From Discrete Structures to Elementary Data Structures

Sets are fundamental to both Mathematics and Computer Science.

In algorithmics, sets can grow, shrink, or otherwise change over time. We shall call such sets dynamic.

Algorithms may require different types of operations to be performed on sets

- Insert elements into a set
- delete elements from a set
- test membership in a set

or more complex operation

- insert an element into and extract the smallest element from a set.

⋮

The best way to implement a dynamic set therefore, depends on the operations to be supported.

Representing the element of a dynamic set

- Each element of the dynamic set is represented by an object which may have multiple fields
- Typically one of the object's fields is an identifying Key field.
- The object may also have
  (a) fields carrying satellite data (information which is unused by the satellite implementation
  (b) fields that are manipulated by set operations

Operations on dynamic sets

Operations

- Queries: Return information about the set
- modifying operations: change the set

Example operations

Search (S, K): Given a set S and a key value K return a pointer x to an element of S:

Key[x] = K or NIL if $\nexists$ any such element $e \in S$

INSERT (S, x) : Augment S with the element pointed to by x

Delete (S, x): Given a pointer x to an element in S, remove x from S.
(Note: this operation uses a pointer to an element in S, not a key value)

MINIMUM (S): A query on the totally ordered set S, that returns an element of S having the smallest key

MAXIMUM (S): returns an element of S having the largest key

Successor (S, x): Given an element x, whose key is from a totally ordered set S, return the next larger element in S or NIL if x is the element with the largest key

Predecessor (S, x): Given an element x, whose key is from a totally ordered set S, return the next smaller element in S or NIL if x is the element with the smallest key

STACKS

Stacks are dynamic sets in which the element removed from the set by the DELETE operation is pre-specified

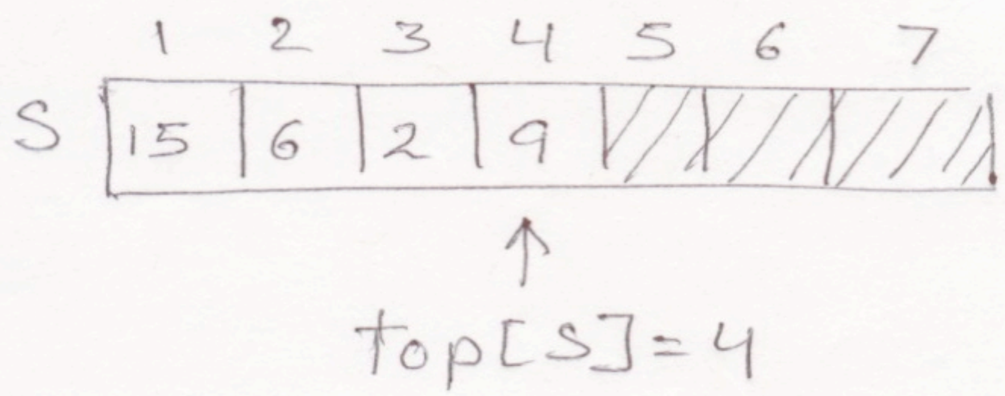
- In a stack the element deleted from the set is the one most recently inserted.

  So the stack implements a LAST-IN, FIRST-OUT (LIFO) policy

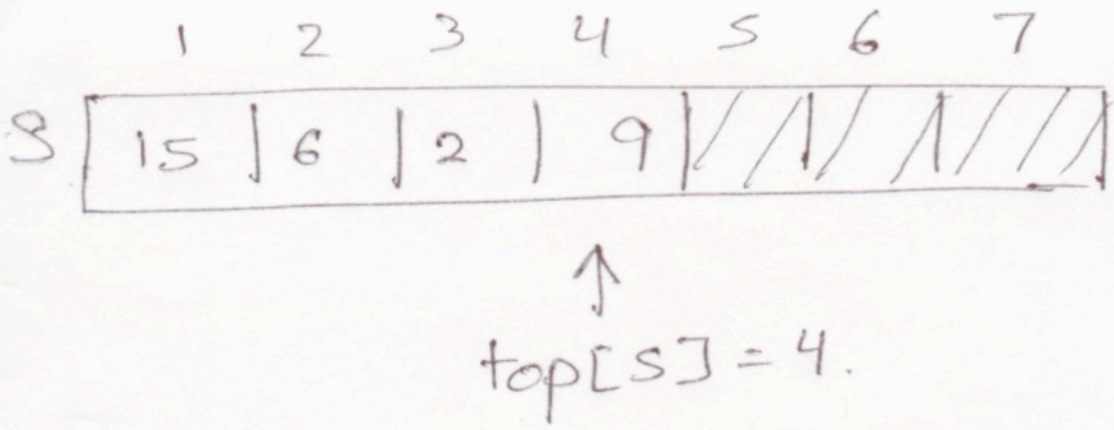
Implementing a stack

A stack of at most n elements can be implemented with an array $S[1, \ldots n]$. the array has an attribute top[S] that indexes the most recent element

The stack consists of elements $S[1, \ldots top[S]]$, where $S[1]$ is the element at the bottom of the stack and $S[top[S]]$ is the element at the top

