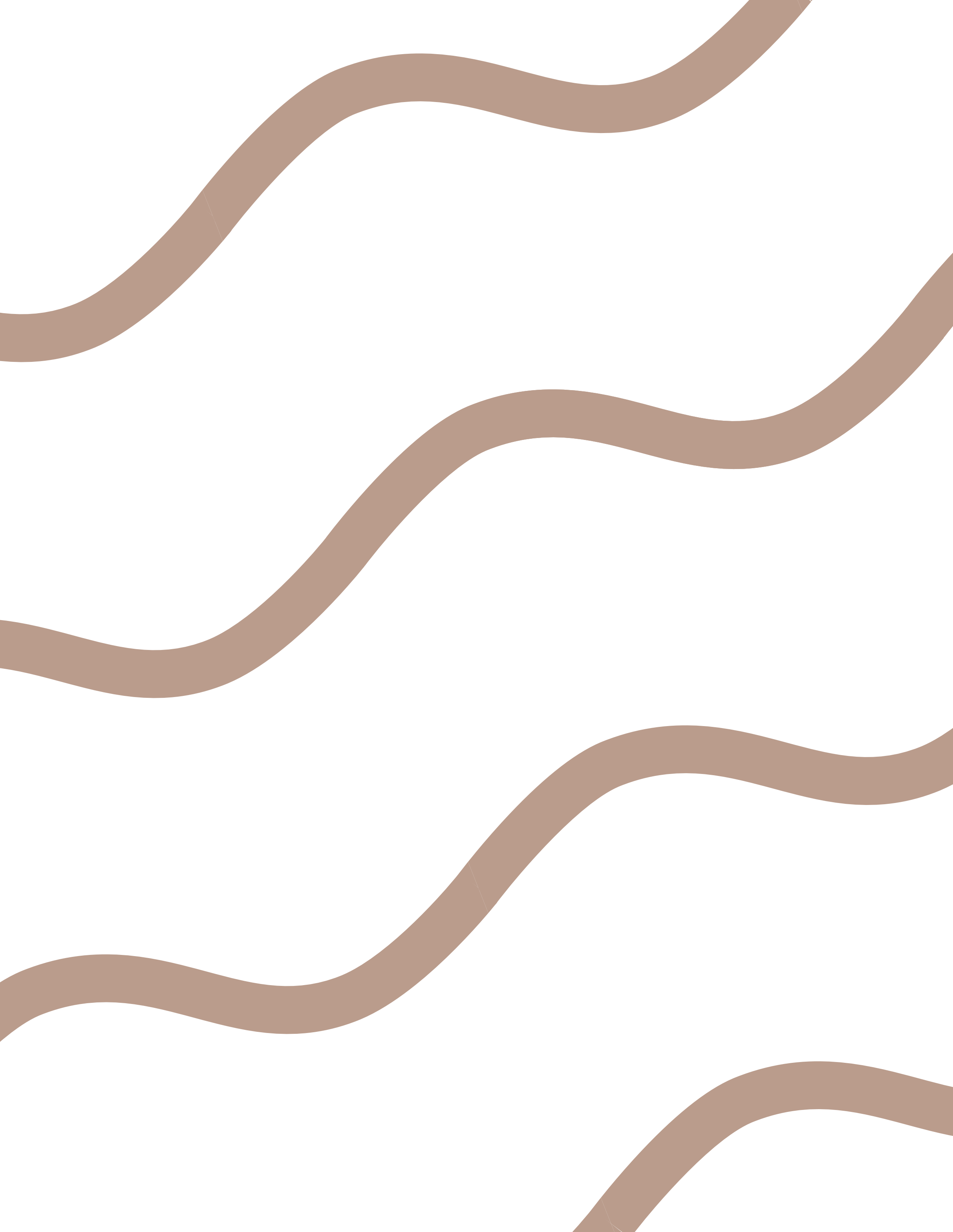
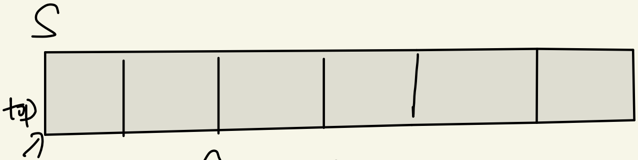
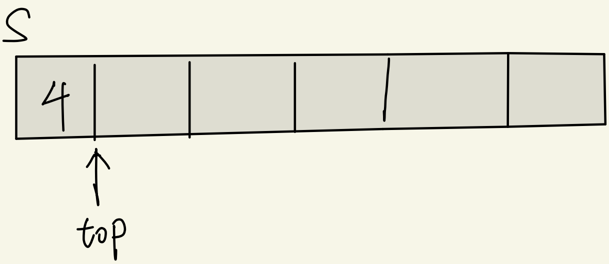
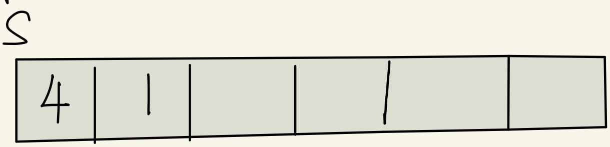




