SOLUCIONARIO

Substitution method

4.3-1. Show that the solution of $T(n) = T(n-1) + n$ is $O(n^2)$

$T(n-1) = T(n-2) + n-1$
$\Rightarrow T(n) = (T(n-2) + n-1) + n = T(n-2) + 2n - 1$

$T(n-2) = T(n-3) + n-2$
$\Rightarrow T(n) = (T(n-3) + n-2) + 2n - 1 = T(n-3) + 3n - 1 - 2$

$T(n-3) = T(n-4) + n-3$
$\Rightarrow \quad T(n) = (T(n-4) + n-3) + 3n - 1 - 2$
$T(n) = T(n-4) + 4n - 1 - 2 - 3$
$T(n) = T(n-4) + 4n - (1 + 2 + 3)$

El patron seria: $T(n) = T(n-k) + kn - (1+2+...+(k-1))$
La iteración continúa hasta que $T(n-k) = T(1)$, entonces:
$n - k = 1 \Rightarrow k = n - 1$

Reemplazando k en el patrón:

$T(n) = T(1) + (n-1)n - (1+2+...+((n-1)-1))$
$= T(1) + (n-1)n - (1+2+...+(n-2))$
$= c + n^2 - n - \sum_{i=1}^{n-2} i$
$= c + n^2 - n - \frac{(n-2)(n-1)}{2}$
$= c + n^2 - n - n^2/2 + 3n/2 - 1$
$= n^2/2 + n/2 + c - 1$

Por lo tanto, $T(n) = O(n^2)$

4.3-2. Show that the solution of $T(n) = T(\text{ceil}(n/2)) + 1$ is $O(\lg n)$
La complejidad para $T(\text{ceil}(n/2)) = T(n/2)$, entonces calculamos para $T(n) = T(n/2) + 1$

$T(n/2) = T(n/2^2) + 1$
$\Rightarrow T(n) = (T(n/2^2) + 1) + 1 = T(n/2^2) + 2$

$T(n/2^2) = T(n/2^3) + 1$
$\Rightarrow T(n) = (T(n/2^3) + 1) + 2 = T(n/2^3) + 3$

$T(n/2^3) = T(n/2^4) + 1$
$\Rightarrow \quad T(n) = (T(n/2^4) + 1) + 3 = T(n/2^4) + 4$

El patron seria: $T(n) = T(n/2^k) + k$

La iteración continúa hasta que $T(n/2^k) = T(1)$, entonces:

$n/2^k = 1$

$n = 2^k \Rightarrow \lg n = \lg 2^k \Rightarrow \lg n = k \lg 2 \Rightarrow k = \lg n$

Reemplazando k en el patrón:

$T(n) = T(n/2^k) + k$

$= T(1) + \lg n$

$= c + \lg n$

Por lo tanto, $T(n) = O(\lg n)$

4.3-3. We saw that the solution of T(n) = 2T(floor(n/2)) + n is O(n lg n). Show that the solution of this recurrence is also Ω(n lg n). Conclude that the solution is Θ(n lg n)

- 1ra suposicion: T(n) <= cnlg(n)

$$\begin{aligned} T(n) &\le 2c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + n \\ &\le cn \lg(n/2) + n \\ &\le cn \lg n - cn \lg 2 + n \\ &\le cn \lg n + (1-c)n \\ &\le cn \lg n \\ &\text{for } c \ge 0 \end{aligned}$$

- 2da suposición: T(n) >= c(n+2)lg(n+2)

$$\begin{aligned} T(n) &\ge 2c(\lfloor n/2 \rfloor + 2)(\lg(\lfloor n/2 \rfloor + 2) + n \\ &\ge 2c(n/2 - 1 + 2)(\lg(n/2 - 1 + 2) + n \\ &\ge 2c\frac{n+2}{2}\lg\frac{n+2}{2} + n \\ &\ge c(n+2)\lg(n+2) - c(n+2)\lg 2 + n \\ &\ge c(n+2)\lg(n+2) + (1-c)n - 2c \qquad \text{for } n \ge 2c/(1-c), 0 < c < 1 \\ &\ge c(n+2)\lg(n+2) \end{aligned}$$

Entonces como para T(n) se cumple que O(nlgn) = Ω(n lg n) => Θ(n lg n)

4.3-4. Show that by making a different inductive hypothesis, we can overcome the difficulty with the boundary condition T(1) = 1 for recurrence (4.19) without adjusting the boundary conditions for the inductive proof.

