Elementary Data Structures (Dynamic Sets)

ITX2010, CSX3003, IT2230

Data Structures and Algorithms, Information Structures

Learning Objectives

Students will be able to:

- Understand what dynamic set is.
- Examine the representation of dynamic sets by simple data structures – such as stack queue, linked list and trees.

Chapter Outline

1. Dynamic set
   1) Elements if a dynamic set
   2) Operations on dynamic set
2. Stack and Queue
   1) Stack
   2) Queue
3. Linked list
   1) Linked list
   2) Searching a linked list
   3) Inserting into a linked list
   4) Deleting from a linked list
   5) Sentinels
4. Representing rooted tree
   1) Binary tree
   2) Rooted tree with unbounded branching

10.1

Dynamic set

1) Elements of a dynamic set
2) Operations on dynamic set

Elements of a dynamic set

Dynamic Set

- Set is as fundamental both computer science and mathematics.
- Mathematica sets are unchanging.

- The set manipulated by algorithms can grow, shrink, or otherwise change over time -> is called “dynamic”. [1]
  - It requires operations to be performed on it.
  - Stack, Queue, Linked list, tree.

- Implement a dynamic set depends upon the operations that must supported. [1]

Elements of a dynamic set

Dynamic Set

- A typical implementation of a dynamic set is represented by an object whose fields can be examined and how we manipulated.

- A dynamic set can be:
  - An identifying key field,
  - A set of key values,

Operations on dynamic sets

Dynamic Set

- Operations on a dynamic set can be grouped into two categories:
  1. Query -> simply returns information about the set and
  2. Modification -> changes the set.

Query	Search(S,k) = Query the element contains k in the set.
	Minimum(S) / Maximum(S) = Query min / max element of the set.
	Successor(S,x) / Predecessor(S,x) = Query the next larger / smaller element pointer of the set
Modify	Insert(S,x) = Insert operation
	Delete(S,x) = Delete operation
	Update(S,x) = Update operation

10.2

Stack and Queue

1) Stack
2) Queue

Stack

Stack and Queue

- It is a dynamic sets -> where the element removed from the set is the one most recently inserted. [1]

- It is Last-in, First-out (LIFO) policy
  - The order in which they are popped will be the reverse of the order in which they were pushed onto the stack.

- Insert operation is called “Push”,
- Delete operation is called “Pop” and it must take nor argument and always remove the top item.

10.2

Stack

Stack and Queue

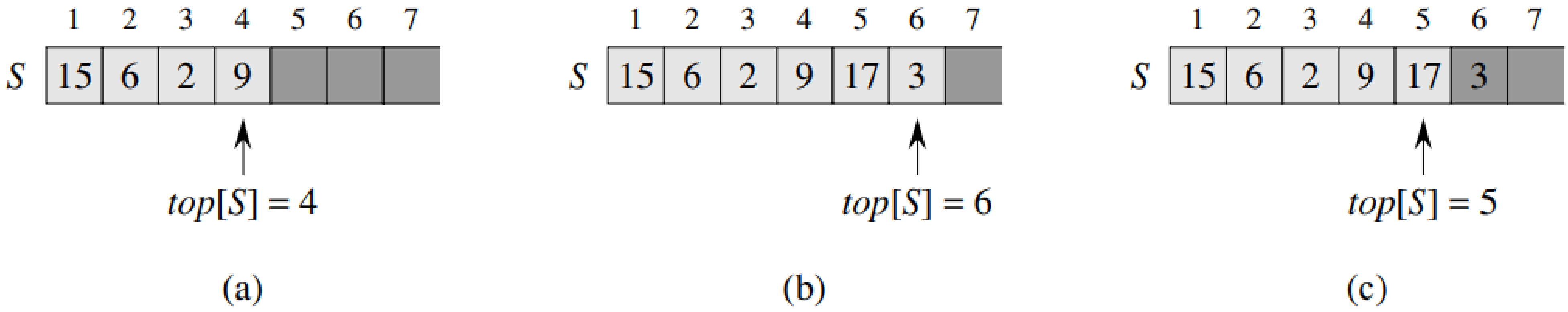


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17. [1]

Stack

Stack and Queue

STACK-EMPTY(S)

```
if top[S] = 0
    then return TRUE
    else return FALSE
```

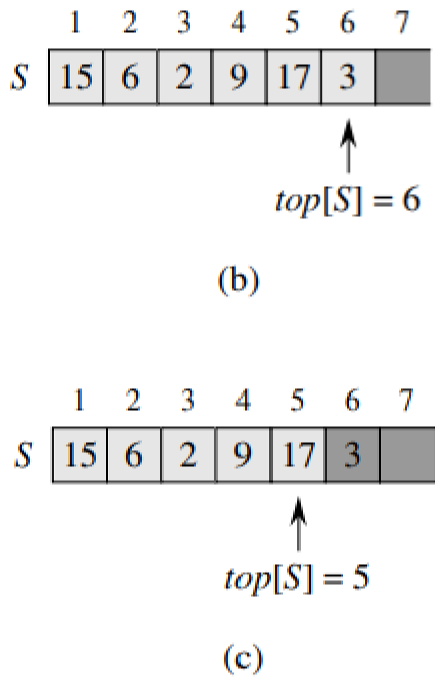
PUSH(S, x)

```
top[S] ← top[S] + 1
S[top[S]] ← x
```

POP(S)

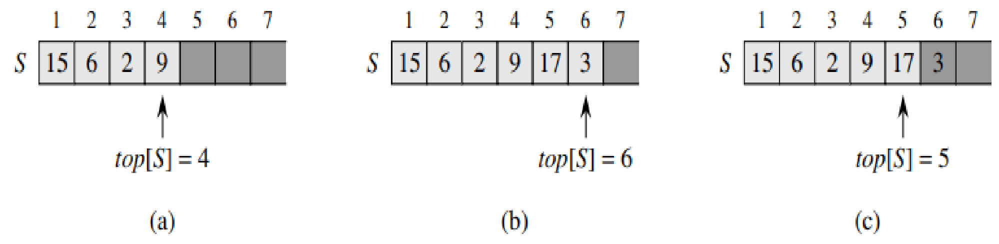
```
if STACK-EMPTY(S)
    then error "underflow"
    else top[S] ← top[S] − 1
         return S[top[S] + 1]
```