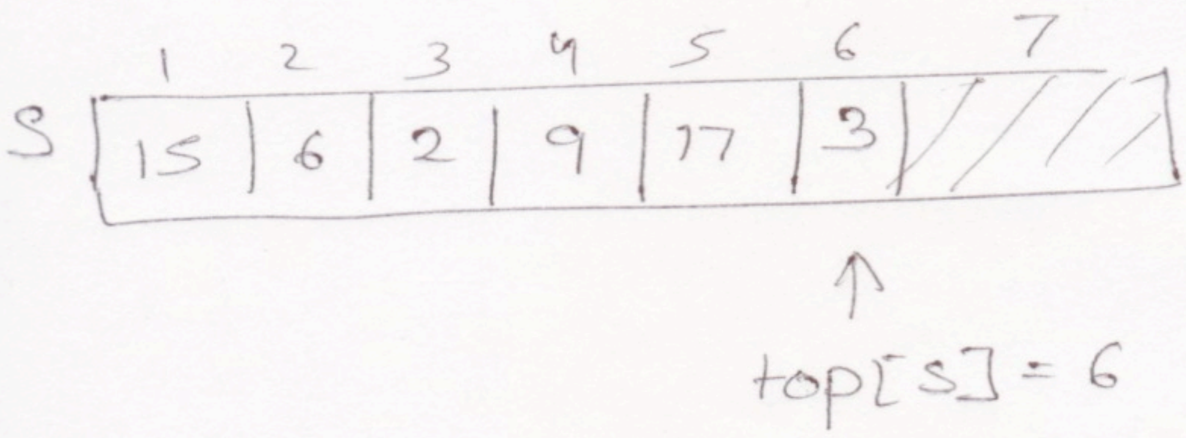


- The insert operation in stacks is called PUSH
- The delete operation is called POP

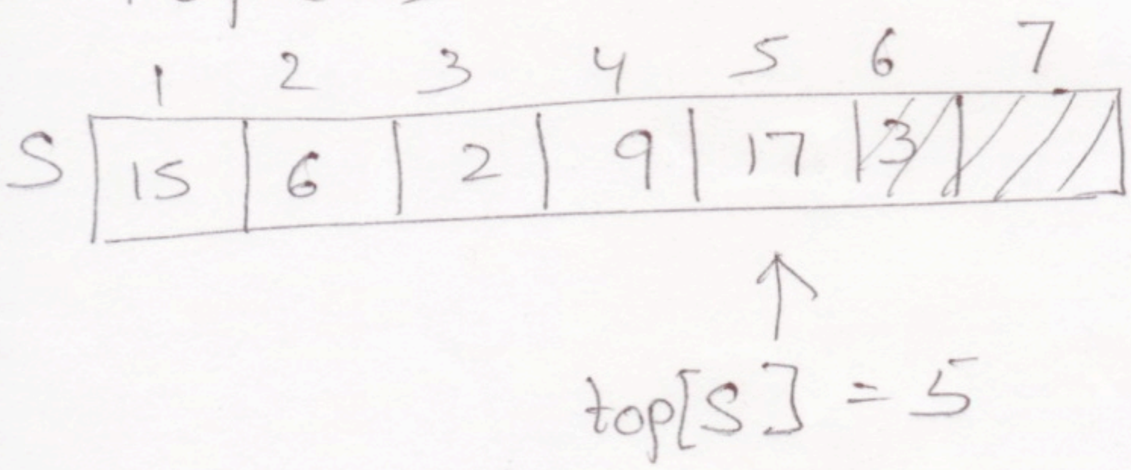


Push(S, 17)

Push (S, 3)



Pop(S)



When top[S] = 0 ⇒ stack is empty

When top[S] = n ⇒ stack is full

Popping an empty stack (stack underflow) is an error

Pushing onto a full stack (stack overflow) is an error

We can devise simple algorithms to check for these conditions as well as the Push and PoP operations

```
STACK-EMPTY(S)
    if top[S] = 0
        then return TRUE
        else return FALSE
```

} if condition true then action1 else (condition false) action 2

```
Push(S, x)
    top[S] ← top[S] + 1        // increment
    S[top[S]] ← x              // assignment
```

```
Pop(S)
    if STACK-EMPTY(S)
        then error underflow
        else top[S] ← top[S] - 1
             return S[top[S] + 1]
```

