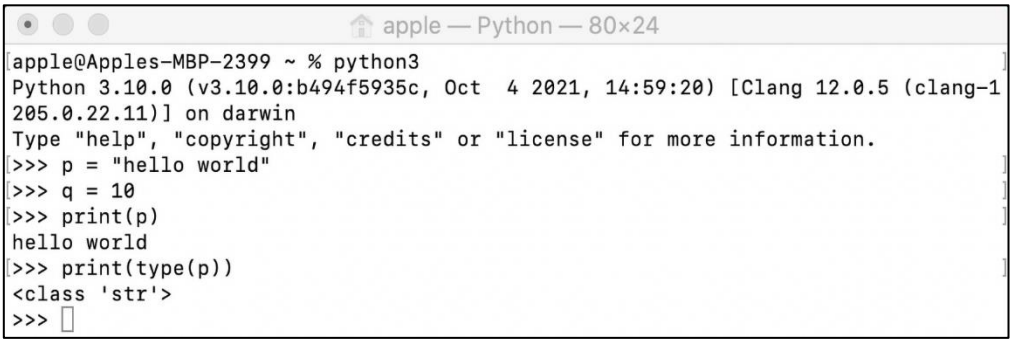
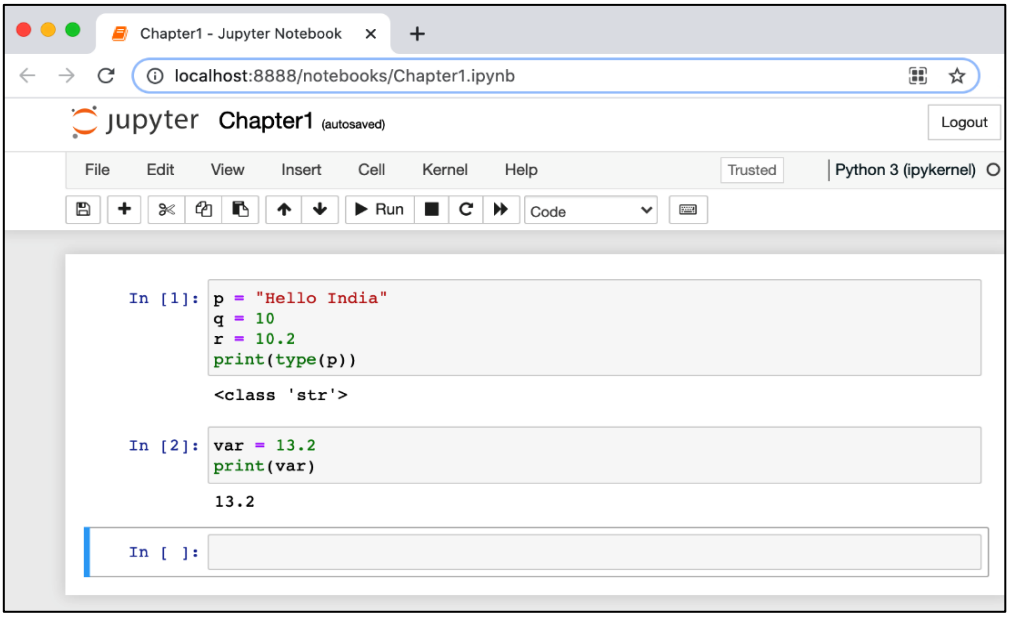
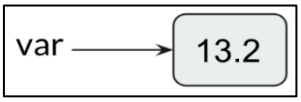
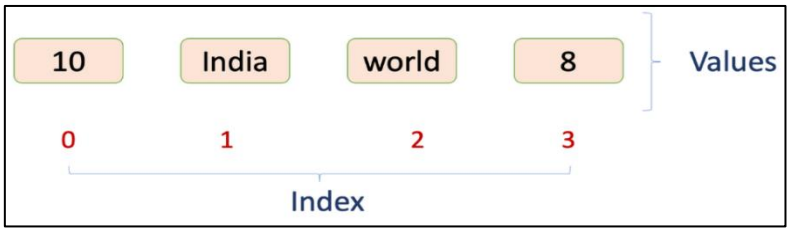
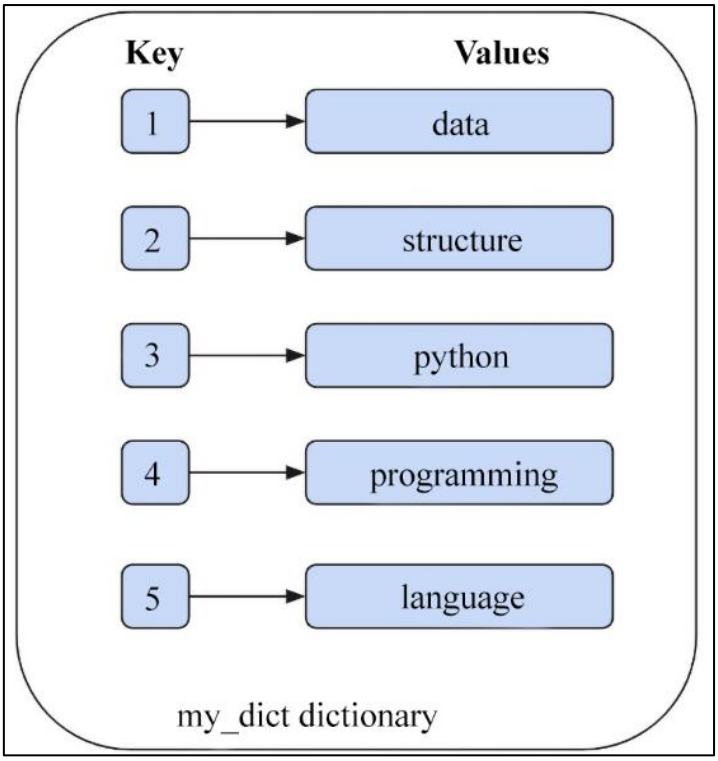
Chapter 1: Python Data Types and Structures

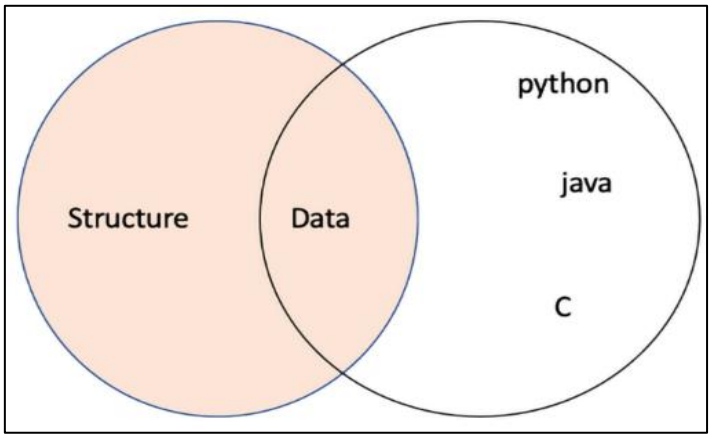




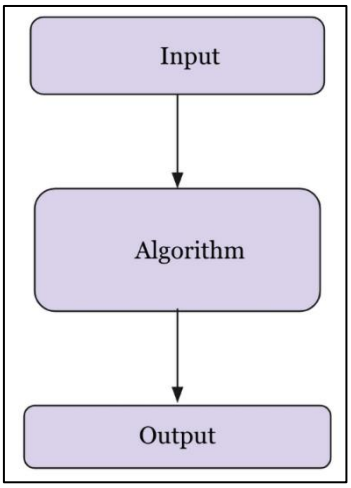


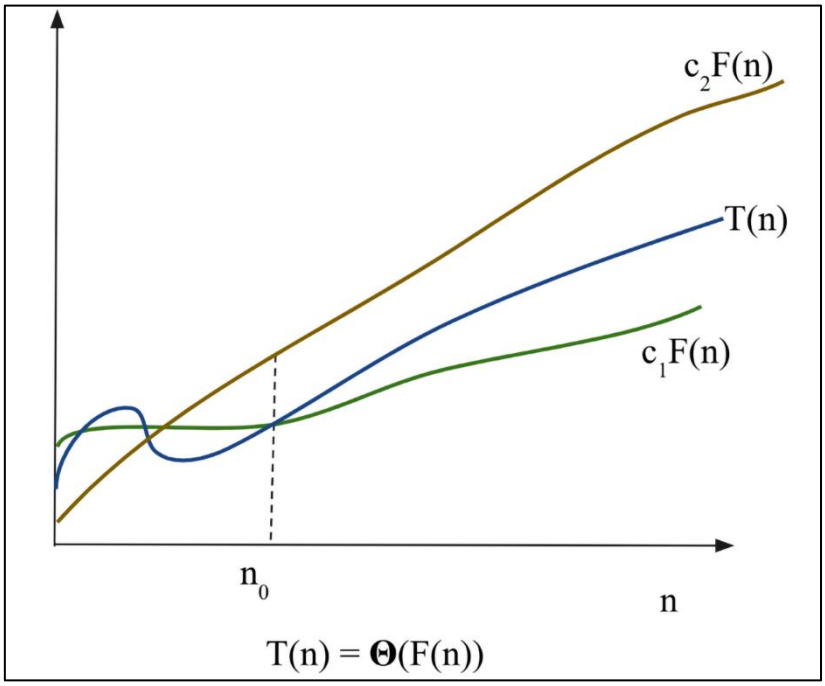


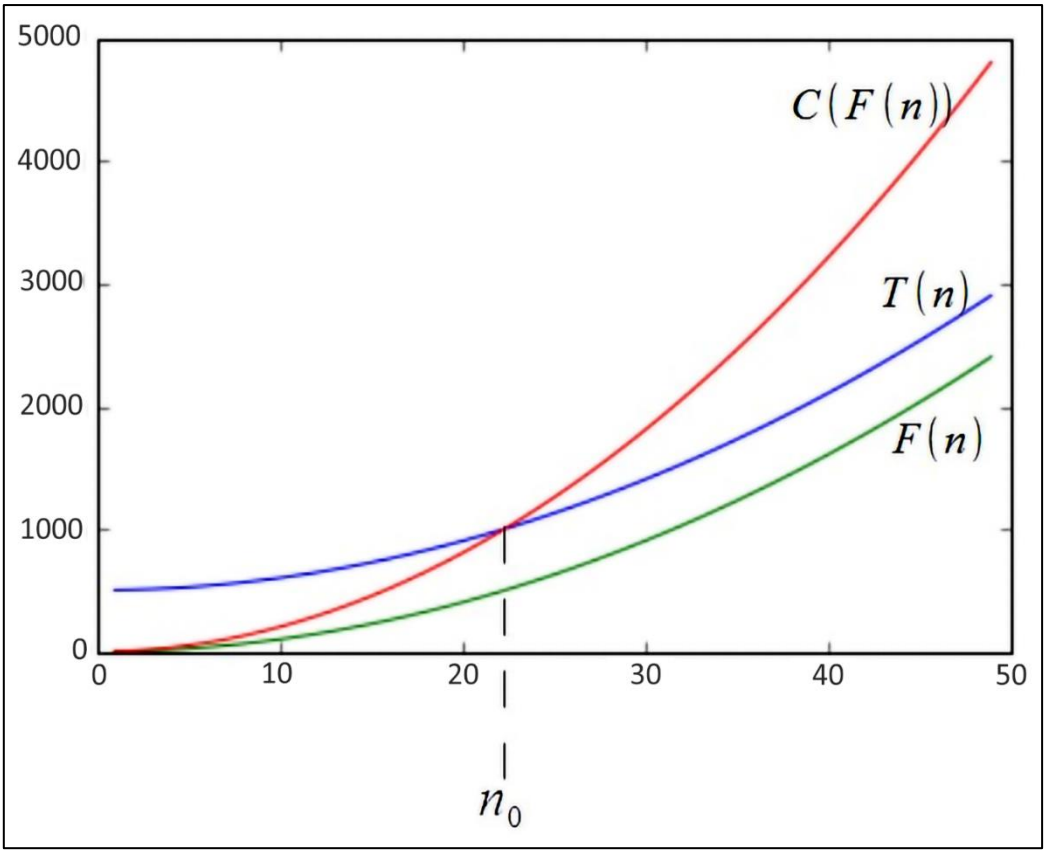

Key
Values
1
data
2
structure
3
python
4
programming
5
language
my_dict dictionary

