Lecture 6: Queue and Stack

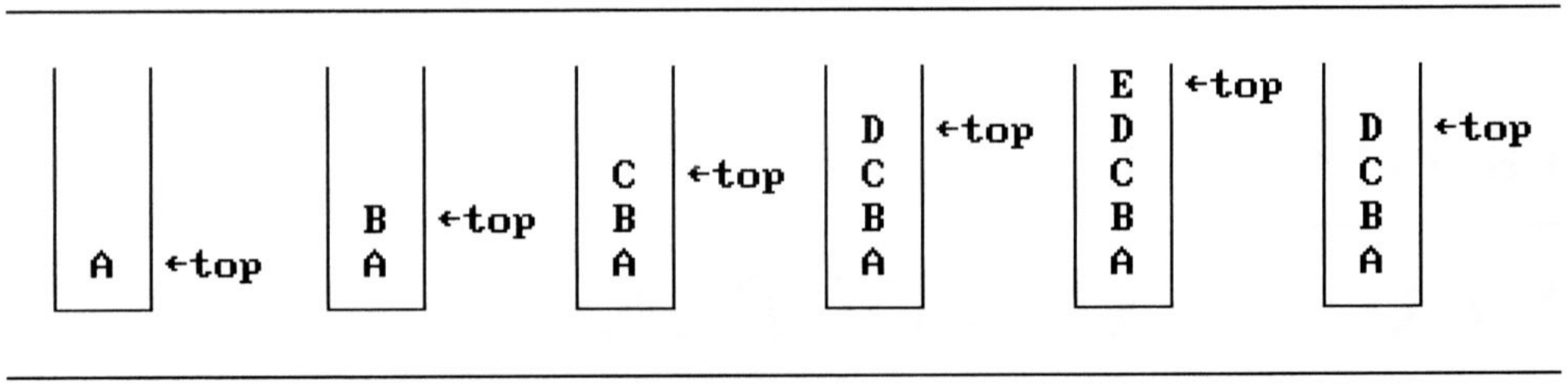
Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

In-class Presentation: 9/14

- **Circuit partition research presentation on 9/14 (in class)**
  - George Karypis and Vipin Kumar, "Multilevel k-way Hypergraph Partitioning," *1999 ACM/IEEE DAC* – ***presented by W-L Lee***
  - Honghua Yang and Martin Wong, "Efficient Network Flow Based Min-Cut Balanced Partitioning," *1994 ACM/IEEE ICCAD* – ***presented by Randy***
  - Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, Kentaro Torisawa, "Automatic Graph Partitioning for Very Large-scale Deep Learning," *2021 IEEE IPDPS 2021* – ***presented by McKay***
- **Instructions**
  - Upload your pptx to https://github.com/tsung-wei-huang/ece5960-physical-design/issues/9
  - Template is available here: https://github.com/tsung-wei-huang/ece5960-physical-design/blob/main/Presentation/template.pptx

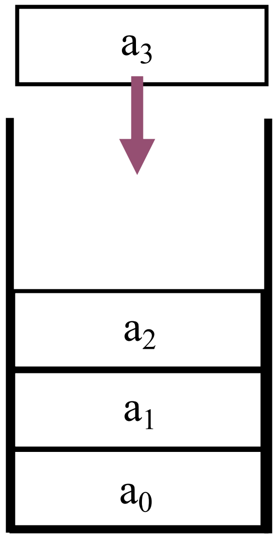
Stack

- **An ordered list supporting insertions and deletions**
  - Push: push an element into the list
  - Pop: pop an element from the list in *Last-In-First-Out* (*LIFO*) order
- If we add the elements *A*, *B*, *C*, *D*, *E* to the stack, in that order, then *E* is the first element we pop from the stack

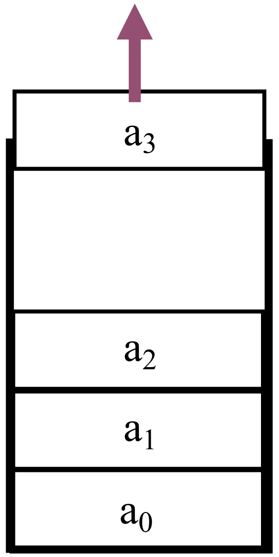


Stack Visualization

- **Three main operations**
  - Push: inserts a new item into the stack
  - Pop: removes the last inserted item from the stack
  - Top: query the top item in the stack



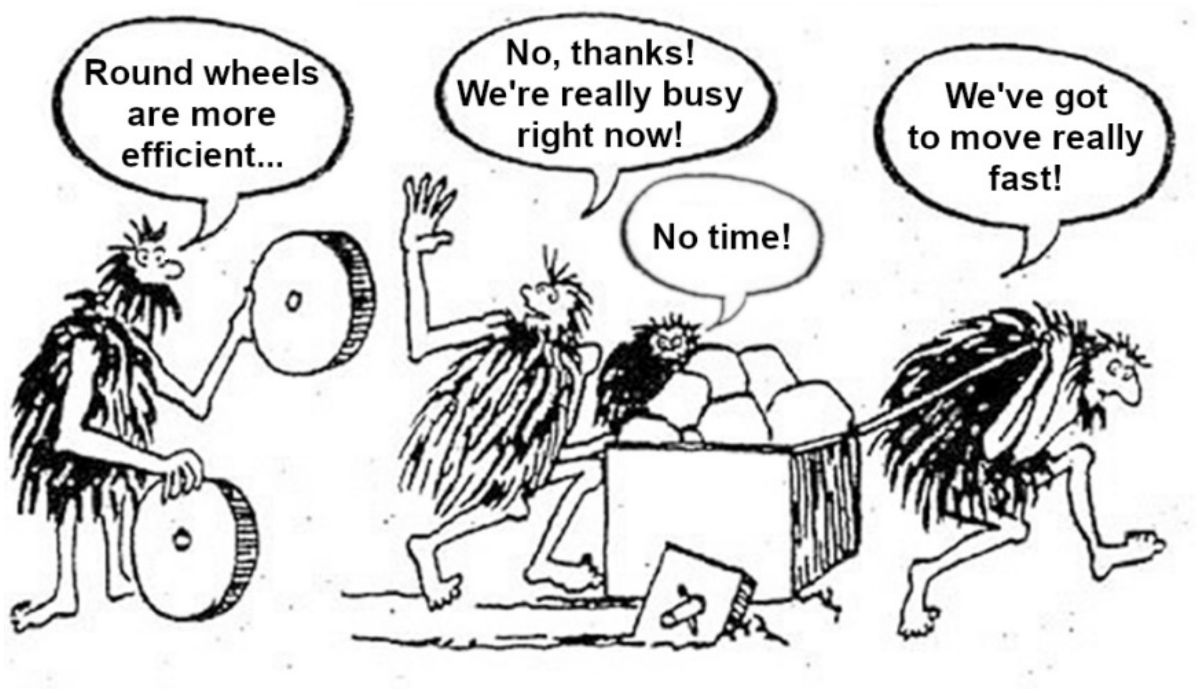
Push (Add)



Pop (Delete)

