Problem Solving using Stack, Queue, and Deque

Problem Solving using Stack

- Write a function rev_string(my_str) that uses a stack to reverse the characters in a string

```
import Stack

def rev_string(my_str):
    myStr = Stack.Stack()

    print("The Input String is: " + my_str)
    for index in range(len(my_str)):
        myStr.push(my_str[index])

    rev_str = ""
    for i in range(len(my_str)):
        rev_str += myStr.pop()

    print("The Reversed String is: " + rev_str)

rev_string("Hello World")
print()
rev_string("0123456789")
```

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0, item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)
```

Problem Solving using Stack

- **Simple Balance Parentheses**
- Arithmetic expressions such as:

  (5 + 6) * (7 + 8)/(4 + 3)

- **Balanced parentheses**: แต่ละวงเล็บเปิดต้องมีวงเล็บปิด วงเล็บ เปิด-ปิด มีได้หลายคู่ ลึกลงไปได้หลายชั้น วงเล็บปิดแต่ละตัวจะคู่กับวงเล็บเปิดที่ใกล้มันมากที่สุด

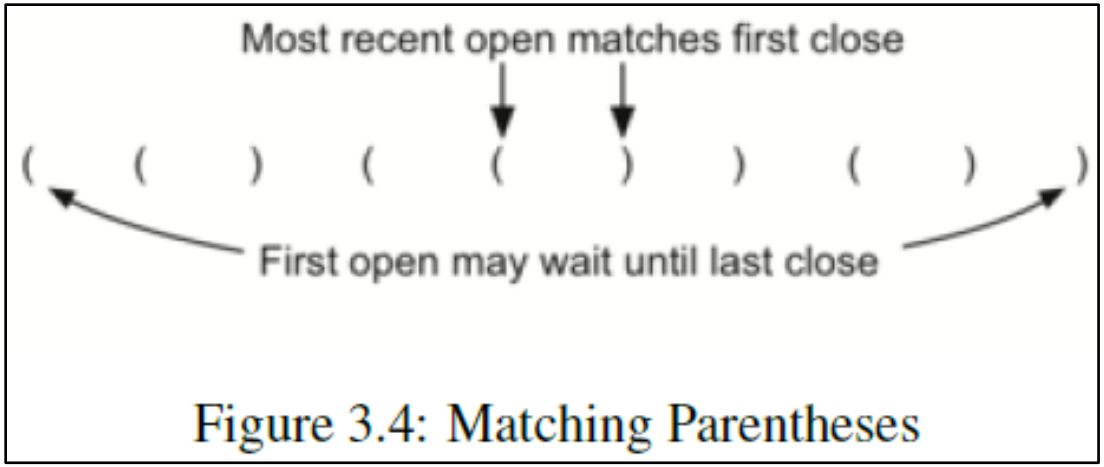


Figure 3.4: Matching Parentheses

Problem Solving using Stack

- **Simple Balance Parentheses (Cont.)**
- ตัวอย่างวงเล็บที่มีหลายคู่

ที่ถูกต้อง	ที่ไม่ถูกต้อง
(()()()())	((((((())
((((()))))	())))
(()((()))()))	(()()(()

- **เราจะตรวจสอบอย่างไร?**

Problem Solving using Stack

- **Simple Balance Parentheses (Cont.)**

```
import Stack #import the Stack class as previously defined

def par_checker(symbol_string):
    s = Stack.Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.is_empty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.is_empty():
        return True
    else:
        return False

print(par_checker('((()))'))
print(par_checker('(()'))
# ถ้า Input String เป็น (5 + 6) * (7 + 8)/(4 + 3) จะต้องเพิ่ม Code ยังไง?
```

Problem Solving using Stack

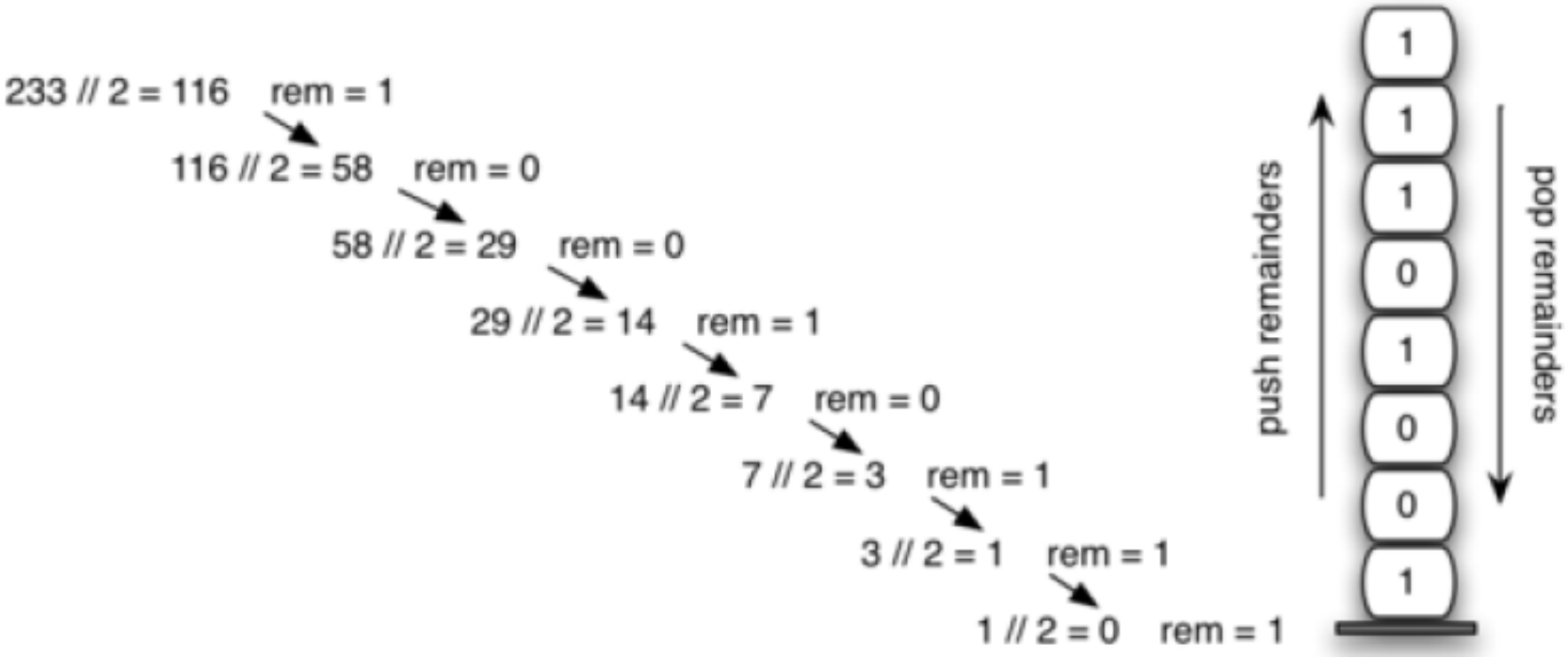
- **Balance Symbols**