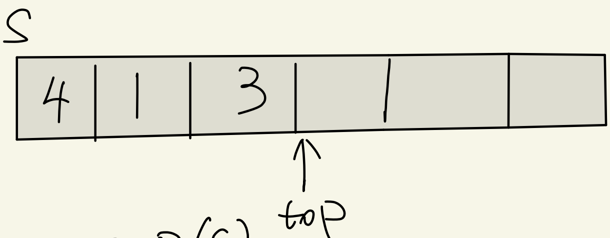
PUSH(S,4)



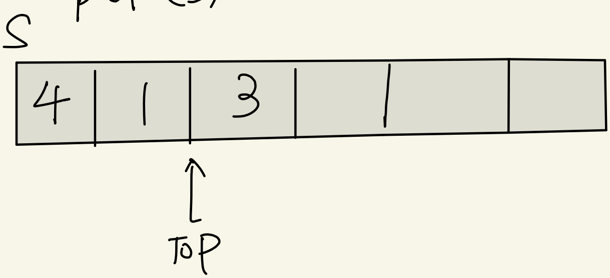
PUSH(S,1)



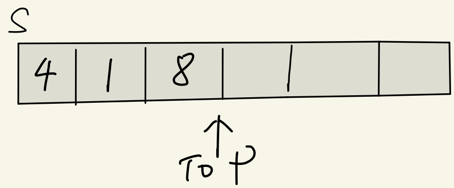
PUSH(S,3)



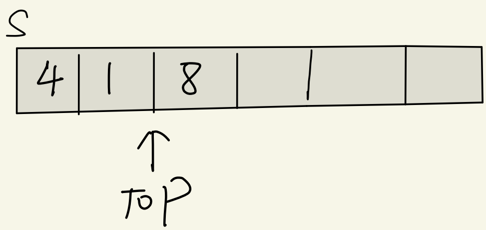
pop(S)



PUSH(S,8)

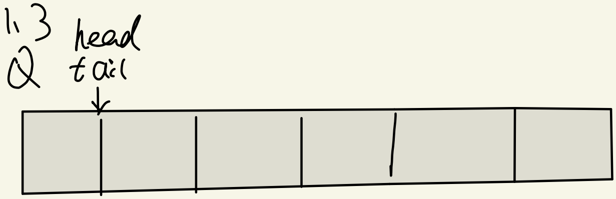


pop(S)

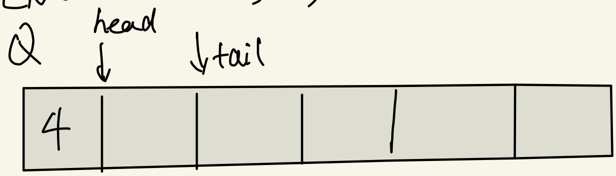


10.1.2 the Tops pointer of the two stacks starts at two ends of the array.