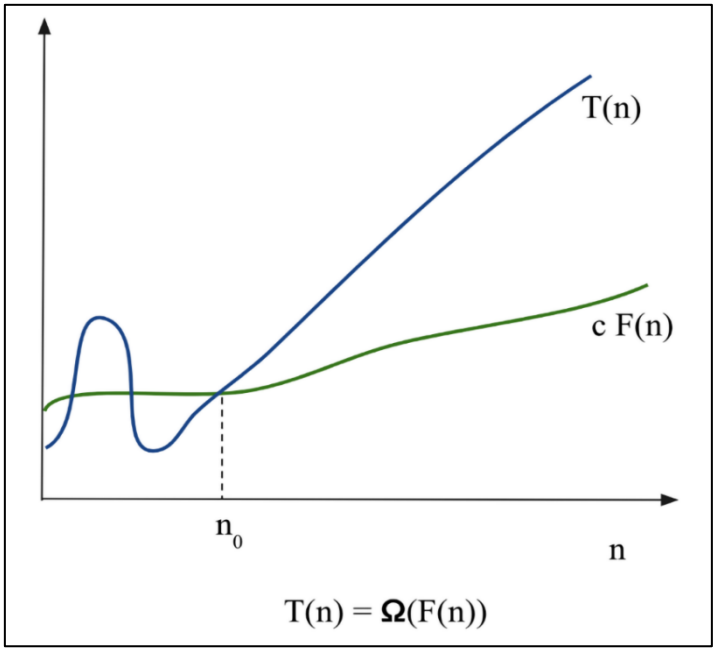

Structure
Data
python
java
C

Chapter 2: Introduction to Algorithm Design

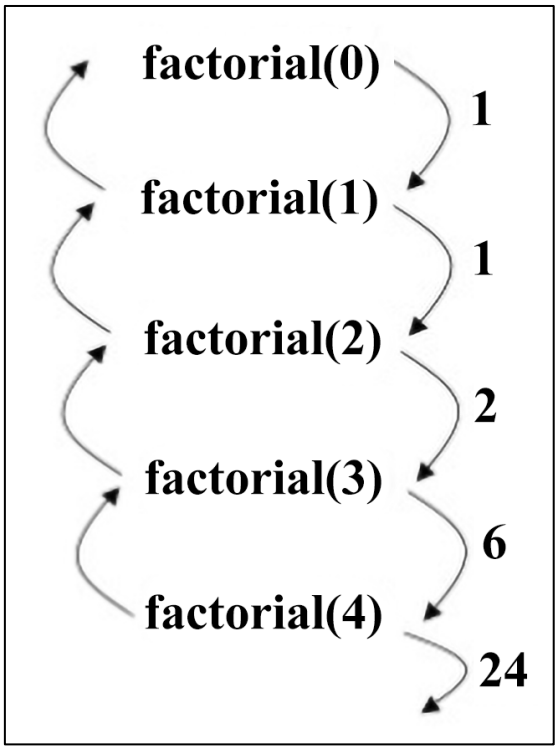


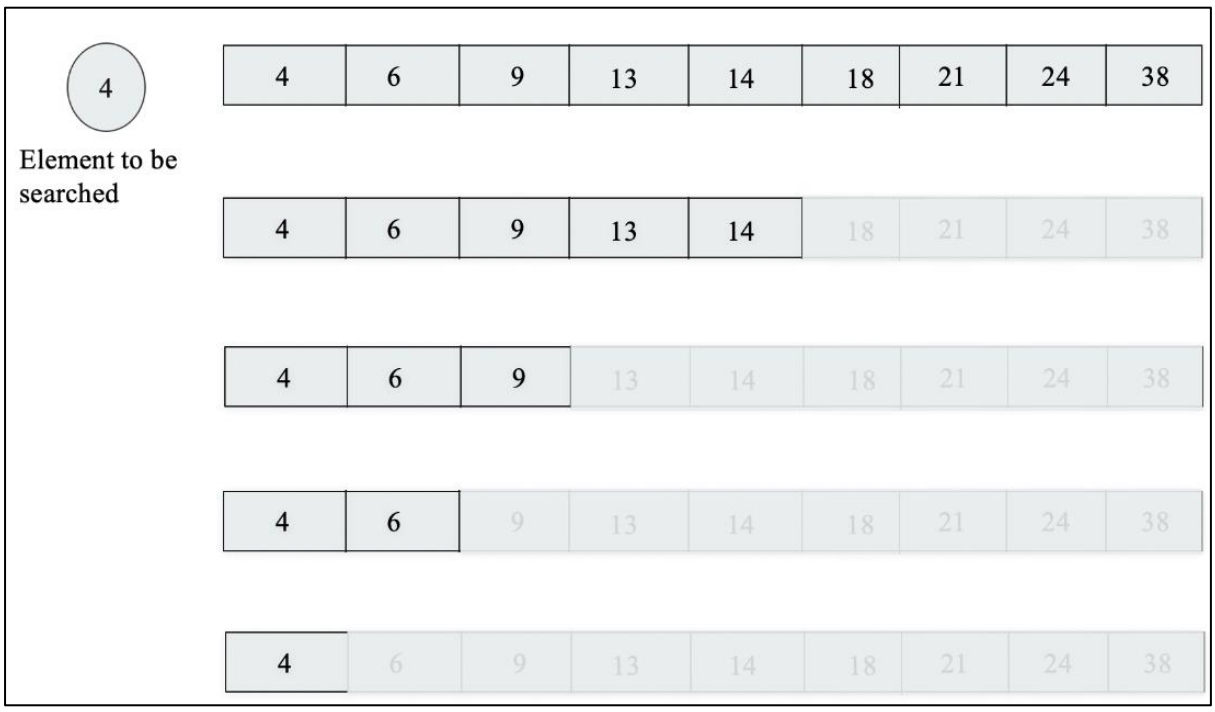


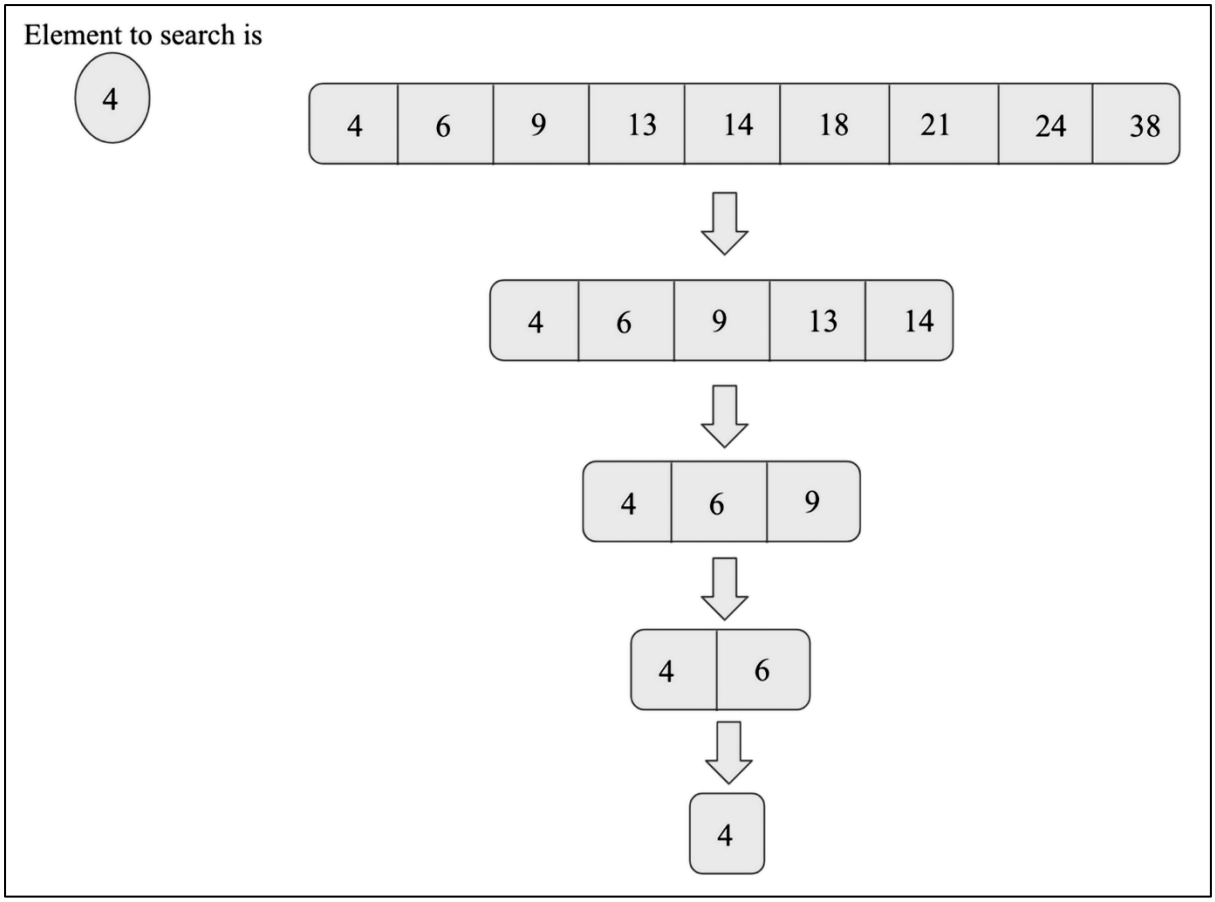

5000
4000
3000
2000
1000
0
0
10
20
30
40
50
C(F(n))
T(n)
F(n)
n0

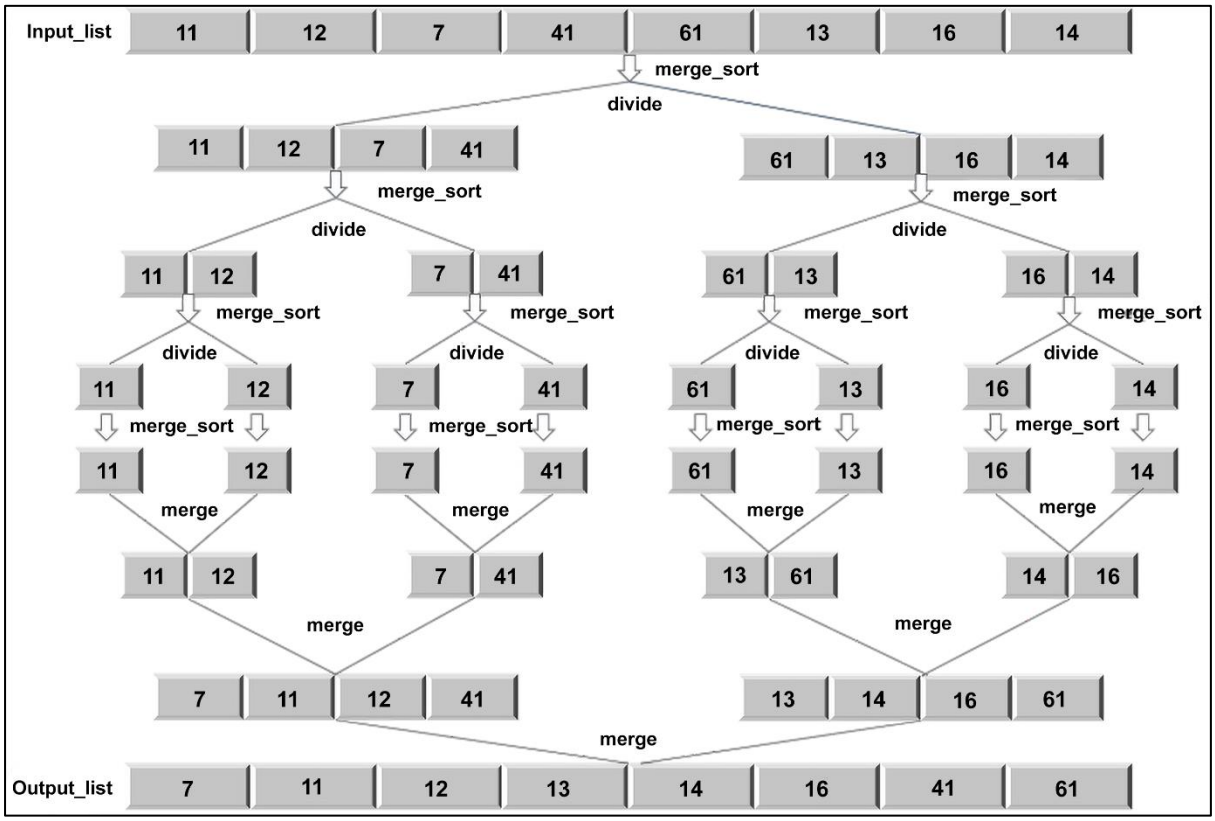

T(n)
c F(n)
n0
n
T(n) = Ω(F(n))

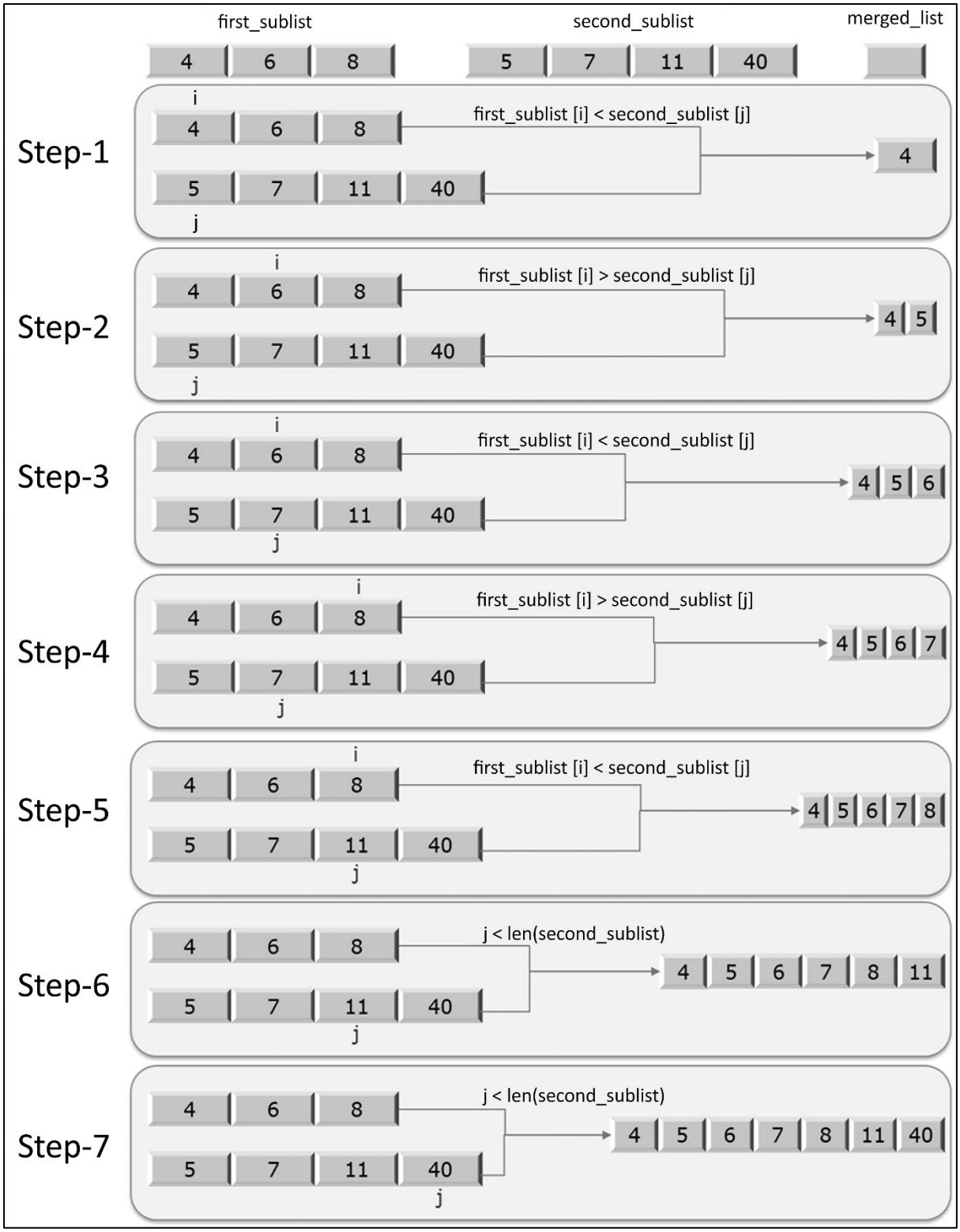
Chapter 3: Algorithm Design Techniques and Strategies

