STMC Coding Team Training

Lesson 8: Data Structure II

Tsai Yun Chen

April 22, 2023

Goal today

Today we will continue the topic of data structure, in particular we will study the commonly used structure: stack, queue. Then we will look into different applications of these data structure.

- A recap on linked-list
- A new implementation
- Stack and Queue, LIFO and FIFO
- implementing stack and queue using linked-list
- application: formula parser
- Shunting yard algorithm (simplified)
- HKOI problem: The Josephus Problem

Linked List

Recall what we have learnt at the end of last lecture.

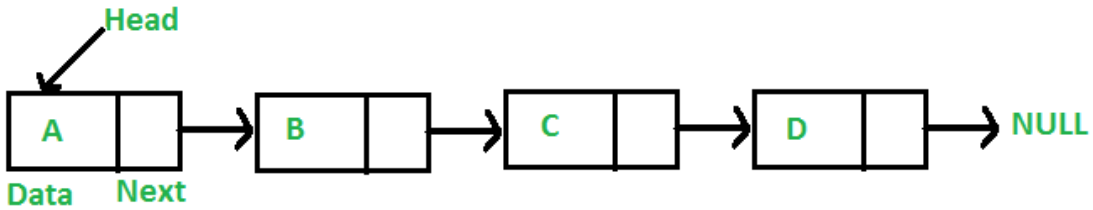

Figure 1: Linked List(Source)

Naive Implementation

We have a simple implementation last time,

```
class LinkedList:
  def __init__(self):
    self.data=None
    self.next=None
  def insert(self,data):
    if self.data==None:
      self.data=data
      self.next=LinkedList()
    else:
      self.next.insert(data)
```

Some problem

Some problem with the previous implementation,

- it takes $O(n)$ time to insert one data to the end
- which makes no difference with array
- we need also remove of data
- Let's slightly improve it!

```
class llnode:
  def __init__(self):
    self.data=None
    self.prev=None #newly added
    self.next=None
class ll:
  def __init__(self):
    self.head=llnode() #empty node
    self.tail=self.head
    self.size=0
  def Insert(self,data,mode='front'):
    if mode=='front':
      temp=llnode()
      temp.data=data
      temp.next=self.head
      self.head.prev=temp
      self.head=temp
    elif mode=='end':
      temp=llnode()
      temp.data=data
      self.tail.next=temp
      temp.prev=self.tail
      self.tail=temp
    else:
      raise Exception("Unrecognized mode of insertion")
```

```
def Pop(self,mode='front'):
  if mode=='front':
    if self.size==0:
      raise Exception("Popping empty list")
    temp=self.head
    self.head=self.head.next
    self.head.prev=None
    return temp.data
  elif mode=='end':
    if self.size==0:
      raise Exception("Popping empty list")
    temp=self.tail
    self.tail=self.head.tail.prev
    self.tail.next=temp.next
    return temp.data
  else:
    raise Exception("Unrecognized mode of insertion")
```

Queue and Stack

Now we are ready to explore another new data structure. We are going to explore two classic and extremely useful data structures, namely queue and stack, these two data structures are inspired from our daily life experience as we will see later

1. Queue
   - Data are stored in the order of adding them
   - Just like you are queuing at a store, first-come-first-serve
   - We also called this type of operation First-In-First-Out (FIFO)
   - Commonly used in task-scheduling and best-solution searching
2. Stack
   - Data are stored in the **reverse** order of adding them
   - Just like you are stacking up a bunch of paper on table, the first one will be at the bottom
   - We also called this type of operation Last-In-First-Out (LIFO) or First-In-Last-Out (FILO)
   - Commonly used in parsing, recursive process and fast solution searching

Queue, a illustration

Figure 2: Queue(Source)