```
import Stack # As previously defined

# Completed extended par_checker for: [,{,(,),},]

def par_checker(symbol_string):
    s = Stack.Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.is_empty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top, symbol):
                    balanced = False
        index = index + 1

    if balanced and s.is_empty():
        return True
    else:
        return False
```

```
def matches(open, close):
    opens = "([{"
    closes = ")]}"
    return opens.index(open) == closes.index(close)


print(par_checker('{{([][])}()}'))
print(par_checker('[{()]'))
```

Problem Solving using Stack

- **Converting Decimal Numbers to Binary Numbers**



- **How to?**

Problem Solving using Stack

- **Operator Precedence**
- **เราจะบอกคอมพิวเตอร์ได้อย่างไรว่า ใน Arithmetic Expression เราจะทำ Arithmetic Operation ไหนก่อน เช่น**

$A + B * C$ → **(A + (B * C))**

$A + B + C$ → **((A + B) + C)**

Problem Solving using Stack

- **Infix Prefix Postfix**
- **เราสามารถนำเสนอ Expression ได้ 3 รูปแบบ**
  - **Infix: operator อยู่ระหว่าง Operand**
  - **Prefix: operator อยู่ก่อน Operand**
  - **Postfix: operator อยู่หลัง Operand**

operator เป็นของ Operand ตัวที่อยู่ใกล้มันที่สุด

Infix Expression	Prefix Expression	Postfix Expression
$A + B$	$+AB$	$AB+$
$A + B * C$	$+A * BC$	$ABC * +$

Table 3.2: Examples of Infix, Prefix, and Postfix

Infix Expression	Prefix Expression	Postfix Expression
$(A + B) * C$	$* + ABC$	$AB + C*$

Table 3.3: An Expression with Parentheses

Problem Solving using Stack

- **Conversion of Infix Expressions to Prefix and Postfix**
- **ตัวอย่างลำดับการทำ Operation ที่ถูกต้อง**

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$++A*BCD$	$ABC*+D+$
$(A + B) * (C + D)$	$*+AB+CD$	$AB+CD+*$
$A * B + C * D$	$+*AB*CD$	$AB*CD*+$
$A + B + C + D$	$+++ABCD$	$AB+C+D+$

Table 3.4: Additional Examples of Infix, Prefix, and Postfix

Problem Solving using Stack

- **Conversion of Infix Expressions to Prefix and Postfix (Cont.)**

- **How to move operator?**

(A + (B * C))

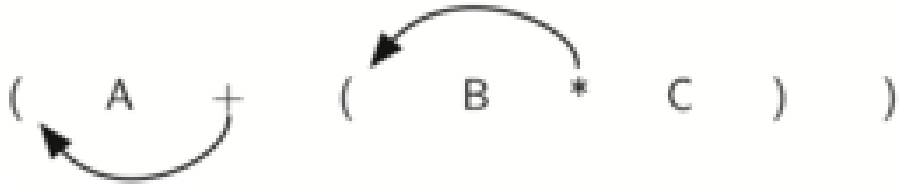


Figure 3.7: Moving Operators to the Left for Prefix Notation)

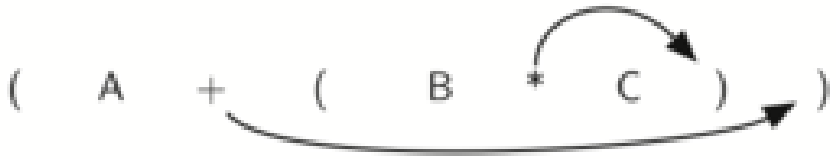
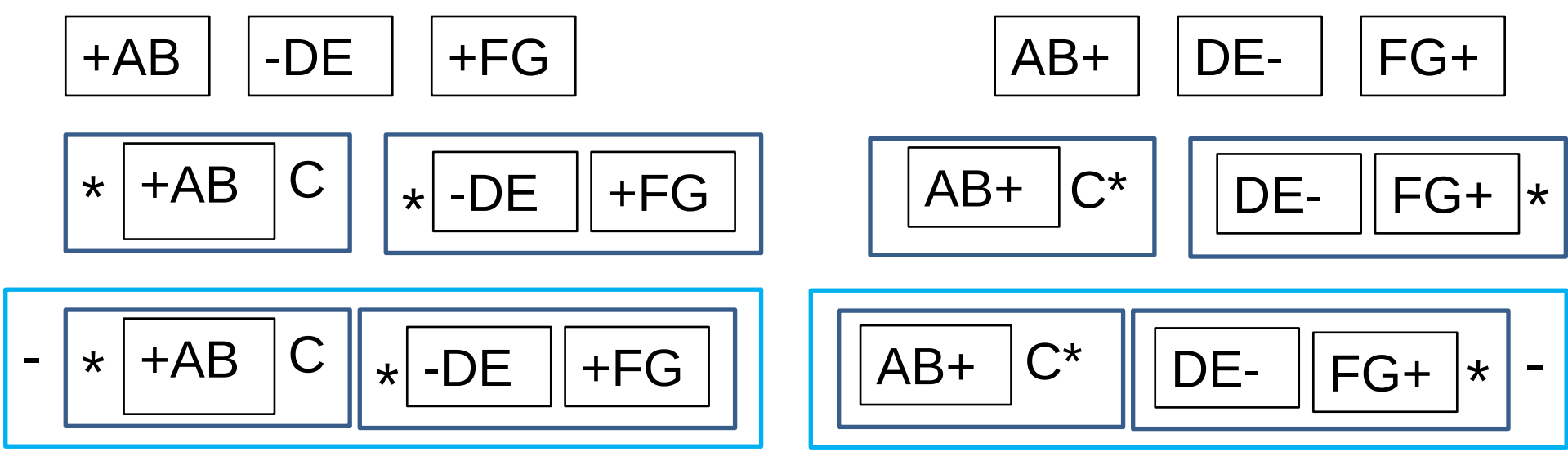