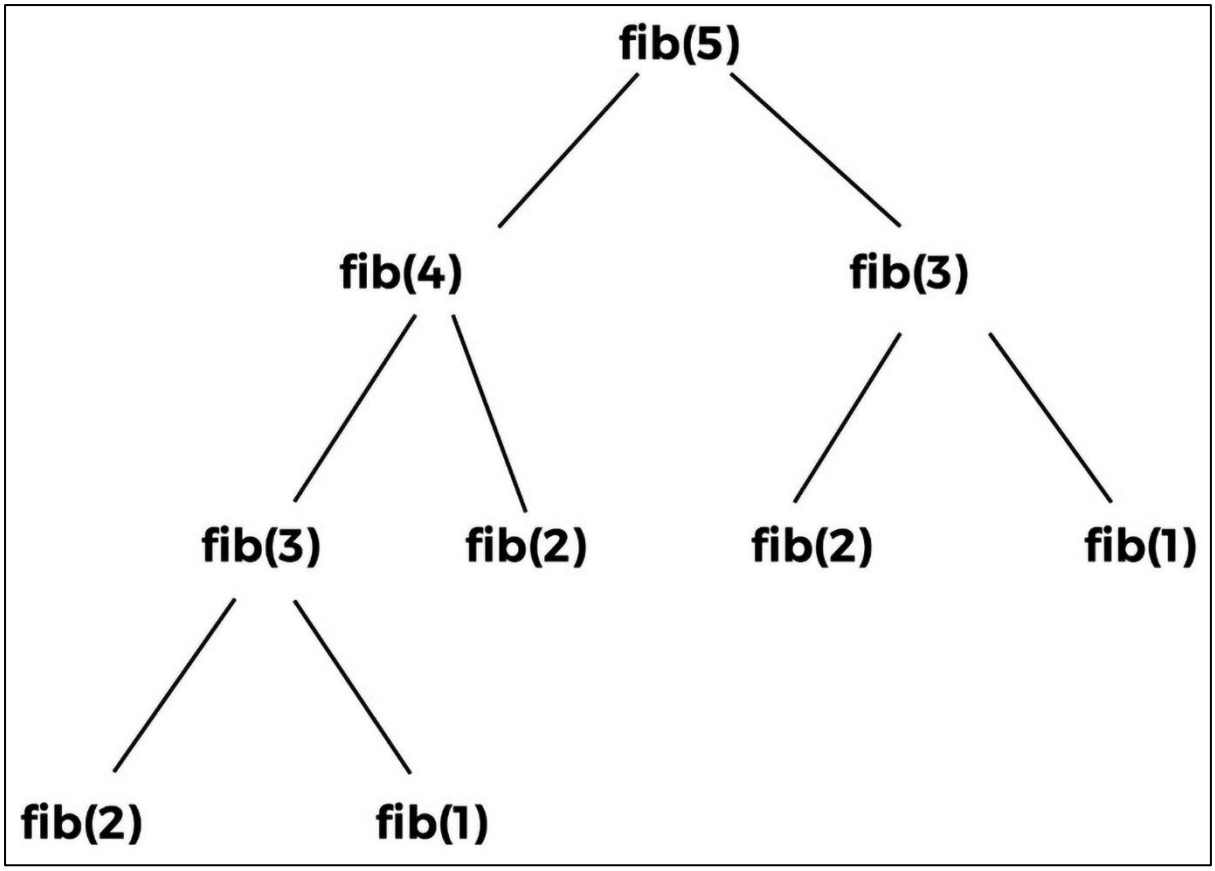


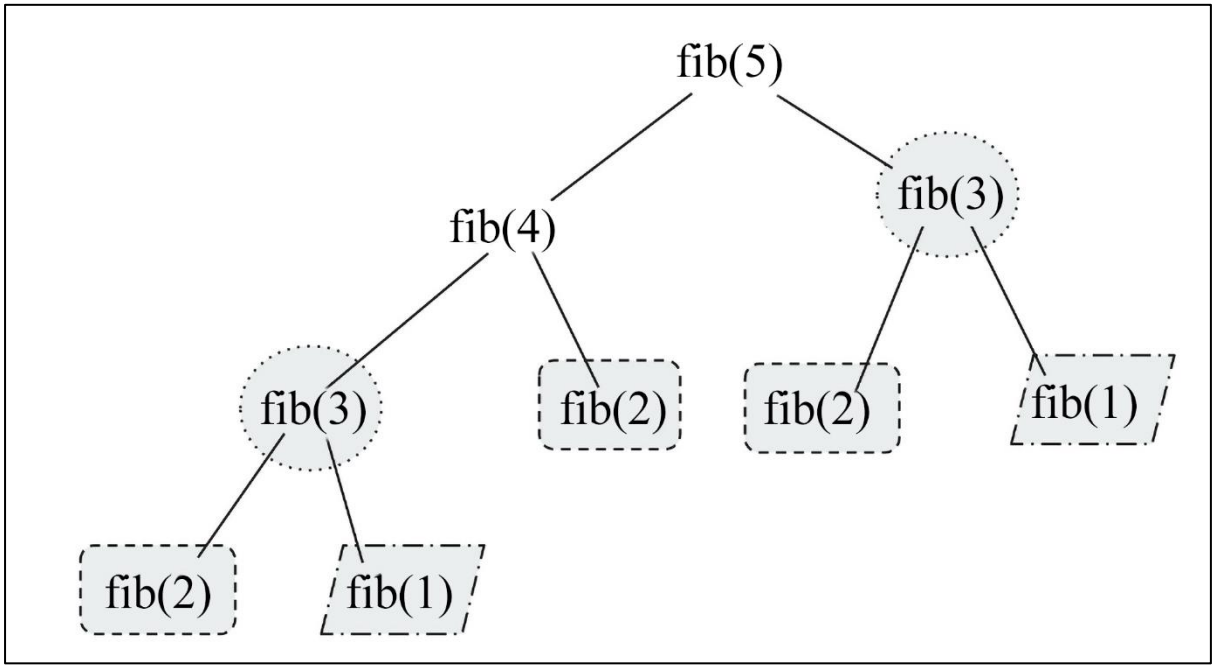


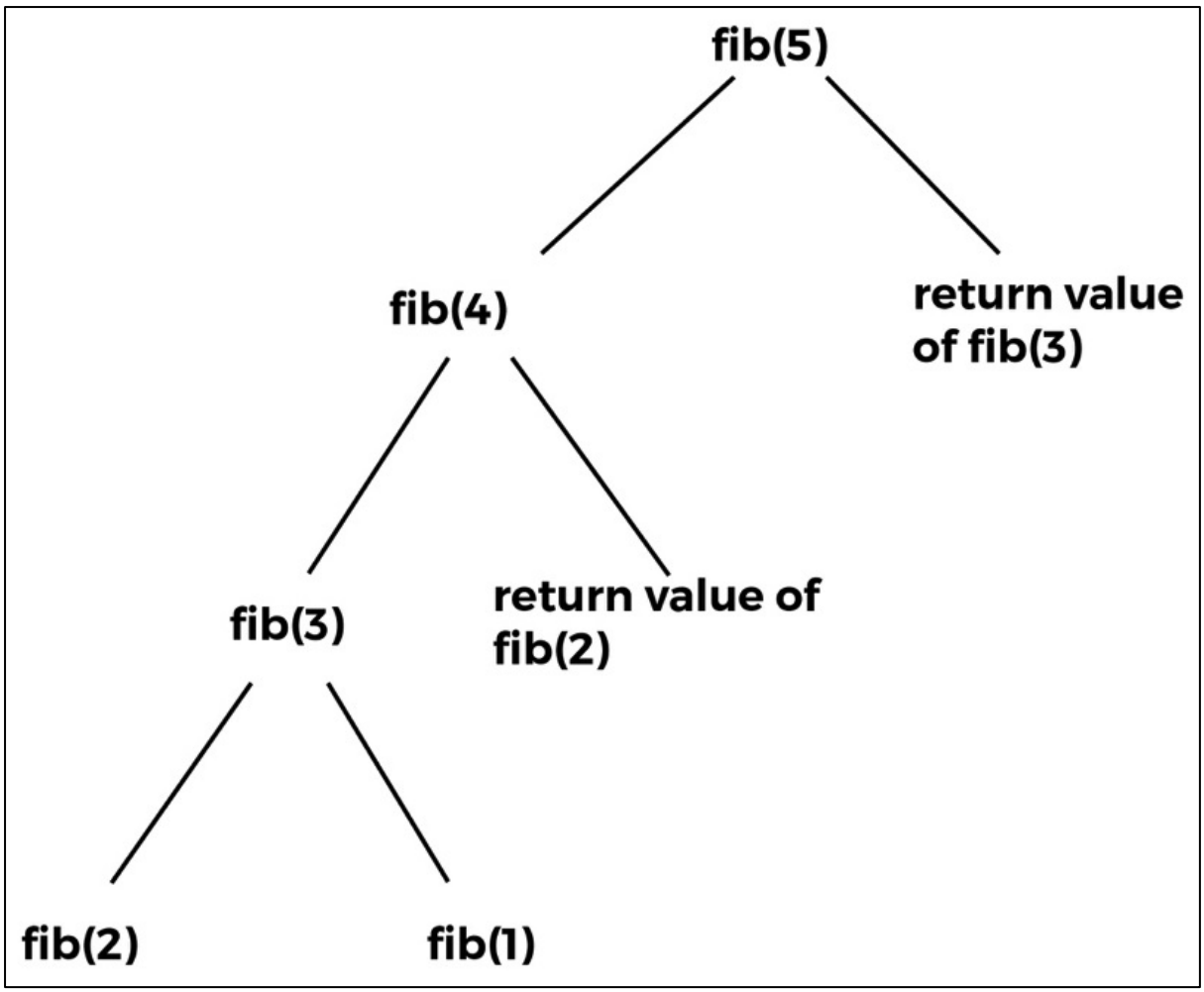

Element to search is
4
4 6 9 13 14 18 21 24 38
4 6 9 13 14
4 6 9
4 6
4

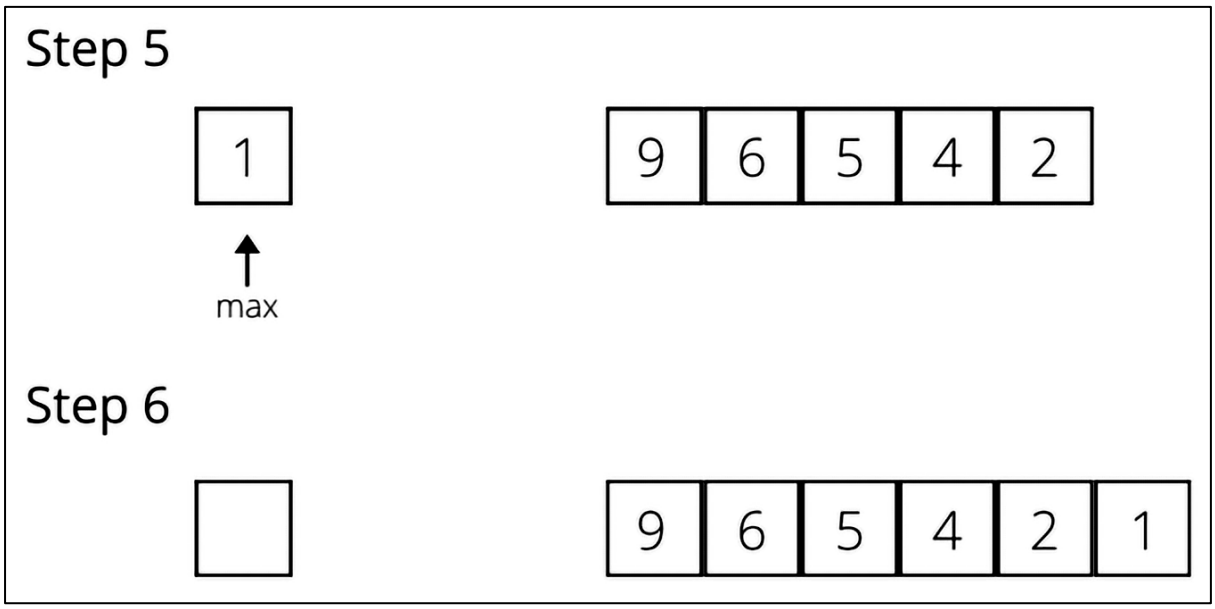

Input_list
11 12 7 41 61 13 16 14
merge_sort
divide
11 12 7 41
61 13 16 14
merge_sort
divide
11 12
7 41
61 13
16 14
merge_sort
divide
11
12
7
41
61
13
16
14
merge
11 12
7 41
13 61
14 16
merge
7 11 12 41
13 14 16 61
merge
Output_list
7 11 12 13 14 16 41 61