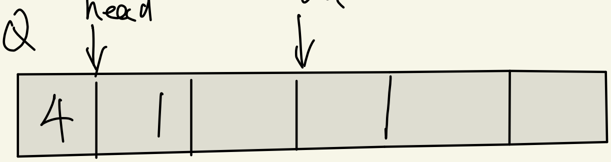
10.1.3



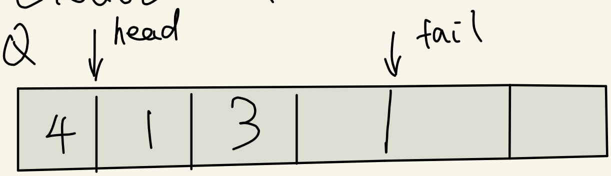
ENQUEUE(Q,4)



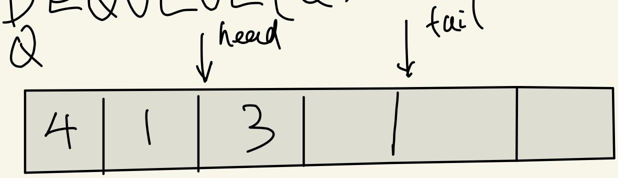
ENQUEQUE(Q,1)



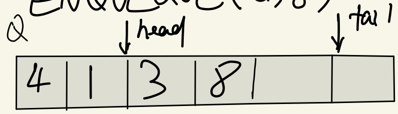
ENQUEQUE(Q,3)



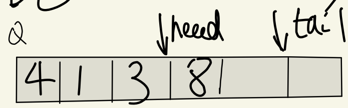
DEQUEUE(Q)



ENQUEQUE(Q,8)



DEQUEQUE(Q)



10.1-4

```
QUEUE-FULL(head, tail)
 if tail == Q.Length,
     head = 1
     return true
 if Q.head = Q.tail + 1
     return true
 else
     reture false


QUEUE-EMPTY(head, tail)
 if tail == head
     return true
 else
     return false

ENQUEUE-OVERFLOW(Q, x)
 if QUEUE-FULL(Q.head, Q.tail)
     error "overflow!"
 else
     ENQUEUE(Q, x)

DEQUEQUE-UNDERFLOW(Q, x)
 if QUEUE-EMPTY(Q.head, Q.tail)
     error "underflow"
 else
     DEQUEUE(Q, x)
```

10.1-5 x

10.1-6

```
ENQUEOUE(A, B, x)
 while not STACK-EMPTY(A)                O(n)
      y = POP[A]
      B.PUSH(y)

 A.PUSH(x)
 while not STACK-EMPTY(B)
      y = POP[B]
      A.PUSH(y)

DEQUEUE(A)
  return A.POP( )          O(1)
```

10.1-7

```
PUSH(A, B, x)
while not A.QUEUE-EMPTY()
    y = A.DEQUEQUE()                O(n)
    B.ENQUEQUE(y)

A.ENQUEQUE(x)
while not B.QUEUE-EMPTY()
    y = B.DEQUEQUE()
    A.ENQUEUE(y)

POP(A)
    return A.DEQUEQUE()
```

10.2 -1

```
INSERT(L, x)
    x.next = L.head
    L.head = x        O(1)

DELET(L, x)
    y = L.head
    while y != nil
        if y.next == x
            y.next = x.next
            return
        y = y.next
```

10.2.2

PUSH(SL, x)

INSERT(SL, X) O(1)

POP(SL)

result = SL.head

SL.head = SL.head.next

return result. O(1)

10.2-3

ENQUEQUE(SL, X)

SL.tail.next = X O(1)

x.next = nil

S.tail = X

DEQUEQUE(SL)

result = SL.head

SL.head = SL.head.next O(1)

return result.