Figure 3.6: Moving Operators to the Right for Postfix Notation)

Problem Solving using Stack

- **Conversion of Infix Expressions to Prefix and Postfix (Cont.)**

(A + B) * C – (D – E) * (F + G)



Problem Solving using Stack

Postfix Order Algorithm

1. Create an empty stack called op_stack for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method split.
3. Scan the token list from left to right.
   - If the token is an operand, append it to the end of the output list.
   - If the token is a left parenthesis, push it on the op_stack.
   - If the token is a right parenthesis, pop the op_stack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
   - If the token is an operator, $*$, $/$, $+$, or $-$, push it on the op_stack. However, first remove any operators already on the op_stack that have higher or equal precedence and append them to the output list.

When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

Problem Solving using Stack

- **Postfix Order Algorithm (Cont.)**

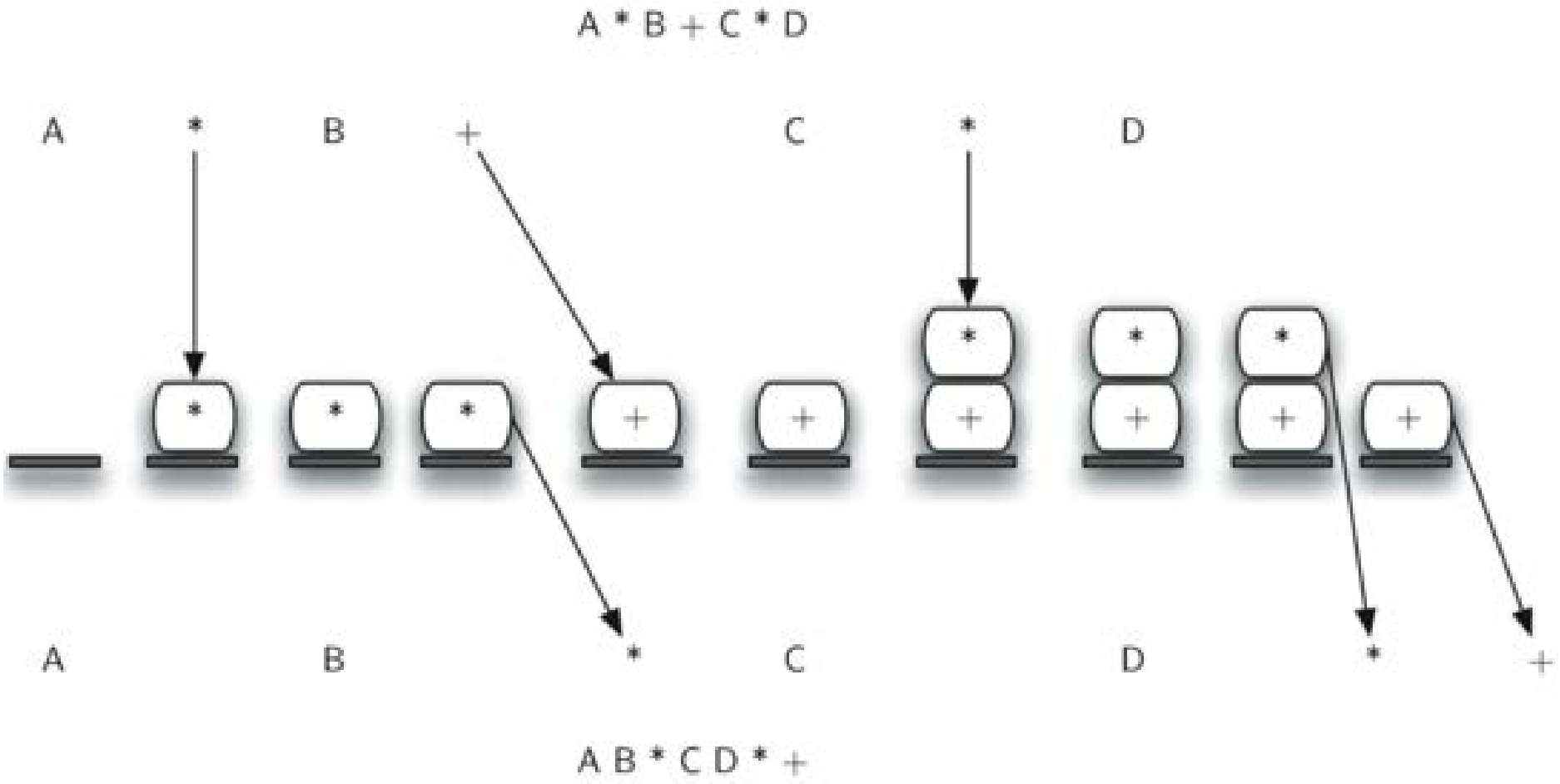


Figure 3.9: Converting $A * B + C * D$ to Postfix Notation)

Problem Solving using Stack

- **Postfix Order Algorithm: Example 1**

A * B + C * D

Step1: A	**Output List: A**	**op_stack:**
Step2: *	**Output List: A**	**op_stack: ***
Step3: B	**Output List: A B**	**op_stack: ***
Step4: +	**Output List: A B ***	**op_stack: +**
Step5: C	**Output List: A B * C**	**op_stack: +**
Step6: *	**Output List: A B * C**	**op_stack: + ***
Step7: D	**Output List: A B * C D**	**op_stack: + ***
Step8:	**Output List: A B * C D ***	**op_stack: + ***
Step9:	**Output List: A B * C D * +**	**op_stack:**

Problem Solving using Stack

- **Postfix Order Algorithm: Example 2** (A + B) * (C + D)

Step1: (	Output List:	op_stack: (
Step2: A	Output List: A	op_stack: (
Step3: +	Output List: A	op_stack: (+
Step4: B	Output List: A B	op_stack: (+
Step5:)	Output List: A B +	op_stack:
Step6: *	Output List: A B +	op_stack: *
Step7: (	Output List: A B +	op_stack: * (
Step8: C	Output List: A B + C	op_stack: * (
Step9: +	Output List: A B + C	op_stack: * (+
Step10: D	Output List: A B + C D	op_stack: * (+
Step11:)	Output List: A B + C D +	op_stack: *
Step12:	Output List: A B + C D + *	op_stack:

Problem Solving using Stack

- **Infix to Postfix: Code**