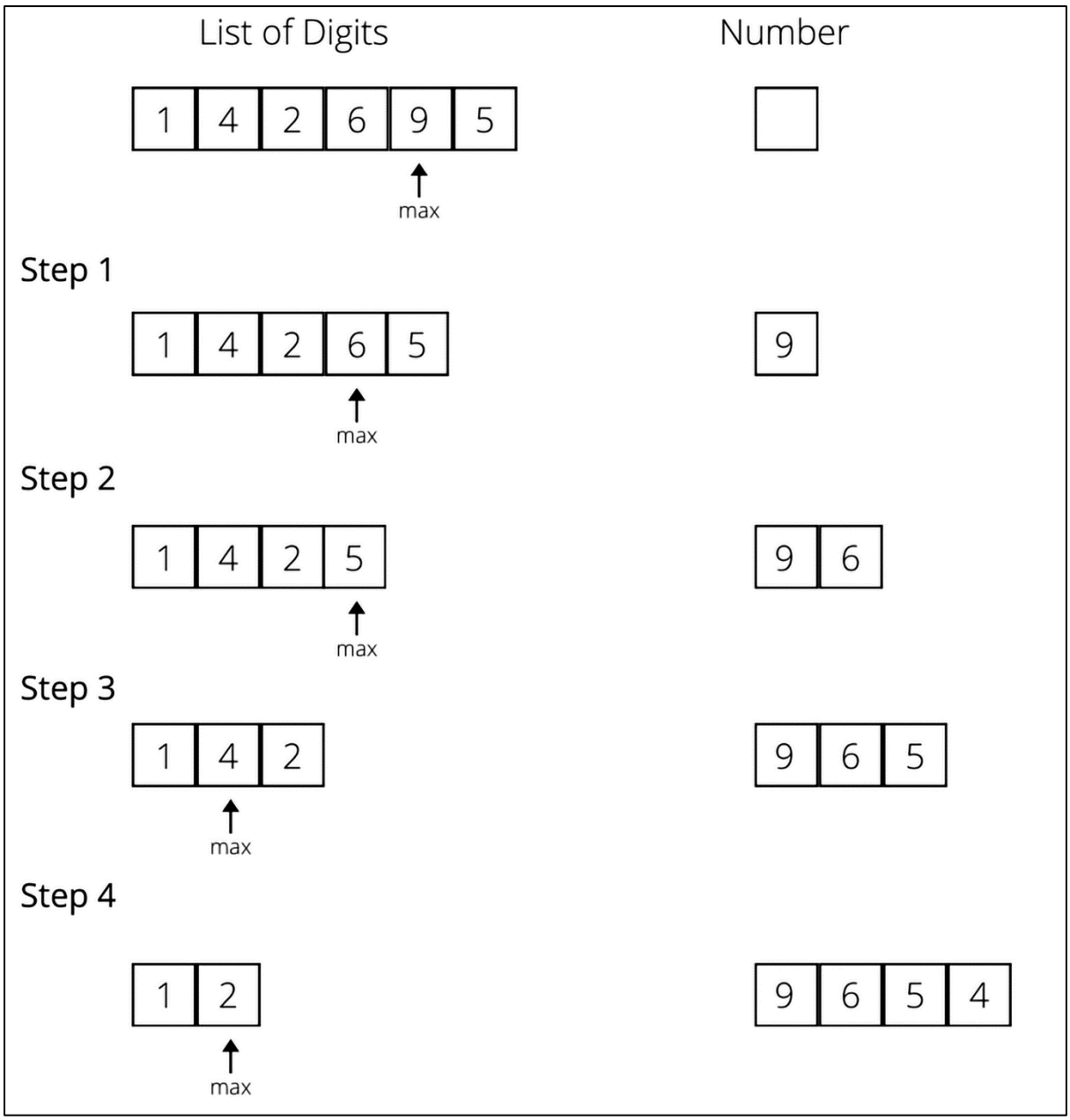

first_sublist
second_sublist
merged_list
4 6 8
5 7 11 40
Step-1
i
j
first_sublist [i] < second_sublist [j]
4
Step-2
first_sublist [i] > second_sublist [j]
4 5
Step-3
first_sublist [i] < second_sublist [j]
4 5 6
Step-4
first_sublist [i] > second_sublist [j]
4 5 6 7
Step-5
first_sublist [i] < second_sublist [j]
4 5 6 7 8
Step-6
j < len(second_sublist)
4 5 6 7 8 11
Step-7
j < len(second_sublist)
4 5 6 7 8 11 40

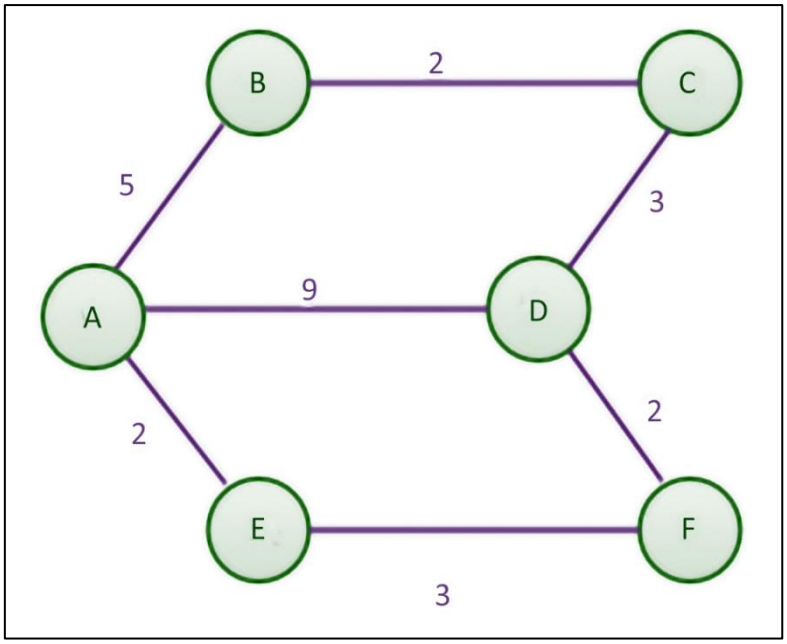

fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)

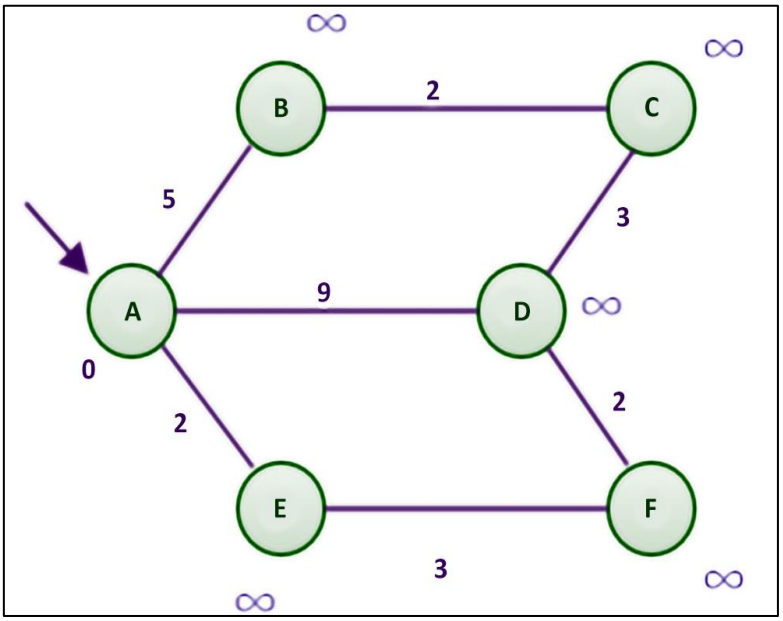

fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)

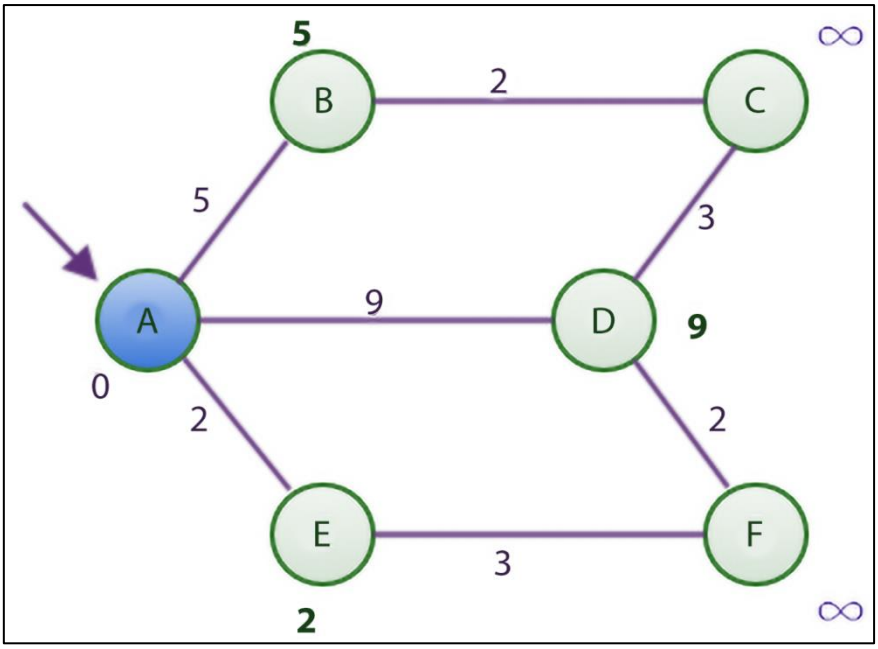

fib(5)
fib(4)
return value of fib(3)
fib(3)
return value of fib(2)
fib(2)
fib(1)

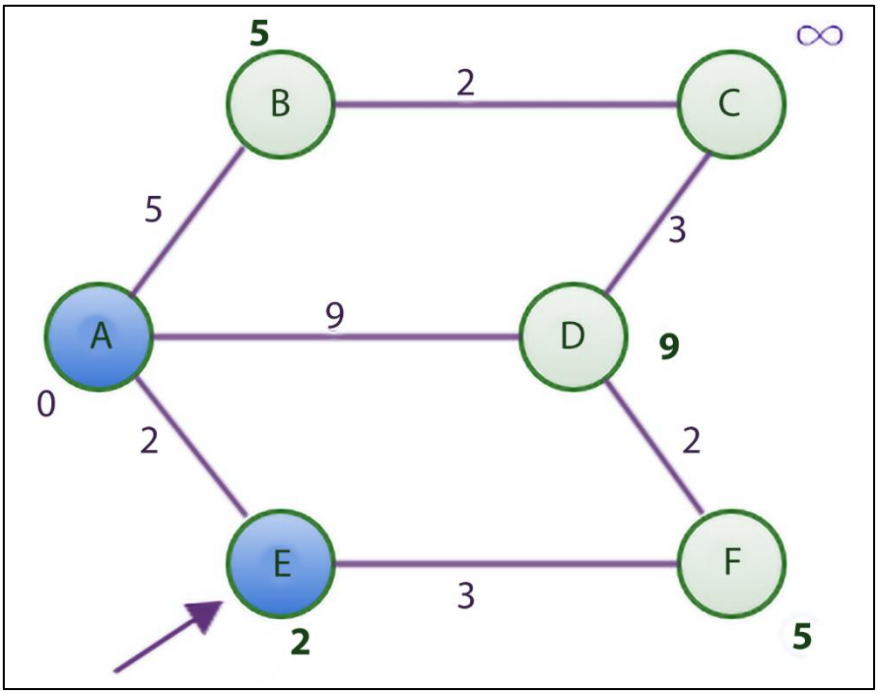

Step 5
1
9 6 5 4 2
max
Step 6
9 6 5 4 2 1

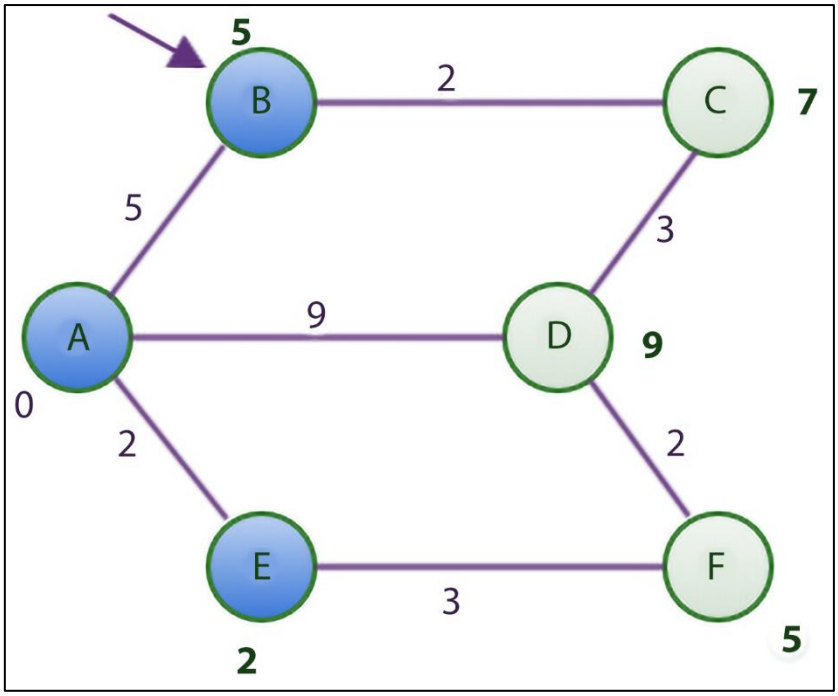

List of Digits
Number
1 4 2 6 9 5
max
Step 1
1 4 2 6 5
9
max
Step 2
1 4 2 5
9 6
max
Step 3
1 4 2
9 6 5
max
Step 4
1 2
9 6 5 4
max

