NANYANG TECHNOLOGICAL UNIVERSITY

Data Structures & Algorithms in Python: Priority Queue and Stacks and Queues Applications

Dr. Owen Noel Newton Fernando

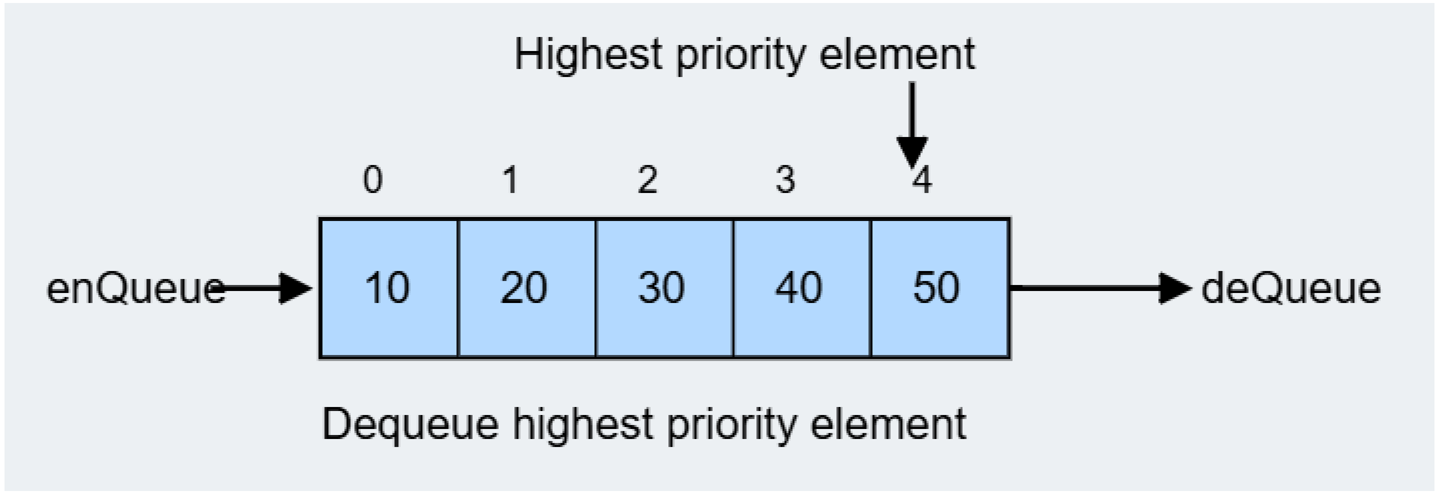
College of Computing and Data Science

Priority Queue

- **Definition:**
  - The **priority queue** is **an abstract data-type similar to a regular queue or stack** data structure in which each element additionally has a "**priority**" associated with it. In a priority queue, an element with high priority is served before an element with low priority.
  - A **priority queue ADT** is a data structure that supports the operations Insert and Delete Min (which returns and removes the minimum element) or Delete Max (which returns and removes the maximum element) and peeking the element(find min or find max).

Priority Queue

- **Difference between Priority Queue and Normal Queue:**
  - In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.



Types of Priority Queue

There are two types in Priority Queue:

- Ascending Order
- Descending order

Types of Priority Queue

There are two types in Priority Queue:

- Ascending Order
- Descending order

Ascending Order Priority Queue:

- An ascending order priority queue gives the highest priority to the lower number in that queue.

Types of Priority Queue

There are two types in Priority Queue:

- Ascending Order
- Descending order

Ascending Order Priority Queue:

- An ascending order priority queue gives the highest priority to the lower number in that queue.
- **Example:** Assume that you have six numbers in the priority queue that are **20, 40, 15, 50, 8, 7**. Firstly, you will arrange these numbers in ascending order.

Types of Priority Queue

There are two types in Priority Queue:

- Ascending Order
- Descending order

Ascending Order Priority Queue:

- An ascending order priority queue gives the highest priority to the lower number in that queue.
- **Example:** Assume that you have six numbers in the priority queue that are **20, 40, 15, 50, 8, 7**. Firstly, you will arrange these numbers in ascending order.
- **The new list is as follows: 7, 8, 15, 20, 40, 50**. In this list, **7** is the smallest number.