Stack Implementation in C++

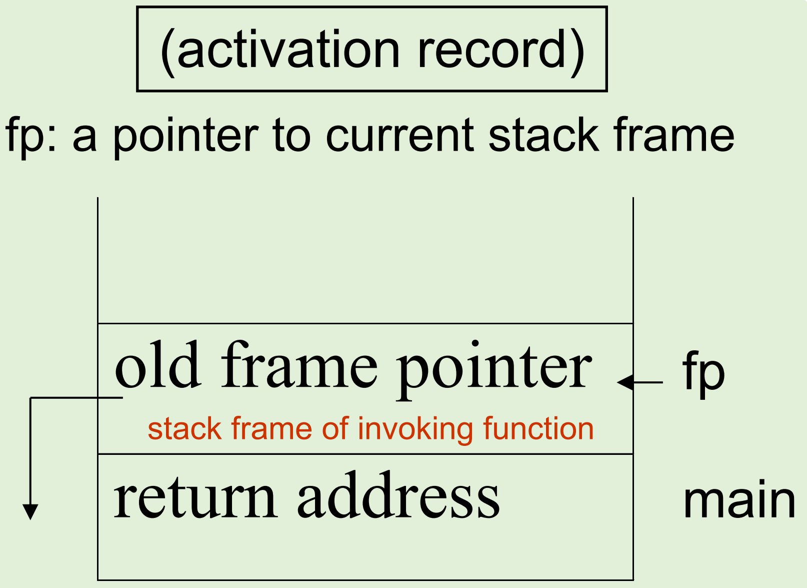
- Standard Template Library (STL)
  - https://en.cppreference.com/w/cpp/container/stack


Round wheels are more efficient...
No, thanks! We're really busy right now!
No time!
We've got to move really fast!