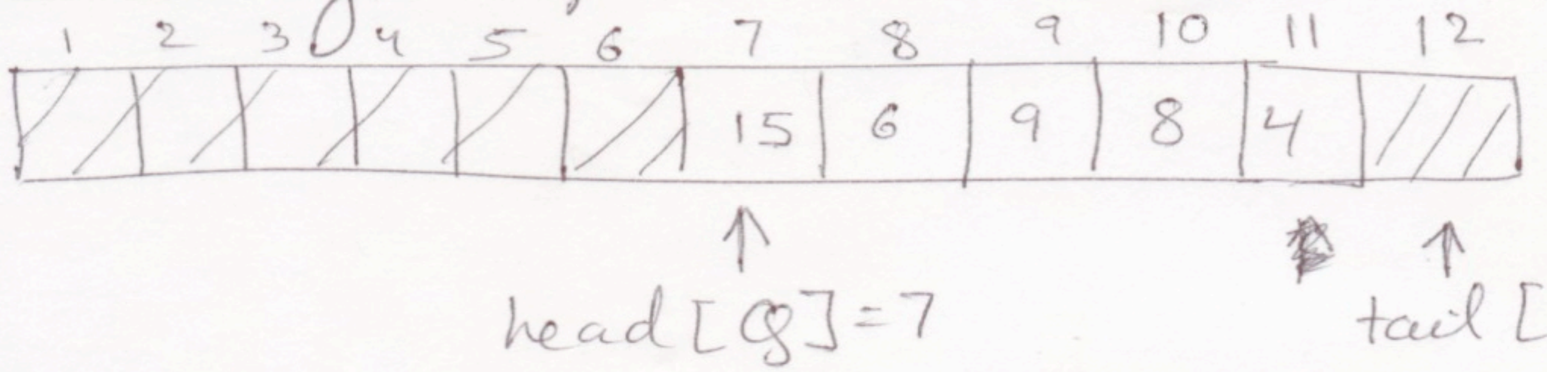
Queues

Dynamic set in which the deleted element is always the one which has been in the set for the longest time. (First-in, First-out FIFO)

- The insert operation is called ENQUEUE
- The delete operation is called DEQUEUE

The queue behaves like a real-world queue. Elements get enqueued at its end and dequeued at its head.

Implementing a Queue

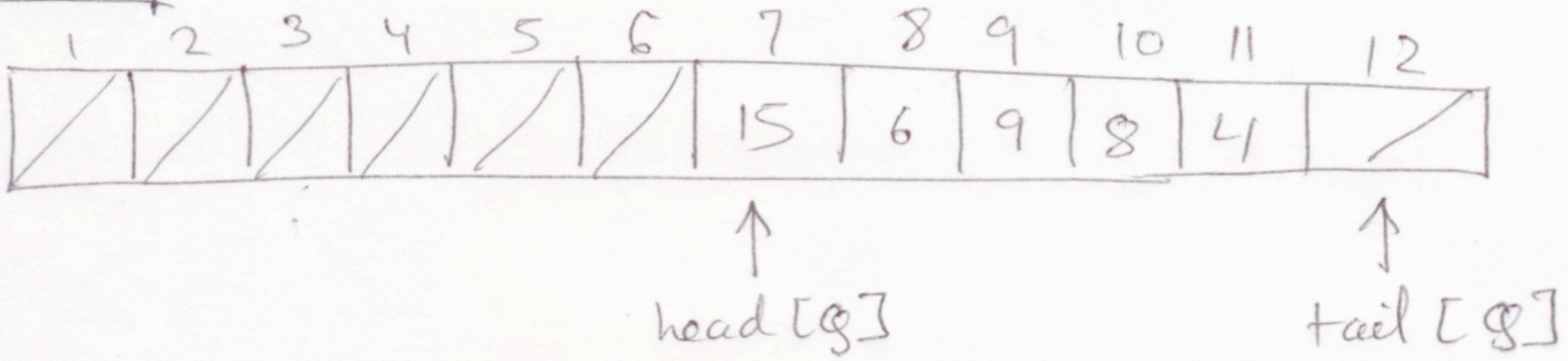


- head[Q]: points to the head
- tail[Q]: indexes the location where the next element will be added

- Elements in the queue are located at head[Q], head[Q]+1, ... tail[Q]-1

- In this implementation, we "wrap-around": Location follows location n in a circular order

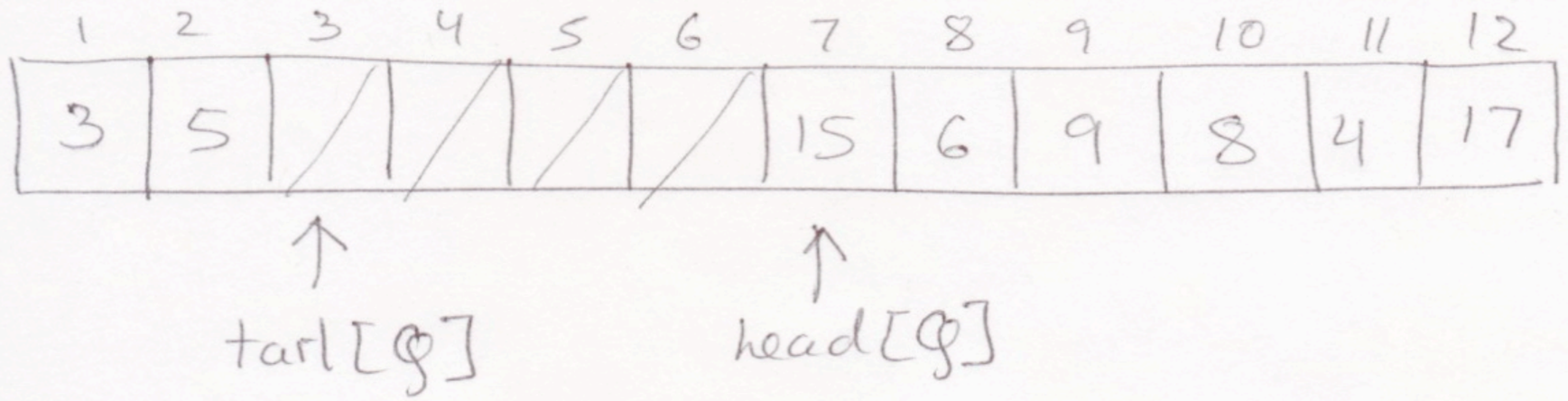
Example:



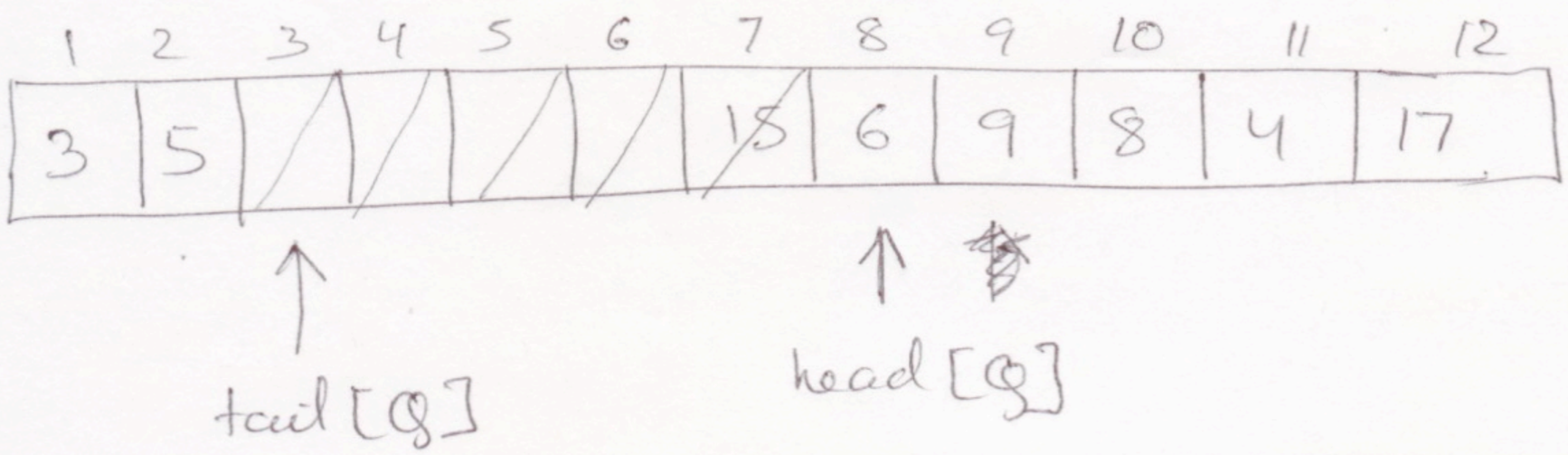
Enqueue (Q, 17)

Enqueue (Q, 3)

Enqueue (Q, 5)



Dequeue (Q)



Note

① When head [Q] = tail [Q]
The queue is empty

② Initially we have
head [Q] = tail [Q] = 1

③ When head [Q] = tail [Q] + 1, the queue is full

Algorithms to Enqueue and Dequeue.

```
ENQUEUE (Q, x)
    Q[tail[Q]] ← x
    if tail[Q] = length[Q]
        then tail[Q] ← 1
        else tail[Q] ← tail[Q] + 1
```

```
Dequeue (Q)
    x ← Q[head[Q]]
    If head[Q] = length[Q]
        then head[Q] ← 1
        else head[Q] ← head[Q] + 1
    return (x)
```

Note: This pseudo code does not check for over/under flow. Try this as an exercise

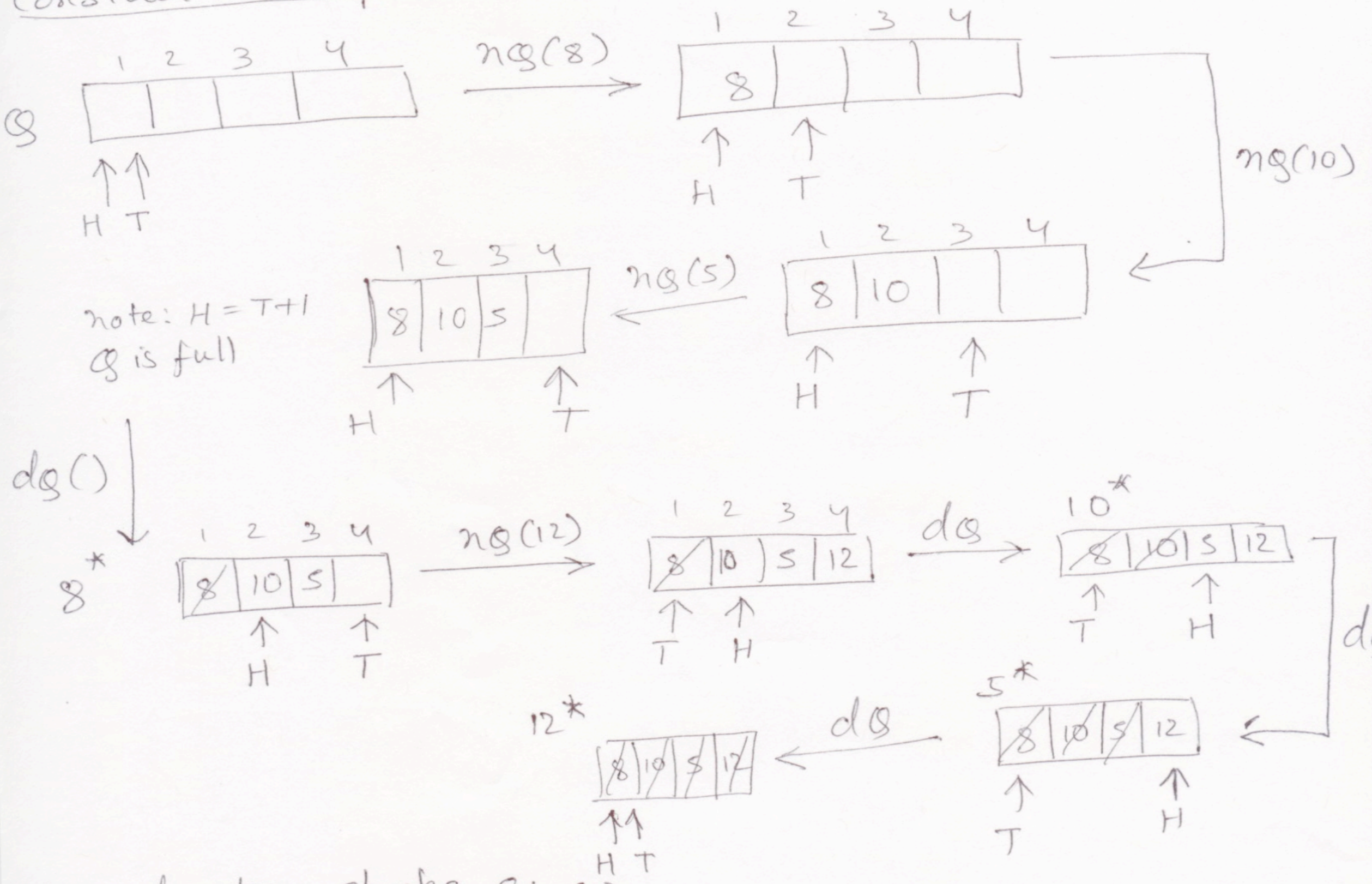
Question: What is the complexity of Enqueue and Dequeue operations?