Types of Priority Queue

There are two types in Priority Queue:

- Ascending Order
- Descending order

Ascending Order Priority Queue:

- An ascending order priority queue gives the highest priority to the lower number in that queue.
- **Example:** Assume that you have six numbers in the priority queue that are **20, 40, 15, 50, 8, 7**. Firstly, you will arrange these numbers in ascending order.
- **The new list is as follows: 7, 8, 15, 20, 40, 50**. In this list, **7** is the smallest number.
- Hence, the ascending order priority queue treats number **7** as the highest priority.
  - **Before applying of ascending order table like this.**

20	40	15	50	6	7

Types of Priority Queue

There are two types in Priority Queue:

- Ascending Order
- Descending order

Ascending Order Priority Queue:

- An ascending order priority queue gives the highest priority to the lower number in that queue.
- **Example:** Assume that you have six numbers in the priority queue that are **20, 40, 15, 50, 8, 7**. Firstly, you will arrange these numbers in ascending order.
- **The new list is as follows: 7, 8, 15, 20, 40, 50**. In this list, **7** is the smallest number.
- Hence, the ascending order priority queue treats number 5 as the highest priority.
  - **Before applying of ascending order table like this.**

20	40	15	50	6	7

  - **After applying of ascending order table like this.**

7	8	15	20	40	50

  - **In the above table 7 has the highest priority, and 50 has the lowest priority.**

Types of Priority Queue

Descending Order Priority Queue:

- A descending order priority queue gives the highest priority to the highest number in that queue.
- **Example:** Assume that you have six numbers in the priority queue that are **20, 40, 15, 50, 8, 7**. Firstly, you will arrange these numbers in descending order.
- **The new list is as follows:** **50, 40, 20, 15, 8, 7**. In this list, **50** is the highest number. Hence, the descending order priority queue treats number **50** as the highest priority.
  - **Before applying of descending order table like this.**

20	40	15	50	8	7

  - **After applying of descending order table like this.**

50	40	20	15	8	7

  - **In the above table 50 has the highest priority, and 7 has the lowest priority.**

Characteristics of a Priority Queue

- **A queue is termed as a priority queue if it has the following characteristics:**
  - Each item has some **priority** associated with it.
  - An item with the **highest priority** is moved at the **front** and **process first (e.g. delete)**.
  - If two elements share the **same priority value**, then the priority queue follows the **first-in-first-out principle** for dequeue operation.

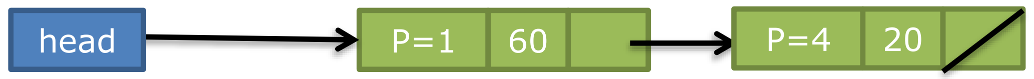
Characteristics of a Priority Queue

- **A queue is termed as a priority queue if it has the following characteristics:**
  - Each item has some priority associated with it.
  - An item with the highest priority is moved at the front and deleted first.
  - If two elements share the same priority value, then the priority queue follows the first-in-first-out principle for dequeue operation.
- **Implementation of the Priority Queue in Data Structure:**
  - You can implement the priority queues in one of the following ways:
    1. Arrays.
       1. Unordered Array
       2. Ordered Array
    2. **Linked list**
    3. Binary heap
       1. Min heap
       2. Max heap
    4. Binary search tree

Implementing Priority Queue using Linked List

```
class PriorityNode:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority
        self.next = None
```

```
class PriorityQueue:
    def __init__(self):
        self.head = None
        self.size = 0
```



Core Priority Queue operations

- `peek()`: Inspect the highest priority item in the queue without removing it.
- `enqueue()`: Add an item to the queue based on its priority.
- `dequeue()`: Remove and return the item with the highest priority from the queue.
- `IsEmpty()`: Check if the queue has no more items remaining
- `getSize()`: Returns the current number of items in the priority queue.

peek()