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time. [1]



```python
# Stack array implementation in python
# Creating a stack
def create_stack():
    stack = []
    return stack
# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0
# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("Push: " + item)
# Removing an element from the stack
def pop(stack):
    if (check_empty(stack)):
        return "Stack array S[] is empty"
    return stack.pop()

stack = create_stack()
print("Stack S array S[]:"+ str(stack))
push(stack, str(15))
push(stack, str(6))
push(stack, str(2))
push(stack, str(9))
print("\nFigure 10.1 (a) Stack S has 4 elements.")
print("Stack array S[]:" + str(stack))
print("\nFigure 10.1 (b) Stack S after calls push(S, 17) and push(S,3).")
push(stack, str(17))
push(stack, str(3))
print("Stack array S[]:" + str(stack))
print("\nFigure 10.1 (c) Stack S after calls pop(S) has returned element 3, which is the one most recently pushed.")
print("Pop: " + pop(stack))
print("Stack array S[]:" + str(stack))
```

```
Stack S array S[]:[]
Push: 15
Push: 6
Push: 2
Push: 9

Figure 10.1 (a) Stack S has 4 elements.
Stack array S[]:['15', '6', '2', '9']

Figure 10.1 (b) Stack S after calls push(S, 17) and push
  (S,3).
Push: 17
Push: 3
Stack array S[]:['15', '6', '2', '9', '17', '3']

Figure 10.1 (c) Stack S after calls pop(S) has returned
  element 3, which is the one most recently pushed.
Pop: 3
Stack array S[]:['15', '6', '2', '9', '17']
```



Queue

Stack and Queue

- A dynamic set is -> where the element removed from one side and inserted to another one side. [1]
- It is First-in, First-out (FIFO) policy.
- Dequeue must take no argument but remove from a side that is not enqueue side.
- Insert operation is called “Enqueue”.
- Delete operation is called “Dequeue”.
- It has a head and tail,
  - Enqueued item will take place at the tail of the queue and
  - Dequeued item will be the one at the head of the queue.

Queue

Stack and Queue

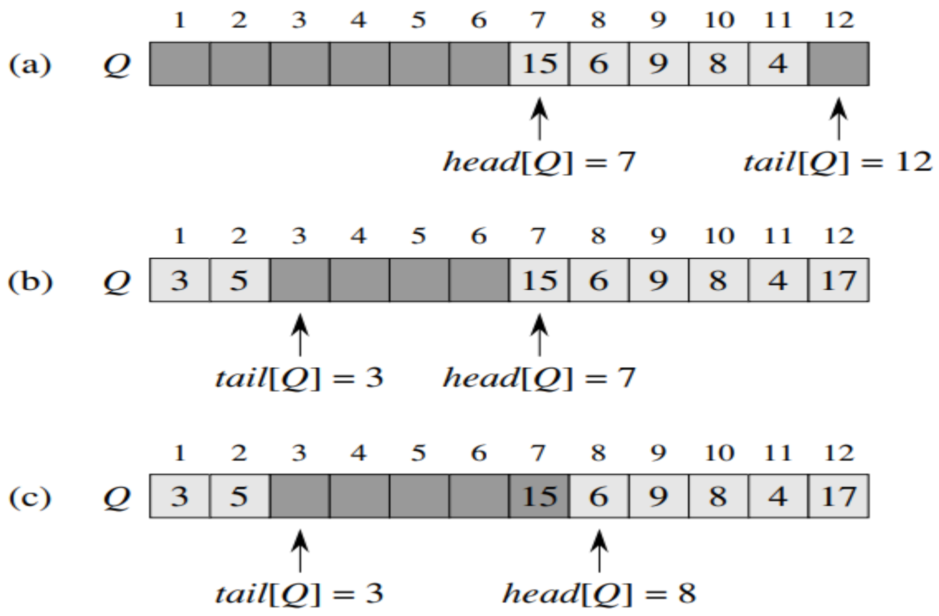


Figure 10.2 A queue implemented using an array $Q[1 \,..\, 12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7 \,..\, 11]$. **(b)** The configuration of the queue after the calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5). **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6. [1]

Queue

Stack and Queue

```
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
```

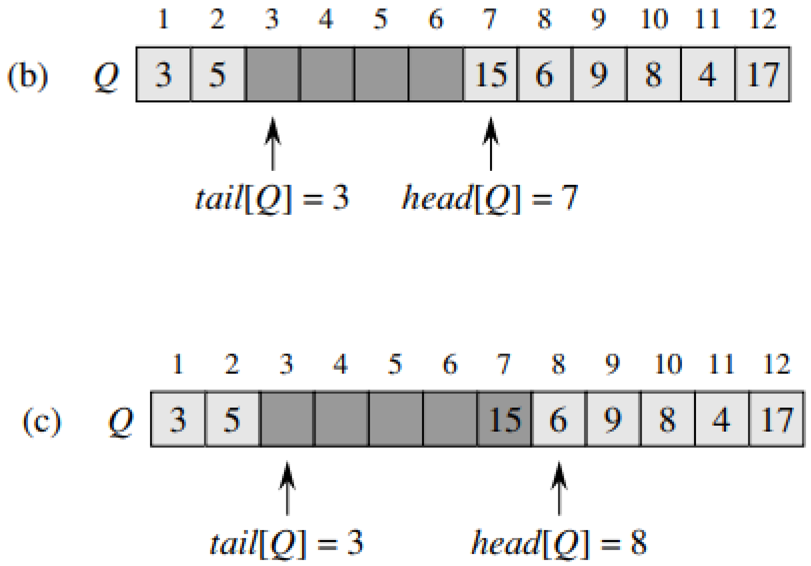
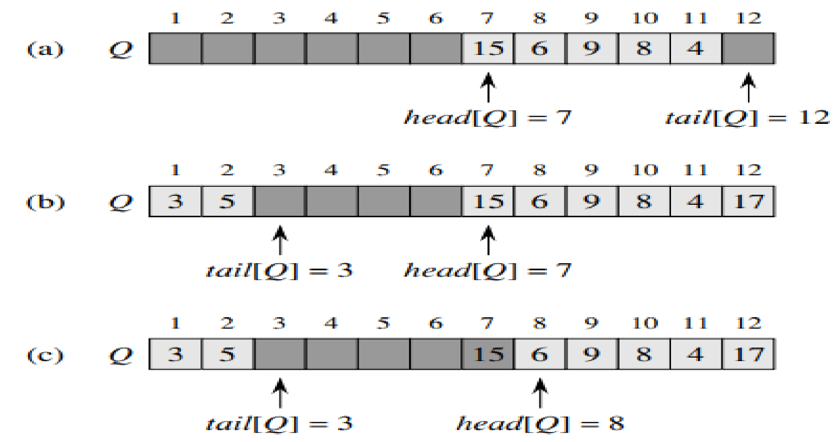


Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time. [1]