Realizamos la siguiente suposición: T(n) <= nlgn + n, entonces:

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + \lfloor n/2 \rfloor) + n \\ &\le 2c(n/2)\lg(n/2) + 2(n/2) + n \\ &\le cn \lg(n/2) + 2n \\ &\le cn \lg(n/2) + 2n \\ &\le cn \lg n - cn \lg 2 + 2n \\ &\le cn \lg n + (2-c)n \qquad (c \ge 1) \\ &\le cn \lg n + n \end{aligned}$$

Verificamos que también se cumpla para el caso base: T(1) = 1

T(1) <= c(1)lg(1) + 1
< = 0 + 1
<= 1

4.3-6. Show that the solution to T(n) = 2T(floor(n/2) + 17) + n is O(nlg n)

La complejidad para T(floor(n/2) + 17) = T(n/2), porque cuando n es muy grande, el 17 no influye significativamente en su complejidad, entonces calculamos para T(n) = 2T(n/2) + n

$T(n/2) = 2T(n/2^2) + n/2$
$\Rightarrow T(n) = 2(2T(n/2^2) + n/2) + n = 2^2T(n/2^2) + 2n$

$T(n/2^2) = 2T(n/2^3) + n/2^2$
$\Rightarrow T(n) = 2^2(2T(n/2^3) + n/2^2) + 2n = 2^3T(n/2^2) + 3n$

$T(n/2^3) = 2T(n/2^4) + n/2^3$
$\Rightarrow T(n) = 2^3(2T(n/2^4) + n/2^3) + 3n = 2^4T(n/2^4) + 4n$

El patron seria: $T(n) = 2^kT(n/2^k) + kn$
La iteración continúa hasta que $T(n/2^k) = T(1)$, entonces:
$n/2^k = 1$
$n = 2^k \Rightarrow \lg n = \lg 2^k \Rightarrow \lg n = k \lg 2 \Rightarrow k = \lg n$

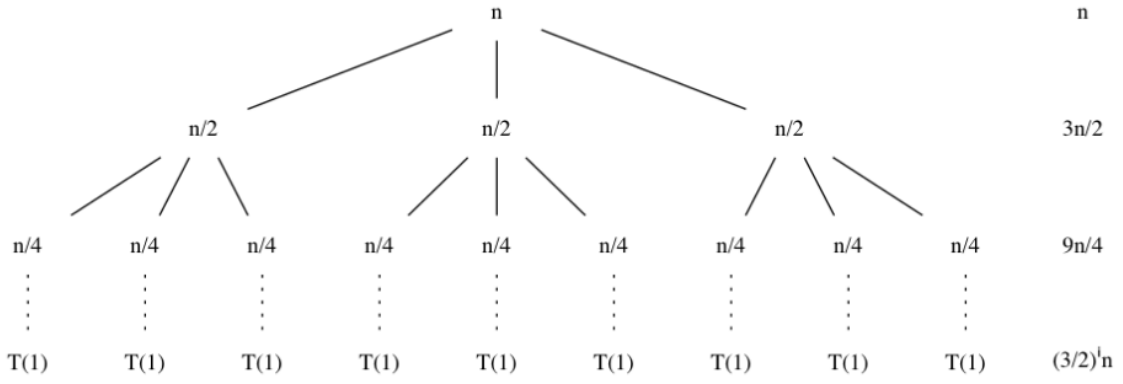
Reemplazando k en el patrón:

$T(n) = 2^kT(n/2^k) + kn$
$= 2^{lg(n)}T(1) + nlg(n)$
$= cn + nlg(n)$

Por lo tanto, T(n) = O(n lg n)

Recursion tree

4.4-1. Use a recursion tree to determine a good asymptotic upper bound on the recurrence T(n) = 3T(floor(n/2)) + n. Use the substitution method to verify your answer.