- Inspect the item at the front of the queue without removing it

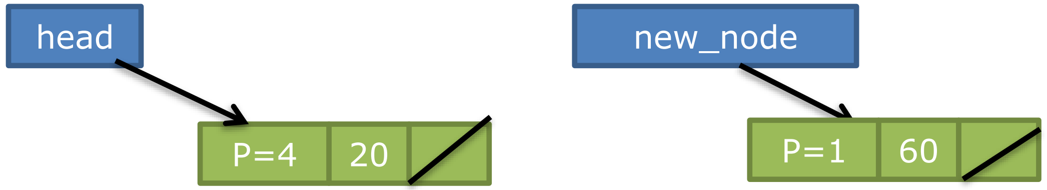
```
def peek(self):
        if self.isEmpty():
            raise IndexError("Queue is empty")
        return self.head.data
```

enqueue(): Add an item to the queue based on its priority

```
class PriorityNode:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority
        self.next = None
```

```
def enqueue(self, data, priority):
       new_node = PriorityNode(data, priority)
       if self.isEmpty() or priority < self.head.priority:
           new_node.next = self.head
           self.head = new_node
       else:
           current = self.head
           while current.next and current.next.priority <= priority:
               current = current.next
           new_node.next = current.next
           current.next = new_node
       self.size += 1
```

enqueue(): Add an item to the queue based on its priority

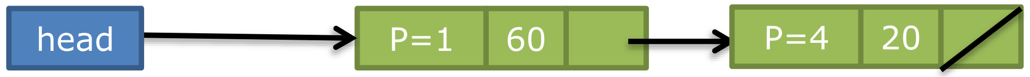


```
class PriorityNode:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority
        self.next = None
```

```
def enqueue(self, data, priority):
    new_node = PriorityNode(data, priority)
    if self.isEmpty() or priority < self.head.priority:
        new_node.next = self.head
        self.head = new_node
    else:
        current = self.head
        while current.next and current.next.priority <= priority:
            current = current.next
        new_node.next = current.next
        current.next = new_node
    self.size += 1
```

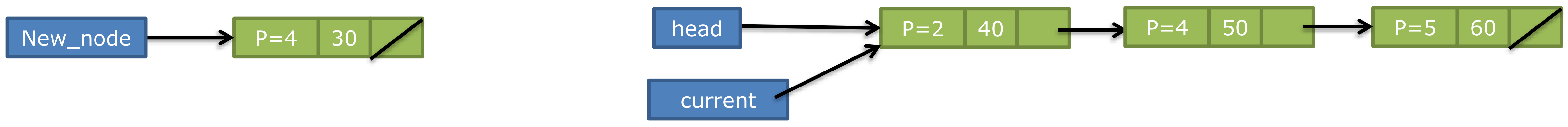
enqueue(): Add an item to the queue based on its priority

```
class PriorityNode:
    def __init__(self, data, priority):
        self.data = data
        self.priority = priority
        self.next = None
```



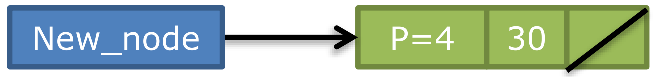
```
def enqueue(self, data, priority):
    new_node = PriorityNode(data, priority)
    if self.isEmpty() or priority < self.head.priority:
        new_node.next = self.head
        self.head = new_node
    else:
        current = self.head
        while current.next and current.next.priority <= priority:
            current = current.next
        new_node.next = current.next
        current.next = new_node
    self.size += 1
```

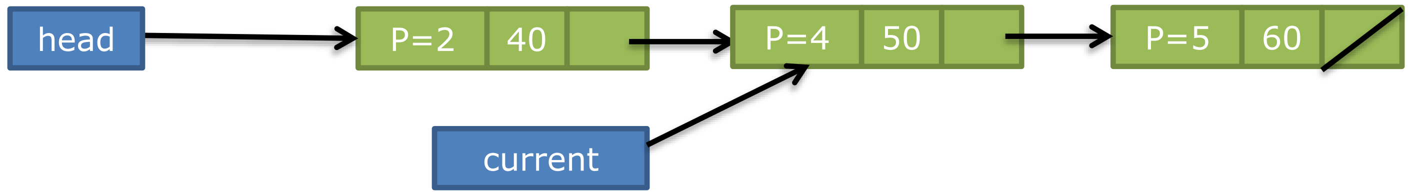
enqueue(): Add an item to the queue based on its priority



```
def enqueue(self, data, priority):
       new_node = PriorityNode(data, priority)
       if self.isEmpty() or priority < self.head.priority:
           new_node.next = self.head
           self.head = new_node
       else:
           current = self.head
           while current.next and current.next.priority <= priority:
               current = current.next
           new_node.next = current.next
           current.next = new_node
       self.size += 1
```