[3]

```python
class CircularQueue():

    def __init__(self, size): # initializing the class
        self.size = size
        # initializing queue with none
        self.queue = [None for i in range(size)]
        self.front = self.rear = -1

    def enqueue(self, data):
        # condition if queue is full
        if ((self.rear + 1) % self.size == self.front):
            print(" Queue is Full\n")
        # condition for empty queue
        elif (self.front == -1):
            self.front = 0
            self.rear = 0
            self.queue[self.rear] = data
        else:
            # next position of rear
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = data

    def dequeue(self):
        if (self.front == -1): # codition for empty queue
            print ("Queue is Empty\n")
        # condition for only one element
        elif (self.front == self.rear):
            temp=self.queue[self.front]
            self.front = -1
            self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return temp
```

```
Queue[]= 0 : 1 ,  1 : 2 ,  2 : 3 ,  3 : 4 ,  4 : 5 ,  5 : 6 ,  6 : 15 ,  7 : 6 ,  8 : 9 ,  9 : 8 ,  10 : 4 ,
The thread 'MainThread' (0x1) has exited with code 0 (0x0).
Dequeue: 1
Dequeue: 2
Dequeue: 3
Dequeue: 4
Dequeue: 5
Dequeue: 6

Figure 10.2 (a)
Queue[]= 6 : 15 ,  7 : 6 ,  8 : 9 ,  9 : 8 ,  10 : 4 ,

Figure 10.2 (b)
Queue[]= 6 : 15 ,  7 : 6 ,  8 : 9 ,  9 : 8 ,  10 : 4 ,  11 : 17 ,

Figure 10.2 (c)
Queue[]= 7 : 6 ,  8 : 9 ,  9 : 8 ,  10 : 4 ,  11 : 17 ,  0 : 3 ,  1 : 5 ,
The program 'python.exe' has exited with code 0 (0x0).
```

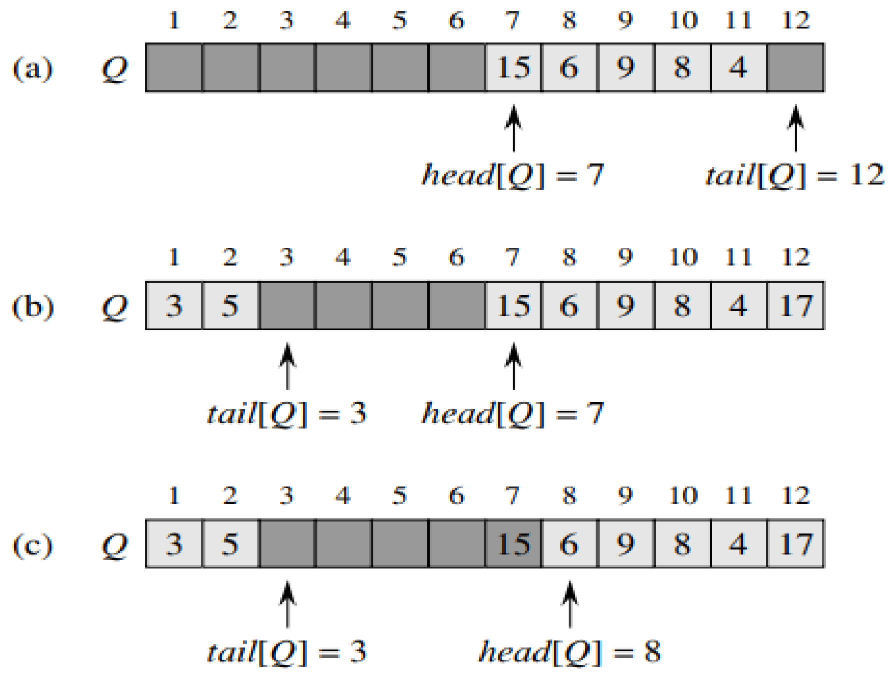


[1]

[3]

```python

    def display(self):
        # condition for empty queue
        if(self.front == -1):
            print ("Queue is Empty")
        elif (self.rear >= self.front):
            print("Queue[]=",
                                              end = " ")
            for i in range(self.front, self.rear + 1):
                print(i,":", self.queue[i],", ", end = " ")
            print ()
        else:
            print ("Queue[]=",
                                           end = " ")
            for i in range(self.front, self.size):
                print(i,":", self.queue[i],", ", end = " ")
            for i in range(0, self.rear + 1):
                print(i,":", self.queue[i],", ", end = " ")
            print ()
        if ((self.rear + 1) % self.size == self.front):
            print("Queue is Full")

# Queue Demonstration
ob = CircularQueue(12)
ob.enqueue(1)
ob.enqueue(2)
ob.enqueue(3)
ob.enqueue(4)
ob.enqueue(5)
ob.enqueue(6)
ob.enqueue(15)
ob.enqueue(6)
ob.enqueue(9)
ob.enqueue(8)
ob.enqueue(4)
ob.display()
print ("Dequeue:", ob.dequeue())
print ("Dequeue:", ob.dequeue())
print ("Dequeue:", ob.dequeue())
print ("Dequeue:", ob.dequeue())
print ("Dequeue:", ob.dequeue())
print ("Dequeue:", ob.dequeue())
print("\nFigure 10.2 (a)")
ob.display()
ob.enqueue(17)
print("\nFigure 10.2 (b)")
ob.display()
ob.dequeue()
ob.enqueue(3)
ob.enqueue(5)
print("\nFigure 10.2 (c)")
ob.display()
```

```
Dequeue: 1
Dequeue: 2
Dequeue: 3
Dequeue: 4
Dequeue: 5
Dequeue: 6

Figure 10.2 (a)
Queue[]= 6 : 15 ,  7 : 6 ,  8 : 9 ,  9 : 8 ,  10 : 4 ,

Figure 10.2 (b)
Queue[]= 6 : 15 ,  7 : 6 ,  8 : 9 ,  9 : 8 ,  10 : 4 ,  11 : 17 ,

Figure 10.2 (c)
Queue[]= 7 : 6 ,  8 : 9 ,  9 : 8 ,  10 : 4 ,  11 : 17 ,  0 : 3 ,  1 : 5 ,
The program 'python.exe' has exited with code 0 (0x0).
```



[1]

10.3

Linked List

1) Searching a linked list
2) Inserting into a linked list
3) Deleting from a linked list
4) Sentinel