Answer : O(1)

Question

How to implement a queue using 2 stacks
What will be the complexity of queue operations?

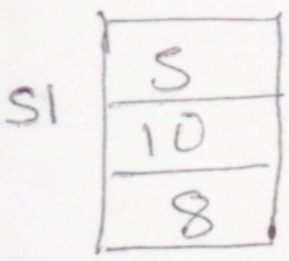
Consider a simple case



Consider two stacks S1, S2

nq = Push(S1)

Push(8), Push(10), Push(5)

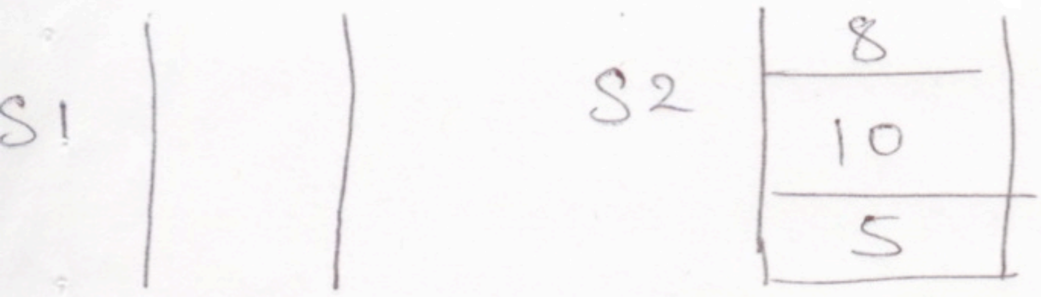


dq = ① Reverse the order. Pop and push on S2

POP(S1) PUSH(S2)

POP(S1) PUSH(S2)

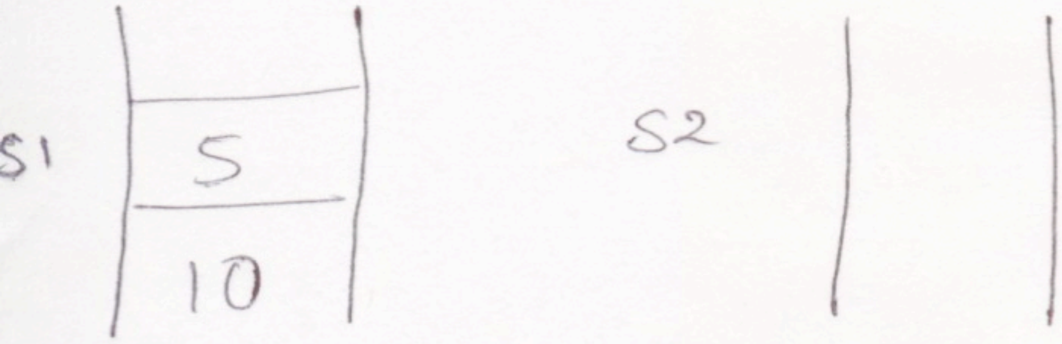
POP(S1) PUSH(S2)



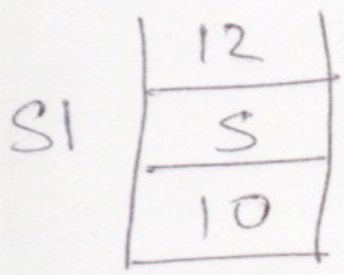
POP (S2) ⟵ 8* // The result of the deque operation

POP (S2) PUSH (S1)