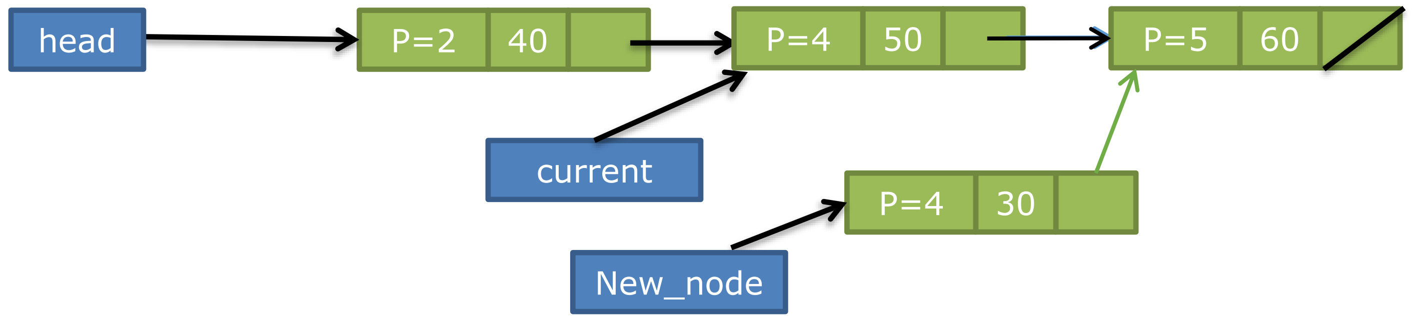
enqueue(): Add an item to the queue based on its priority





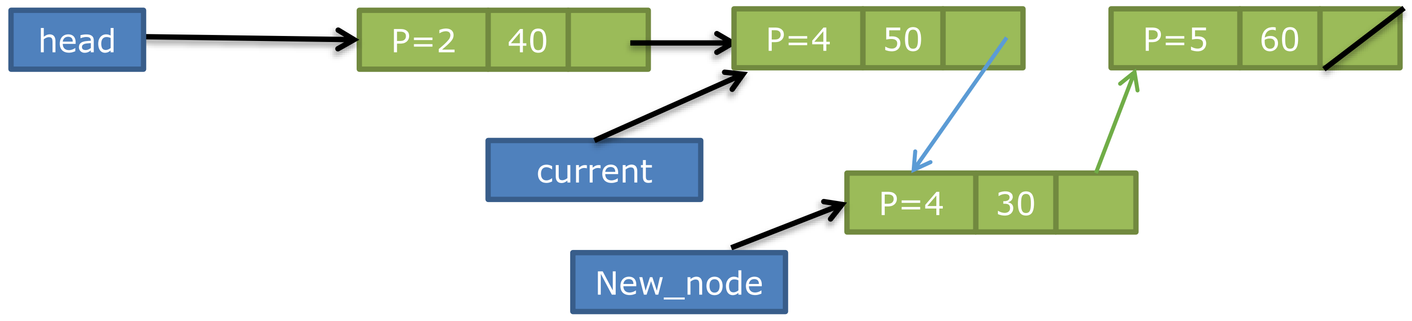
```
def enqueue(self, data, priority):
       new_node = PriorityNode(data, priority)
       if self.isEmpty() or priority < self.head.priority:
           new_node.next = self.head
           self.head = new_node
       else:
           current = self.head
           while current.next and current.next.priority <= priority:
               current = current.next
           new_node.next = current.next
           current.next = new_node
       self.size += 1
```

enqueue(): Add an item to the queue based on its priority



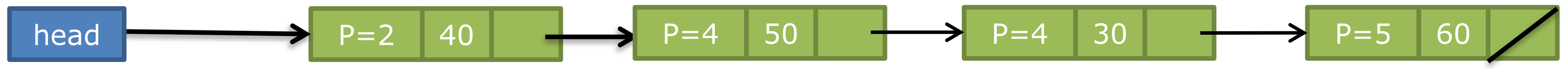
```
def enqueue(self, data, priority):
        new_node = PriorityNode(data, priority)
        if self.isEmpty() or priority < self.head.priority:
            new_node.next = self.head
            self.head = new_node
        else:
            current = self.head
            while current.next and current.next.priority <= priority:
                current = current.next
            new_node.next = current.next
            current.next = new_node
        self.size += 1
```