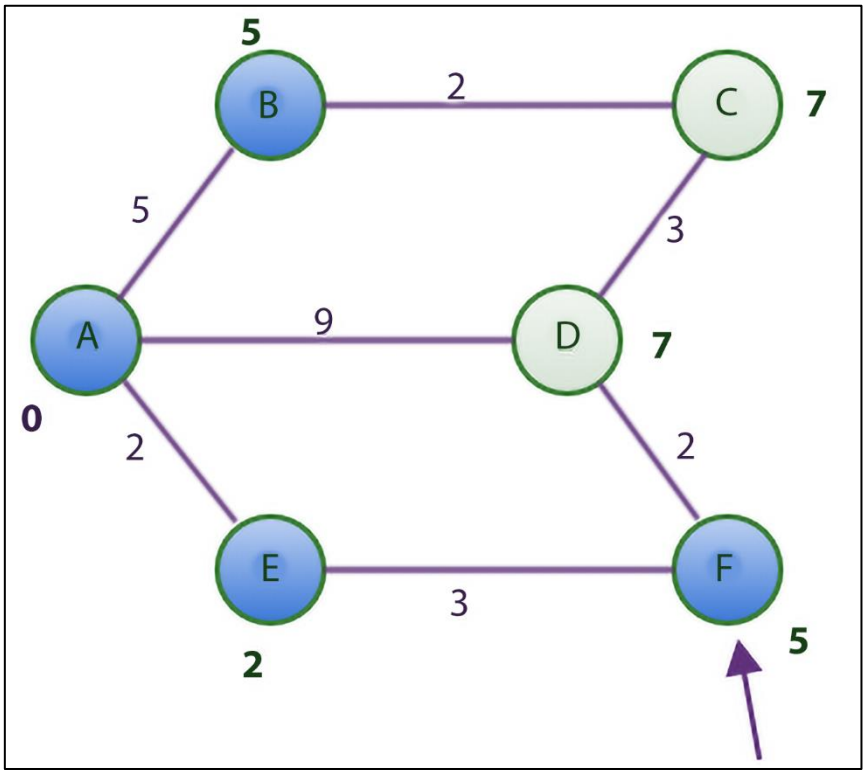

B
2
C
5
3
A
9
D
2
2
E
F
3

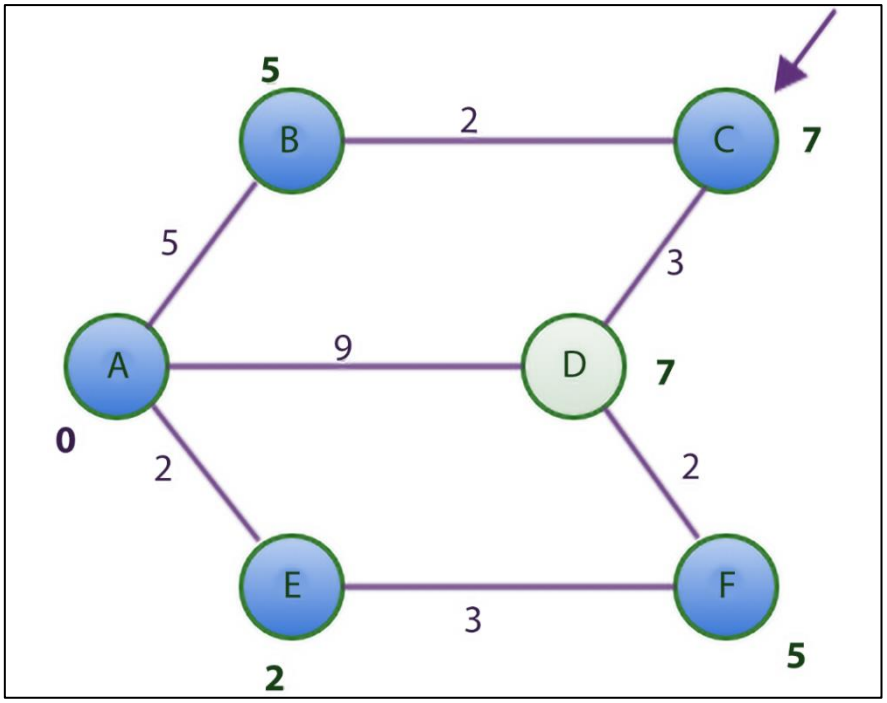

∞
∞
B
2
C
5
3
A
9
D
∞
0
2
2
E
F
3
∞
∞

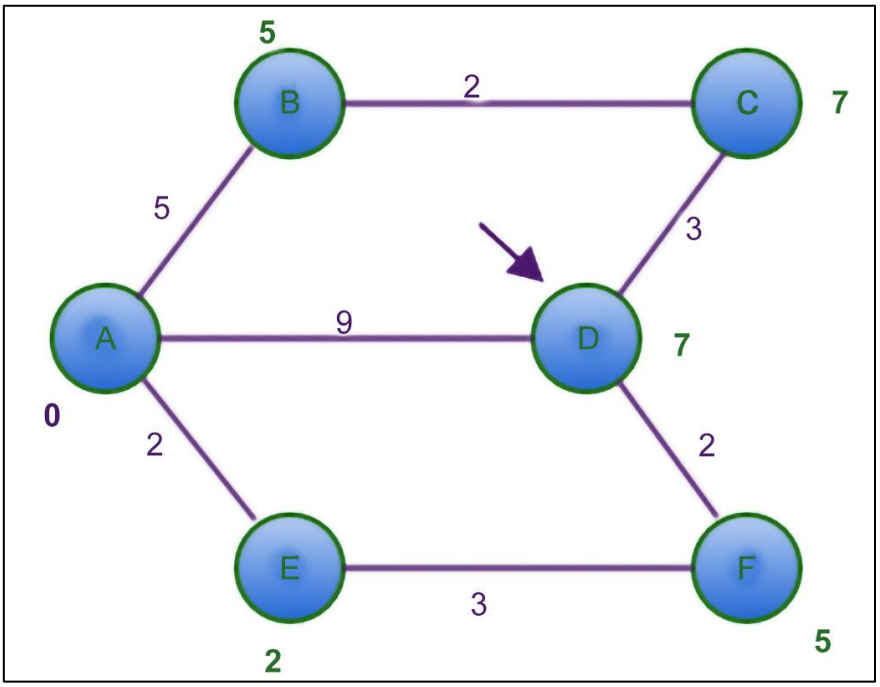

5
∞
B
2
C
5
3
A
9
D
9
0
2
2
E
F
3
2
∞

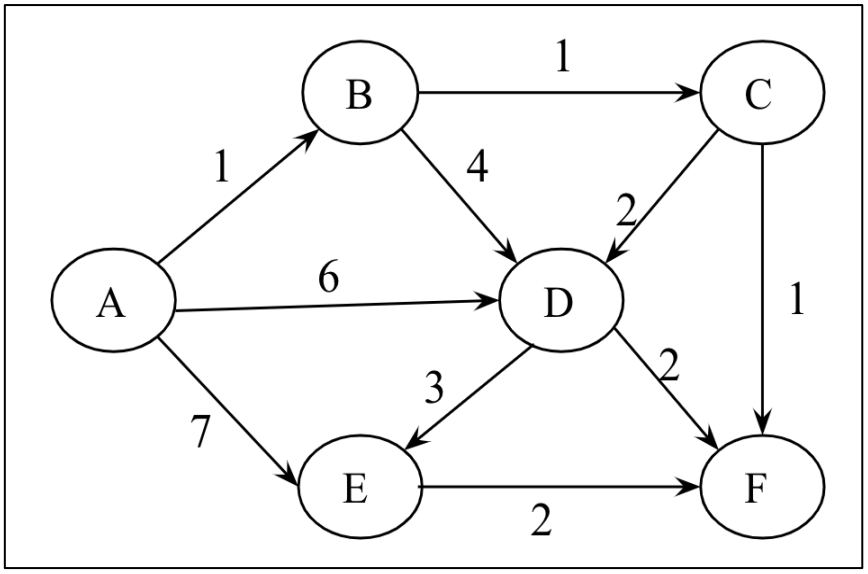

5
∞
B
2
C
5
3
A
9
D
9
0
2
2
E
F
3
2
5


5
B
2
C
7
5
3
A
9
D
9
0
2
2
E
F
3
2
5

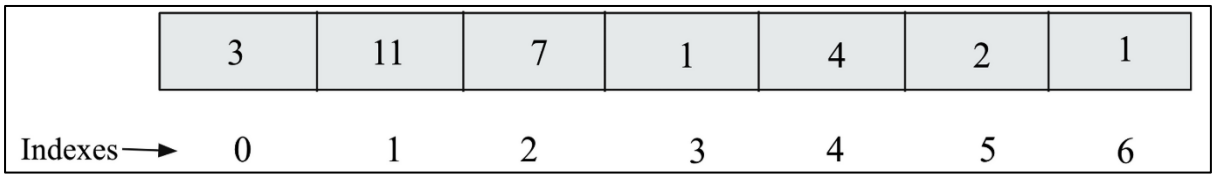

5
B
2
C
7
5
3
A
9
D
7
0
2
2
E
F
3
2
5

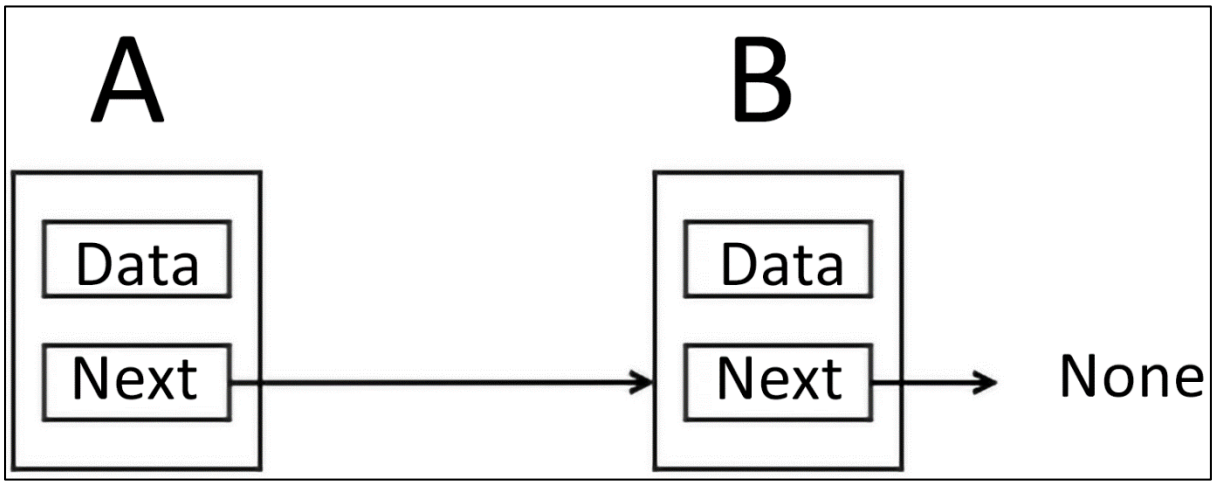

5
B
2
C
7
5
3
A
9
D
7
0
2
2
E
F
3
2
5

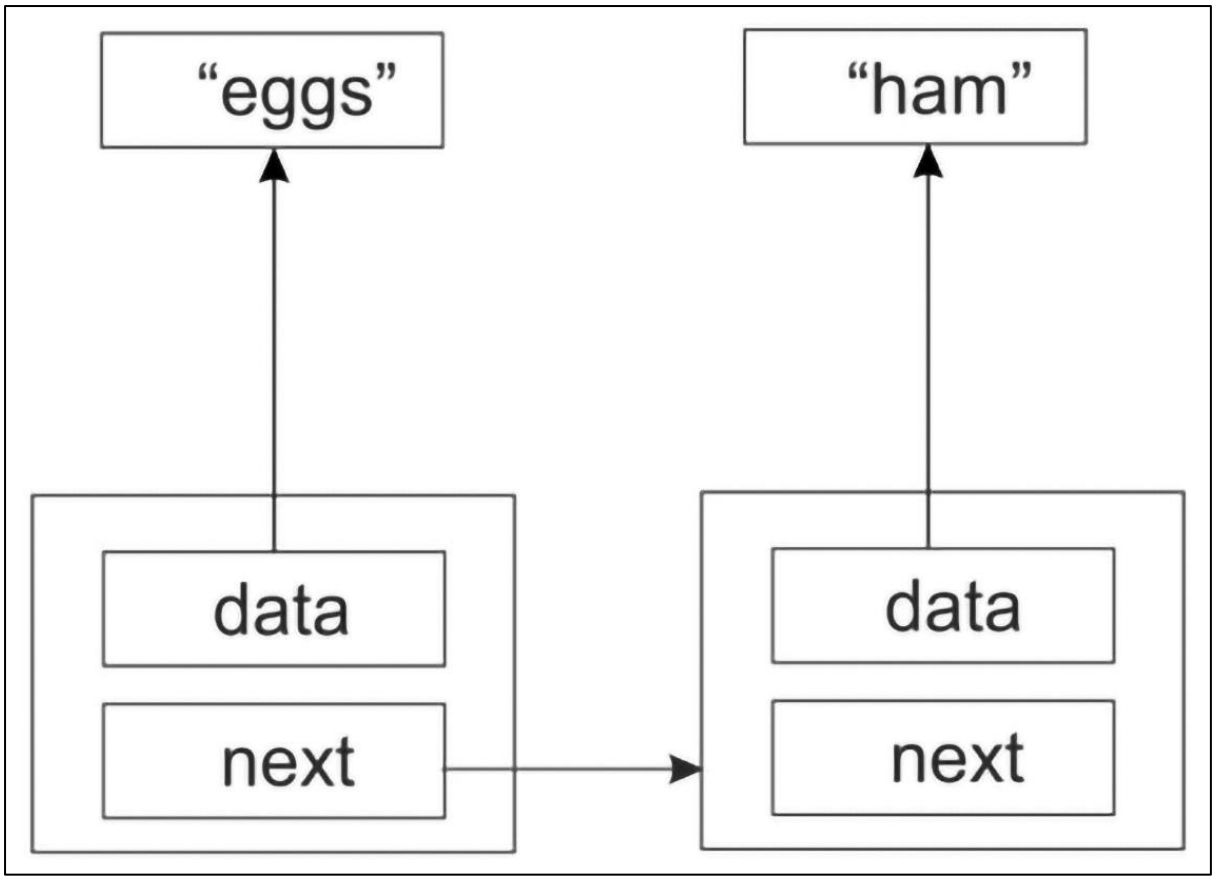

5
B
2
C
7
5
3
A
9
D
7
0
2
2
E
F
3
2
5

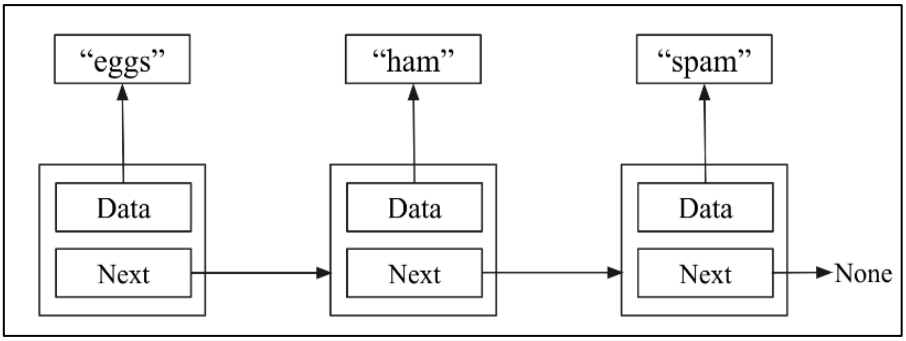

B
C
A
D
E
F
1
1
4
2
6
1
3
2
7
2

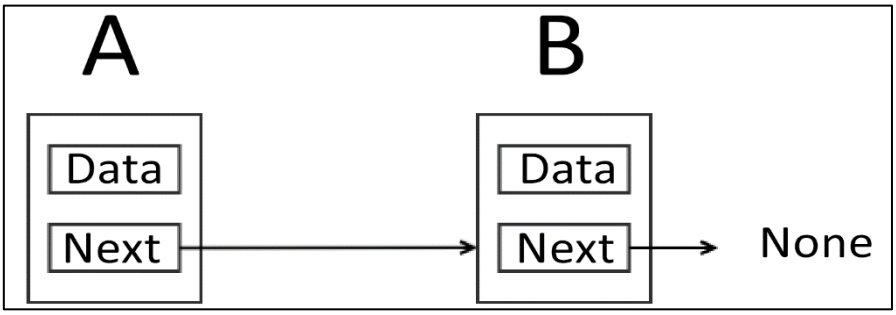
Chapter 4: Linked Lists

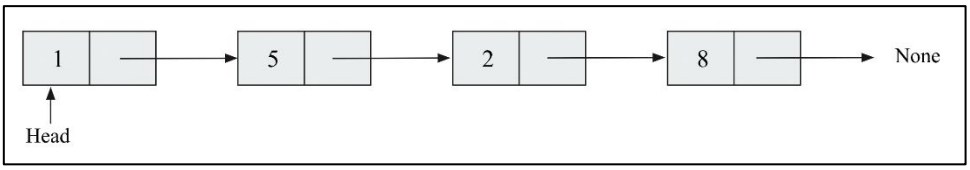


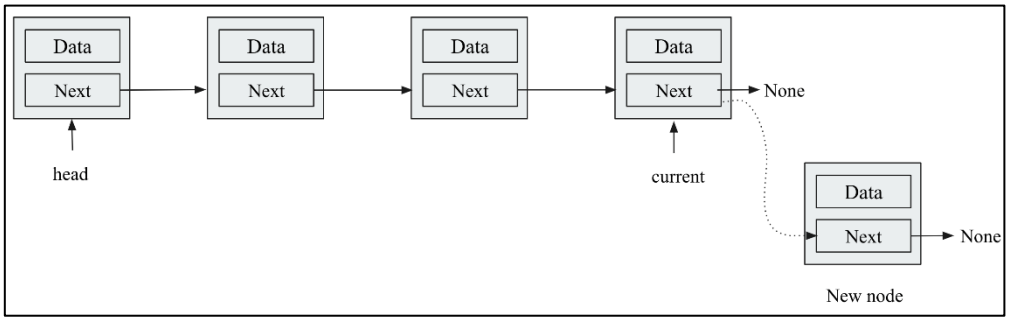


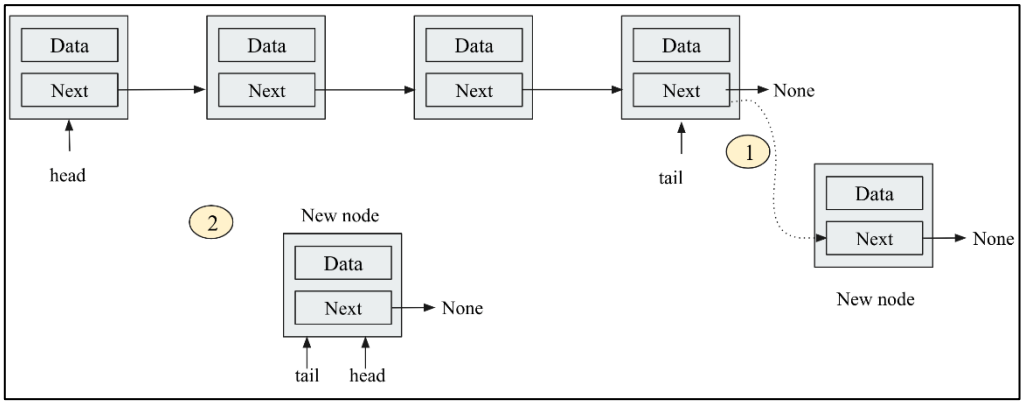


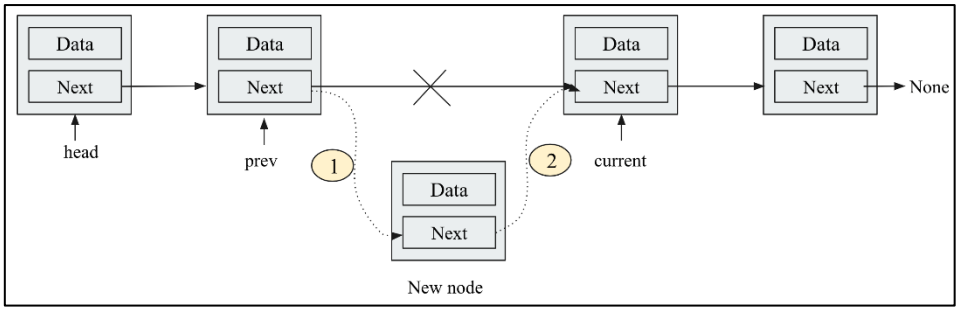

“eggs”
“ham”
“spam”
Data
Next
Data