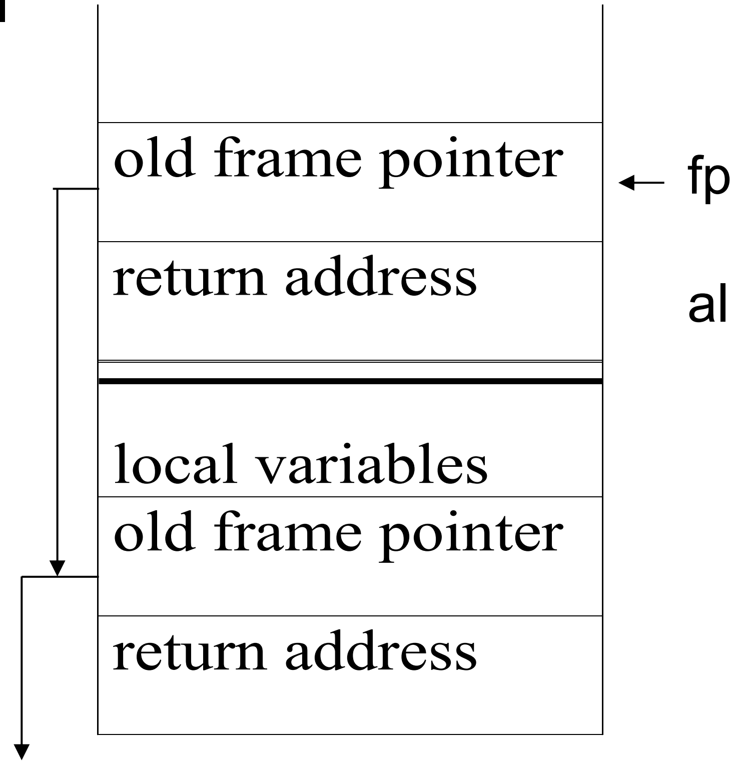
```
/* stack example */
#include <iostream>
#include <stack>

int main() {

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

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

Stack in Recursive Programs

- **Stack frames of recursive function call**
  - Always returns from the last recursion


(activation record)
fp: a pointer to current stack frame
old frame pointer
fp
stack frame of invoking function
return address
main

system stack before a1 is invoked



system stack after a1 is invoked

Application: Parenthesis Checking

- **Input: a string of brackets '(', ')', '{', '}', '[' and ’]’**
- **Goal: determine if the input string is valid**
  - 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.
- **A very frequently asked question in job interview**

