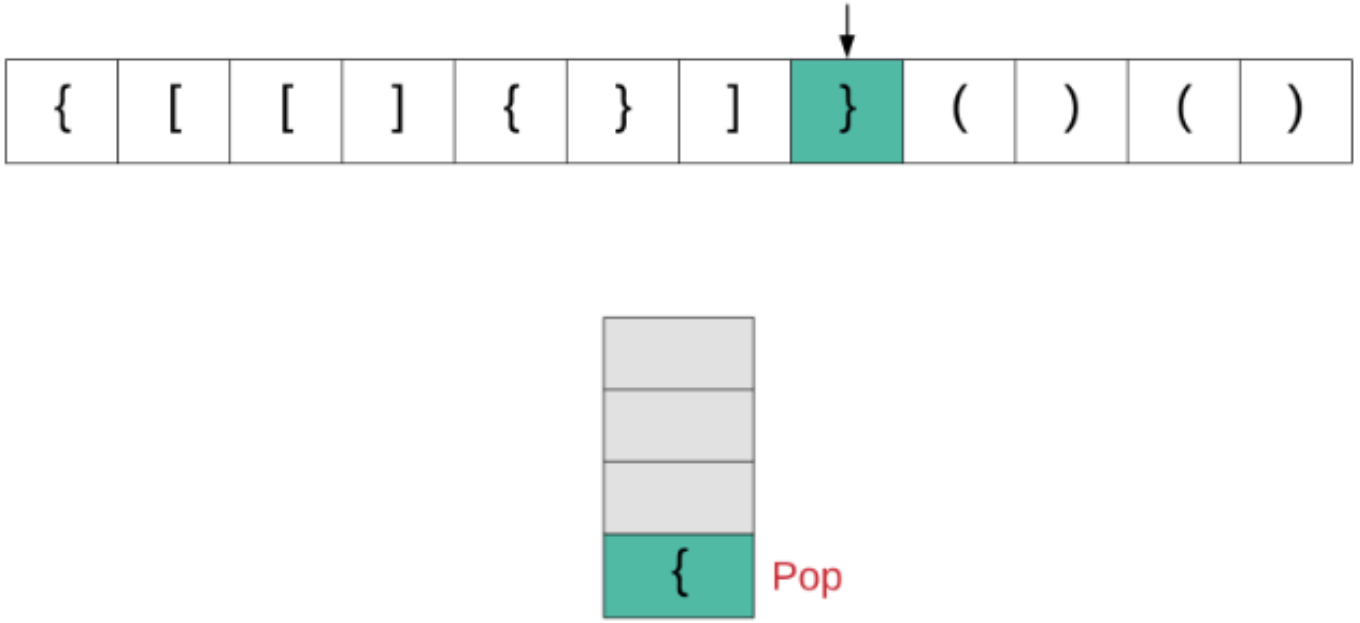


Illustration

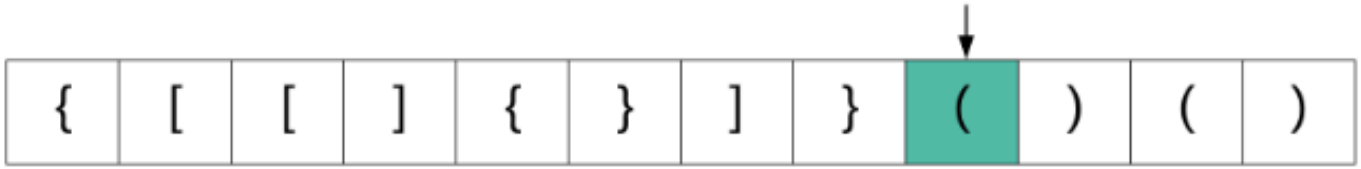




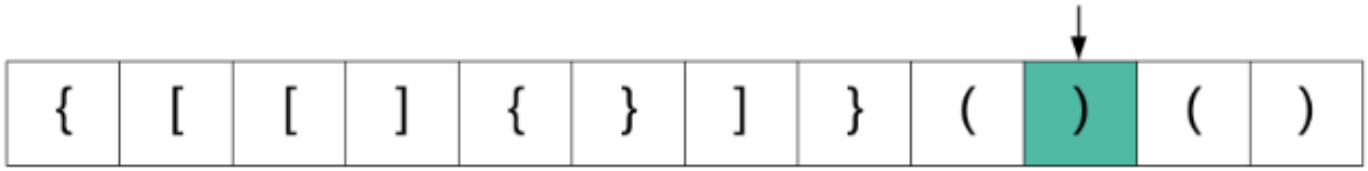
Illustration

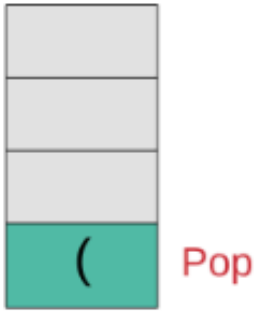


Illustration

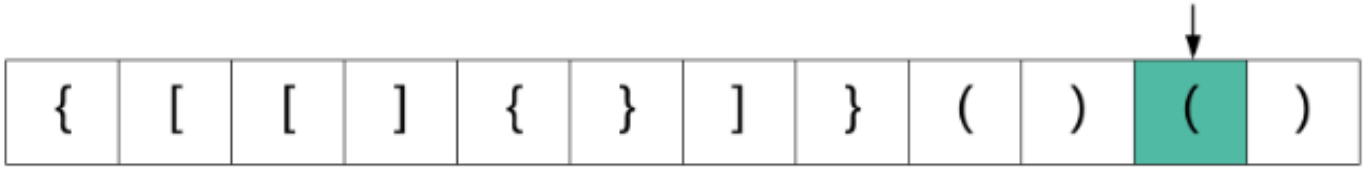


Illustration

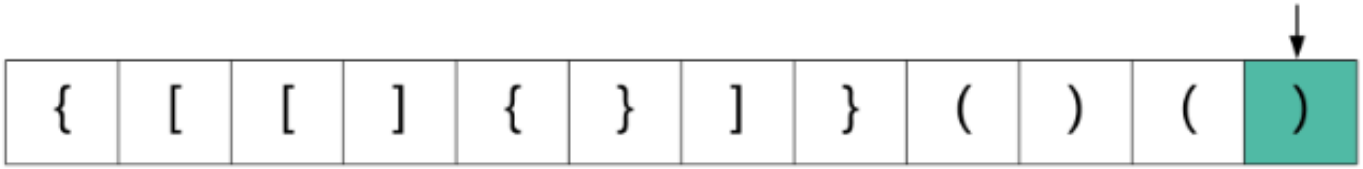


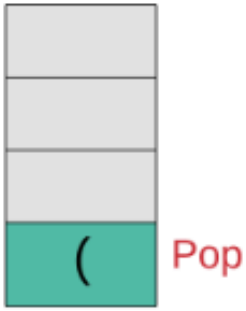


Illustration



Illustration





Complexity

- Time complexity : $O(n)$
  - We traverse the given string one character at a time and push and pop operations on a stack take $O(1)$ time.
- Space complexity : O(n)
  - We push all opening brackets onto the stack and in the worst case, we will end up pushing all the brackets onto the stack. e.g. ((((((((((.