Next
Data
Next
None

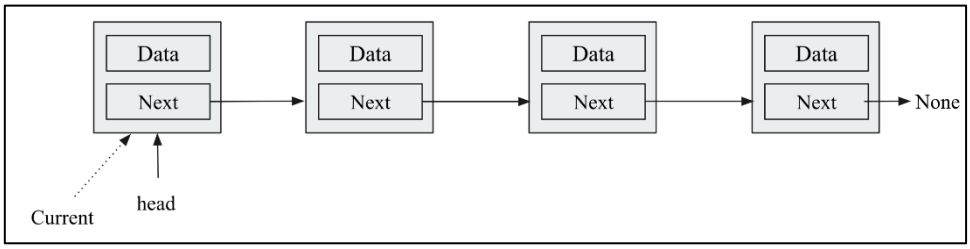

A
B
Data
Next
Data
Next
None

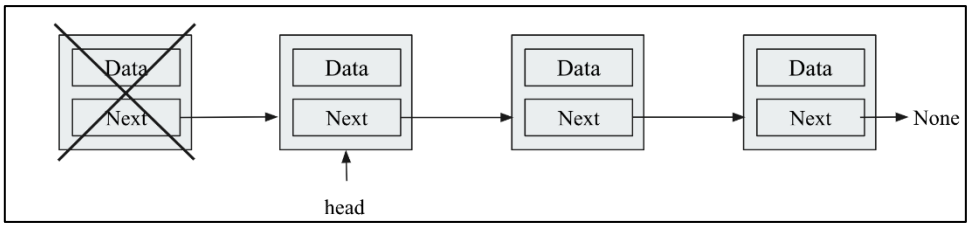

1
5
2
8
None
Head

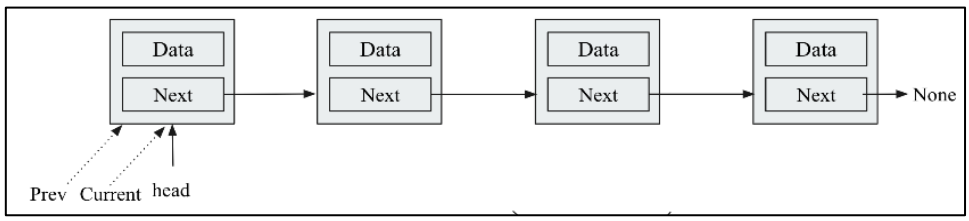

Data
Next
head
current
None
New node

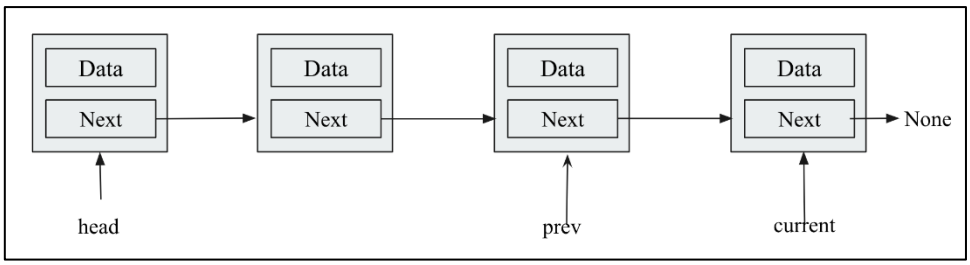

head
tail
None
1
2
New node
tail
head
New node

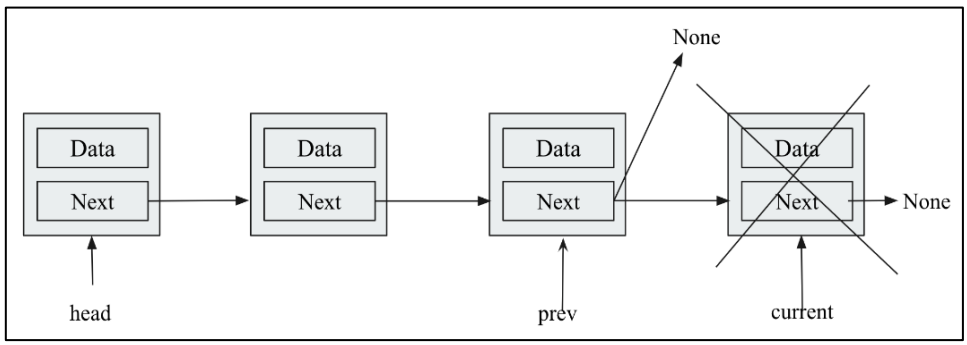

Data
Next
head
prev
1
2
current
New node
None

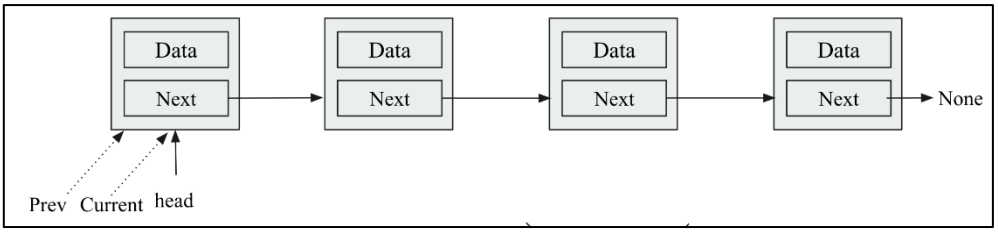

Data
Next
Current
head
None

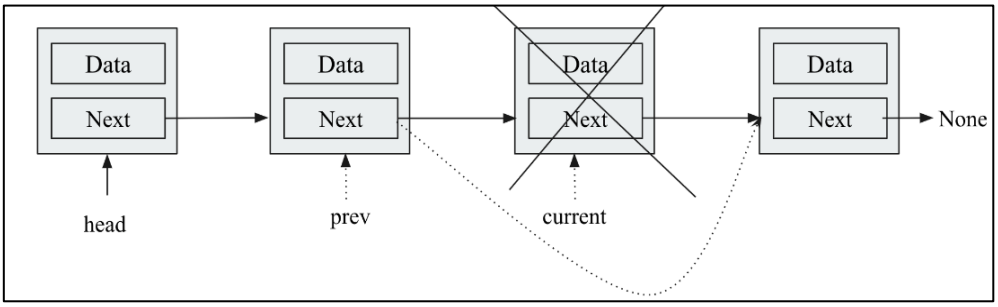

Data
Next
head
None

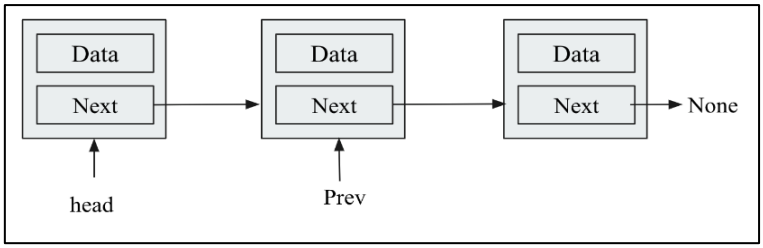

Data
Next
Prev
Current
head
None

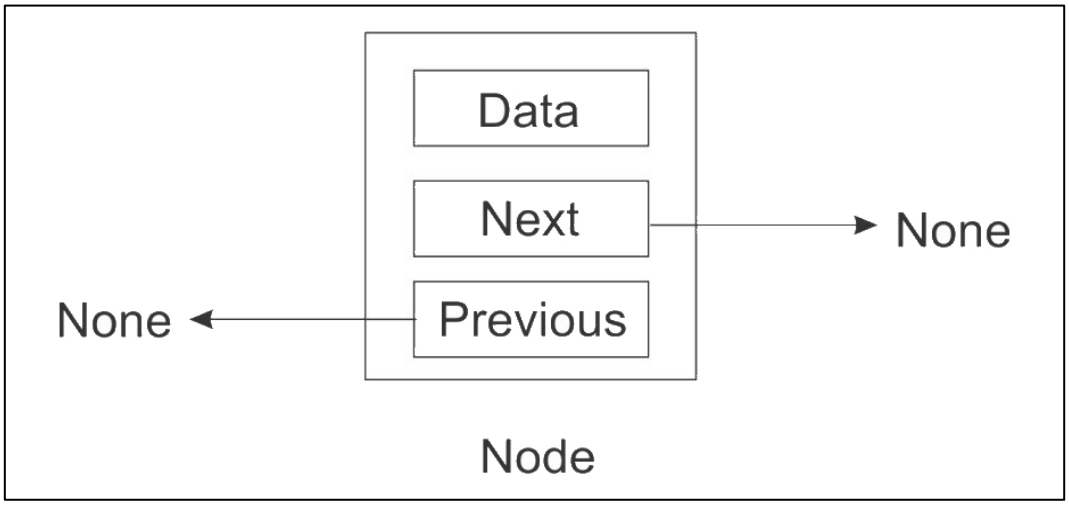

Data
Next
head
prev
current
None

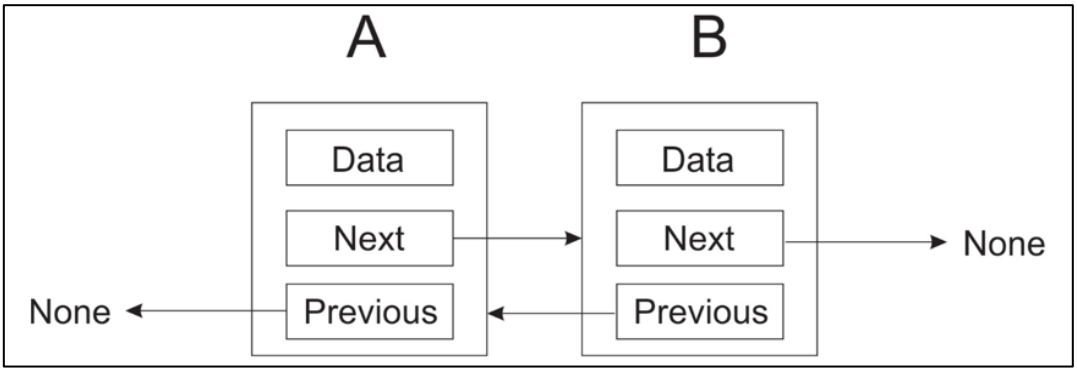

None
Data
Next
head
prev
current
None

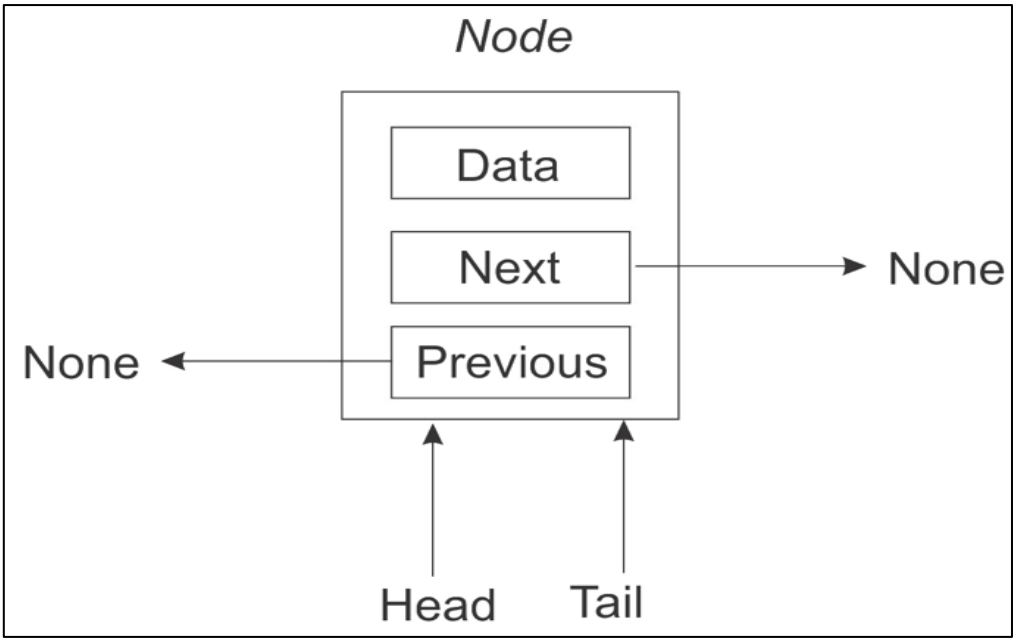

Data
Next
Data
Next
Data
Next
Data
Next
None
Prev
Current
head

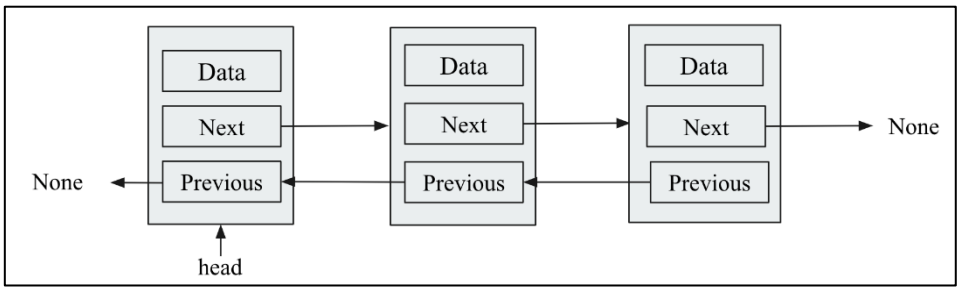

Data
Next