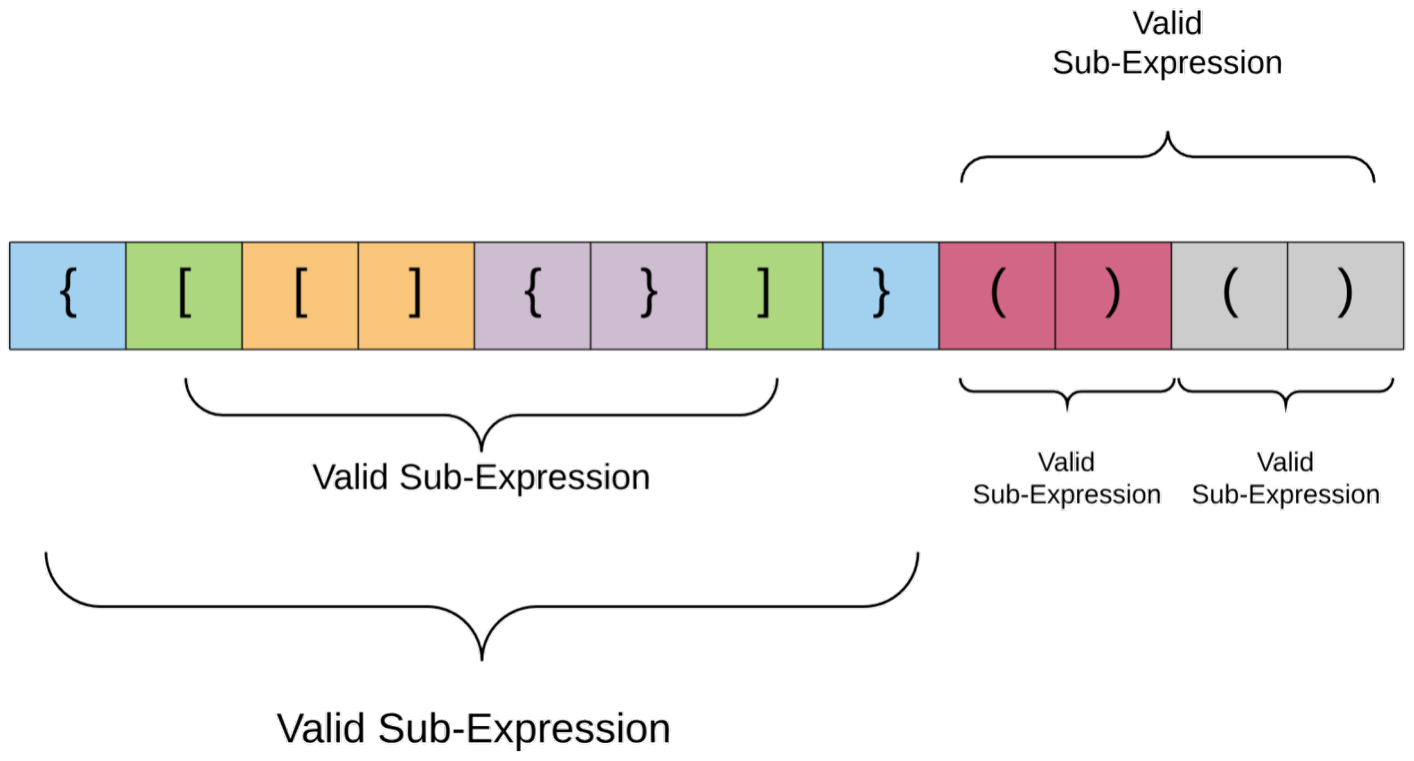
()	valid
()[]{}	valid
(]	invalid
([)]	invalid
{[]}	valid

Importance of Parenthesis Checking

- **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(); }
- **Many applications in physical design (e.g., floorplan)**

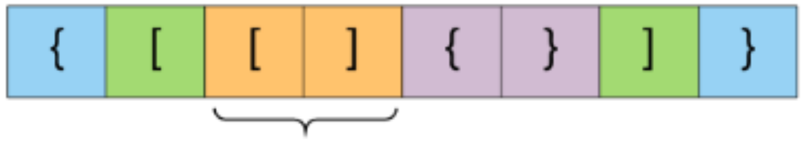
Property of a Valid Expression

- A valid parenthesis string/expression must imply:
    - All subexpressions are valid



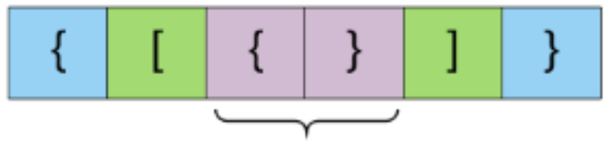
Recursive Validness

- Remove the yellow pair



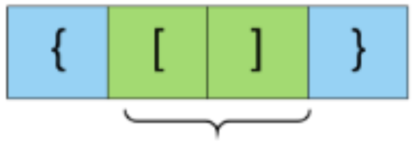
Recursive Validness

- Remove the purple pair



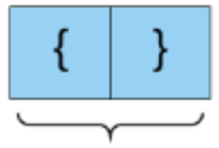
Recursive Validness

- Remove the green pair

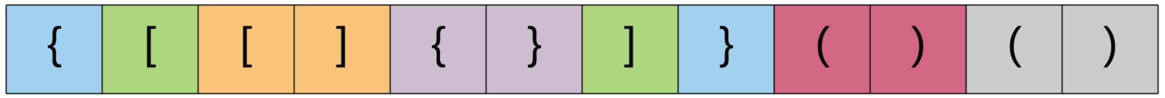


Recursive Validness

- Every subexpression is valid

