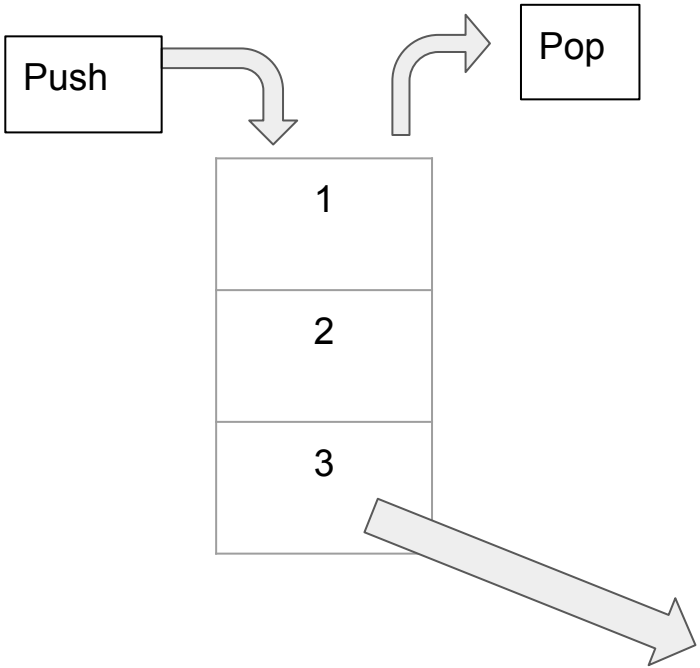
Abstract data type

A model representing a concept.

Programmer worries about the implementation

Whereas, the end user just needs to understand the functionality.(the operations performed on the data structures).

The Stack ADT



Stack ADT generally has the following operations:

- **push**(element) - add element at the top
- **pop** - delete element at top
- **peek** - returns top element but doesn't delete it
- **isEmpty** - tells if stack is empty

3 was the first element
that was pushed

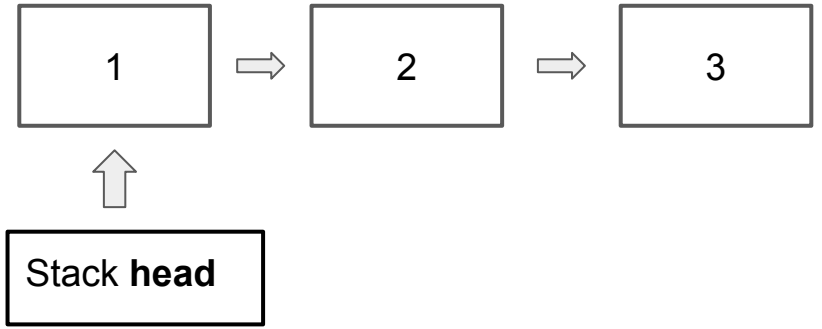
Implementing the Stack ADT:

- We can implement it using either a **linked list or an array**
- Let’s look at implementing using a **linked list**
- In the following slides, we have a parallel representation of the ADT and it’s implementation