enqueue(): Add an item to the queue based on its priority



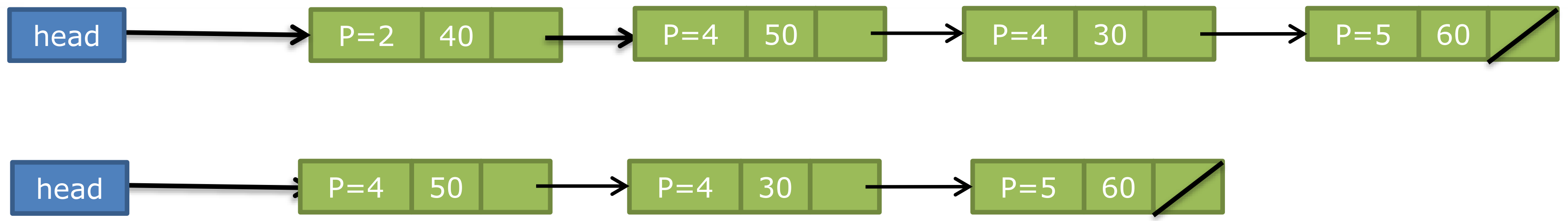
```
def enqueue(self, data, priority):
       new_node = PriorityNode(data, priority)
       if self.isEmpty() or priority < self.head.priority:
           new_node.next = self.head
           self.head = new_node
       else:
           current = self.head
           while current.next and current.next.priority <= priority:
               current = current.next
           new_node.next = current.next
           current.next = new_node
       self.size += 1
```

enqueue(): Add an item to the queue based on its priority



```
def enqueue(self, data, priority):
       new_node = PriorityNode(data, priority)
       if self.isEmpty() or priority < self.head.priority:
           new_node.next = self.head
           self.head = new_node
       else:
           current = self.head
           while current.next and current.next.priority <= priority:
               current = current.next
           new_node.next = current.next
           current.next = new_node
       self.size += 1
```

dequeue()



```
def dequeue(self):
        if self.isEmpty():
            raise IndexError("Queue is empty")
        removed_data = self.head.data
        self.head = self.head.next
        self.size -= 1
        return removed_data
```

isEmpty()

- Check whether the queue is empty

```
def isEmpty(self):
        return self.head is None
```

getSize()

- Returns the current size of the queue

```
def getSize(self):
        return self.size
```

Stack and Queues: Applications

Stack and Queues: Applications

Stack

- Balanced Parentheses Problem (Lab)
- Algebraic expression conversion (infix, prefix and postfix)
- Recursive functions to Iterative functions