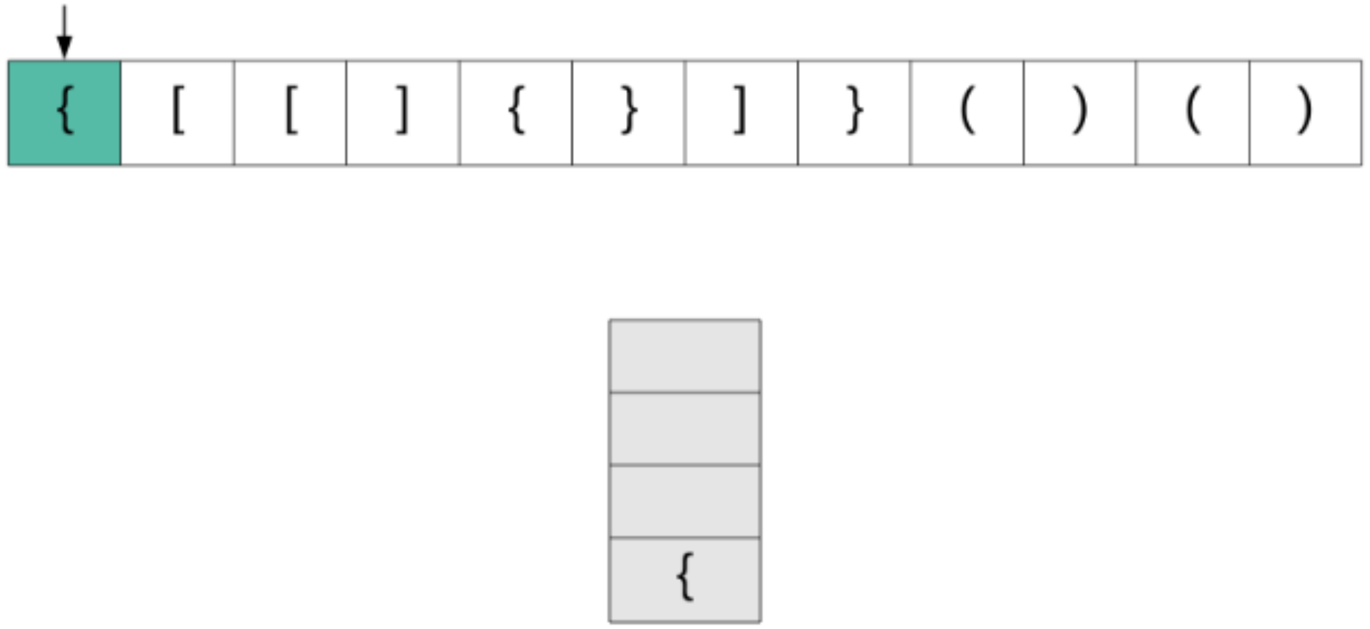


Algorithm

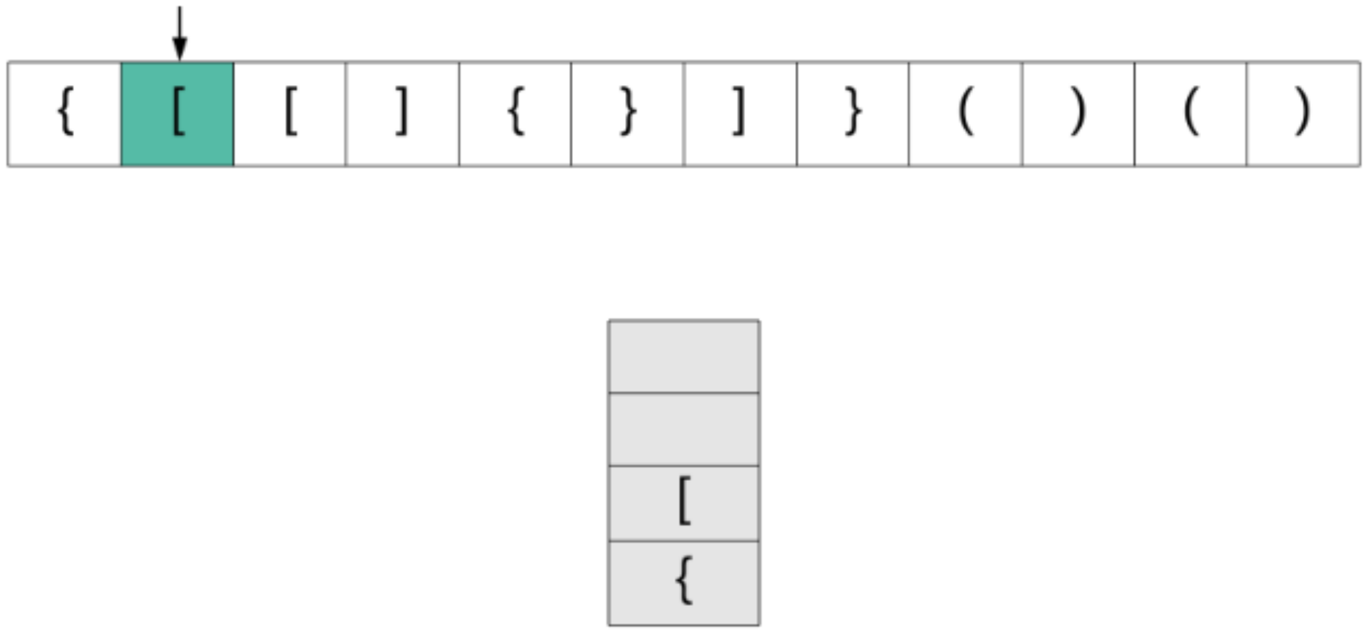


1. Initialize a stack S.
2. Process each bracket of the expression from left to right.
3. If we encounter an opening bracket, we simply push it onto the stack. This means we will process it later, let us simply move onto the sub-expression ahead.
4. If we encounter a closing bracket, then we check the 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.

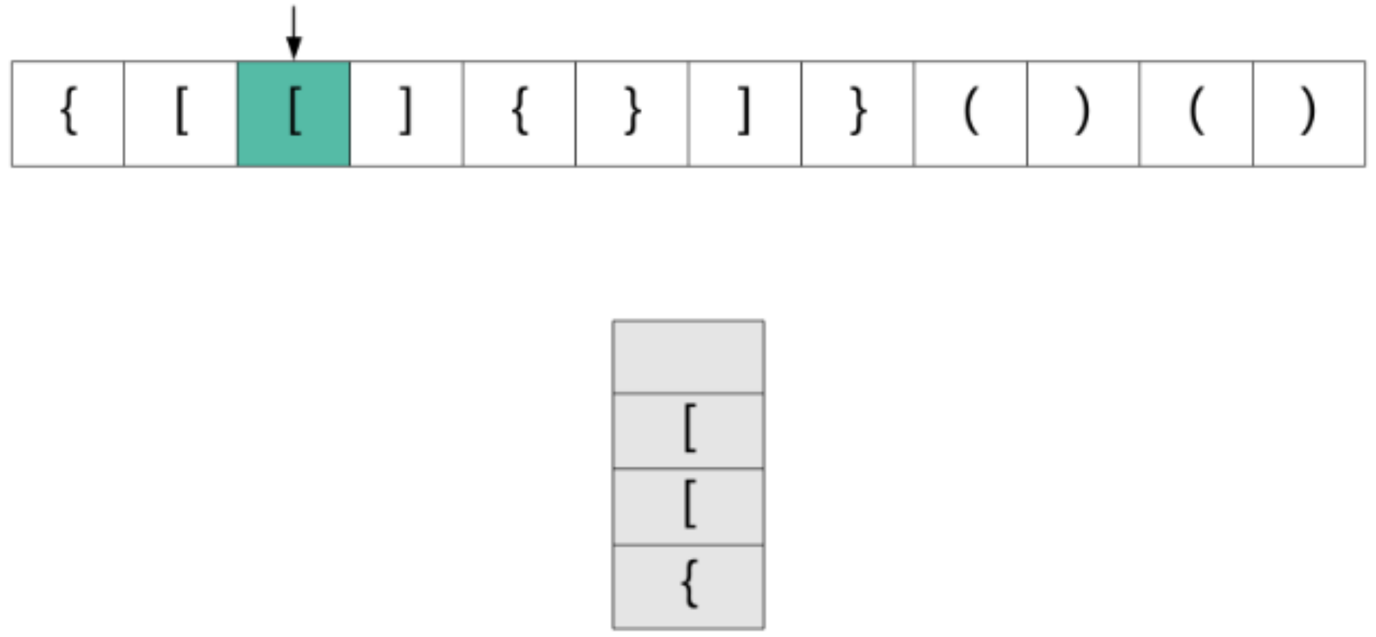
Algorithm Walkthrough – 1



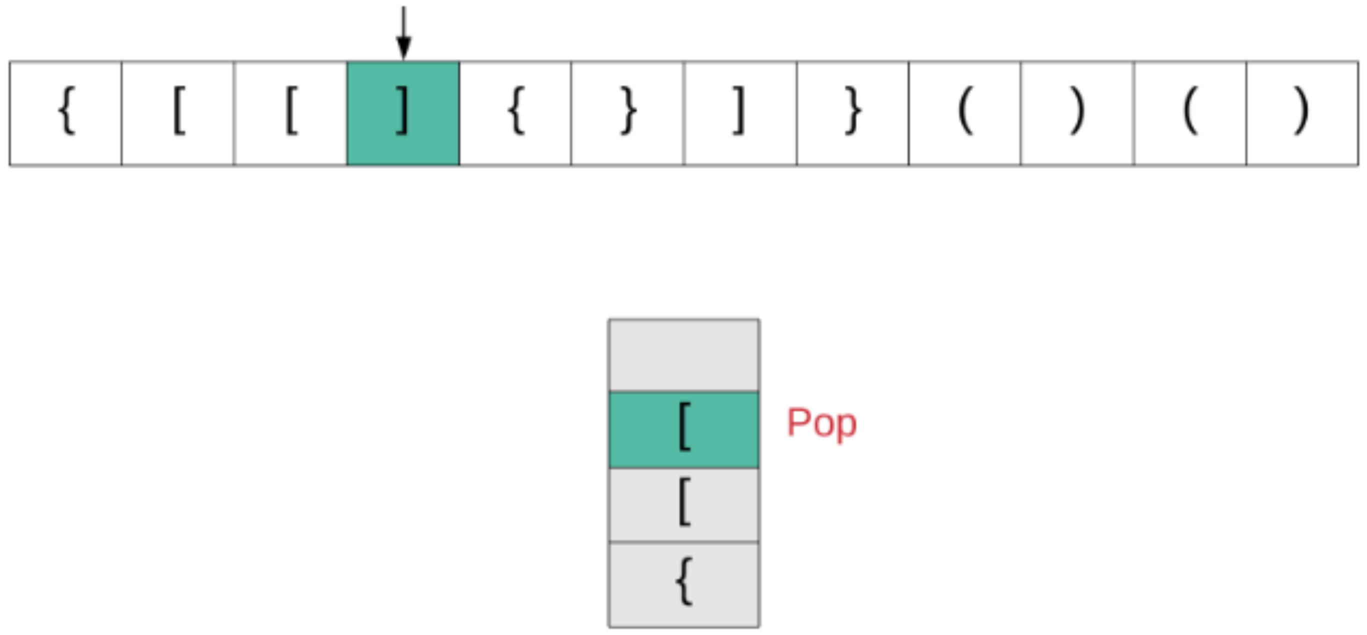
Algorithm Walkthrough – 2



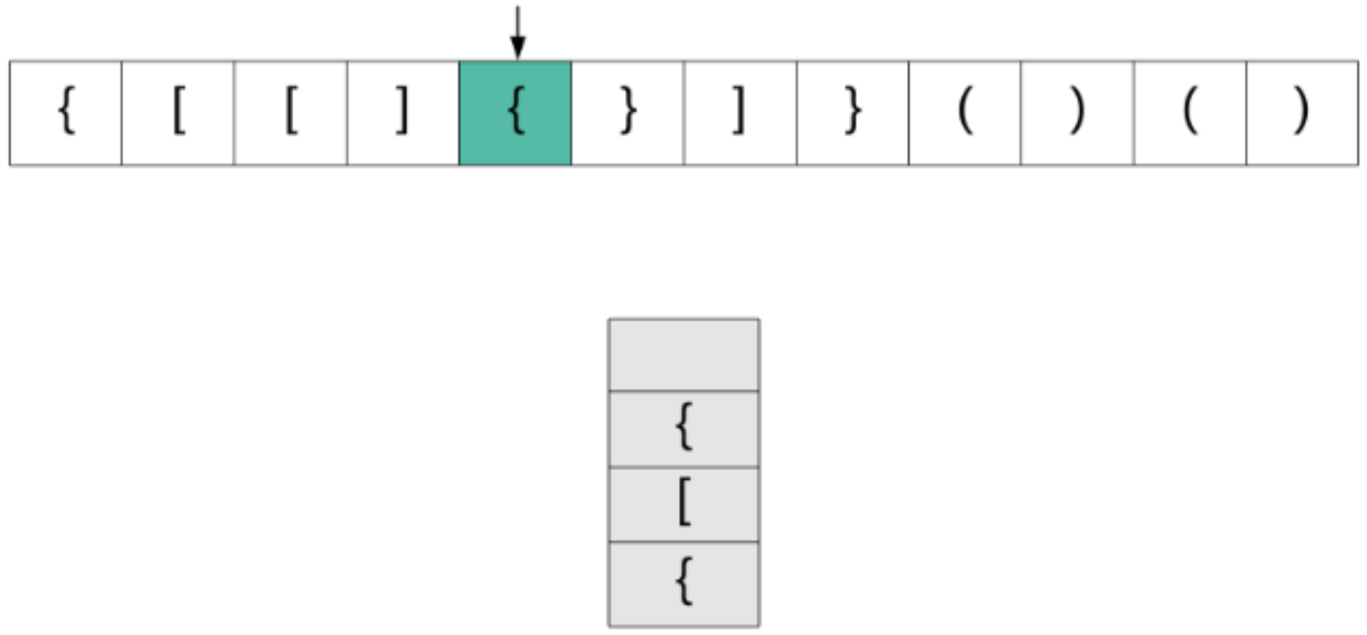
Algorithm Walkthrough – 3



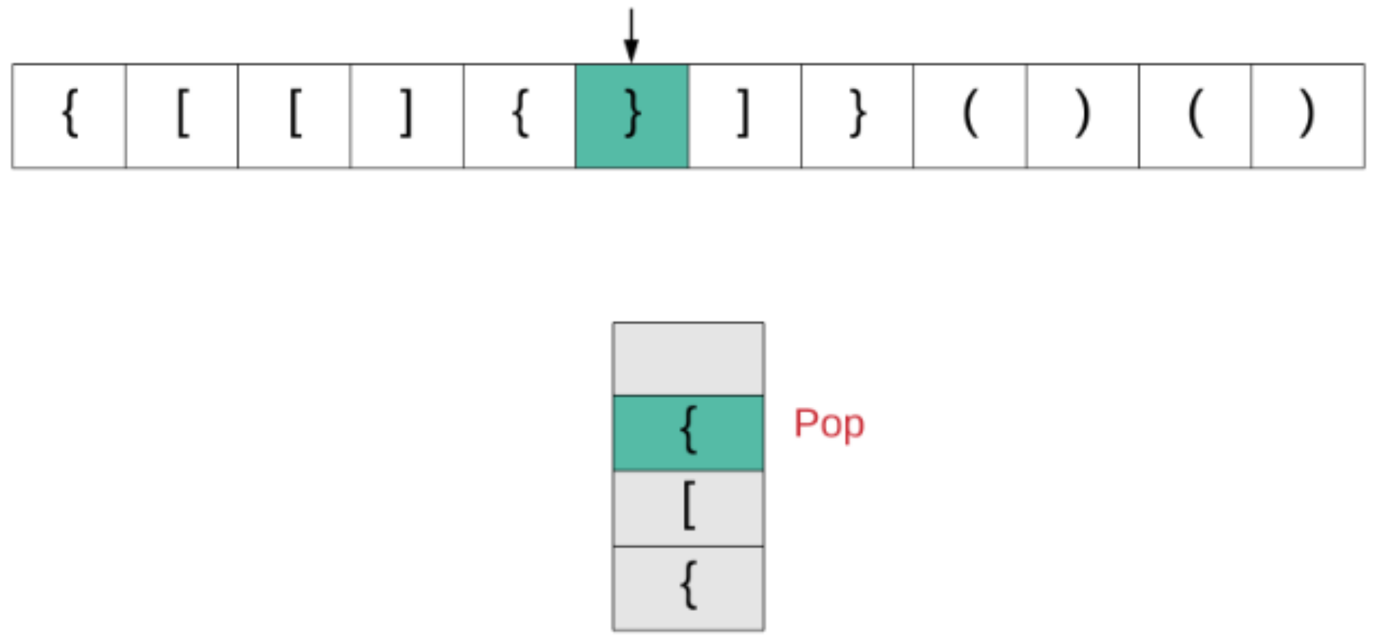
Algorithm Walkthrough – 4



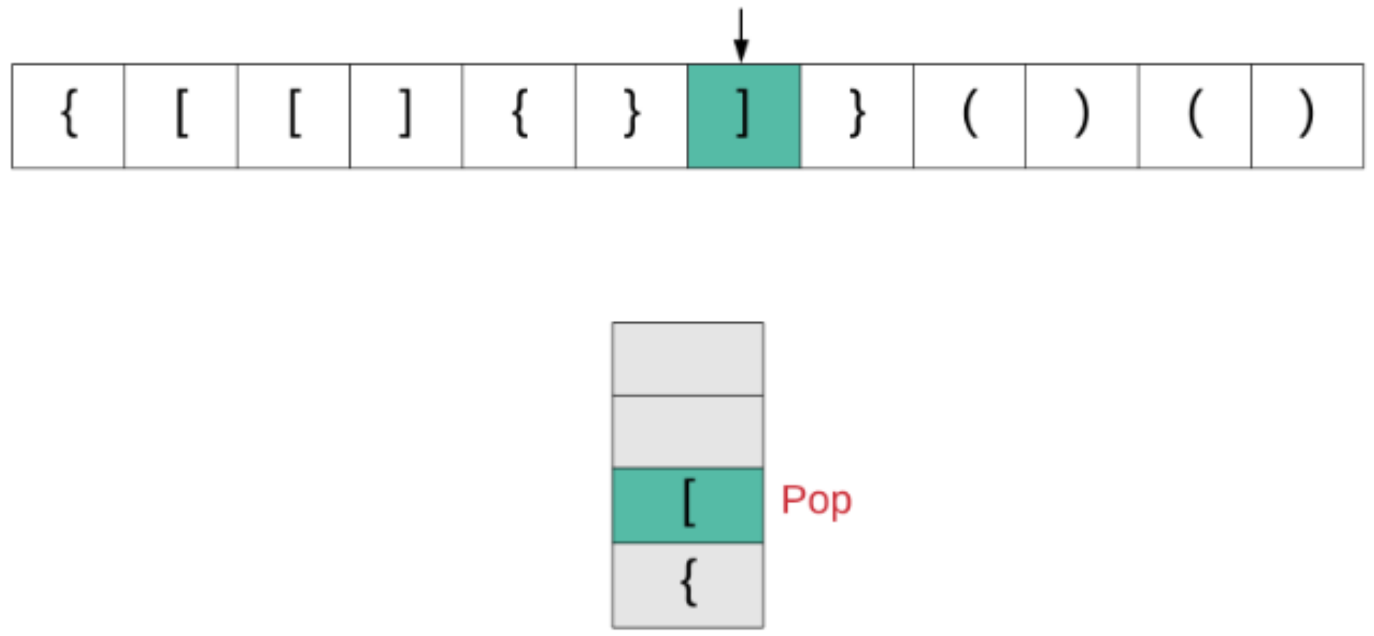
Algorithm Walkthrough – 5