Stacks as a Linked List

Abstraction of the stack concept

Linked List implementation of a stack



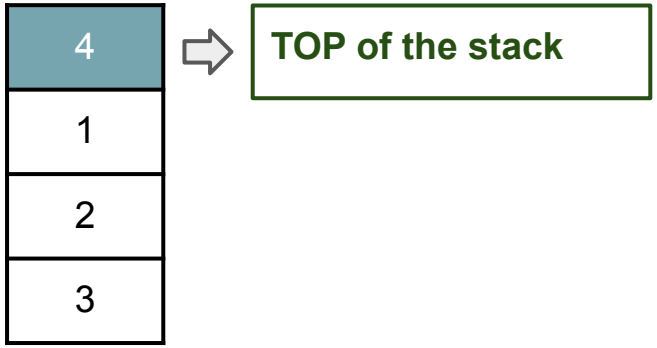
Node struct for the linked list

Head pointer to the linked list

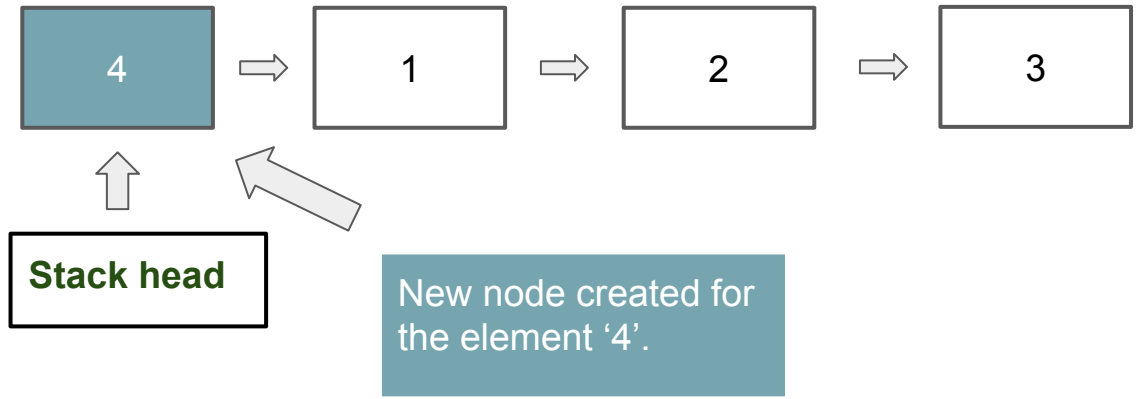
Functions of the stack

Push operation in a stack ==
Add new node as the head of linked list

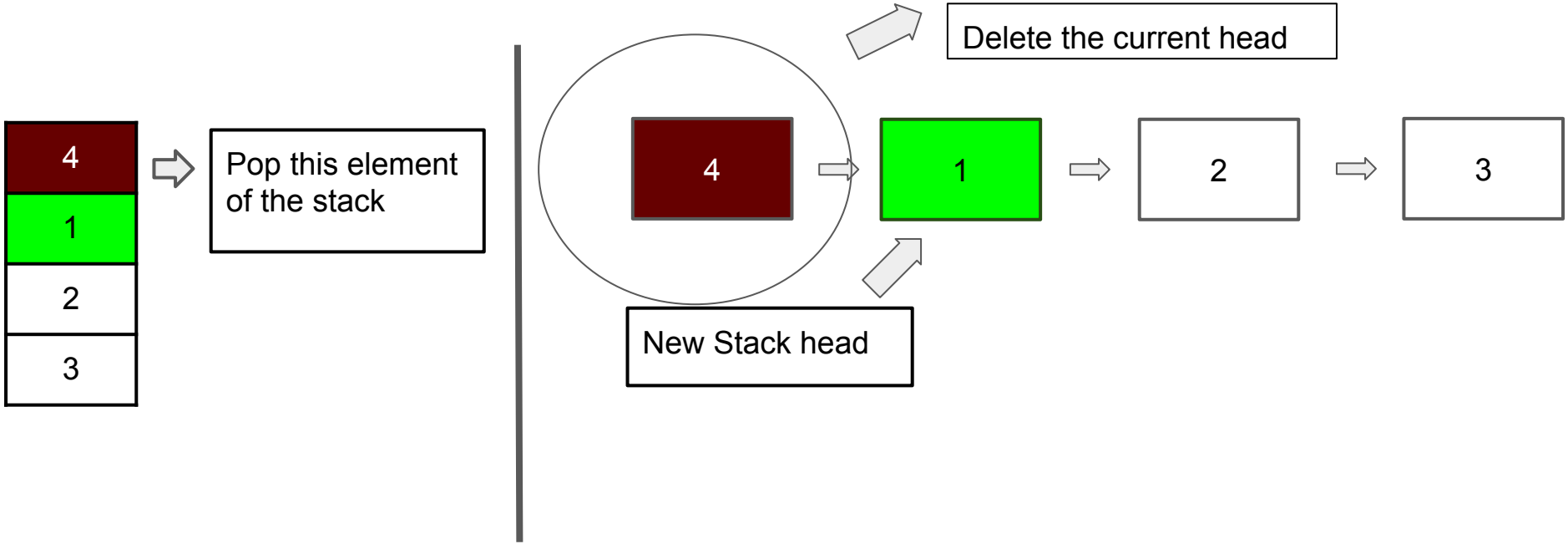
Abstraction of the stack concept



Linked List implementation of a stack



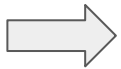
Pop operation in a stack == Delete the head of linked list



```
bool StackLL::isEmpty()
{
    return (stackHead == nullptr);
}

void StackLL::push(char key)
{
    Node* nn = new Node;
    nn->key = key;
    nn->next = nullptr;
    nn->next = stackHead;
    stackHead = nn;
}
```