Data
Next
Data
Next
Data
Next
None
head
prev
current

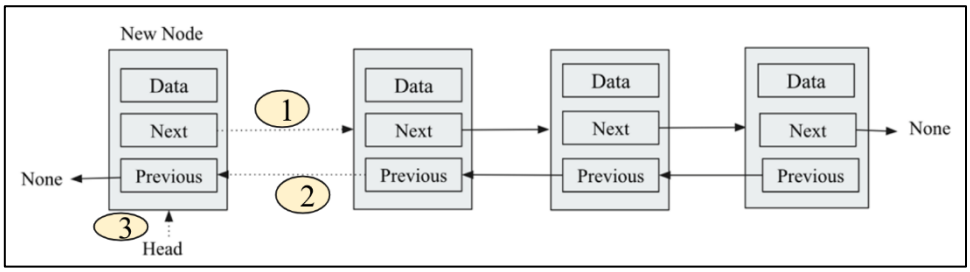

Data
Next
Data
Next
Data
Next
None
head
Prev

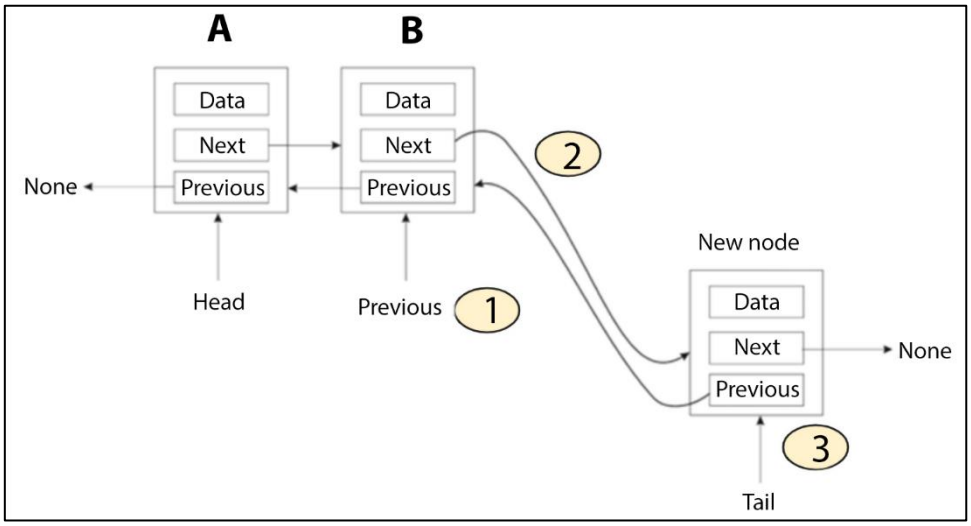

Data
Next
Previous
None
None
Node

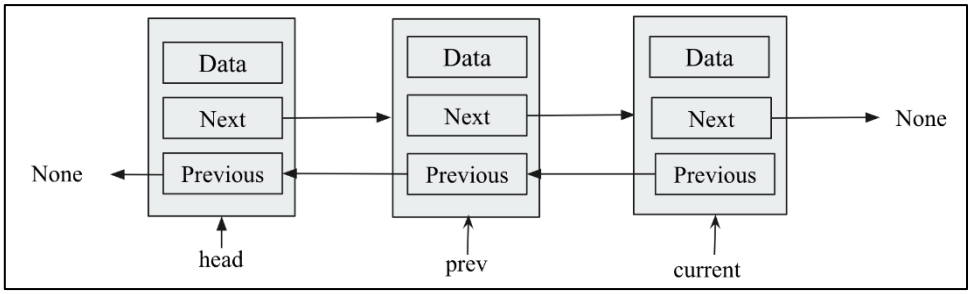

A
B
Data
Next
Previous
Data
Next
Previous
None
None

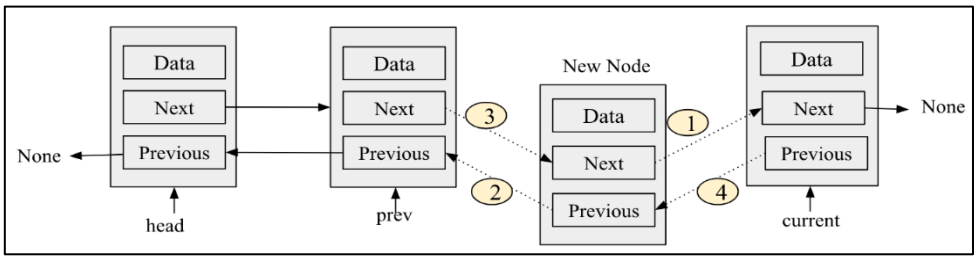

Node
Data
Next
Previous
None
None
Head
Tail

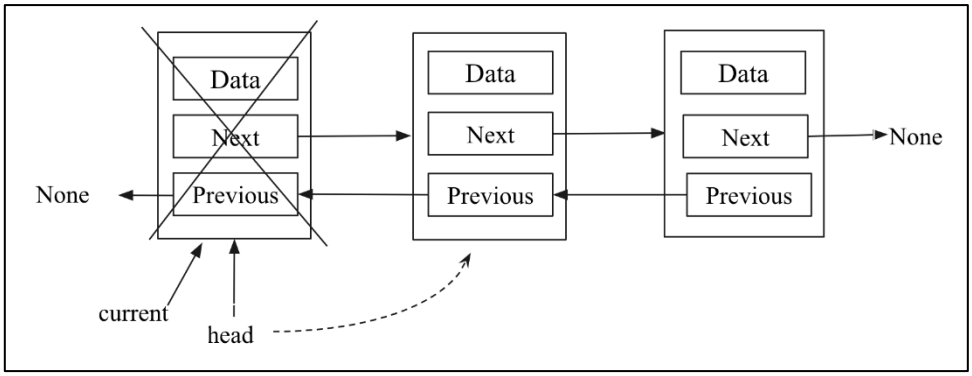

Data
Next
Previous
Data
Next
Previous
Data
Next
Previous
None
None
head

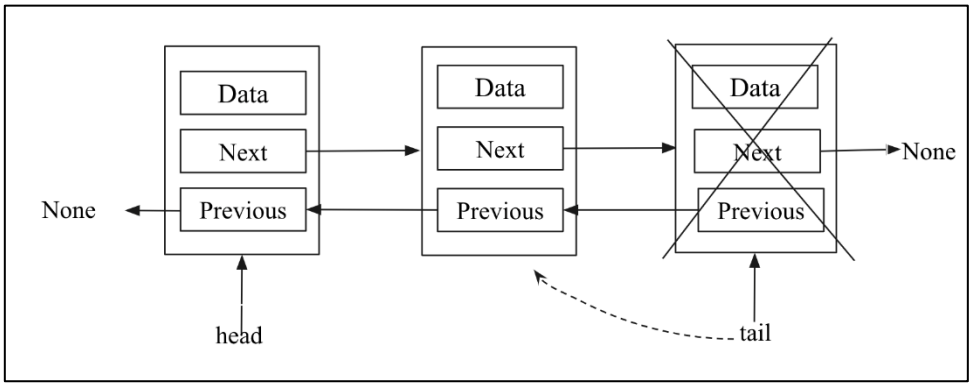

New Node
Data
Next
Previous
1
2
3
Head
Data
Next
Previous
Data
Next
Previous
Data
Next
Previous
None
None

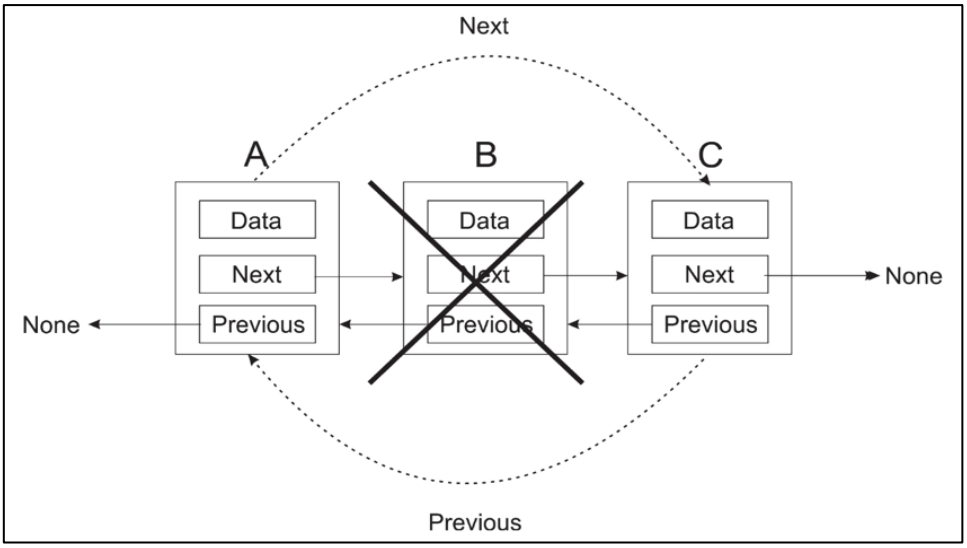

A
B
Data
Next
Previous
Data
Next
Previous
2
None
Head
Previous
1
New node
Data
Next
Previous
None
3
Tail

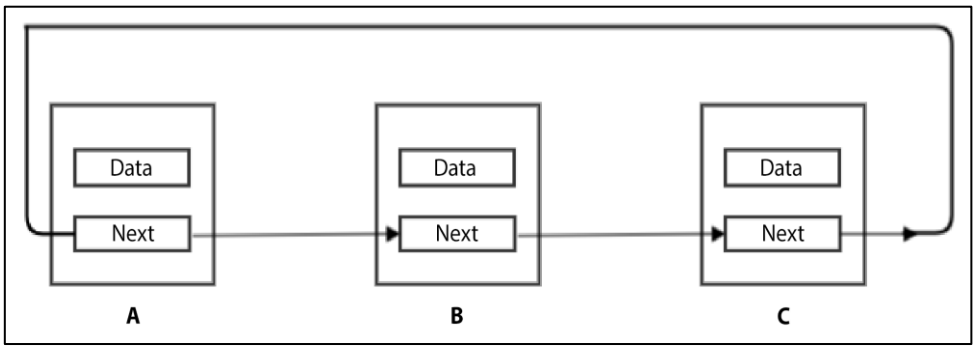

Data
Next
Previous
Data
Next
Previous
Data
Next
Previous
None
None
head
prev
current