10.2-4

L.nil.key = x.key $\boxed{?}$ check k cannot be nil $\boxed{?}$

10.2-5

SL:

INSERT(SL, x) O(1)

DELETE(SL, x) O(n)

SEARCH(SL, x) O(n)

CL:

INSERT(SL, x) O(1)

DELETE(SL, x) O(1)

SEARCH(SL, x) O(n)

10.2-6

circular array with senitile

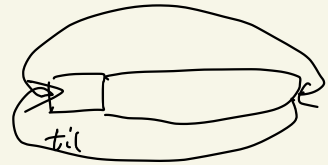
UNION(S_1, S_2)

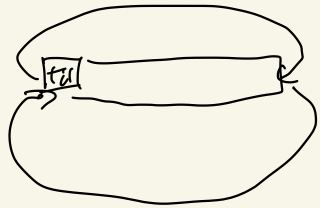
$S_2.nil.prev.next = S_1.nil$

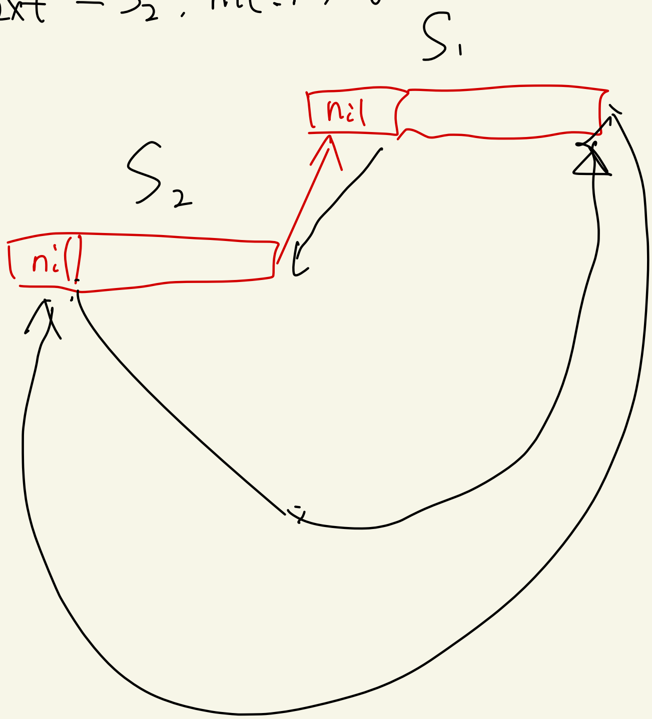
$S_2.nil.next.prev = S_1.nil.prev$

$S_1.nil.prev = S_2.nil.prev$

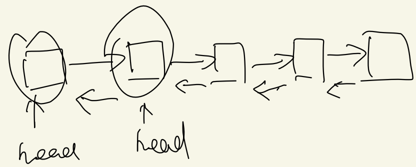
$S_1.nil.prev.next = S_2.nil.next$







10.2-7




```
REVERSE(SL)
  if SL.head == nil
      return
  x = SL.head
  next = x.next
  x.next = nil

  while next != nil
    nextNext = next.next
    next.next = x

    x = next
    next = nextNext

SL.head = x
```

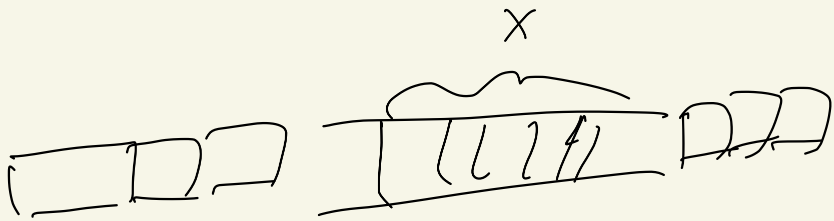
$O(n)$

10.3-1

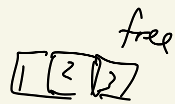
prev	7	1	2	3	4	5	6
	1	2	3	4	5	6	7
key	nil	13	4	8	19	5	11
next	2	3	4	5	6	7	1