```
import Stack  # As previously defined

def infix_to_postfix(infix_expr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    op_stack = Stack.Stack()
    postfix_list = []
    token_list = infix_expr.split()

    for token in token_list:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in
            "0123456789":
            postfix_list.append(token)
        elif token == '(':
            op_stack.push(token)
        elif token == ')':
            top_token = op_stack.pop()
            while top_token != '(':
                postfix_list.append(top_token)
                top_token = op_stack.pop()
        else:
            while (not op_stack.is_empty()) and \
               (prec[op_stack.peek()] >= prec[token]):
                 postfix_list.append(op_stack.pop())
            op_stack.push(token)

    while not op_stack.is_empty():
        postfix_list.append(op_stack.pop())
    return " ".join(postfix_list)
```

```
myString = "( 345 + 456 ) * 2".split()
print(myString)
['(', 345', '+', '456', ')', '*', '2']
```

Problem Solving using Stack

- **Postfix Evaluation: Code**

```
import Stack # As previously defined

def postfix_eval(postfix_expr):
    operand_stack = Stack.Stack()
    token_list = postfix_expr.split()
    for token in token_list:
        if token in "0123456789":
            operand_stack.push(int(token))
        else:
            operand2 = operand_stack.pop()
            operand1 = operand_stack.pop()
            result = do_math(token, operand1, operand2)
            operand_stack.push(result)
    return operand_stack.pop()
```