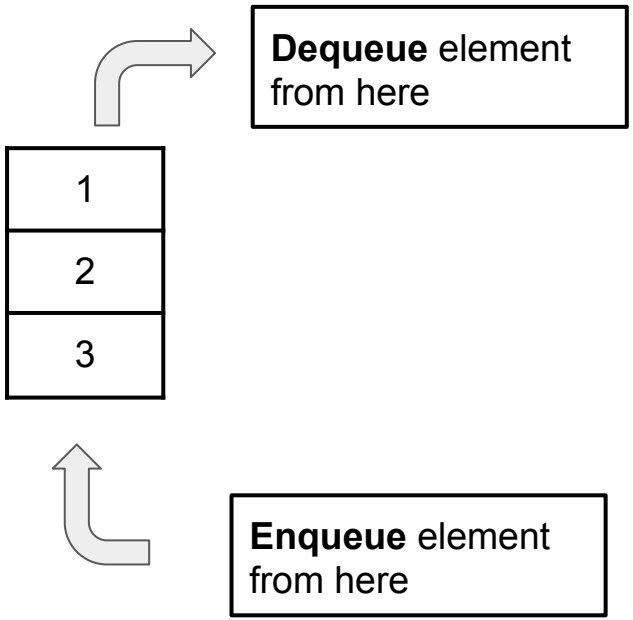
Push of a stack == Insertion at the head of the Linked list

```
}

void StackLL::pop()
{
    if(!isEmpty())
    {
        Node* temp = stackHead;
        stackHead = stackHead->next;
        delete temp;
    }
    else
    {
        cout<<"empty stack. can not pop"<<endl;
    }
}
```



Pop of a stack == Deletion of linked list head

The queue ADT

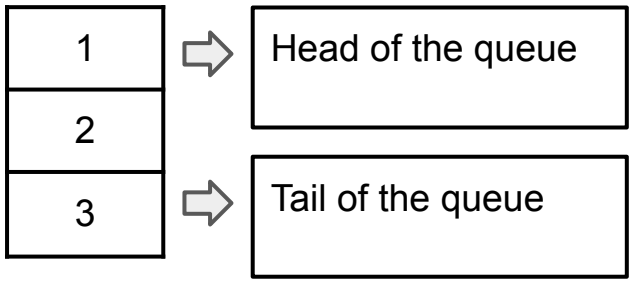


QueueADT generally has the following operations:

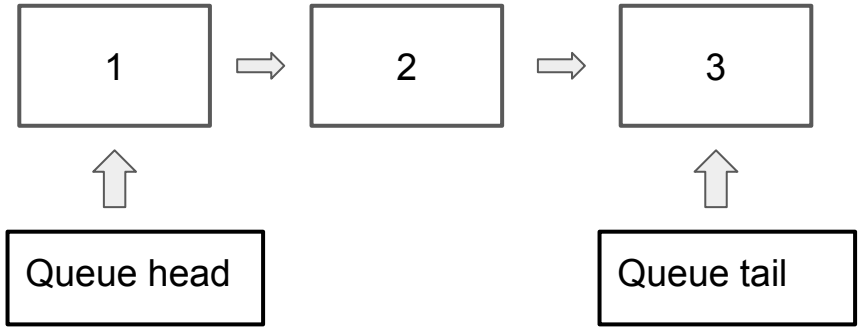
- **enqueue**(element) - add element at end of queue
- **dequeue** - fetch element at top and delete
- **isEmpty** - tells if queue is empty
- **peak**- return the first element of the queue

Queues as a Linked List

Abstraction of the queue concept



Linked List implementation of a queue



We need a head and a tail pointer because we are ***dequeuing at head and enqueuing at the tail***.