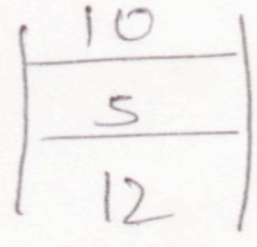
POP (S2) PUSH (S1)



enQ (12) ≡ PUSH (12)

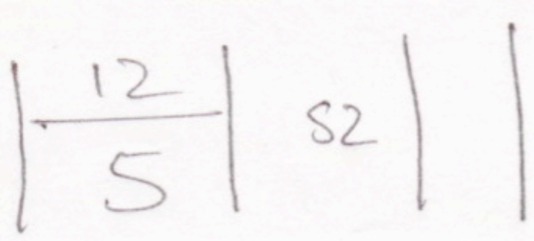


deQ



POP(S2) ⟵ 10*

POP(S2) PUSH(S1)
POP(S2) PUSH(S1)



deQ

POP(S1) PUSH(S2)
POP(S1) PUSH(S2)
POP(S2) ⟵ 5*

⋮

⟵ 12*

```
ENQUEUE 2STACK (Q, x)
    Push (S1, x)

DEQUEUE 2 STACK (Q)
    WHILE (STACK-EMPTY(S1)) = FALSE
        PUSH (S2, POP(S1))
    POP(S2)
    WHILE (STACK-EMPTY (S2)) = FALSE
        PUSH (S1, POP(S2))
```

Enqueue 2 STACK — $O(1)$

Dequeue 2 STACK — $O(n) + O(1) + O(n)$

$= O(2n) + O(1) = O(n)$

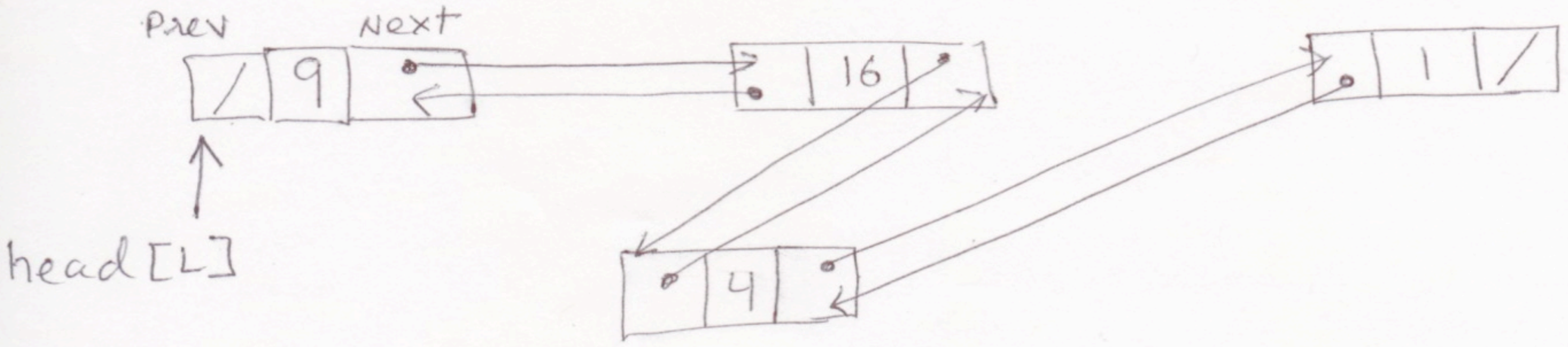
EXERCISE

SHOW HOW to Implement a STACK using two queues. Analyze the running time of the stack operations

LINKED LISTS

A Linked list is a data structure, in which the objects are arranged in a linear order.
Unlike an array, where the order is defined by the array indices, the order in a linked list is defined by a pointer contained in each object

Example



- Each element of this list is an object with
  - a key field
  - two pointer fields : previous and next (prev)

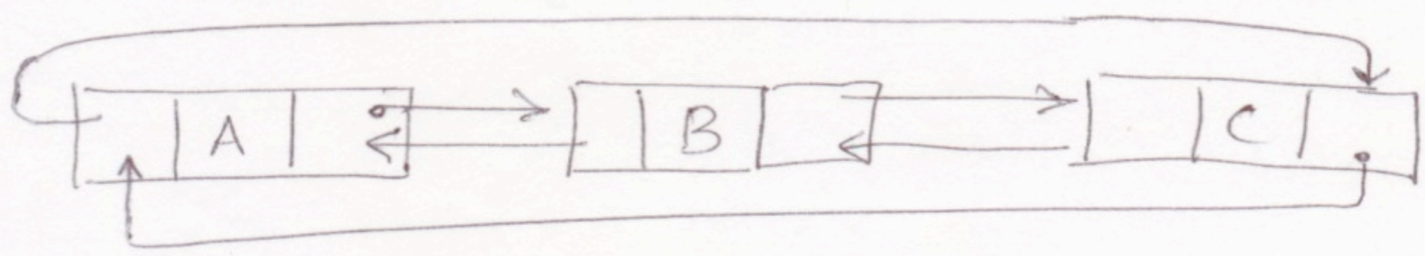
  Given an element x in this list,
  prev : points to its predecessor
  next : points to its successor

- If prev[x] = NIL, x is the first element. It is also called the head
- If next[x] = NIL, x is the last element and

- The attribute head[L] points to the first element of the list. If head[L] = NIL, then the list is empty.