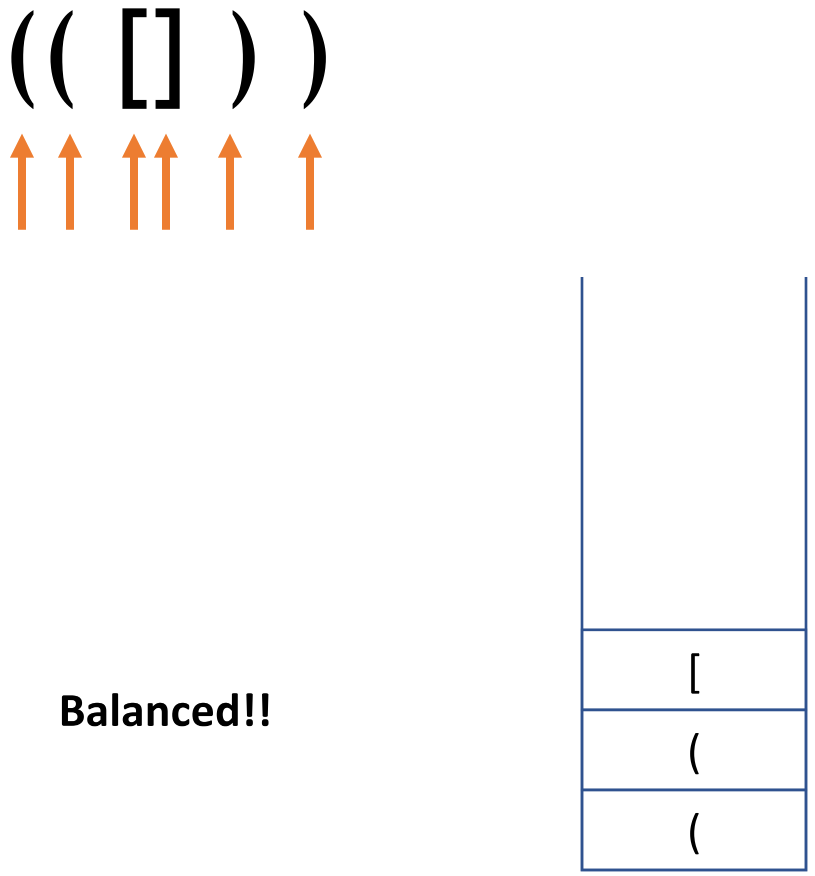
Queue

- Palindromes (Lab)
- Scheduling in multitasking, network, job, mailbox etc.

Application: Using a Stack to Ensure Brackets and Parentheses Are Balanced

- ([ ] ({ ( ) } [( )])) is balanced;
  ([ ] ({ ( ) } [( ))]) is not
- Simple counting is not enough to check the balance
- You can do it with a stack: going left to right,
  - If you see a **(**, **[**, or **{**, **push** it on the stack
  - If you see a **)**, **]**, or **}**, **pop** the stack and check whether you popped the corresponding (, [, or {
  - When you reach the end, check that the **stack is empty**

Application: Using a Stack to Ensure Brackets and Parentheses Are Balanced



Balanced!!

Infix

Operators	Precedence
*, /, %	Highest
+, -	
<<, >>	
&&	
=	Lowest

- While writing an arithmetic expression using **Infix** notation, the operator is placed between the operands.
  - For example, *A+B;* here, plus operator is placed between the two operands A and B.
- **A * (B + C) / D** means: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.

Postfix Notation

- Postfix notation which is better known as Reverse Polish Notation or RPN.
- In Postfix notation, the operator is placed after the operands. For example, if an expression is written as *A+B* in **Infix** notation, the same expression can be written as *AB+* in **Postfix** notation.
- A postfix operation does not follow the rules of **operator precedence**. The operator which occurs first in the expression is operated first on the operands.
- For example, given a postfix notation **AB+C***. While evaluation, addition will be performed prior to multiplication.
- The order of evaluation of a postfix expression is always from left to right.

Prefix Notation

- In a **Prefix notation**, the operator is placed before the operands.
- For example, if A+B is an expression in **Infix notation**, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence and associativity.
- The expression (A + B) * C is written as: *+ABC in the prefix notation
- So, computers work more efficiently with expressions written using Prefix and Postfix notations.

Tutorial 04

Given the precedence of some operators,

Operators	Precedence
*, /, %	Highest
+, -	
<<, >>	
&&	
=	Lowest

a) Convert an **infix** expression, x = a + b * c%d >> e, to a **postfix** expression

b) Convert a **prefix** expression, = y&& << ab >> c + de, to an **infix** expression

c) Convert a **postfix** expression, xabc * d% + e >>=, to a **prefix** expression

Prefix to infix

Do:

- Read the given prefix expression from right to left
- If the character is an operand, push it into the stack.
- But if the character is an operator, pop the top two values from stack.
- Concatenate this operator with these two values (***1st top value+operator+2nd top value***) to get a new string.
- Push this resulting string back into the stack.

Until:

- End of prefix expression
- Value in the stack is the desired infix expression.

Question 4 – (b)

Convert a **prefix** expression, **= y&& << ab >> c + de**, to an **infix** expression

Do:

- Read the given prefix expression from right to left
- If the character is an operand, push it into the stack.
- But if the character is an operator, pop the top two values from stack.
- Concatenate this operator with these two values (***1st top value+operator+2nd top value***) to get a new string.
- Push this resulting string back into the stack.