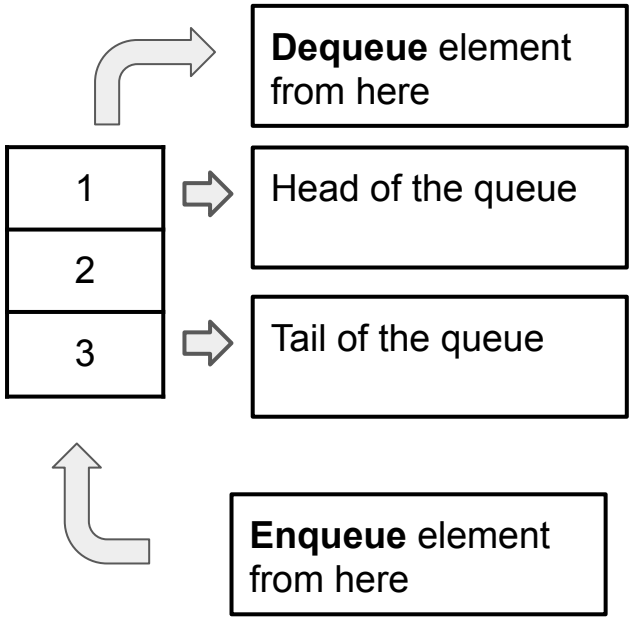
```cpp
#ifndef QUEUELL_HPP
#define QUEUELL_HPP

#include <string>

class QueueLL
{
  private:
    struct Node
    {
        int key;
        Node *next;
    };

    // item in list to be dequeued next
    Node* queueFront;
    // item in list that was most recently enqueued
    Node* queueEnd;

  public:
    QueueLL();
    ~QueueLL();
    bool isEmpty();
    void enqueue(int key);
    void dequeue();
    int peek();
};

#endif
```

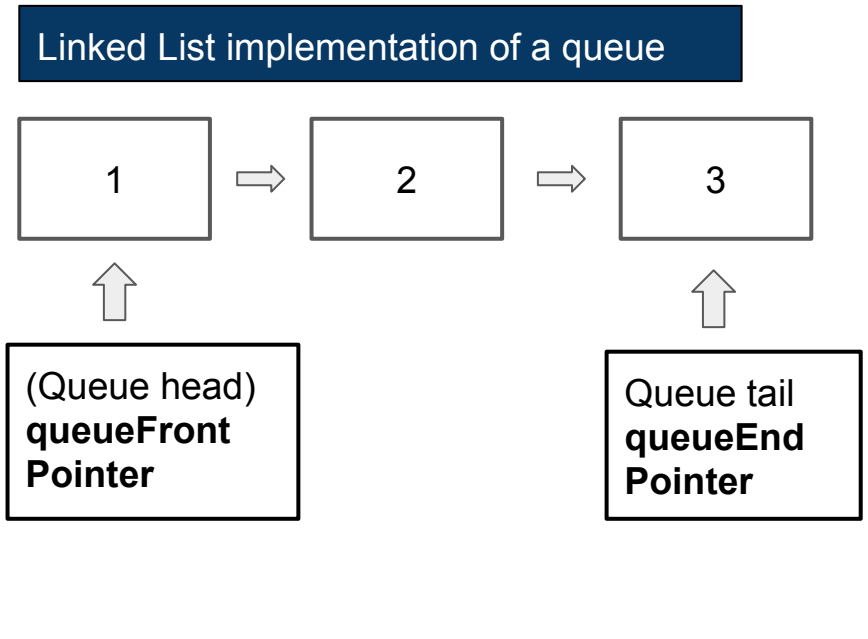
Node struct for the linked list

Front and end pointers to the linked list

Functions of the Queue

Queues as a Linked List





Implementing **enqueue**

Edge case - empty queue :

Both **front and tail pointers** should be updated to the new node

Generic case - non empty queue :

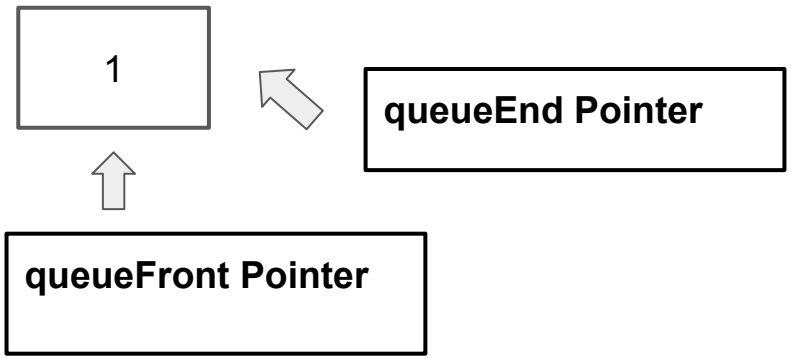
Tail pointer should be pointing to the newly created node