Linked List

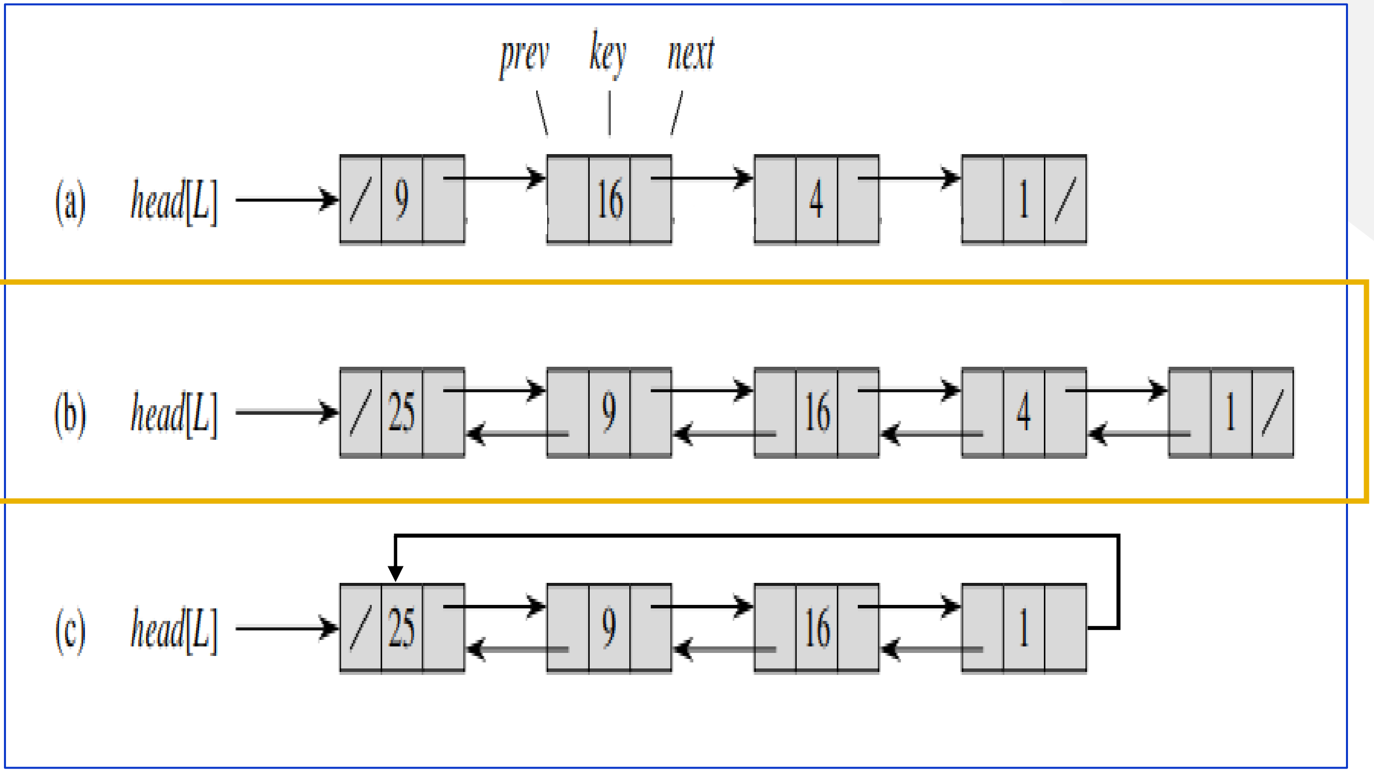
Linked List

- It is a data structure in which the objects are arranged in a linear order. [1]
- It can be sorted or unsorted list.
- Unlike an array, the linked list order is determined by a pointer in each object.
- Like list, it provides a simple, flexible representation for dynamic set.

Linked List

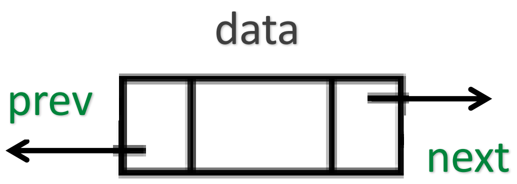
Linked List

- Linked List
  - Singly Linked List
  - Doubly Linked List
  - Circularly Doubly Linked List



Doubly Linked Node

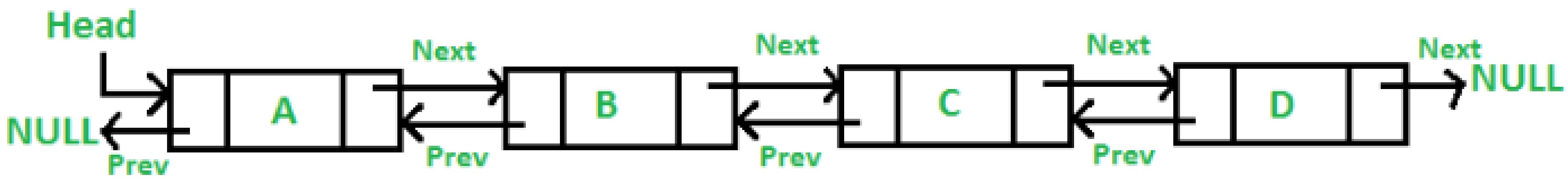
Linked List




```python
# Node of a doubly linked list  [3]
class Node:
    def __init__(self, next=None, prev=None, data=None):
        self.next = next # reference to next node in DLL
        self.prev = prev # reference to previous node in DLL
        self.data = data
```

Doubly Linked List

Linked List



[3]

```python
# Node of a doubly linked list
class Node:
    def __init__(self, next=None, prev=None, data=None):
        self.next = next # reference to next node in DLL
        self.prev = prev # reference to previous node in DLL
        self.data = data

# Class to create a Doubly Linked List
class DoublyLinkedList:
    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None
```

Linked List

Linked List

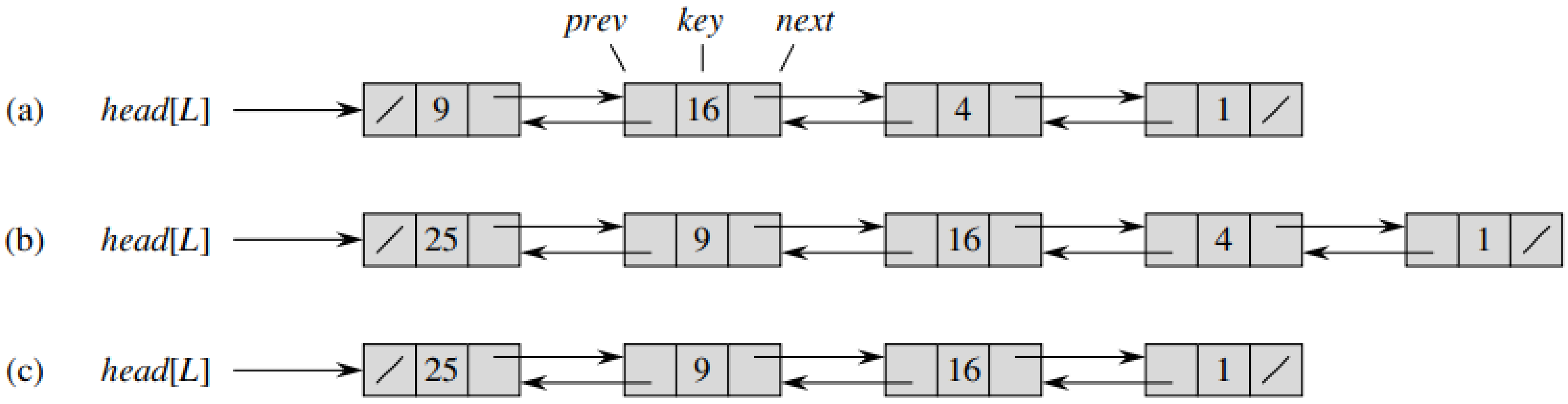


Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The *next* field of the tail and the *prev* field of the head are NIL, indicated by a diagonal slash. The attribute *head*$[L]$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where *key*$[x] = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4. [1]