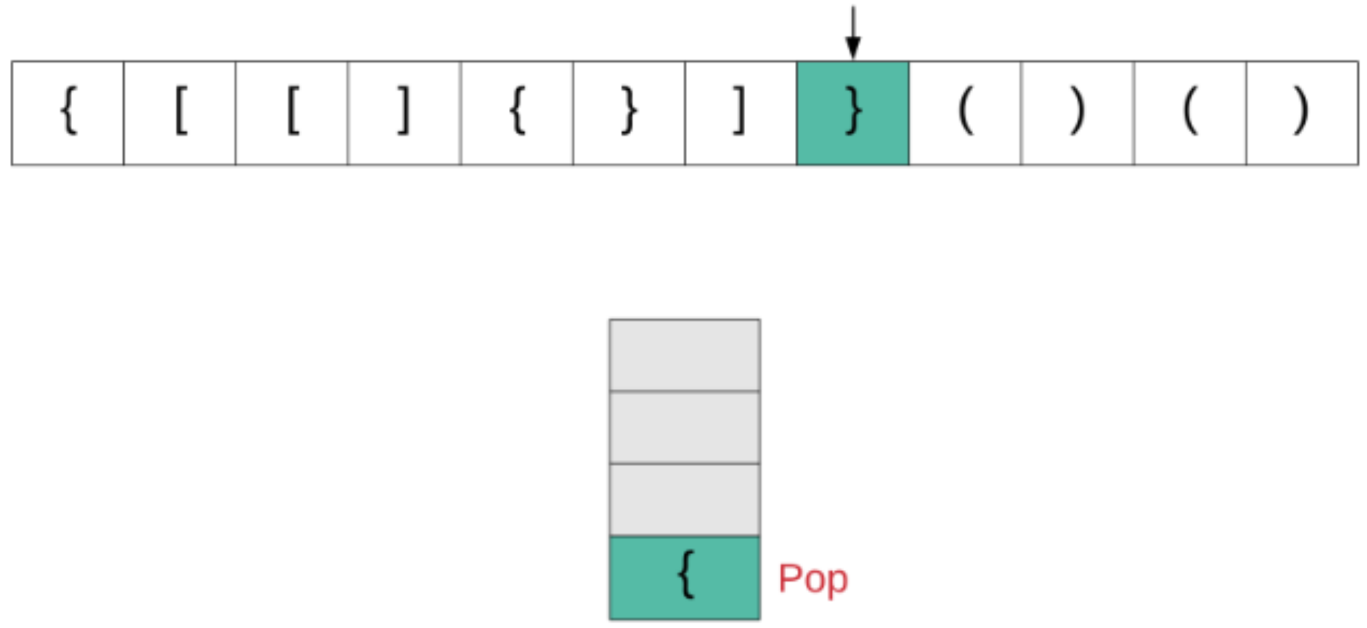
Algorithm Walkthrough – 6



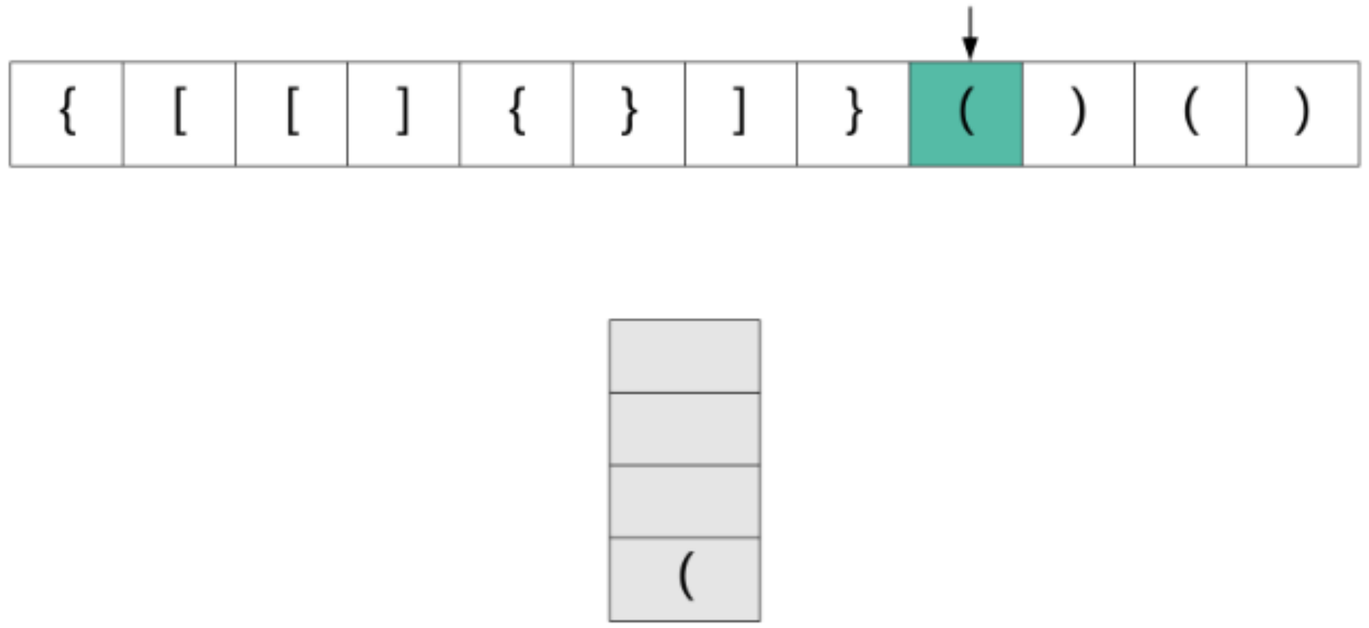
Algorithm Walkthrough – 7



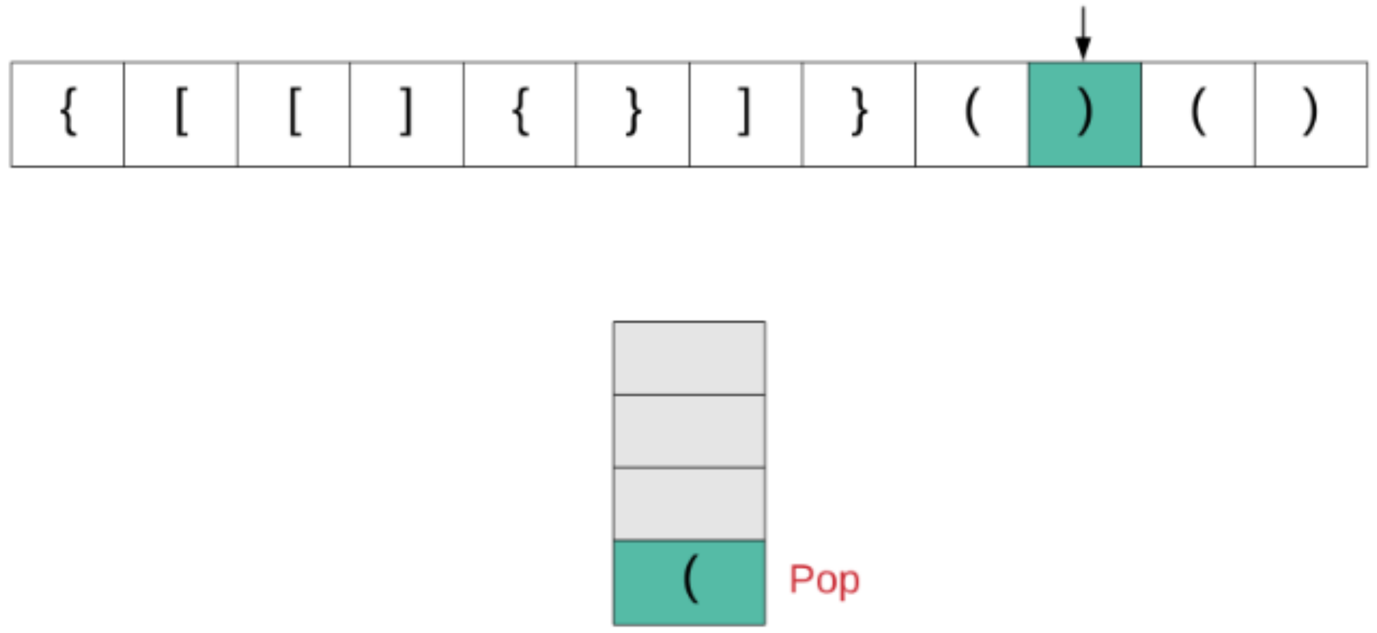
Algorithm Walkthrough – 8



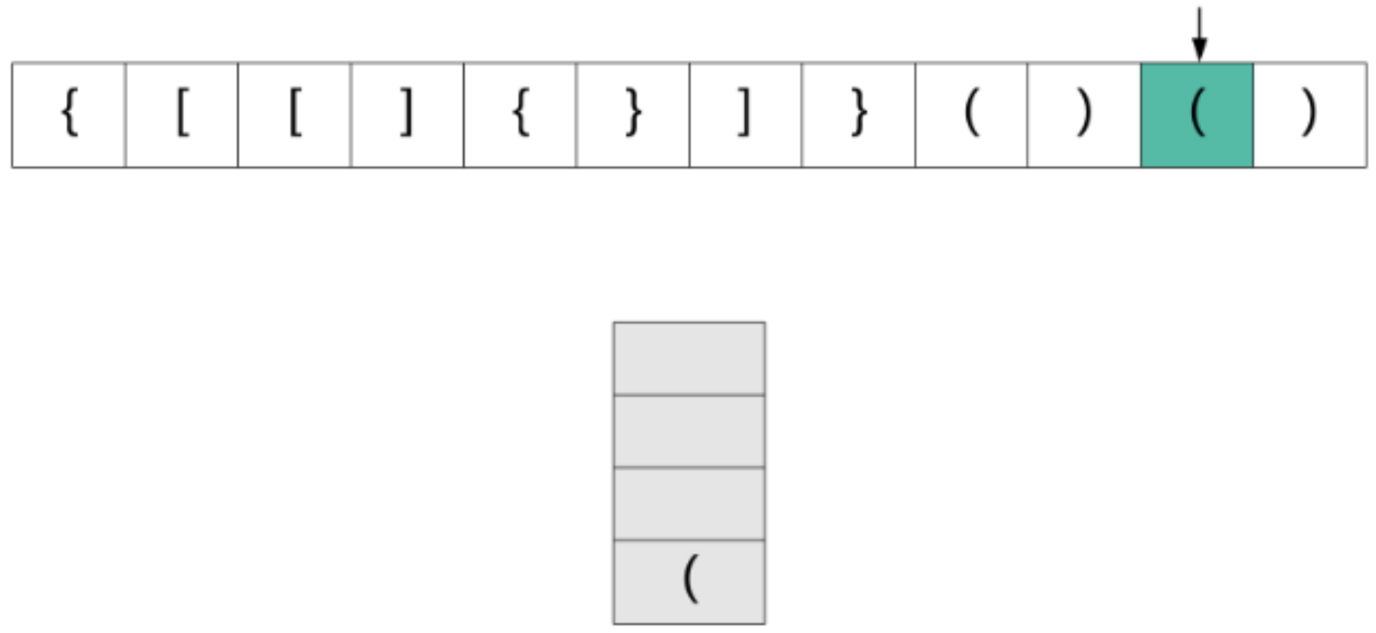
Algorithm Walkthrough – 9



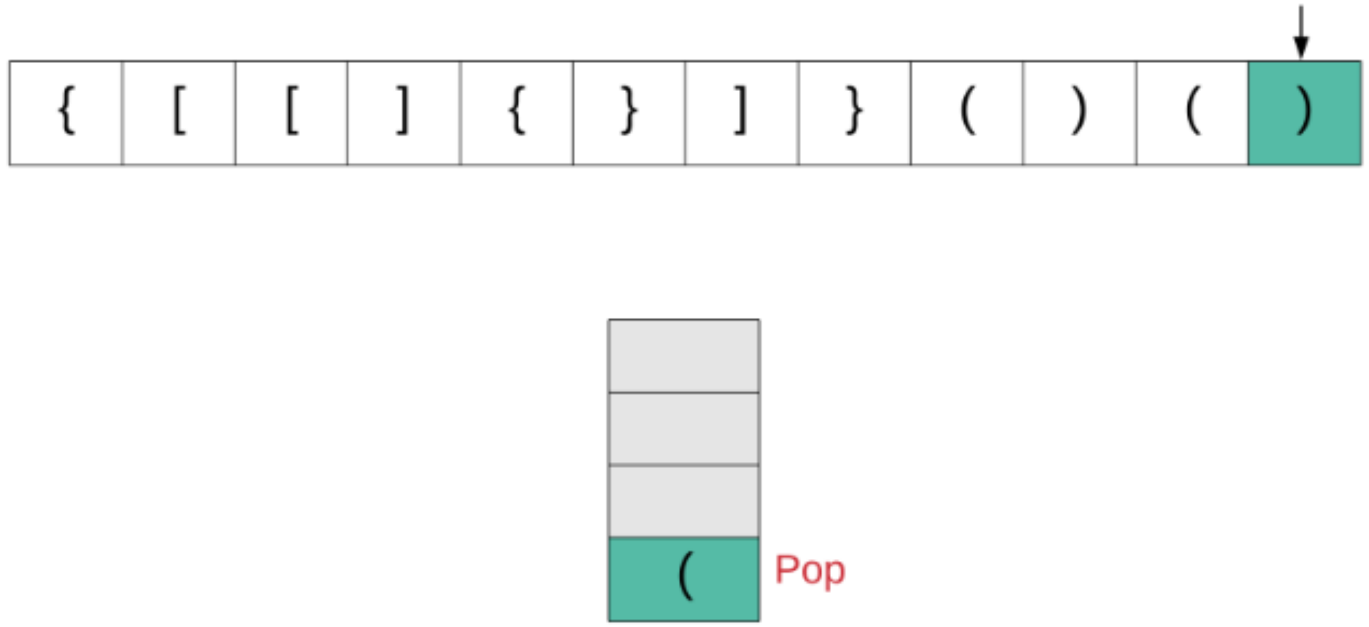
Algorithm Walkthrough – 10



Algorithm Walkthrough – 11



Algorithm Walkthrough – 11



Algorithm 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. ((((((((((.

Variant: Longest Valid Parenthesis Substr

- **Input: a string of round brackets ‘(' and ')'**
- **Goal: find the length of the longest parenthesis substring**
    - Open brackets must be closed by the same type of brackets
    - Open brackets must be closed in the correct order
    - The entire string might be invalid but substrings are valid