Stack, a illustration

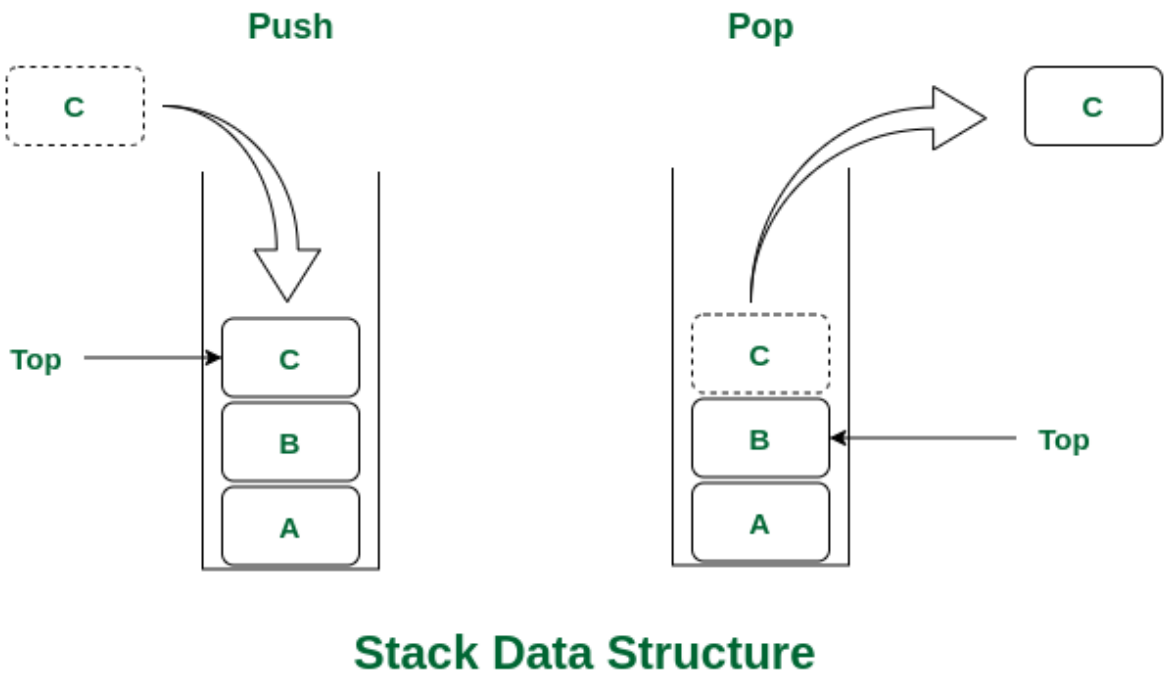

Figure 3: Stack(Source)

Implementation

Now let's do the implementation, since the major feature of such strctures are update and query (FIFO,FILO), it makes sense for us to utilize the linked list we just learnt (and that's why we are introducing it). I will do the stack as an example and the queue will be your exercise. What we will need is

- A class holding a structure `stack`
- operator pop(): removes a data from the top and return it
- operator push(): add a data to the top
- operator top(): return the data currently at top without removing it
- operator empty(): return true of the stack is empty now, else return false
- operator size(): return the current size of the stack

```
class stack:
  def __init__(self):
    self.data=ll()
  def pop(self):
    return self.data.Pop()  #default popping from the front
  def push(self):
    self.data.Insert()  #default insert to the front
  def top(self):
    if self.data.size>0:
      return self.data.head.data
    else:
      raise Exception('Accessing empty stack')
  def empty():
    return self.data.size==0
  def size():
    return self.data.size
```

The implementation is actually pretty simple with the help of linked-list.

Implementation Queue

Now it’s your turn, let’s try to implement a queue, make sure you work it out since we will be using it later.

Application of Stack: Formula Parser

Now let's look at some application, consider now we are designing a calculator, our job is to receive a formula, evaluate it and return the answer. For example the input might be

$$1 + 2 + 3$$

Then our program should be able to evaluate the answer 6 and return it.
But the problem is there might be formula like this

$$1 + 2 \times 3$$

Then we must perform the $\times$ first and then perform the $+$. In general this problem could be hard to solve and there is a famous algorithm called the Shunting-yard algorithm (Developed by Dijkstra, a great computer scientist).