```
def do_math(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2

print(postfix_eval('7 8 + 3 2 + /'))
```

Practice: Implementation
- **Infix to Prefix**
- **Prefix Evaluation**

Problem Solving using Queue

- **Hot Potato**
- ให้เด็กยืนเป็นวงกลม และให้ส่งต่อมันฝรั่งร้อนให้คนที่ยืนถัดไปเรื่อยๆ ถ้าถึงเวลาที่กำหนด (อาจจะสุ่มก็ได้) มันฝรั่งอยู่ที่ใคร คนนั้นก็จะถูกออกจากวง

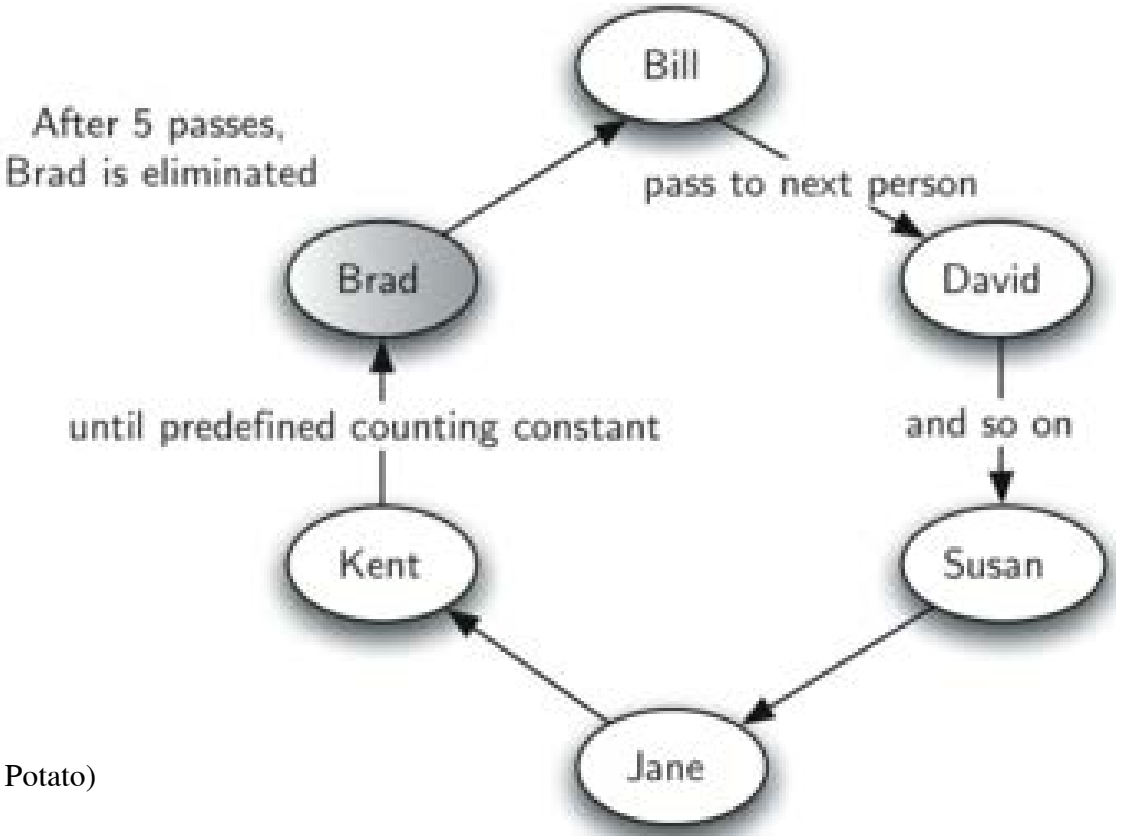


Figure 3.13: A Six Person Game of Hot Potato)