Enqueue - Edge case - Empty Queue

```
QueueLL::QueueLL()
{
    queueFront = nullptr;
    queueEnd = nullptr;
}
```

If Queue is empty,

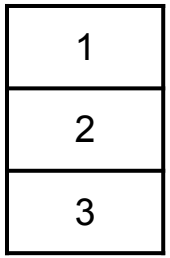
- **Create** a new node
- Both **queueFront** and **queueEnd** point to this Node

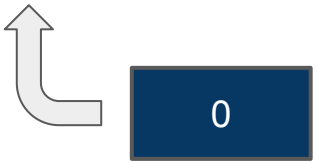


Enqueue = Insert Node at the end of a linked list

If Queue is not empty,

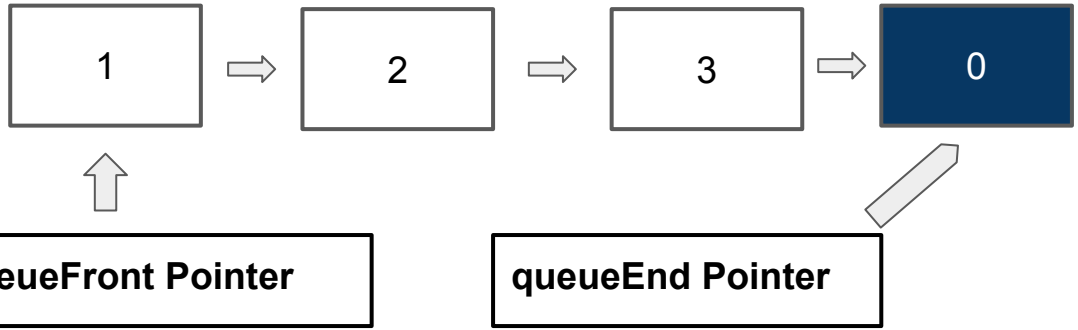
- **Create** a new node
- Update **queueEnd** to point to this Node





Enqueuing the element 0

Linked List implementation of a queue



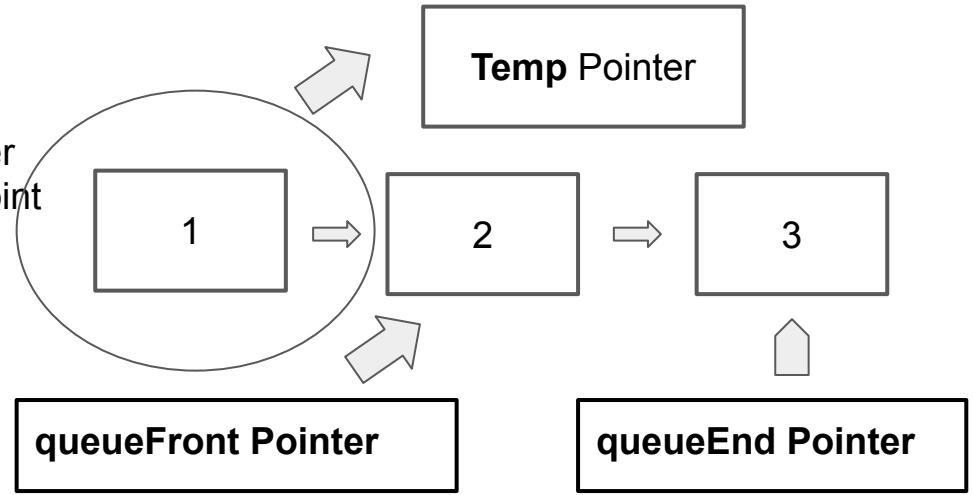
Dequeue

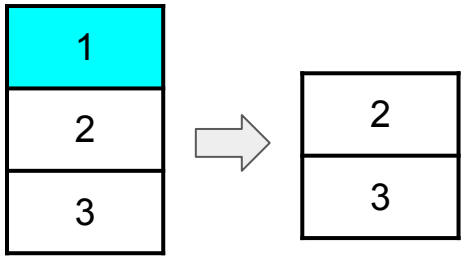
What should the **dequeue** function do for the above **queue**?