Shunting-yard algorithm

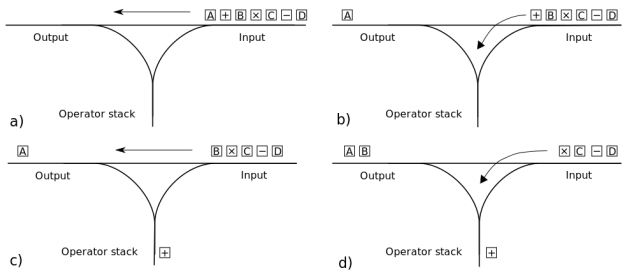

Figure 4: Shunting-yard algorithm (Source)

Figure 5: Real Shunting Yard (Source)

A simpler case

Let's consider a simpler case, we assume the formula we will get only consist of 3 types of things, number, $+$ and $\times$, no subtraction, no division (actually there are by changing the number). We also do not allow parenthesis, whenever there are terms like

$$a \times b \times c$$

it is equal to

$$(a \times b) \times c$$

We call this type of rule **left-associative**.

Idea for implementation I

We consider walking through the array from left to right, then we can always see only 3 things by our restriction,

- Numbers: let's first leave this alone since we don't know what to do yet, we continue walking.
- $+$: we found an operator, but there might be $\times$ behind and we need to evaluate that first, we continue walking.
- $\times$: we found an operator again, and since here $\times$ must have the highest order of evaluation, we can always evaluate it, grab the thing infront and behind to compute it!

Idea for implementation II

So you can imagine, after first round of walking, we will evaluate all the $\times$, but the problem is how to keep track of the things infront? What if we encounter things like

$$a \times b \times c \times d \times \ldots$$

We need something to store the things we walked pass and also be able to **keep track of** what is the **last thing added** to the storage and since we are doing this repeatly thus we also want this to be **efficient** as well.

Stack is what we need!!!

Idea for implementation III

So what we will be doing is maintain a stack that stores the the things passed by, then whenever we encounter a $\times$, we pop the last elements stored and multiply it with the next elements in the formula. For example the equation

$$1 \times 2 \times 3$$

The stack we maintained will be something like

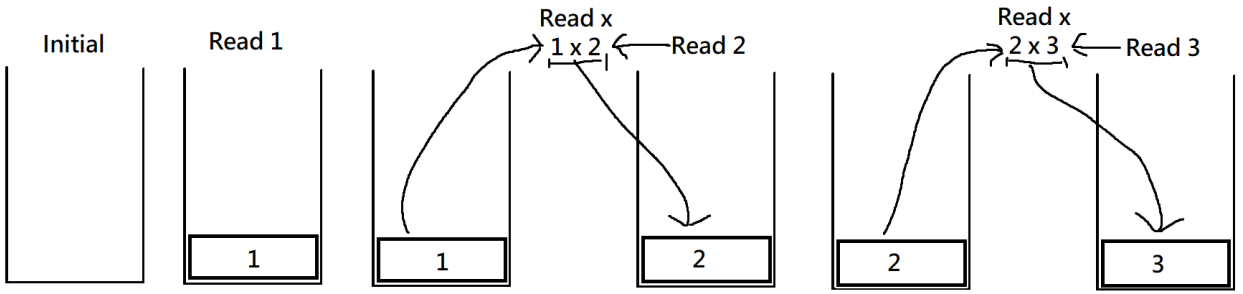

Figure 6: Stack state explained (self-drawn)

Idea for implementation IV

Now we are left with the last pieces, after the above process, we will obtain a stack that contains of only two types of things:

1. Numbers
2. Operator $+$

Hence we simply pop all of them out and add them up.

```
def EquationParser(formula):
  tokens=formula.split()
  walked=stack()
  flag=False
  for token in tokens:
    if flag==True:
      left=int(walked.pop())
      right=int(token)
      walked.push(str(left*right))
      flag=False
    elif token!='*':
      walked.push(token)
    else:
      flag=True
  res=0
  while not walked.empty():
    temp=walked.pop()
    if temp!='+':
      res=res+int(temp)
  return res
```

Your turn now!

Let's have some practices. This time we will have something different, we will borrow the problem from **HKOI**, the Hong Kong Olympiad of Informatics, for the practice. Please visit the following link

https://judge.hkoi.org/task/01030

and login with your **school** gmail account, read the problem and you can first try to solve the problem on your own computer/Google Colab, then once you are confidence with your solution try to submit it.
Hint: you can try to use the queue here, it will be helpful.