nil 22 4 13 1 7 4 4 10 8 7 13 19 10 16 5 13 19 6 16 22

10.3-2



FREE-OBJECT(A, X, offset)
  for i = 0 to offset
    A[i+X] = free
    free = i+X



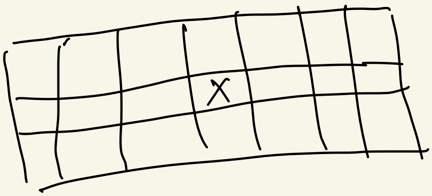
ALLOCATE-OBJECT(A, offset)
  if free == nil
    error "out of space"
  else
    x = free
    free = free - offset
    return X

10.3-3

we only need the next for the singly-linked list

10.3-4 FIFO

next
key
prev



ALLOCATE()

if next[0] == next.length

error "out of space" O(1)

return next[0] + 1

FREE-OBJECT(X) O(n)

for i = x to next[0]

key[x] = [x+1]

next[x] = next[x+1]

prev[x] = prev[x+1]

next[0] = next[0] - 1

prev[next[0]] = 1

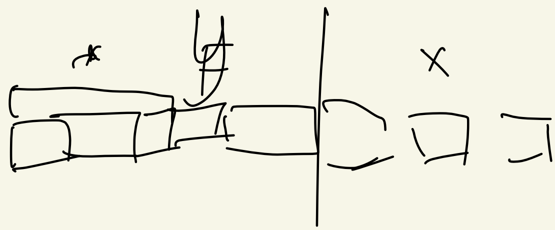
10.3.5

COMPACTIFY-LIST(L, F)

```
i = j = 1
x = L.head
y = F.head

while i < n and j < m-n
    while x ≤ n
        x = x.next
        i = i+1
    while y ≥ n+1
        y = y.next
        j = j+1
    y.prev.next = x
    x.prev.next = y
    temp = x.key
    x.key = y.key
    y.key = temp

    temp = x.next
    x.next = y.next
    y.next = temp
    temp = x.prev
    x.prev = y.prev
    y.prev = x.prev
```

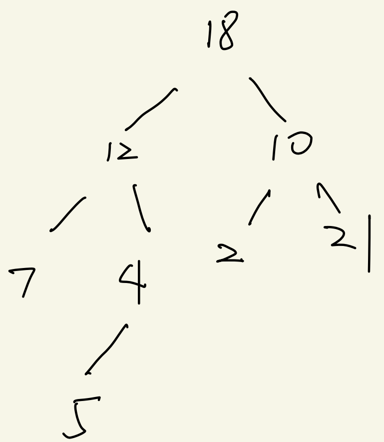


loop-invariant:

1, ..., i are located ≤ n,

1 ... j are located ≥ n+1

10.4-1



10.4-2 $O(n)$

```
PRINT-BINARY-TREE(
if root != nil
    print( root. key)
    PRINT-TREE-RECURSIVE( root.left)
    PRINT-TREE-RECURSIVE( root. right)
```

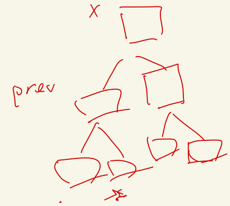
10.4-3 Stack : Depth-first   Queue : Breadth-First.

```
PRINT-BINARY-TREE-NONE-RECURSIVE(root)
    S = new stack
    S.push(root)
    while !STACK-EMPTY(S)
        node = S.pop
        print(node.key)
        if node.left != nil
            S.push(node.left)
        if node.right != nil
            S.push(node.right)
```

10.4-4 O(n)

```
PRINT-TREE-RECURSIVE(root)
    PRINT(root.key)
    if root.left-child != nil
        PRINT-TREE-RECURSIVE(root.left-child)
    if root.right-sibling != nil
        PRINT-TREE-RECURSIVE(root.right-sibling)
```