Dequeue = Delete Node at the head of a linked list

If Queue is not empty,

- **Store** head as a **temp** pointer
- Update **queueFront** to point to next Node(2nd node)
- Delete **temp**





Balanced parentheses problem

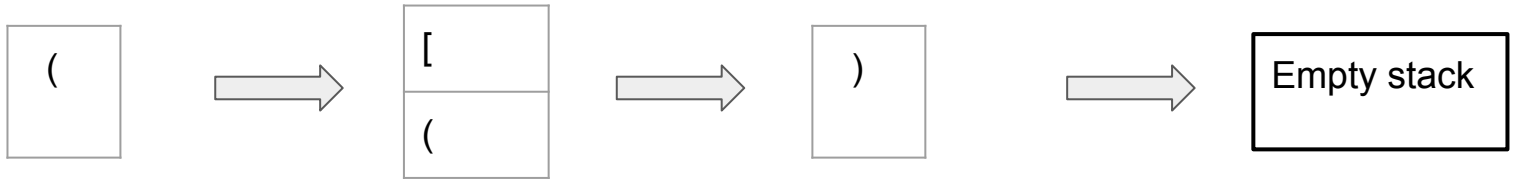
Balanced	UnBalanced
()	(]
[]	[}
([])	((])
{}	([{])
{[()]}	{(]

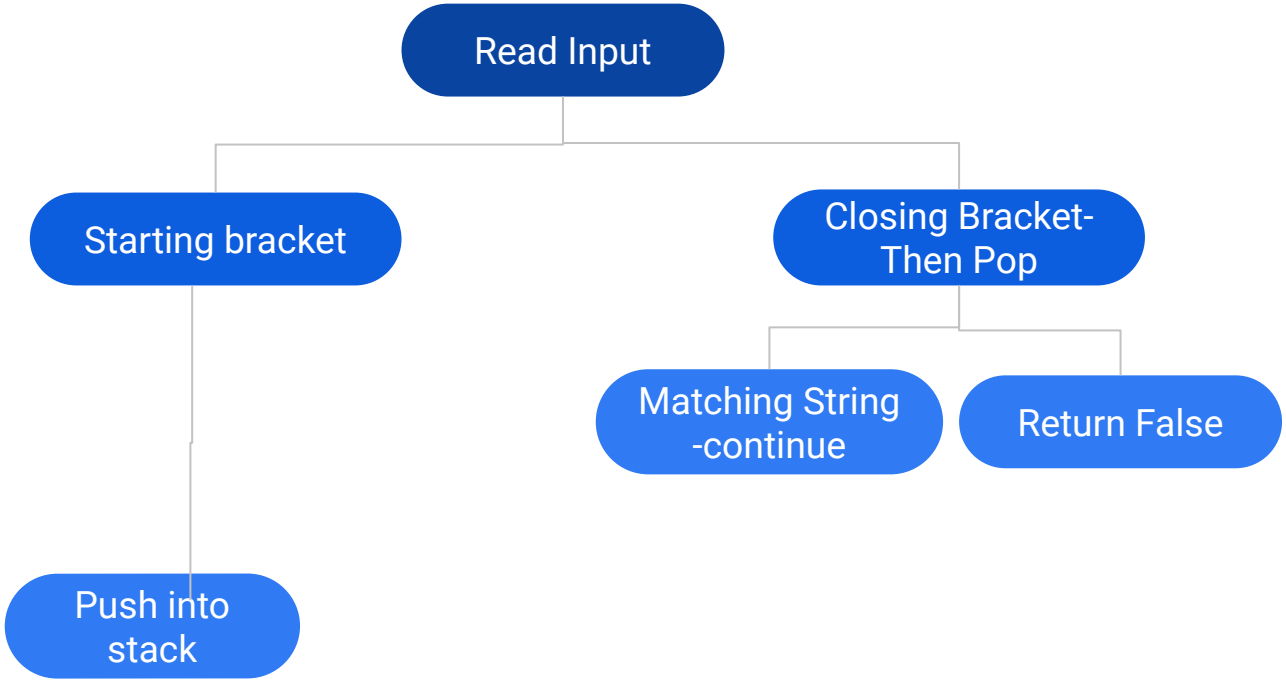
Solve using a stack

String S = “([])”;

Input	(	[	]	)
Action	Push	Push	Pop	Pop

Stack




Read Input
Starting bracket
Closing Bracket-
Then Pop
Push into
stack
Matching String
-continue
Return False