- A List may be singly-linked (no prev pointer) or doubly-linked

- A list may be sorted or not and may be circular or not. In a circular list, the prev pointer of the head points to the tail and the next pointer of the tail points to the head.



Searching a List

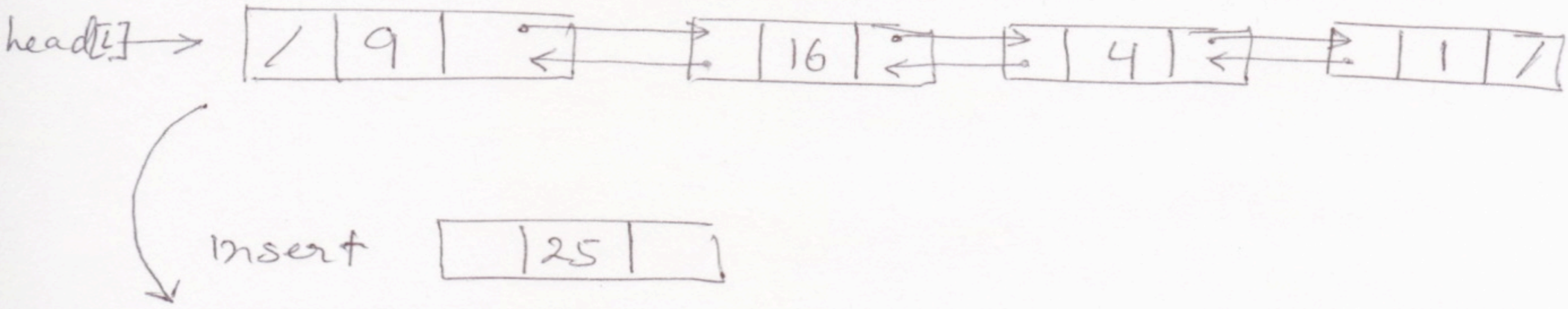
Find the first element with key = k in the list

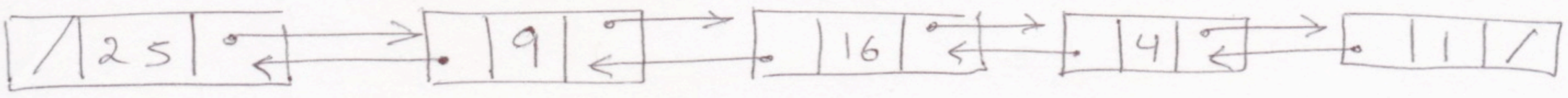
```
List-Search (L, k)
    x ← head[L]
    while x ≠ NIL AND key[x] ≠ k
        do x ← next[x]
    return x
```

Complexity ? O(n) in the worst-case.

Inserting into a Linked List

Given an element x whose key field has already been set, the procedure LIST-INSERT splices x onto the front of the linked list



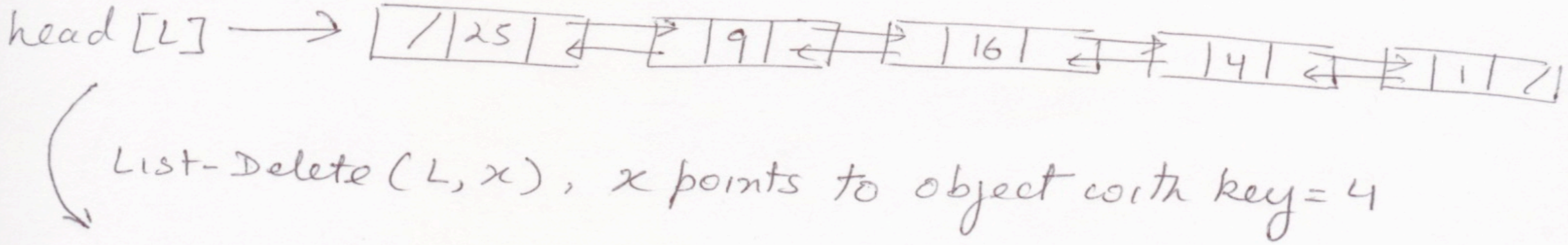



```
LIST-INSERT(L, x)
    next[x] ← head[L]
    if head[L] ≠ NIL
        then prev[head[L]] ← x
    head[L] ← x
    Prev[x] ← NIL
```

Complexity $O(1)$

Deleting from a Linked List

The procedure LIST-DELETE, removes an element x from a linked dist L. It is given a pointer to x and splices x out of L by updating pointers




```
LIST-DELETE (L, x)
    if prev [x] ≠ NIL
        then next (prev [x]) ← next (x)
        else head [L] ← next (x)
    If next (x) ≠ NIL
        then prev (next [x]) ← prev (x)
```

Complexity = O (1)