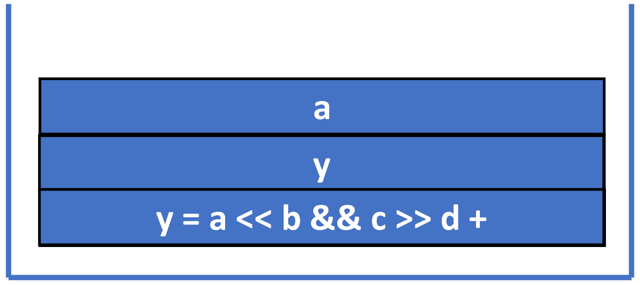
Until:

- End of prefix expression
- Value in the stack is the desired infix expression.

Prefix : An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2). **Example** : ***+AB-CD (Infix : (A+B) * (C-D))**

Infix : An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2). **Example** : **(A+B) * (C-D)**

=	y	&&	<<	a	b	>>	c	+	d	e



y = a << b && c >> d + e

y	=	a	<<	b	&&	c	>>	d	+	e

Operators	Precedence
*, /, %	Highest
+, -	
<<, >>	
&&	
=	Lowest

Performing calculations

- To evaluate any expression, such as 1+2*3+4 we make two passes through the expression:
  - Convert the "**infix notation**" expression to "**postfix notation**" (also known as "Reverse Polish Notation", RPN)
  - Evaluate the postfix expression to compute its value
  - Each pass uses a stack

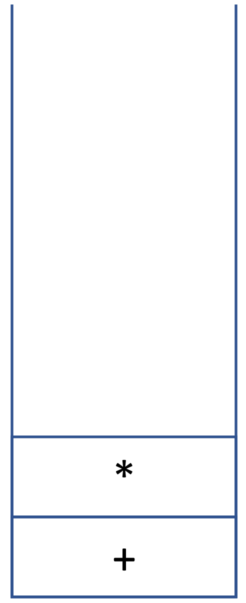
How to convert Infix to Postfix

Input	Action
Operand	Write to output
Operator	- While operator on stack top is >= precedence of the input operator – pop and output the operator from stack top. - Push the input operator onto stack
<END>	While stack not empty, pop and output the operator at top of stack

How to convert Infix to Postfix

Converting 1+2*3+4 to postfix

Input token	Action	Output so far	Stack
1	Add “1” to postfix expression	1	
+	Push “+” onto stack	1	+
2	Add “2” to postfix expression	1 2	+
*	Push “*” onto stack (higher precedence than +)	1 2	+ *
3	Add “3” to postfix expression	1 2 3	+ *
+	Pop “*” and “+”, add to postfix, then push “+” (* and + >= precedence (+).	1 2 3 * +	+
4	Add “4” to postfix expression	1 2 3 * + 4	+
<END>	Pop remaining “+” and add to postfix	1 2 3 * + 4 +	



Output:

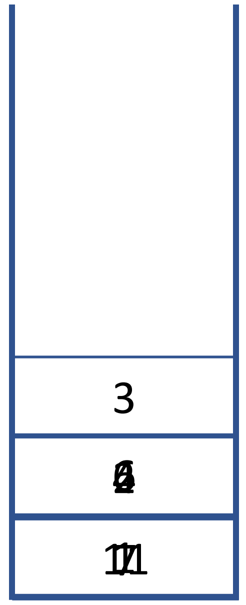
1 2 3 * + 4 +

Evaluating a Postfix expression

Infix: 1+2*3+4   Postfix: 1 2 3 * + 4 +

Input	Action
Operator + - * /	- Pop two operands from stack - Combine them depending upon the operator - Push result back onto stack
Operand	Push int() of operand to stack [Init(): converts a string or character into an integer.]
End (empty input list)	Pop result from stack

	Input	Stack
●	"1"	1
●	"2"	1 2
●	"3"	1 2 3
●	"*"	1 6
●	"+"	7
●	"4"	7 4
●	"+"	11



Output:

11

Algorithm: Infix to Postfix Conversion

```
1. Append ')' to the end of the infix expression.
2. Push '(' onto the stack.
3. For each character in the infix expression from left to right:
     a. If the character is '(':
          •Push it onto the stack.
     b. If the character is an operand:
          •Add it to the postfix expression.
     c. If the character is ')':
          •Pop operators from the stack and append them to the postfix expression until '(' is encountered.
          •Discard the '(' (pop it from the stack but do not add to the postfix expression).
     d. If the character is an operator X:
          •While the stack is not empty and the operator at the top of the stack has equal or higher precedence
          than X:
               •Pop the operator from the stack and append it to the postfix expression.
          •Push X onto the stack.
4.After scanning all characters:
     •Pop any remaining operators from the stack and append them to the postfix expression.
5.Terminate the algorithm.
```