Inserting into a linked list

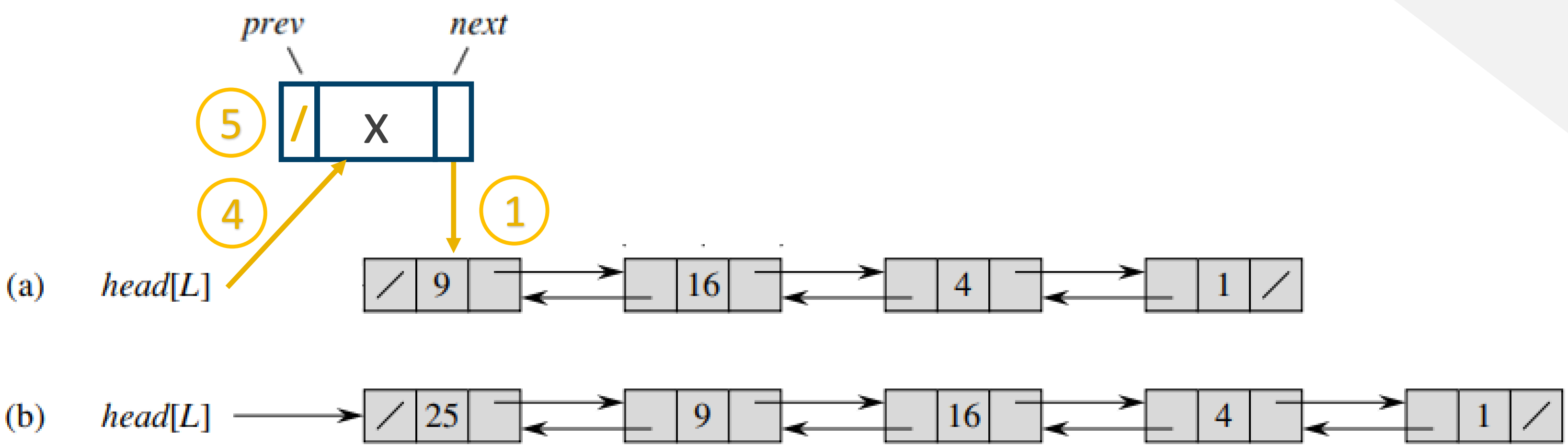
Linked List

- An inserted node can be added in several conditions:

1. Add at front
    - -> can be implemented in Push of a stack list
2. Add between two nodes
    - -> This should provide for supporting a normal insertion and carefully be dropped in the restricted list such as stack list or queue list.
3. Add at end
    - -> can be adopted to be Enqueue of a queue list

Inserting into a linked list

Linked List



[1]

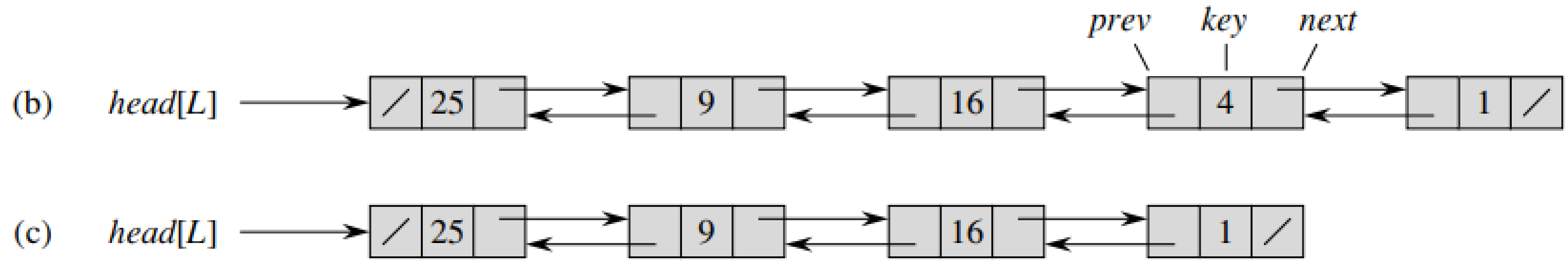
LIST-INSERT(L, x)

$next[x] \leftarrow head[L]$
if $head[L] \neq$ NIL
  then $prev[head[L]] \leftarrow x$
$head[L] \leftarrow x$
$prev[x] \leftarrow$ NIL

The running time for LIST-INSERT on a list of n elements is $O(1)$.

Deleting from a linked list

Linked List



LIST-DELETE(L, x)

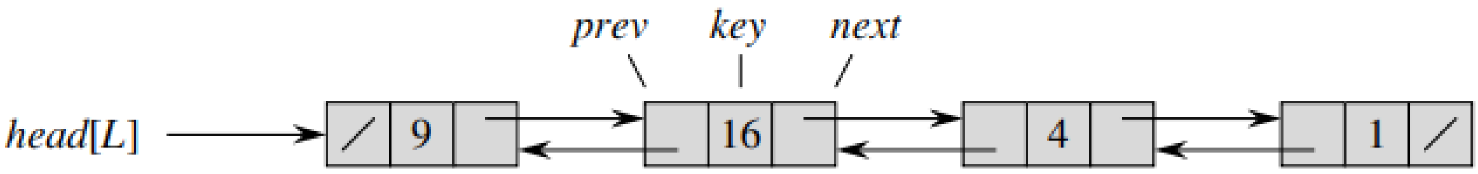
if $prev[x] \neq$ NIL
    then $next[prev[x]] \leftarrow next[x]$
    else $head[L] \leftarrow next[x]$
if $next[x] \neq$ NIL
    then $prev[next[x]] \leftarrow prev[x]$

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH. [1]

Searching a linked list

Linked List

(a)