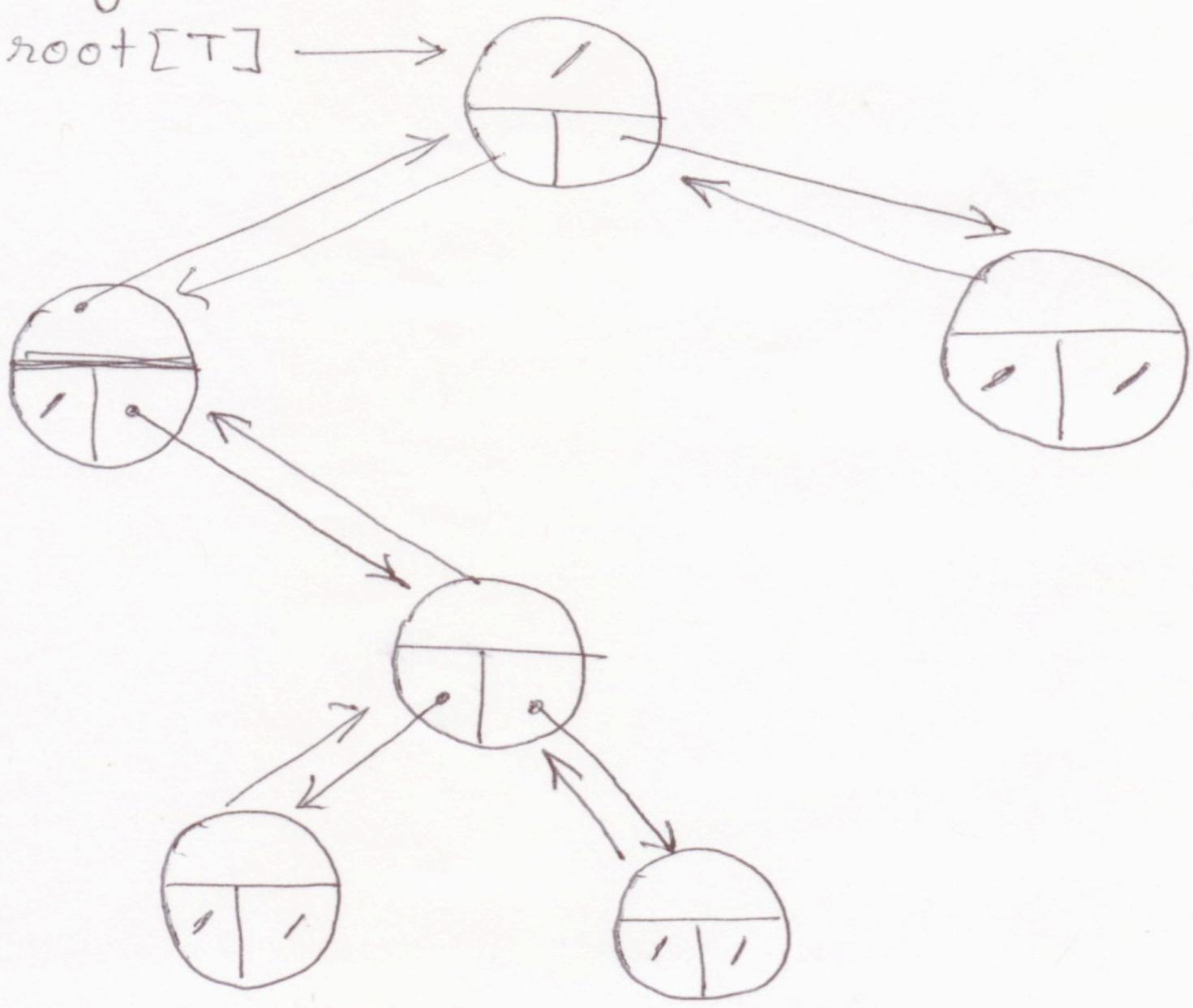
complexity of finding an element and then deleting it

= O(n) + O(1) = O(n)

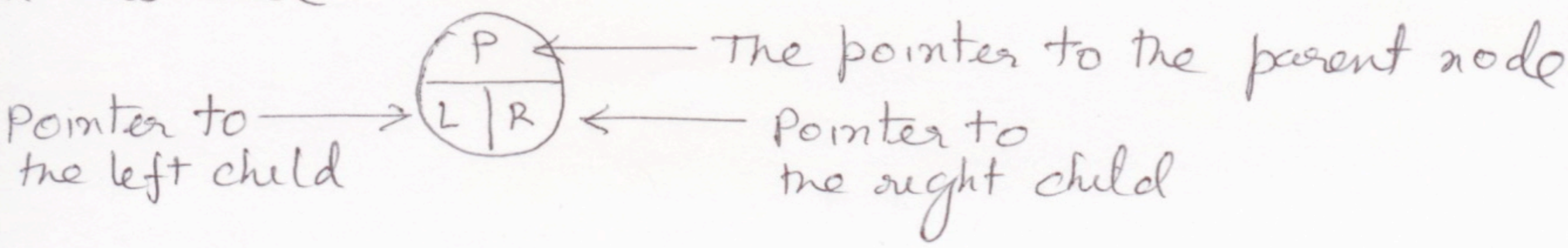
Representing Trees using linked structures

- Each node of a tree is represented by an object
- Each node contains a key field. The remaining fields of interest are pointers to other nodes

Binary trees



In this case



- If P[x] = NIL, this is the root node
- If L[x] and R[x] = NIL, this is a leaf node (no children)

	Unsorted SLL	Sorted SLL	Unsorted DLL	Sorted DLL
Search (L, k)				
Search (L, x)				
Insert (L, k)				
Insert (L, x)				
Delete (L, k)				
Delete (L, x)				
Successor (L, k)				
Successor (L, x)				
Predecessor (L, k)				
Predecessor (L, x)				
minimum (L)				
maximum (L)				

(L, k): denotes a pointer to a list element
(L, x): value of an element (key-value)

FILL THE TABLE with worst case complexity bounds