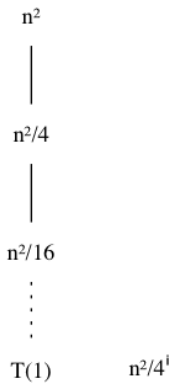
El ultimo nivel tiene $(3/2)^i n$ hojas, donde i es el último nivel (altura del árbol (lg(n))), entonces el número de hojas en el último nivel es:

$(3/2)^{lg(n)}n = n(n^{lg(3/2)}) = n^{lg(3/2)+1} = n^{lg3-lg2+1} = n^{lg3}$

$$
\begin{aligned}
T(n) &= n + 3n/2 + (3/2)^2 n + \ldots + n^{lg3} \\
&= n^{lg3} + n \sum_{k=0}^{lg(n)-1} (3/2)^k \\
&= n^{lg3} + n\left(\frac{(3/2)^{lg(n)}-1}{3/2-1} \right) \\
&= n^{lg3} + 2n(n^{lg3-1} - 1) \\
&= 3n^{lg3} - 2n
\end{aligned}
$$

Por lo tanto, $T(n) = O(n^{lg3})$

4.4-2. Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$. Use the substitution method to verify your answer.

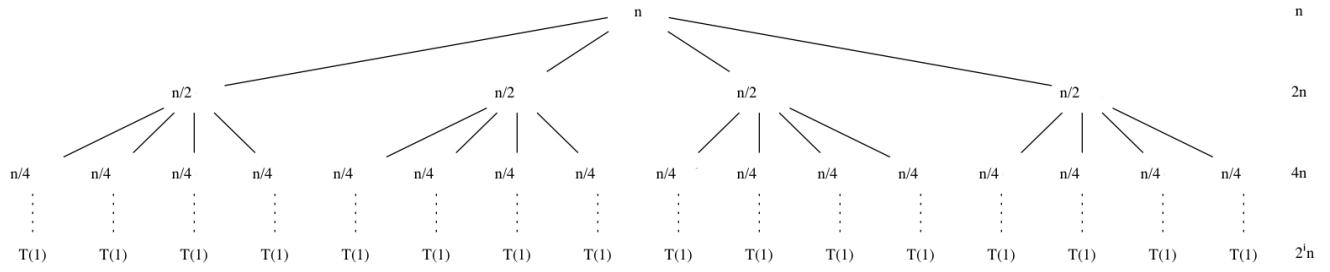


Altura del árbol = lg(n)
hojas en ultimo nivel = 1

$$
\begin{aligned}
T(n) &= n^2 + n^2/4 + n^2/4^2 + \ldots + c \\
&= n^2(1/4^0 + 1/4^1 + 1/4^2 + \ldots) + c \\
&= c + n^2 \sum_{i=0}^{lg(n)-1} (1/4)^i \\
&= c + n^2\left(\frac{(1/4)^{lg(n)}-1}{1/4-1} \right) \\
&= c + n^2\left(\frac{n^{-2}-1}{-3/4} \right) \\
&= 4/3n^2 - 4/3 + c
\end{aligned}
$$

Por lo tanto, $T(n) = O(n^2)$

4.4-3. Use a recursion tree to determine a good asymptotic upper bound on the recurrence
$T(n) = 4T(n/2 + 2) + n$. Use the substitution method to verify your answer.
La complejidad para $T(n/2+2) = T(n/2)$, porque cuando n es muy grande, el +2 no influye significativamente en su complejidad, entonces calculamos para $T(n) = 4T(n/2) + n$

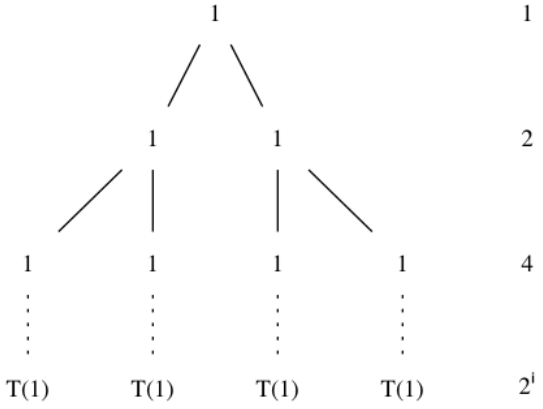


Altura del arbol (i) = $\log_2 n$
hojas ultimo nivel = $2^i n = n(2^{\lg n}) = n^2$