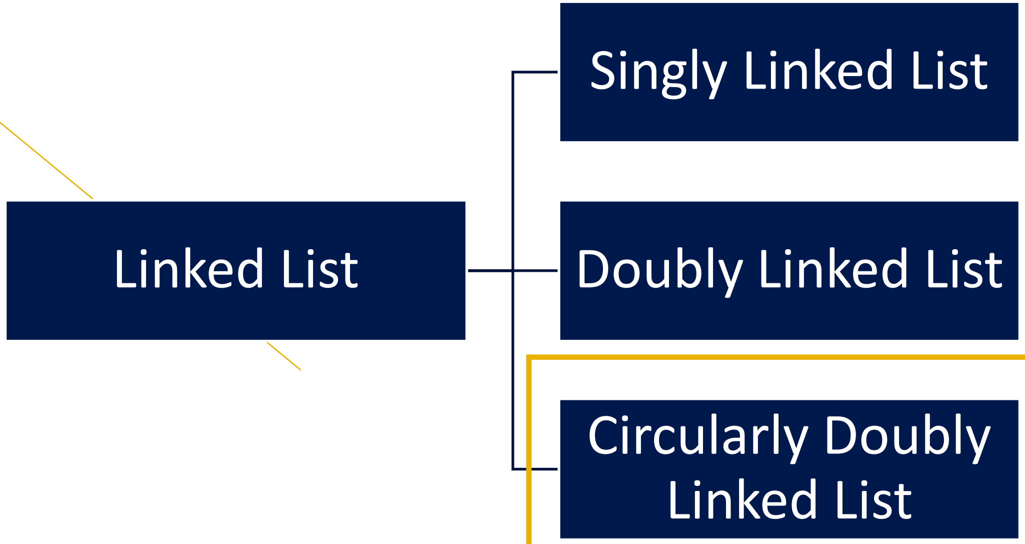

```
LIST-SEARCH(L, k)
x ← head[L]
while x ≠ NIL and key[x] ≠ k
      do x ← next[x]
return x
```

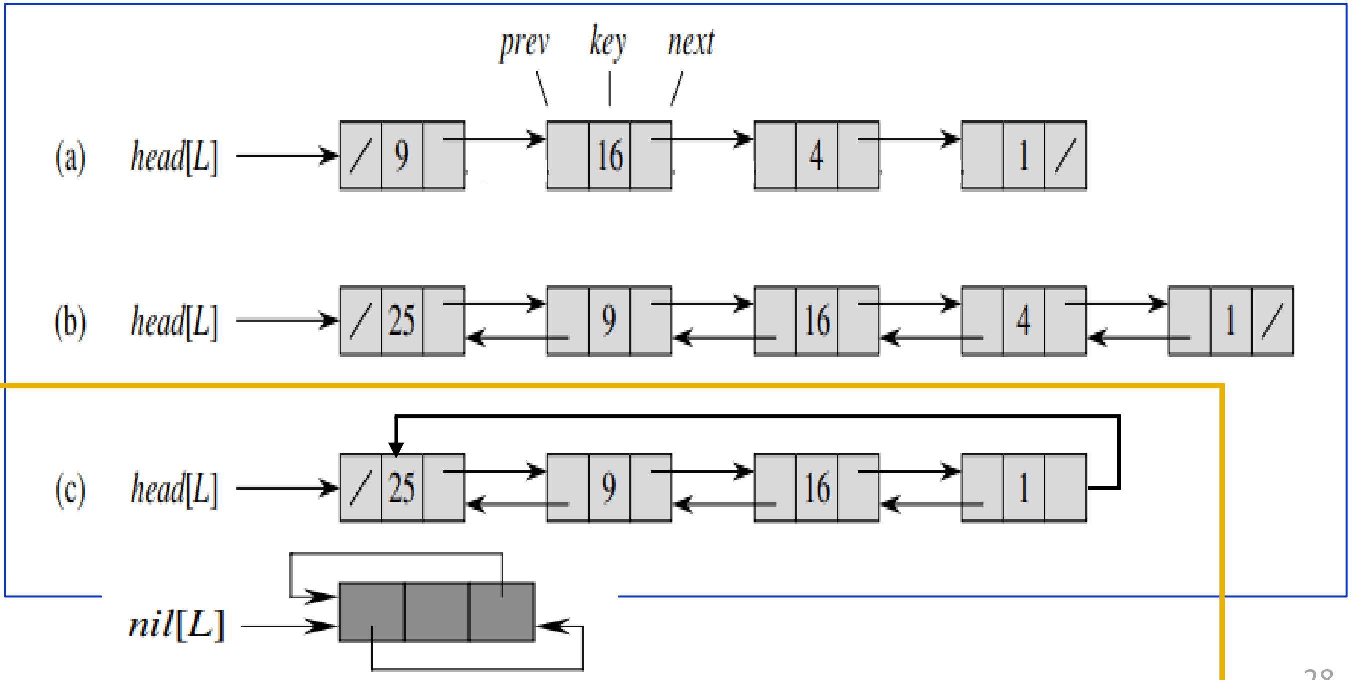
To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list. [1]

Sentinels

Linked List

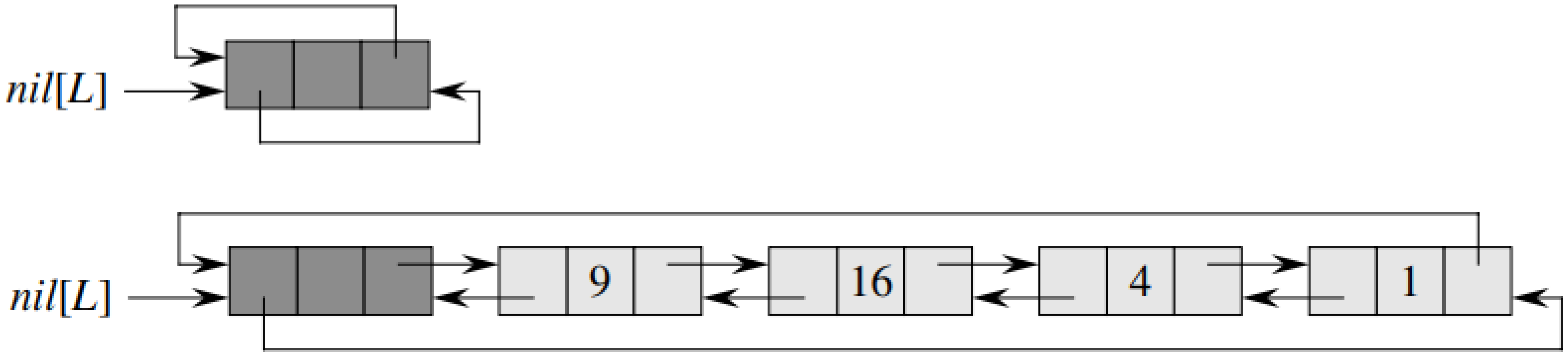
- It is a dummy object (nil[L]) that allows us to simplify the boundary condition.
- If linked list is a circular doubly linked list, the sentinel nil[L] is placed between head and tail. [1]





Linked List

Linked List



LIST-SEARCH(L, k)