Problem Solving using Queue

- **Hot Potato: Queue Implementation**

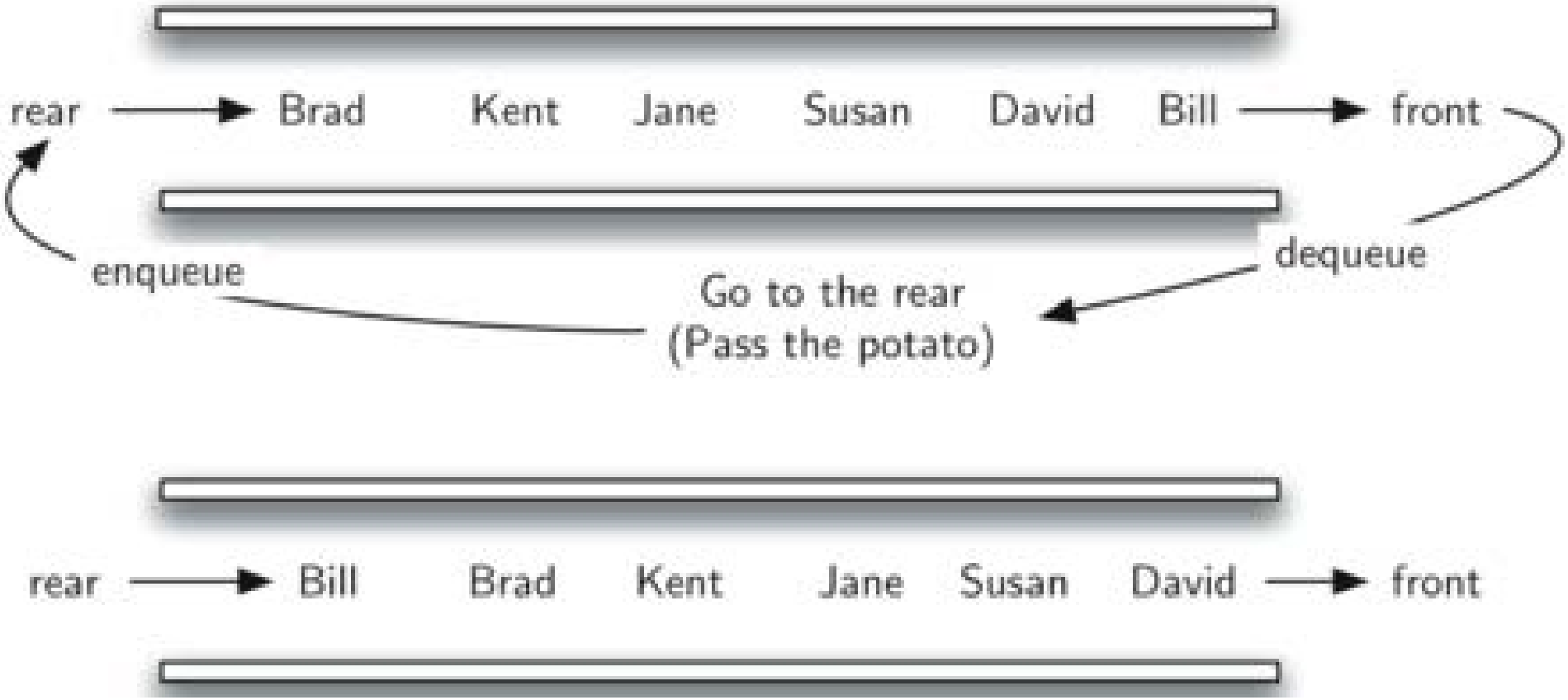
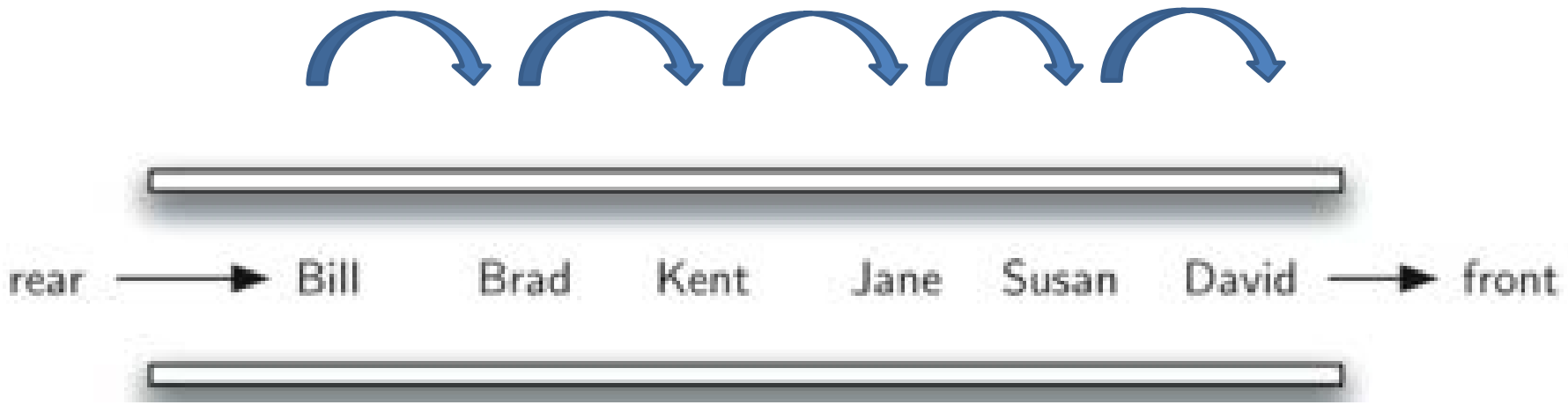


Figure 3.14: A Queue Implementation of Hot Potato)

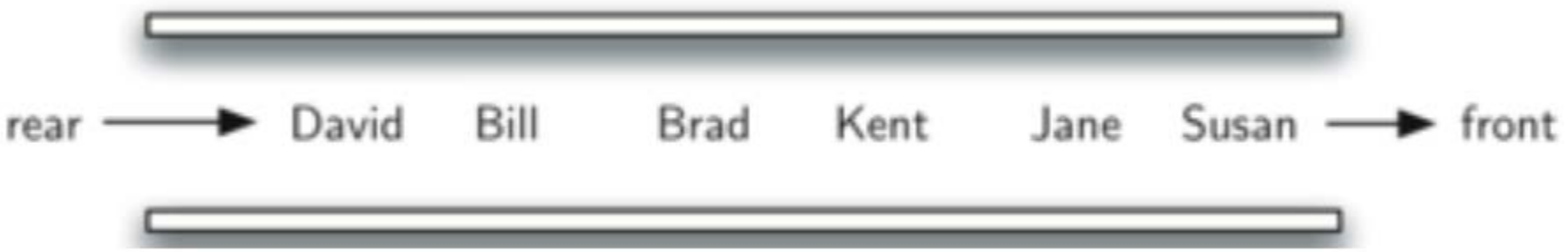
Problem Solving using Queue

- **Hot Potato: Forward Potato**

After forward potato 5 times



After forward potato 6 times



Problem Solving using Queue

- **Hot Potato: Code**