T(n) = $n + 2n + 4n + \ldots + n^2$
= $n^2 + n(1 + 2 + 4 + \ldots)$
= $n^2 + n \sum_{k=0}^{lg(n)-1} 2^k$
= $n^2 + n \left(\frac{2^{lgn} - 1}{2 - 1} \right)$
= $n^2 + n \left(\frac{n - 1}{1} \right) = 2n^2 - n$

Por lo tanto, $T(n) = O(n^2)$

4.4-4. Use a recursion tree to determine a good asymptotic upper bound on the recurrence T(n)= 2T(n-1) + 1. Use the substitution method to verify your answer.



Altura del arbol = n
hojas en el último nivel = $2^i = 2^n$

T(n) = $1 + 2 + 4 + \ldots + 2^n c$
= $2^n c + \sum_{k=0}^{n-1} 2^k$
= $2^n c + \frac{2^n - 1}{2 - 1}$
= $2^n(c + 1) - 1$

Por lo tanto, $T(n) = O(2^n)$

Master method

4.5-1. Use the master method to give tight asymptotic bounds for the following recurrences.

a. T(n) = 2T(n/4) + 1
a=2 b=4 f(n) = 1 = n^0 => $n^{\log_4 2} = n^{0.5}$

Caso 1:
f(n) = $O(n^{\log_b a - \varepsilon})$
$n^0 = O(n^{0.5 - \varepsilon})$, se cumple el caso 1 porque ε = 0.5 y ε > 0

=> $T(n) = \Theta(n^{0.5}) = \Theta(\sqrt{n})$

b. $T(n) = 2T(n/4) + \sqrt{n}$
a=2 b=4 f(n) = $\sqrt{n} = n^{0.5}$ => $n^{\log_4 2} = n^{0.5}$

Caso 2:

f(n) = Θ($n^{log_b a}$)

$n^{0.5}$ = Θ($n^{0.5}$)

=> T(n) = Θ($\sqrt{n}\ lg\ n$)

c. T(n) = 2T(n/4) + n

a=2 b=4 f(n) = n => $n^{log_4 2} = n^{0.5}$

Caso 3:

f(n) = Ω($n^{log_b a + \varepsilon}$)

n^1 = Ω($n^{0.5 + \varepsilon}$) , donde ε = 0.5 para que se cumpla la condición

Verificamos la condición: af(n/b) <= cf(n)

2f(n/4) <= cf(n)

2(n/4) <= cn

½(n) <= cn y se cumple la condición de c < 1

=> T(n) = Θ(f(n)) = Θ(n)

d. T(n) = 2T(n/4) + n^2

a=2 b=4 f(n) = n^2 => $n^{log_4 2} = n^{0.5}$

Caso 3:

f(n) = Ω($n^{log_b a + \varepsilon}$)

n^2 = Ω($n^{0.5 + \varepsilon}$) , donde ε = 1.5 para que se cumpla esta igualdad

Verificamos la condición: af(n/b) <= cf(n)

2f(n/4) <= cf(n)

$2(n/4)^2$ <= cn^2

⅛(n^2) <= cn^2 y se cumple la condición de c < 1

=> T(n) = Θ(f(n)) = Θ(n^2)

4.5-2. Professor Caesar wishes to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into pieces of size n/4 x n/4, and the divide and combine steps together will take Θ(n^2) time. He needs to determine how many subproblems his algorithm has to create in order to beat Strassen's algorithm. If his algorithm creates a subproblems, then the recurrence for the running time T(n) becomes T(n) = aT(n/4) + Θ(n^2). What is the largest integer value of a for which Professor Caesar's algorithm would be asymptotically faster than Strassen's algorithm?

a = ? b = 4 f(n) = Θ(n^2)

Strassen: T2(n) = Θ(n^{lg7}) = Θ($n^{2.81}$)

Caso 1) Cuando T(n) = Θ($n^{log_b a}$) = Θ($n^{log_4 a}$)