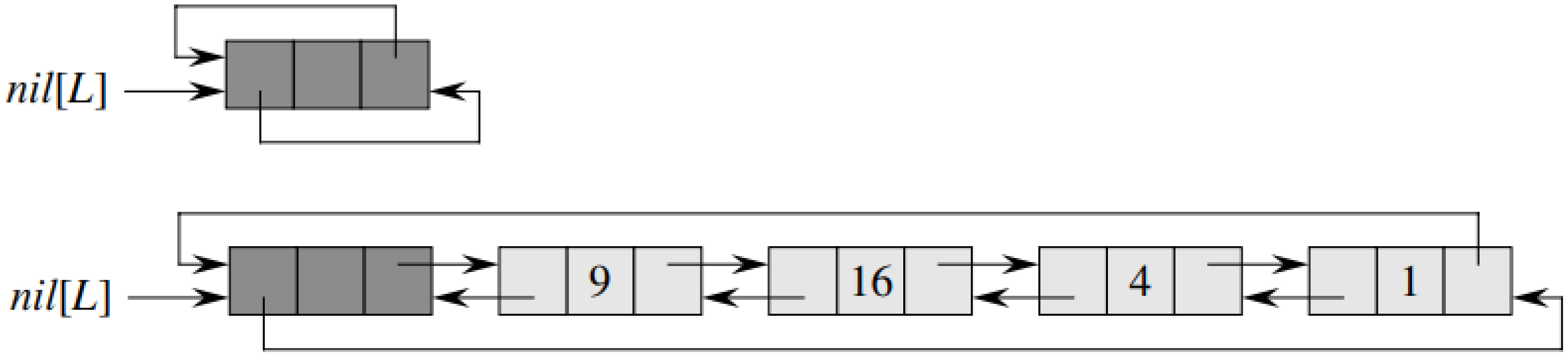
```
x ← head[L]
while x ≠ NIL and key[x] ≠ k
      do x ← next[x]
return x
```

LIST-SEARCH'(L, k)

```
x ← next[nil[L]]
while x ≠ nil[L] and key[x] ≠ k
      do x ← next[x]
return x
```

Linked List

Linked List



LIST-INSERT (L, x)

$next[x] \leftarrow head[L]$
if $head[L] \neq$ NIL
  then $prev[head[L]] \leftarrow x$
$head[L] \leftarrow x$
$prev[x] \leftarrow$ NIL

LIST-INSERT′ (L, x)

$next[x] \leftarrow next[nil[L]]$
$prev[next[nil[L]]] \leftarrow x$
$next[nil[L]] \leftarrow x$
$prev[x] \leftarrow nil[L]$

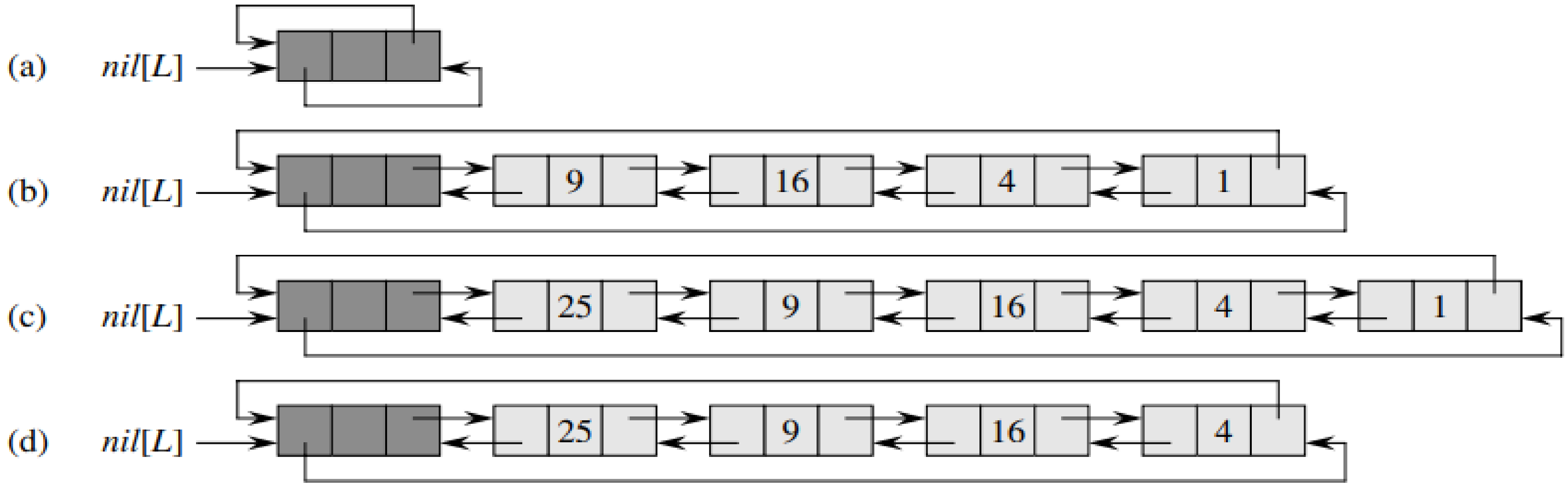


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel *nil*[L] appears between the head and tail. The attribute *head*[L] is no longer needed, since we can access the head of the list by *next*[*nil*[L]]. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where *key*[x] $= 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4. [1]

Sentinels

Linked List

- Sentinel rarely reduces the asymptotic time bounds of data structure operations, but they can reduce constant factors. [1]
  - Sentinel may help to tighten the code in the loop, thus reducing the coefficient of term in the running time.

- The gain from using sentinel within loop -> is usually used in simplifying the code rather than improving the speed.
  - Searching, Insertion and Deletion running time require O(1) time.

- Sentinel should not be used indiscriminately.
  - If there are many small lists, the extra storage can represent wasted memory.

10.4

Representing rooted tree

1) Binary trees
2) Root trees with unbounded branching

Binary trees

Representing rooted tree

As shown in Figure 10.9, we use the fields p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $p[x] = \text{NIL}$, then x is the root. If node x has no left child, then $left[x] = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $root[T]$. If $root[T] = \text{NIL}$, then the tree is empty.