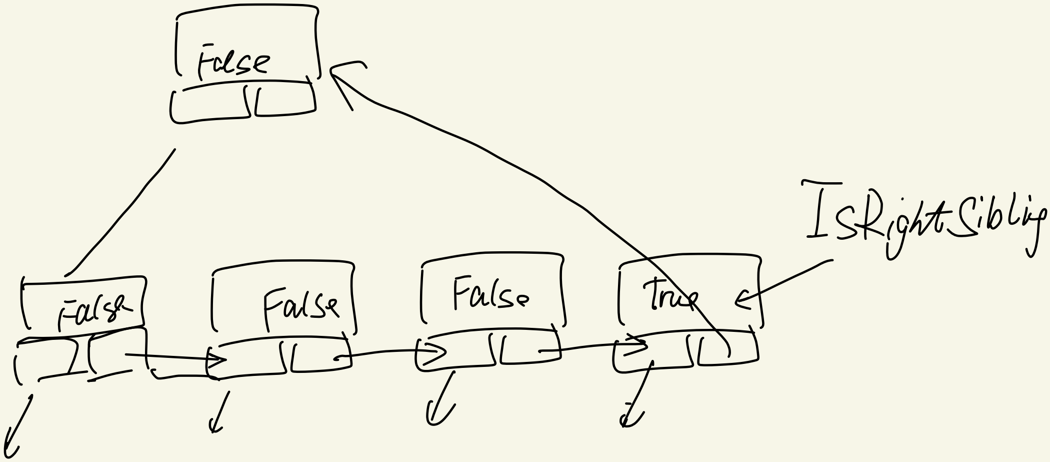
10.4-5




```
PRINT-BINARY-TREE(T)
 x = T.root
 prev = x.parent      // nil
 while x != nil
    if prev == x.parent
        PRINT(x.key)
        prev = x
        if x.left != nil
            x = x.left
        else if x.right != nil
            x = x.right


        else
            x = x.parent
    if prev == x.left   and   x.right != nil
        prev = x
        x = x.right

    else
        prev = x
        x = x.parent
```



PARENT(node)

```
while !node.IsRightSibling
  node = node.next

return node next
```

10-1

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
INSERT(L, x)	$O(1)$	$O(n)$	$O(1)$	$O(n)$
DELETE(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
SUCCESSOR(L, x)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
PREDECESSOR(L, x)	$O(n)$	$O(n)$	$O(n)$	$O(1)$
MINIMUM(L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
MAXIMUM(L)	$O(n)$	$O(n)$	$O(n)$	$O(1)$

10-2.

a.

sorted Double Linked list

```
MAKE-HEAP ( )
  L = new Double Linked List    O(1)
  return L

INSERT(L, x)
  node = L.head
  next = node.next
  while next != nil and next.key ≤ x.key
      node = next
      next = node.next
                                  O(n)
  x.prev = node
  x.next = node.next
  node.next.prev = x
  node.next = x

MINIMUM(L) O(1)
  return L.head
```

```
RIGHT(i)
  return 2i+1

LEFT(i)
  return 2i

PARENT(i)
  return ⌊i/2⌋

EXTRACT-MIN(L)   O(1)
  L.nil.next = L.head.next
  L.head.prev = L.nil
  temp = L.head
  L.head = L.head.next
  return temp.
```

union (L_1 , L_2)

L = new double linked list   O(n)

x = L_1 . head

y = L_2 . head

while x != nil or y != nil

    if y == nil or y.key ≥ x.key

        L.insert(x)

        x = x.next

    if x == nil or x.key > y.key

        L.insert(y)

        y = y.next

return L

b. Unsorted lists

$O(1)$

MAKEHEAP( )

return new Double-linked list L

Aussumes that the two lists are already a Min-Heap.

UNION(L_1, L_2)

10.3 a.