Comparamos con T2(n) para obtener una complejidad menor:

T(n)	< T2(n)
Θ($n^{log_4 a}$)	< Θ(n^{lg7})
$n^{log_4 a}$	< n^{lg7}
$\log_4 a$	< lg7
lg(a) / lg(4)	< lg7
lg(a)	< 2lg(7)
lg(a)	< $lg(7^2)$
$2^{lg(a)}$	< $2^{lg(49)}$
a	< 49

Caso 2) Cuando T(n) = Θ($n^{log_b a} lg(n)$) = Θ($n^{log_4 a} lg(n)$)

Θ($n^{log_4 a} lg(n)$)	< Θ(n^{lg7})
Θ($n^{log_4 a} lg(n)$)	< Θ($n^{2.81}$)

Si evaluamos la desigualdad con a = 50 (mayor valor entero en comparación al caso anterior):

Θ($n^{log_4 50} lg(n)$)	< Θ($n^{2.81}$)
Θ($n^{2.82} lg(n)$)	< Θ($n^{2.81}$)

Como se puede observar no se cumple la condicion de ser menor a la complejidad de Strassen.

Caso 3) Cuando T(n) = Θ(f(n)) = Θ(n^2)

Θ(n^2) < Θ(n^{lg7}) Se cumple la condicion, pero debemos evaluar la condición del C3

af(n/b) <= cf(n)

$ad(n/4)^2$ <= $cd(n^2)$

a/16 <= c , como c debe ser menor a 1, entonces:

a/16 < 1 => a < 16

Por lo tanto, el máximo valor entero de “a” para que la complejidad de este algoritmo sea menor a Strassen seria 48.

4.5-3. Use the master method to show that the solution to the binary-search recurrence T(n) = T(n/2) + Θ(1) es T(n) = Θ(lg n). (See Exercise 2.3-5 for a description of binary search)

a = 1 b = 2 f(n) = Θ(1)

$n^{log_b a} = n^{log_2 1} = n^0 = 1$

Caso 2)

f(n) = Θ($n^{log_b a}$) = Θ(1)

Θ(1) = Θ(1)

=> T(n) = Θ(lg n)

4.5-4. Can the master method be applied to the recurrence T(n) = 4T(n/2) + n^2 lg n? Why or why not? Give an asymptotic upper bound for this recurrence.

a = 4 b = 2 f(n) = $n^2 \lg n$ => $n^{\log_b a} = n^{\log_2 4} = n^2$

El master method clásico no puede aplicarse porque no se cumple ninguno de sus casos:

Caso 1) $f(n) \neq O(n^{2-\varepsilon})$

Caso 2) $f(n) \neq \Theta(n^2)$

Caso 3) $f(n) \neq \Omega(n^{2+\varepsilon})$

En este caso se puede aplicar el master method modificado:

a = 4 b = 2 k = 1

Caso 2)

$f(n) = \Theta(n^2 \log^k n)$

$n^2 \log n = \Theta(n^2 \log^k n)$, k = 1 para que se cumpla la igualdad

=> T(n) = $\Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^2 \log^2 n)$

4.5-5. Consider the regularity condition af af(n/b) <= cf(n) for some constant c<1, which is part of case 3 of the master theorem. Give an example of constants a >= 1 and b > 1 and a function f(n) that satisfies all the conditions in case 3 of the master theorem except the regularity condition.

a = 1 b = 2 f(n) = n(2 - cos n)

Caso 3)

$f(n) = \Omega(n^{\log_b a + \varepsilon})$

$n(2 - cos(n)) = \Omega(n^{\log_2 1 + \varepsilon})$

$n(2 - cos(n)) = \Omega(n^{0+\varepsilon})$, existe un ε > 0 para el cual se cumple esta condición.