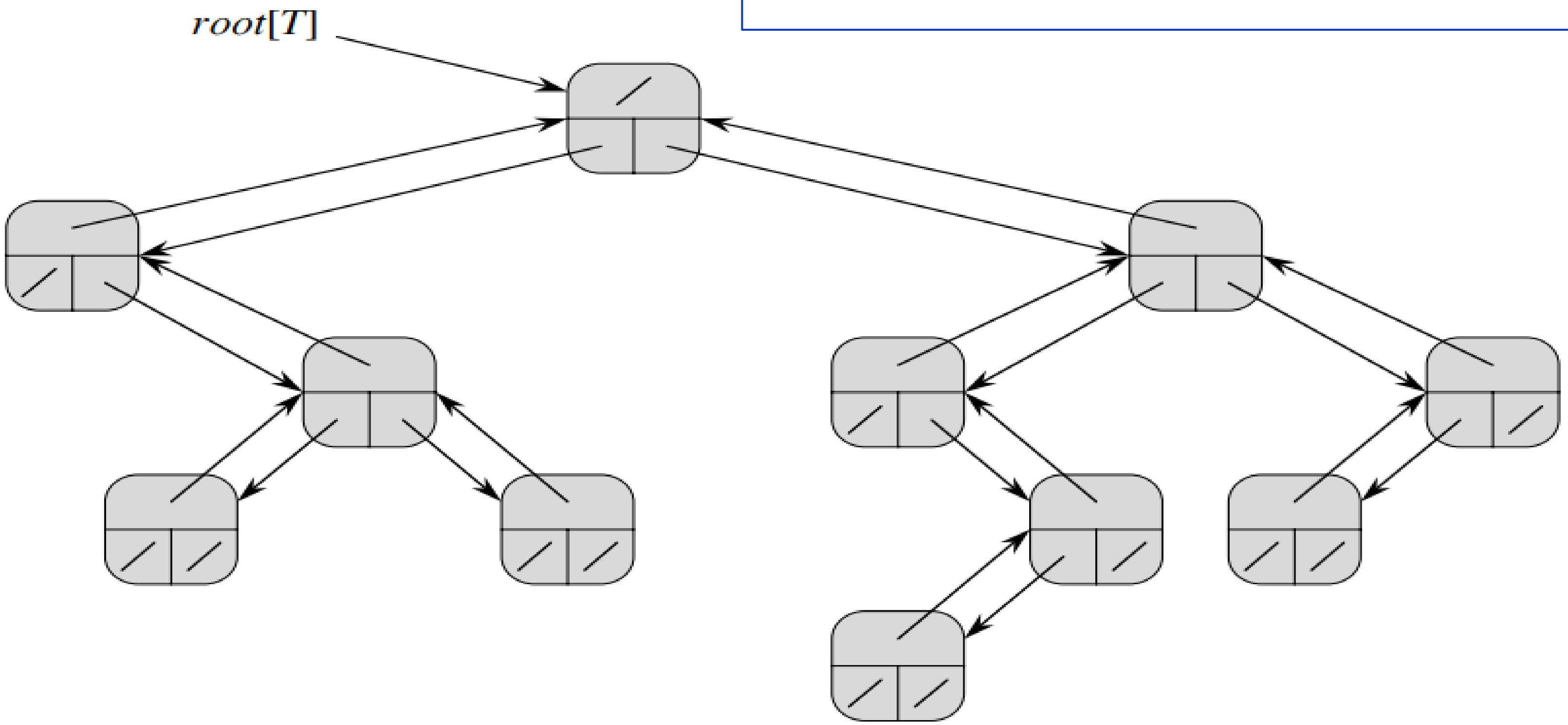


Figure 10.9 The representation of a binary tree T. Each node x has the fields $p[x]$ (top), $left[x]$ (lower left), and $right[x]$ (lower right). The *key* fields are not shown. [1]

Root trees with unbounded branching

Representing rooted tree

- The scheme for representing a binary tree (k=2) -> can be extended to any class of trees in which the member of children of each node is at most some constant k (k>2).
  - It no longer works when the number of children of a node is unbounded (k is not defined or unknown).
  - The advantage is only O(n) space for any n-node rooted tree.
- It is called "**left-child, right-sibling representation**". [1]
- Instead of having a pointer to each of its children, each node x has only two pointers:

  1. *left-child*[x] points to the leftmost child of node x, and
  2. *right-sibling*[x] points to the sibling of x immediately to the right.

1. *left-child*[x] points to the leftmost child of node x, and
2. *right-sibling*[x] points to the sibling of x immediately to the right.

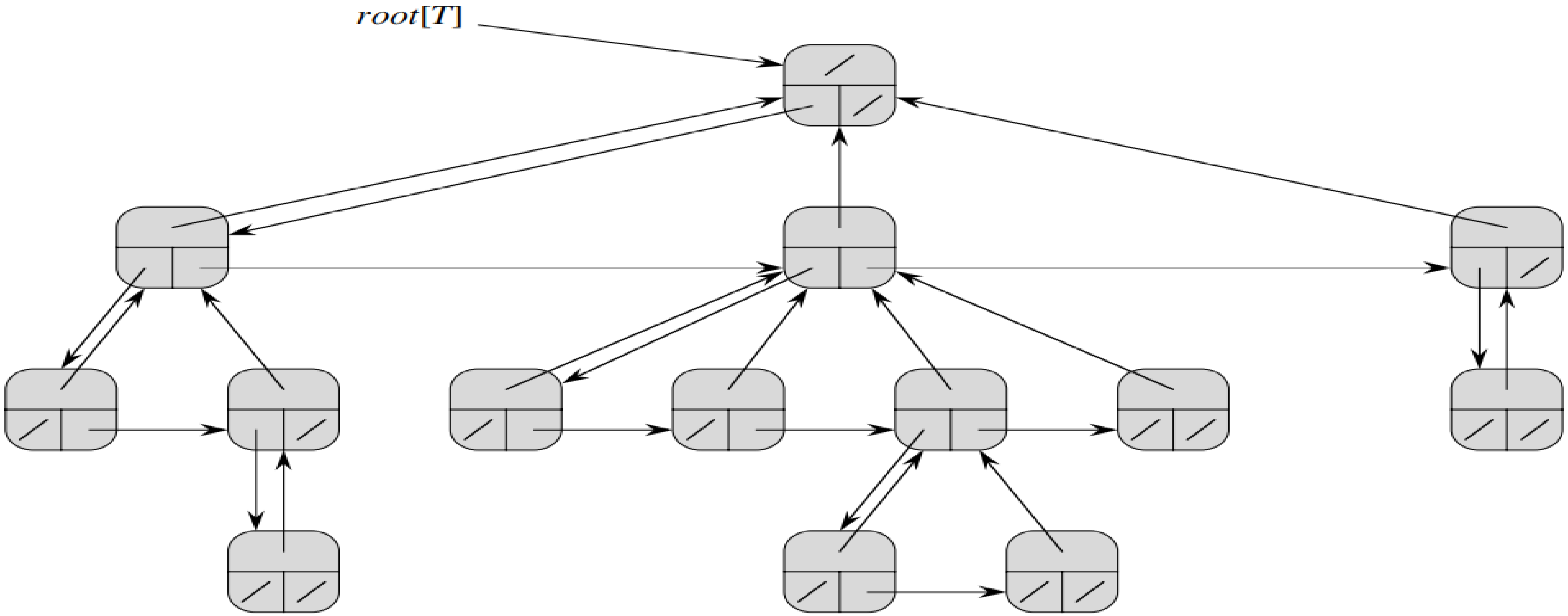


Figure 10.10 The left-child, right-sibling representation of a tree T. Each node x has fields $p[x]$ (top), *left-child*[x] (lower left), and *right-sibling*[x] (lower right). Keys are not shown. [1]

References

Texts | Integrated Development Environment (IDE)

[1] Introduction to Algorithms, Second Edition, Thomas H. C., Charles E. L., Ronald L. R., Clifford S., The MIT Press, McGraw-Hill Book Company, Second Edition 2001.

[2] https://visualstudio.microsoft.com/

[3] https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/