Convert Infix to Postfix : 7 - ((3 * 4) + 8) / 5

Step 1: infix "(7 - ((3 * 4) + 8) / 5)" => postfix "7 3 4 * 8 + 5 / -"

NO	infix	Stack	postfix
1	(	(	
2	7	(	7
3	-	(-	7
4	(	(-(	7
5	(	(-((	7
6	3	(-((	73
7	*	(-((*	7 3
8	4	(-((*	7 3 4
9	)	(-(	7 3 4 *
10	+	(-(+	7 3 4 *
11	8	(-(+	7 3 4 * 8
12	)	(-	7 3 4 * 8 +
13	/	(-/	7 3 4 * 8 +
14	5	(-/	7 3 4 * 8 + 5
15	)		7 3 4 * 8 + 5 / -

Algorithm: Evaluate Postfix Expression

```
1. Create an empty stack S.
2. For each character c in the postfix expression from left to right:
    a. If c is an operand:
        • Push c onto the stack S.
    b. If c is an operator:
        • Pop the top element from the stack and store it in operand1.
        • Pop the next top element from the stack and store it in operand2.
        • Evaluate the result of operand2 <operator> operand1.
        • Push the result back onto the stack S.
3. After scanning all characters:
    • Pop the result from the stack.
4. Terminate the algorithm.
```

Convert Infix to Postfix and Evaluate: 7 - ((3 * 4) + 8) / 5

Step 1: infix “(7 - ((3 * 4) + 8) / 5)" => postfix “7 3 4 * 8 + 5 / -“

NO	infix	Stack	postfix
1	(	(	
2	7	(	7
3	-	(-	7
4	(	(-(	7
5	(	(-((	7
6	3	(-((	73
7	*	(-((*	7 3
8	4	(-((*	7 3 4
9	)	(-(	7 3 4 *
10	+	(-(+	7 3 4 *
11	8	(-(+	7 3 4 * 8
12	)	(-	7 3 4 * 8 +
13	/	(-/	7 3 4 * 8 +
14	5	(-/	7 3 4 * 8 + 5
15	)		7 3 4 * 8 + 5 / -

Step 2 evaluate “7 3 4 * 8 + 5 / -“

No	Character scanned	Stack
1	7	7
2	3	7, 3
3	4	7, 3, 4
4	*	7, 12
5	8	7, 12, 8
6	+	7, 20
7	5	7, 20, 5
8	/	7, 4
9	-	3

Stack and Queues: Applications

Stack

- Balanced Parentheses Problem
- Algebraic expression conversion (infix, prefix and postfix)
- Recursive functions to Iterative functions

Queue

- Palindromes
- Scheduling in multitasking, network, job, mailbox etc.

Applications of Priority Queue

- Prim's algorithm implementation can be done using priority queues.
- Dijkstra's shortest path algorithm implementation can be done using priority queues.
- A* Search algorithm implementation can be done using priority queues.
- Priority queues are used to sort heaps.
- Priority queues are used in operating system for load balancing and interrupt handling.
- Priority queues are used in huffman codes for data compression.
- In traffic light, depending upon the traffic, the colours will be given priority.

Implement Infix to Postfix Conversion in python

```
1. Append ')' to the end of the infix expression.
2. Push '(' onto the stack.
3. For each character in the infix expression from left to right:
      a. If the character is '(':
            •Push it onto the stack.
      b. If the character is an operand:
            •Add it to the postfix expression.
      c. If the character is ')':
            •Pop operators from the stack and append them to the postfix expression until '(' is encountered.
            •Discard the '(' (pop it from the stack but do not add to the postfix expression).
      d. If the character is an operator X:
            •While the stack is not empty and the operator at the top of the stack has equal or higher precedence
            than X:
                  •Pop the operator from the stack and append it to the postfix expression.
            •Push X onto the stack.
4.After scanning all characters:
      •Pop any remaining operators from the stack and append them to the postfix expression.
5.Terminate the algorithm.
```