Verificamos la condición de regularidad:

af(n/b) <= cf(n)

(n/2) (2 - cos(n/2)) <= cn(2 - cos(n))

½ (2 - cos(n/2)) <= c (2 - cos(n))

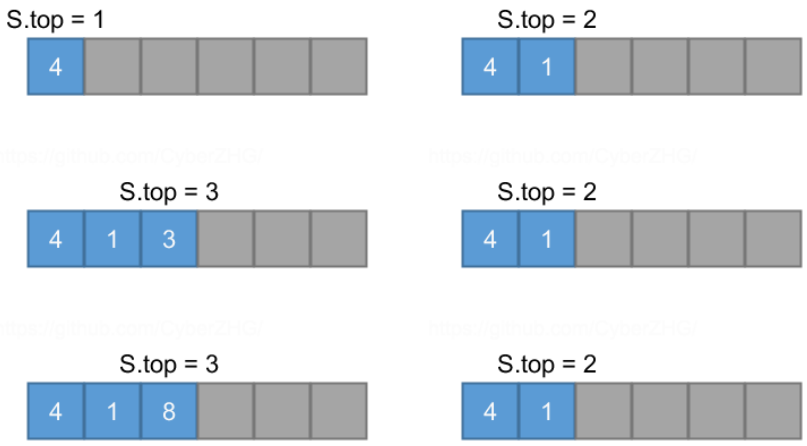
½ (2 - cos(n/2)) <= c (2 + 1 - $2\cos^2(n/2)$)

$\frac{2 - cos(n/2)}{2(3 - 2cos^2(n/2))}$ <= c

Analizando la desigualdad, el min(cos(n/2)) = -1, esto implica que c >= 3/2, pero se debe cumplir que c < 1. Por lo tanto para este f(n) no se cumple la función de regularidad del caso 3.

Pilas y colas

10.1-1. Using Figure 10.1 as a model, illustrate the result of each operation in the sequence PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S,8), and POP(S) on an initially empty stack S stored in array S[1..6]



10.1-2. Explain how to implement two stacks in one array A[1..n] in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in O(1) time.

En este caso, Stack1 podría iniciar desde la posición 1 de A e insertar sus elementos incrementando hacia n, mientras que Stack2 iniciaría desde la posición n hacia la posición 1. El overflow ocurre cuando S1.top y S2.top son adyacentes.

```python
class Stack1:
    def __init__(self):
        self.top = -1

    def is_empty(self):
        return self.top == -1

    def push(self, x):
        global a
        self.top += 1
        a[self.top] = x

    def pop(self):
        global a
        self.top -= 1
        return a[self.top + 1]
```

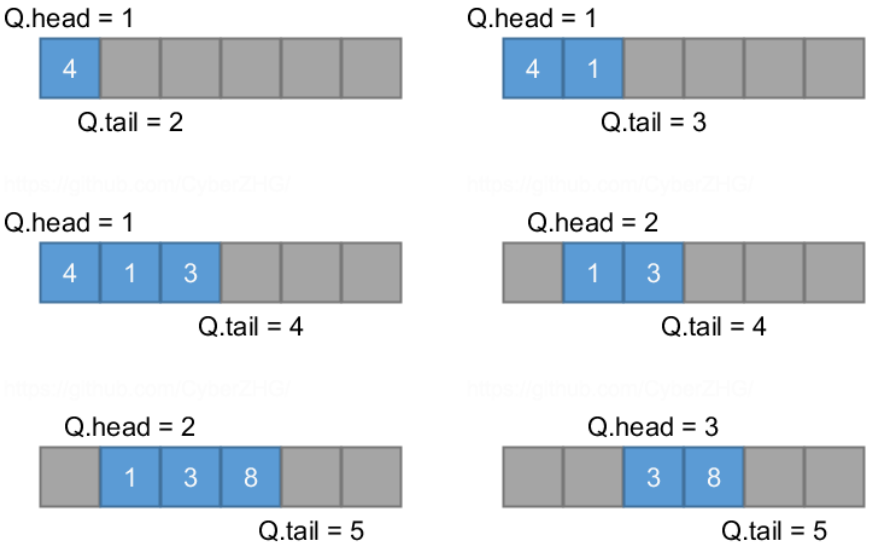
```python
class Stack2:
    def __init__(self):
        self.top = n

    def is_empty(self):
        return self.top == n

    def push(self, x):
        global a
        self.top -= 1
        a[self.top] = x

    def pop(self):
        global a
        self.top += 1
        return a[self.top - 1]
```