        - What is the longest one of such a substring?

()	2
)()())	4
(]	invalid
([)] (()	2
{[]}	2

Algorithm

1) Create an empty stack and push -1 to it. The first element of the stack is used to provide a base for the next valid string.
2) Initialize result as 0.
3) If the character is '(' i.e. str[i] == '('), push index 'i' to the stack.

   Else (if the character is ')’)

   a) Pop an item from the stack (an opening bracket)

   b) If the stack is not empty, then find the length of current valid substring by taking the difference between the current index and top of the stack. Then update the result of maximum value.

   c) If the stack is empty, push the current index as a base for the next valid substring.

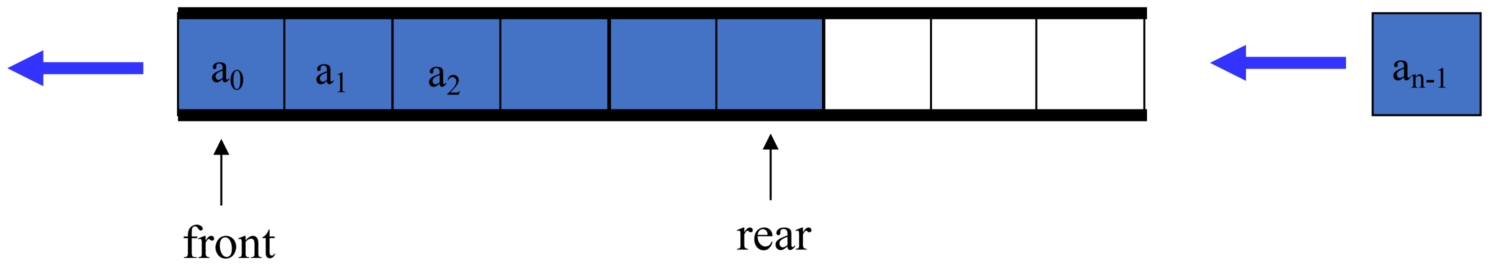
3) Return result.

Implementation