Since both algorithms returns correctly. Thus, they return the same anwser,

COMPACT-LIST-SEARCH'(L, n, k, t) has t iteration for the for loop. Thus the number of iterations are at least n.

b. $O(t + E(X_t))$

$\swarrow$ for loop $\searrow$ while loop

c. $E(X_t) \le \sum_{r=1}^{n} (1 - r/n)^t$

$\boxed{C.25}$ $E(X) = \sum_{i=0}^{\infty} i \cdot \Pr\{X = i\}$

$$= \sum_{i=0}^{\infty} i(\Pr\{X \ge i\} - \Pr\{X \ge i+1\})$$

$$= \sum (i-1)(\Pr\{X \ge i-1\} - \Pr\{X \ge i\})$$

$$+ i(\Pr\{X \ge i\} - \Pr\{X \ge i+1\})$$

$$= \sum_{i=1}^{\infty} \Pr\{X \ge i\}$$

Assume k 's order m

$m \geq n$ or $m < n$

$x_t \in [0, n-1]$

$$E[x_t] = \sum_{d=0}^{n-1} d \Pr\{x_t = d\}$$

$$= \sum_{d=1}^{n-1} \Pr\{x_t \geq d\}$$

$$\Pr\{x_t \geq d\} = (1 - d/n)^t$$

$$E[x_t] = \sum_{d=1}^{n-1} (1 - d/n)^t$$

$$\leq \sum_{r=1}^{n} (1 - r/n)^t$$

$n=1$ equal

$$f(x) = \frac{x^{t+1}}{t+1}$$

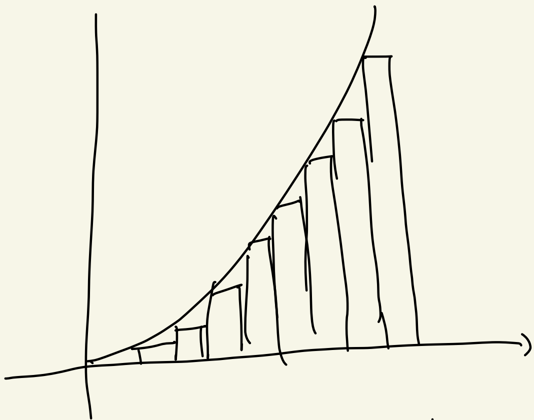
$$f'(x) = x^t$$

$$d . \sum_{r=0}^{n-1} r^t \leq n^{t+1}/t+1$$

$$\frac{1}{\cancel{t+1}} . \cancel{(t+1)} \quad n^t$$

d. $\sum_{r=0}^{n-1} r^t = \int_0^{n-1} \lfloor r^t \rfloor \, dr$

since the $y = r^t$ is convex function.



$$\sum_{r=0}^{n-1} r^t = \int_0^n \lfloor r^t \rfloor \, dr \le \int_0^n r^t \, dr$$

$$= \left. \frac{r^{t+1}}{t+1} \right|_0^n \le \frac{n^{t+1}}{t+1}$$

e. $E[x^t] \le \sum_{r=1}^{n} \left(1 - \frac{r}{n}\right)^t$

$$= \frac{1}{n^t} \sum_{r=1}^{n} (n-r)^t$$

$$= \frac{1}{n^t} \sum_{x=0}^{n-1} x^t$$

$$\le \frac{1}{n^t} \cdot \frac{n^{t+1}}{t+1} = \frac{n}{t+1}$$

f. $O(t + n/t+1) = O(t + n/t)$

g. $O(t) = O(t + n/t)$

runtime of COMPACT-LIST-SEARCH'(

runtime of COMPACT-LIST-SEARCH(L, n, k)

$\Rightarrow O(t) = O(\sqrt{n})$

h. when there are repeated key values in the list, they decrease the the probability that the index will skip. In a extreme situation when all keys are repeated, line 2 to line 7 will never be execut