```
import Queue # As previously defined

def hot_potato(name_list, num):
    sim_queue = Queue.Queue()

    for name in name_list:
        sim_queue.enqueue(name)

    while sim_queue.size() > 1:
        for i in range(num):
            sim_queue.enqueue(sim_queue.dequeue())

        sim_queue.dequeue()
    return sim_queue.dequeue()

print(hot_potato(["Bill", "David", "Susan", "Jane", "Kent", "Brad"], 7))
```

Problem Solving using Queue

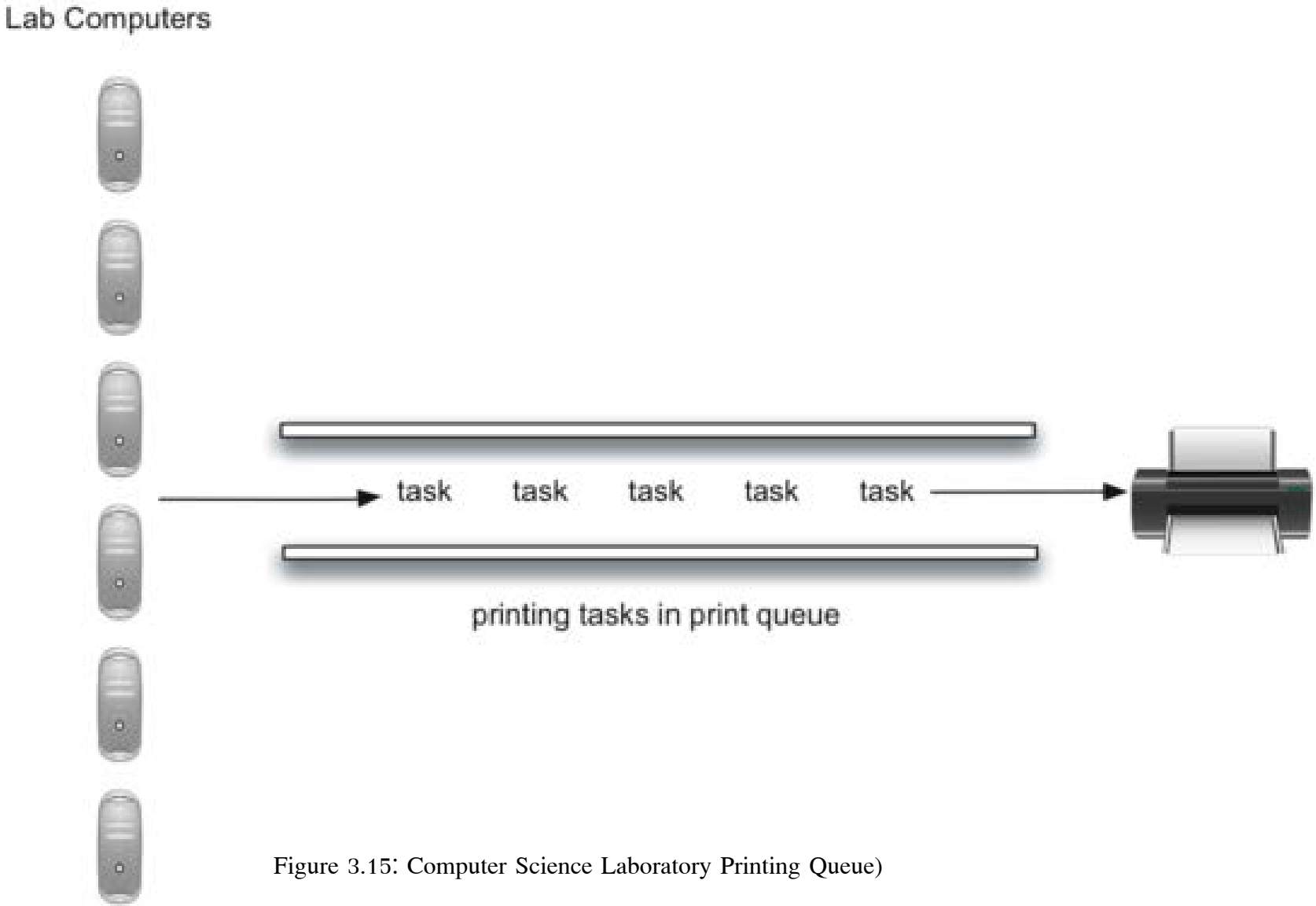


Figure 3.15: Computer Science Laboratory Printing Queue)

Problem Solving using Queue

- **Simulation: Printer Task**
  - การจำลองการเข้าคิวใช้เครื่อง **Printer** ของห้อง **Lab**
  - นักศึกษาแต่ละคน พิมพ์ในช่วง **1-20** หน้า โดยการสุ่มตัวเลข
  - ในแต่ละวินาที เราสามารถ **simulate** จำนวน **print task** โดยการสุ่มตัวเลข
  - ดังนั้น สิ่งที่เกี่ยวข้องกับปัญหาหลัก ๆ มีดังนี้คือ
    - เครื่อง **Printer**
    - **Task** ที่มารอเข้าคิว
    - คิวในการพิมพ์

Problem Solving using Queue

- **Simulation: Printer Task (Cont.)**

Algorithm ที่ใช้ในการ Simulation Print Task

1. Create a queue of print tasks. Each task will be given a timestamp upon its arrival. The queue is empty to start.
2. For each second (`currentSecond`):
    - Does a new print task get created? If so, add it to the queue with the `currentSecond` as the timestamp.
    - If the printer is not busy and if a task is waiting,
        - Remove the next task from the print queue and assign it to the printer.
        - Subtract the timestamp from the `currentSecond` to compute the waiting time for that task.
        - Append the waiting time for that task to a list for later processing.
        - Based on the number of pages in the print task, figure out how much time will be required.
    - The printer now does one second of printing if necessary. It also subtracts one second from the time required for that task.
    - If the task has been completed, in other words the time required has reached zero, the printer is no longer busy.
3. After the simulation is complete, compute the average waiting time from the list of waiting times generated.

http://interactivepython.org/runestone/static/pythonds/BasicDS/SimulationPrintingTasks.html

Problem Solving using Queue

- **Simulation: Printer Task Implementation**
- To design this simulation we will create classes for the three real-world objects described above:
  - **Printer:** track whether it has a current task
  - **Task**: represent a single printing task
  - **PrintQueue:** manage print queue

If there are 10 students in the lab and each prints twice, then there are 20 print tasks per hour on average. What is the chance that at any given second, a print task is going to be created? The way to answer this is to consider the ratio of tasks to time. Twenty tasks per hour means that on average there will be one task every 180 seconds:

$$\frac{20\ tasks}{1\ hour} \times \frac{1\ hour}{60\ minutes} \times \frac{1\ minute}{60\ seconds} = \frac{1\ task}{180\ seconds}$$

For every second we can simulate the chance that a print task occurs by generating a random number between 1 and 180 inclusive. If the number is 180, we say a task has been created. Note that it is possible that many tasks could be created in a row or we may wait quite a while for a task to appear. That is the nature of simulation. You want to simulate the real situation as closely as possible given that you know general parameters.

Problem Solving using Queue

- **Simulation: Printer Task Implementation (Cont.)**

Class Printer:

- Track whether it has a current task
- If it does, then it is busy (lines 13–17) and the amount of time needed can be computed from the number of pages in the task.
- The constructor will also allow the pages-per-minute setting to be initialized.
- The tick method decrements the internal timer and sets the printer to idle (line 11) if the task is completed.

Problem Solving using Queue

- **Simulation: Printer Task Implementation (Cont.)**

```
class Printer:
    def __init__(self, ppm):
        self.pagerate = ppm
        self.currentTask = None
        self.timeRemaining = 0

    def tick(self):
        if self.currentTask != None:
            self.timeRemaining = self.timeRemaining - 1
            if self.timeRemaining <= 0:
                self.currentTask = None

    def busy(self):
        if self.currentTask != None:
            return True
        else:
            return False

    def startNext(self,newtask):
        self.currentTask = newtask
        self.timeRemaining = newtask.getPages() * 60/self.pagerate
```

Problem Solving using Queue

- **Simulation: Printer Task Implementation (Cont.)**

Class Task:

- Represent a single printing task.
- When the task is created, a random number generator will provide a length from 1 to 20 pages.