10.1-3. Using Figure 10.2 as a model, illustrate the result of each operation in the sequence ENQUEUE(Q,4), ENQUEUE(Q,1), ENQUEUE(Q,3), DEQUEUE(Q), ENQUEUE(Q,8), and DEQUEUE(Q) on an initially empty queue Q stored in array Q[1..6]



10.1-4. Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

```
ENQUEUE(Q, x)
  if Q.head == Q.tail
      error "Queue overflow"

  Q[Q.tail] = x

  if Q.head == NIL
      Q.head = Q.tail

  if Q.tail == Q.length
      Q.tail = 1
  else
      Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
  if Q.head == NIL
      error "Queue underflow"

  x = Q[Q.head]

  if Q.head == Q.length
      Q.head = 1
  else
      Q.head = Q.head + 1

  if Q.head == Q.tail
      Q.head = NIL
  return x
```

Listas enlazadas

10.2-1. Can you implement the dynamic-set operation INSERT on a singly linked list in O(1) time? How about DELETE?

Si se puede implementar el INSERT en O(1), siempre y cuando el nuevo elemento sea insertado al inicio de la lista enlazada. Caso contrario del DELETE porque previamente se debe buscar el elemento dentro de la lista, en el peor de los casos sería O(n).

```
def insert(head, x):
    new_node = LinkListNode(x)
    new_node.next = head.next
    head.next = new_node


def delete(head, x):
    while head is not None:
        if head.next is not None and head.next.value == x:
            head.next = head.next.next
        else:
            head = head.next
```

10.2-2. Implement a stack using a singly linked list L. The operations PUSH and POP should still take O(1) time.

```
class LinkListNode:
    def __init__(self, value):
        self.value = value
        self.next = None


def push(head, x):
    new_node = LinkListNode(x)
    new_node.next = head.next
    head.next = new_node


def pop(head):
    if head.next is None:
        return None
    x = head.next.value
    head.next = head.next.next
    return x
```

10.2-3. Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take O(1) time.

```
class Queue:
    def __init__(self):
        self.head = None
        self.tail = LinkListNode(None)

    def enqueue(self, x):
        new_node = LinkListNode(x)
        if self.tail.next is None:
            self.head = new_node
            self.tail.next = self.head
        else:
            self.head.next = new_node
            self.head = new_node

    def dequeue(self):
        if self.tail.next is None:
            return None
        x = self.tail.next.value
        self.tail = self.tail.next
        return x
```

10.2-4. As written, each loop iteration in the LIST-SEARCH' procedure requires two tests: one for $x \neq L.nil$ and one for $x.key \neq k$. Show how to eliminate the test for $x \neq L.nil$ in each iteration.

```
LIST-SEARCH'(L, k):
  x = L.nil.next
  L.nil.key = k
  while x.key ≠ k
      x = x.net
  return x
```

10.2-5. Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

```python
class LinkListNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
        self.prev = None


class Dict:
    def __init__(self):
        self.nil = LinkListNode(None, None)
        self.nil.next = self.nil
        self.nil.prev = self.nil

    def insert(self, key, value):
        x = self.search_node(key)
        if x is None:
            x = LinkListNode(key, value)
            x.next = self.nil.next
            x.prev = self.nil
            x.next.prev = x
            x.prev.next = x
        else:
            x.value = value

    def delete(self, key):
        x = self.search_node(key)
        if x is not None:
            x.next.prev = x.prev
            x.prev.next = x.next

    def search_node(self, key):
        self.nil.key = key
        x = self.nil.next
        while x.key != key:
            x = x.next
        if x == self.nil:
            return None
        return x

    def search(self, key):
        x = self.search_node(key)
        if x is None:
            return None
        return x.value
```

- La función **search_node** en el peor de los casos debe recorrer todo el array para encontrar el elemento que se busca, por eso su tiempo sería O(n).
- INSERT: previamente se debe buscar si el nuevo key ya se encuentra en el diccionario, para eso se usa la función search_node, por lo tanto también tendría O(n)
- DELETE: se debe buscar el key que se va a eliminar, se usa la función search_node, por lo tanto sería O(n).
- SEARCH: En el peor de los casos se tendrá que recorrer todo el array, seria O(n)