```
int longest_valid_subparenthesis_string(string S) {
  std::vector<int> stack = {-1};
  int ans = 0;
  for (int i = 0; i < S.size(); i++) {
    if (S[i] == '(') stack.push_back(i);
    else if (stack.size() == 1) stack[0] = i;
    else {
      stack.pop_back();
      ans = max(ans, i - stack.back());
    }
  }
  return ans;
}
```

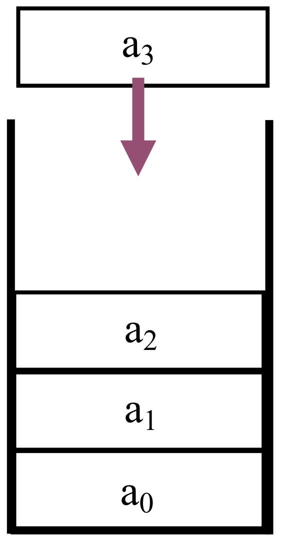
Queue

- **An ordered list supporting insertions and deletions**
  - Push: insert a new item into the list
  - Pop: remove an item from the list in a *First-In-First-Out* (*LIFO*) order
- 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

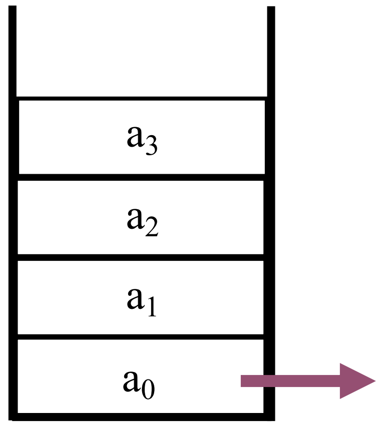


Queue Visualization

- **Three main operations**
  - Push: inserts a new item into the queue
  - Pop: removes the earliest inserted item from the queue
  - front: query the front item in the queue



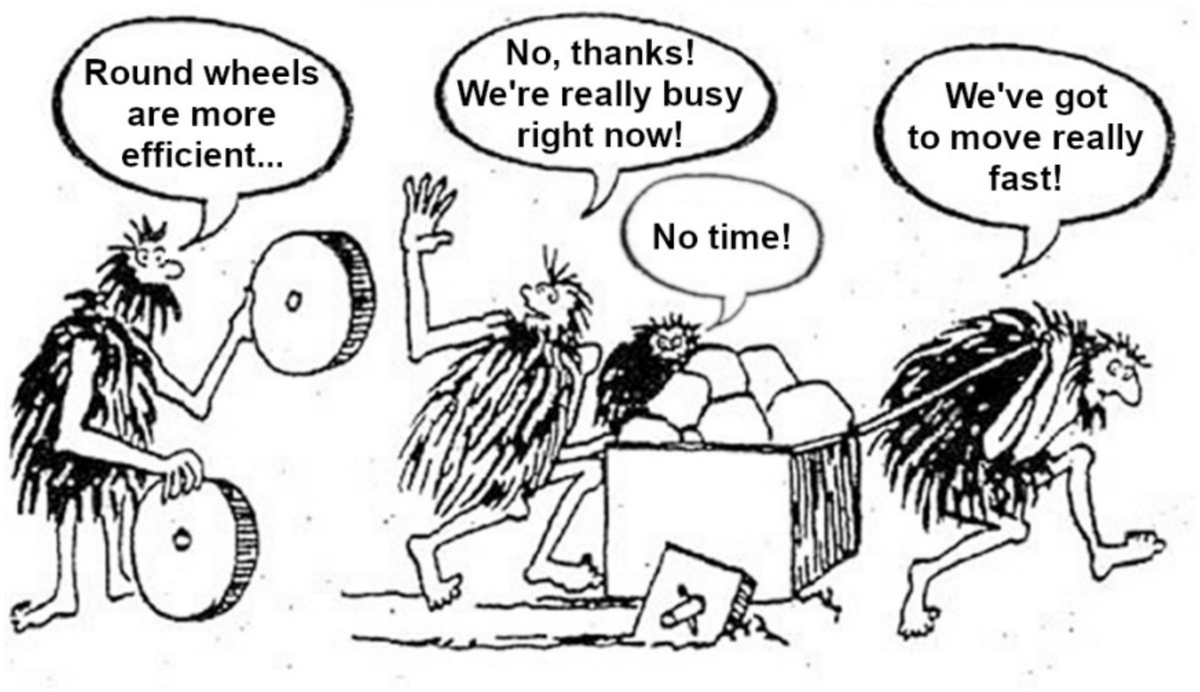
Push (Add)



Pop (Delete)

Queue Implementation in C++

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


Round wheels are more efficient...
No, thanks! We're really busy right now!
No time!
We've got to move really fast!

```
#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();
}
```

Practice: Implement Stack using Queue

- **Input: Given standard queue data structure**
  - push: insert an element into the queue
  - pop: remove an element from the queue
- **Output: Implement stack using only one queue**
  - stack::push: insert an element into the stack
  - stack::pop: remove an element from the stack

Practice: Implement Queue using Stack

- **Input: Given standard stack data structure**
  - push: insert an element into the stack
  - pop: remove an element from the stack
- **Output: Implement stack using only one queue**
  - queue::push: insert an element into the queue
  - queue::pop: remove an element from the queue

Summary

- **We have discussed two fundamental data structures**
  - Stack organizes data in last-in-first-out order
  - Queue organizes data in first-in-first-out order
- **We have discussed application of stack**
  - Recursive program implementation
  - Parenthesis checking
- **Both data structures are widely used in physical design**
  - For instance, slicing-tree Floorplan will leverage stack to recover the chip area from a linear representation string