```
>>> import random
>>> random.randrange(1,21)
18
```

  - We have chosen to use the randrange function from the random module.
- Each task will also need to keep a timestamp to be used for computing waiting time. This timestamp will represent the time that the task was created and placed in the printer queue.
  - The waitTime method can then be used to retrieve the amount of time spent in the queue before printing begins.

Problem Solving using Queue

- **Simulation: Printer Task Implementation (Cont.)**

```
import random

class Task:
    def __init__(self,time):
        self.timestamp = time
        self.pages = random.randrange(1,21)

    def getStamp(self):
        return self.timestamp

    def getPages(self):
        return self.pages

    def waitTime(self, currenttime):
        return currenttime - self.timestamp
```

Problem Solving using Queue

- **Simulation: Printer Task Implementation (Cont.)**

Class printQueue:

- Manage Task Queue.
- A boolean helper function, **newPrintTask**, decides whether a new printing task has been created.
  - We have again chosen to use the randrange function from the random module to return a random integer between 1 and 180.
  - Print tasks arrive once every 180 seconds. By arbitrarily choosing 180 from the range of random integers (line 32), we can simulate this random event.
- The simulation function allows us to set the total time and the pages per minute for the printer.

Problem Solving using Queue

- **Simulation: Printer Task Implementation (Cont.)**

```
import Queue
import Printer
import random
import Task
def simulation(numSeconds, pagesPerMinute):

    labprinter = Printer(pagesPerMinute)
    printQueue = Queue.Queue()
    waitingtimes = []

    for currentSecond in range(numSeconds):

      if newPrintTask():
         task = Task(currentSecond)
         printQueue.enqueue(task)

      if (not labprinter.busy()) and (not printQueue.isEmpty()):
        nexttask = printQueue.dequeue()
        waitingtimes.append(nexttask.waitTime(currentSecond))
        labprinter.startNext(nexttask)

      labprinter.tick()

    averageWait=sum(waitingtimes)/len(waitingtimes)
    print("Average Wait %6.2f secs %3d tasks remaining."%(averageWait,printQueue.s
ize()))
```

```
>>>for i in range(10):
      simulation(3600,5)

Average Wait 165.38 secs 2 tasks remaining.
Average Wait  95.07 secs 1 tasks remaining.
Average Wait  65.05 secs 2 tasks remaining.
Average Wait  99.74 secs 1 tasks remaining.
Average Wait  17.27 secs 0 tasks remaining.
Average Wait 239.61 secs 5 tasks remaining.
Average Wait  75.11 secs 1 tasks remaining.
Average Wait  48.33 secs 0 tasks remaining.
Average Wait  39.31 secs 3 tasks remaining.
Average Wait 376.05 secs 1 tasks remaining.
```

```
def newPrintTask():
    num = random.randrange(1,181)
    if num == 180:
        return True
    else:
        return False

for i in range(10):
    simulation(3600,5)
```

Problem Solving using Deque

- **Palindrome:** is a string that reads the same forward and backward, for example, radar, toot, and madam. We would like to construct an algorithm to input a string of characters and check whether it is a palindrome.

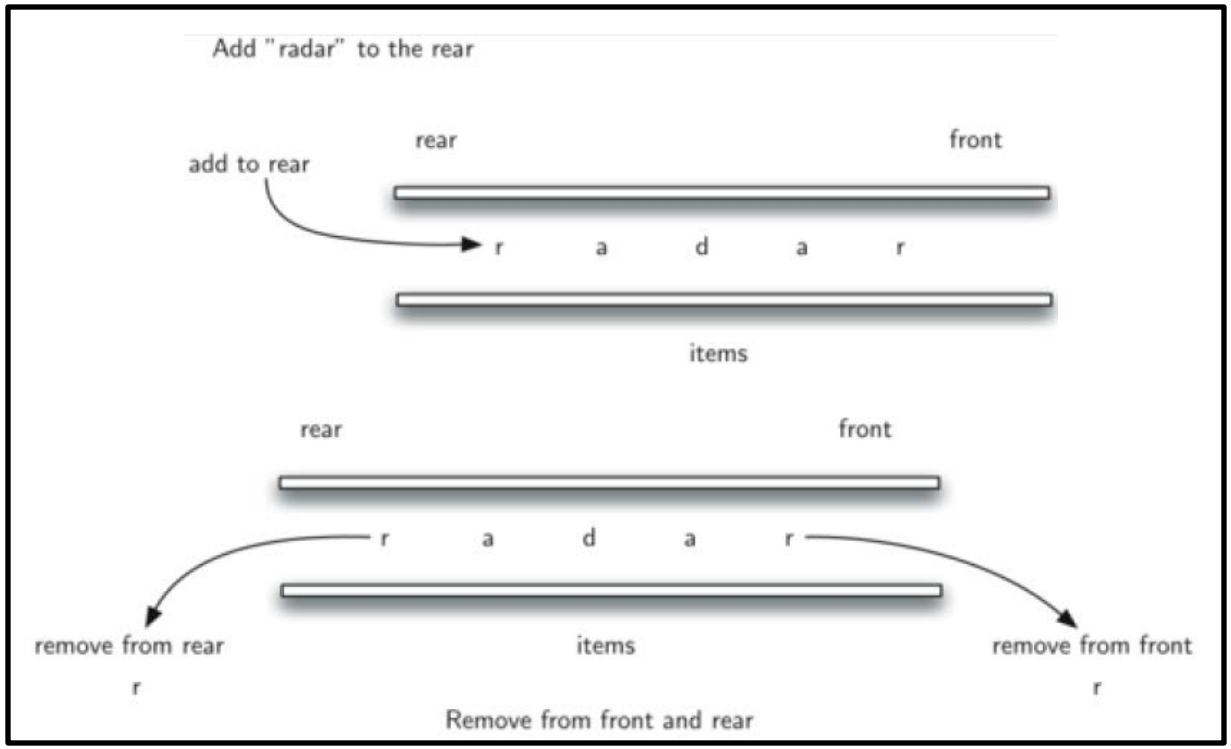


Figure 3.17: A Deque)

Problem Solving using Deque

- **Palindrome:** Implementation

```
import Deque # As previously defined
def pal_checker(a_string):
    char_deque = Deque()

    for ch in a_string:
        char_deque.add_rear(ch)

    still_equal = True

    while char_deque.size() > 1 and still_equal:
        first = char_deque.remove_front()
        last = char_deque.remove_rear()
        if first != last:
            still_equal = False

    return still_equal

print(pal_checker("lsdkjfskf"))
print(pal_checker("radar"))
```