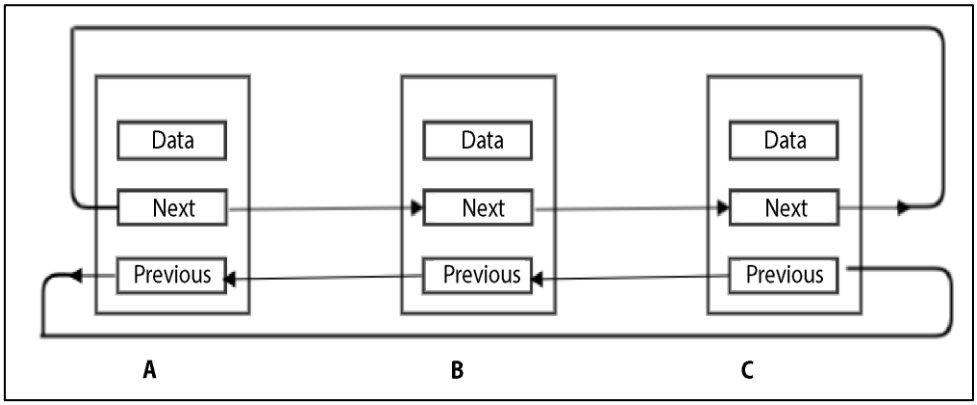

Data
Next
Previous
Data
Next
Previous
New Node
Data
Next
Previous
Data
Next
Previous
None
None
head
prev
current
1
2
3
4

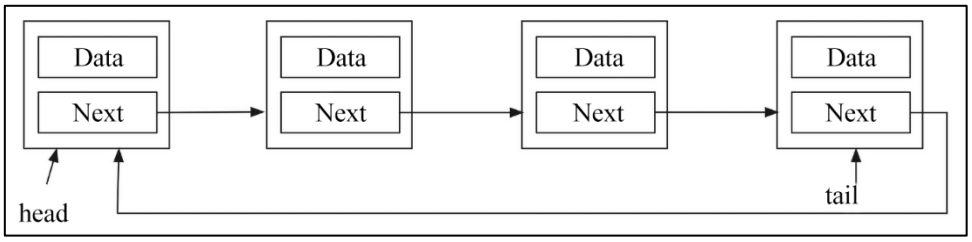

Data
Next
Previous
Data
Next
Previous
Data
Next
Previous
None
None
current
head

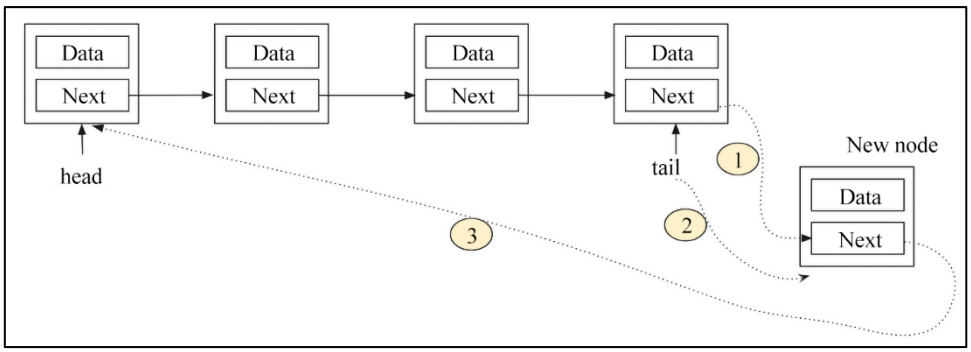

Data
Next
Previous
Data
Next
Previous
Data
Next
Previous
None
None
head
tail

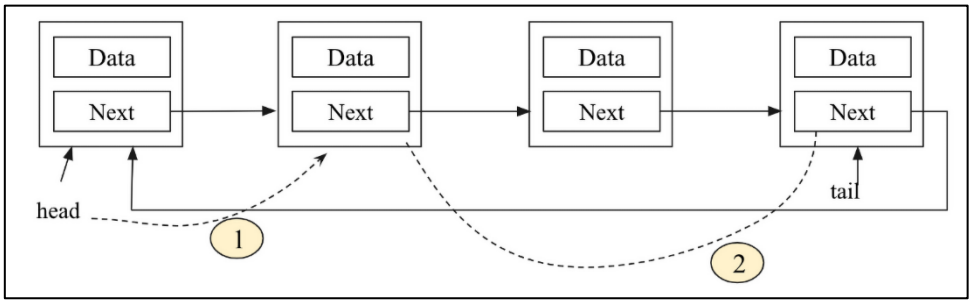

Next
A
B
C
Data
Next
Previous
Data
Next
Previous
Data
Next
Previous
None
None
Previous

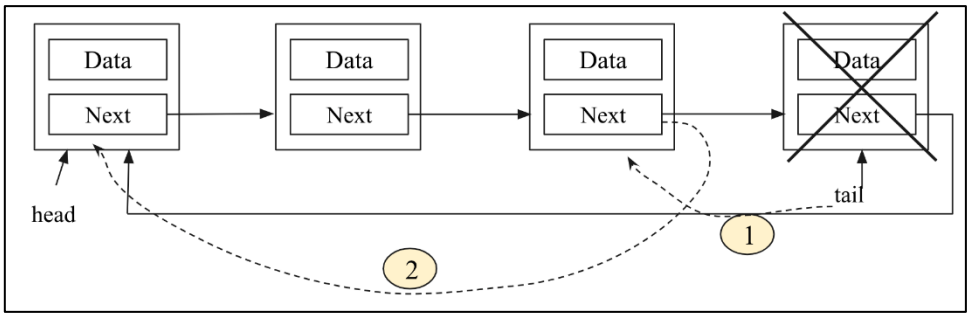

Data
Next
Data
Next
Data
Next
A
B
C

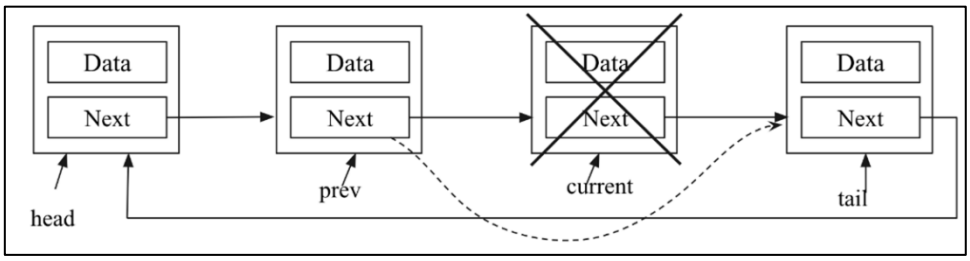

Data
Next
Previous
Data
Next
Previous
Data
Next
Previous
A
B
C


Data
Next
Data
Next
Data
Next
Data
Next
head
tail

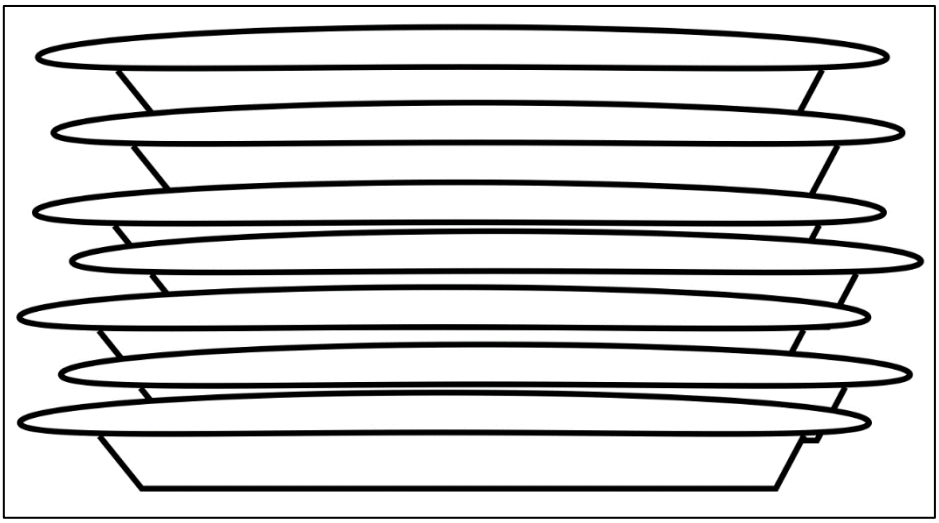

Data
Next
head
tail
New node
1
2
3

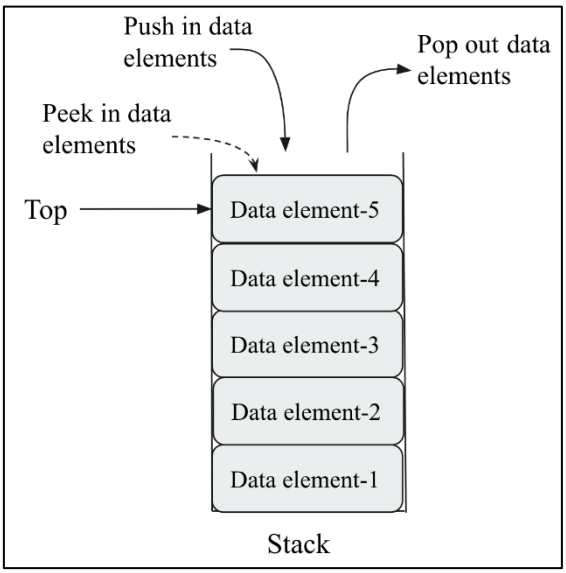

Data
Next
head
tail
1
2

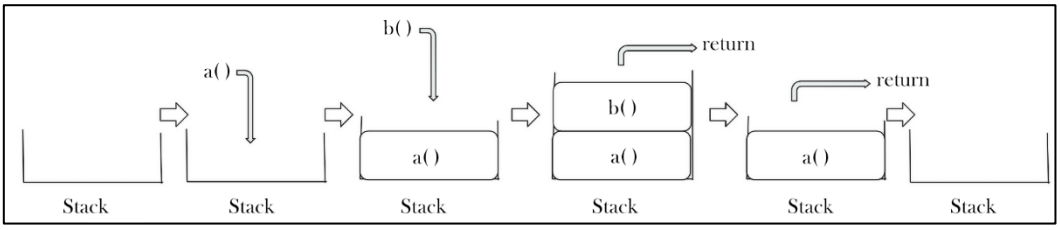

Data
Next
head
tail
1
2

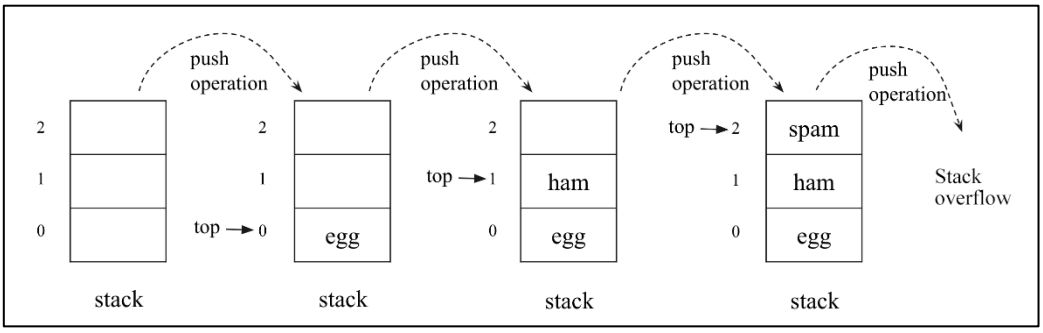

Data
Next
head
prev
current
tail

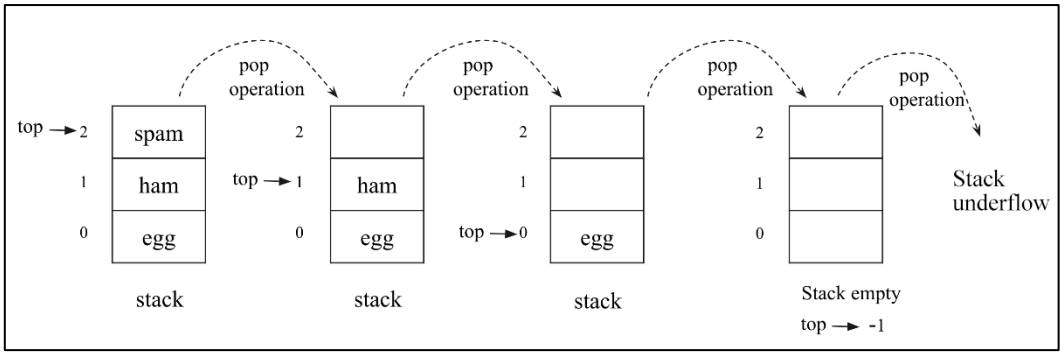
Chapter 5: Stacks and Queues

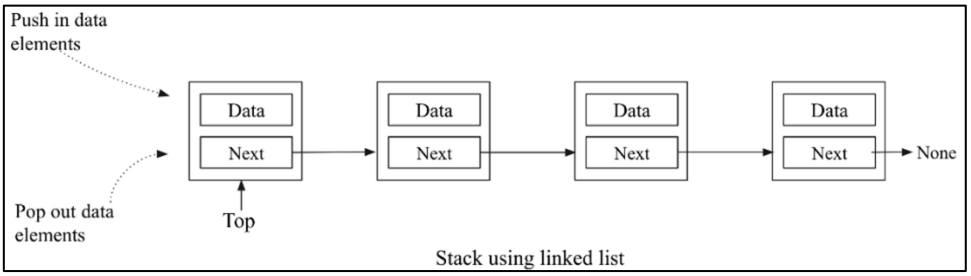


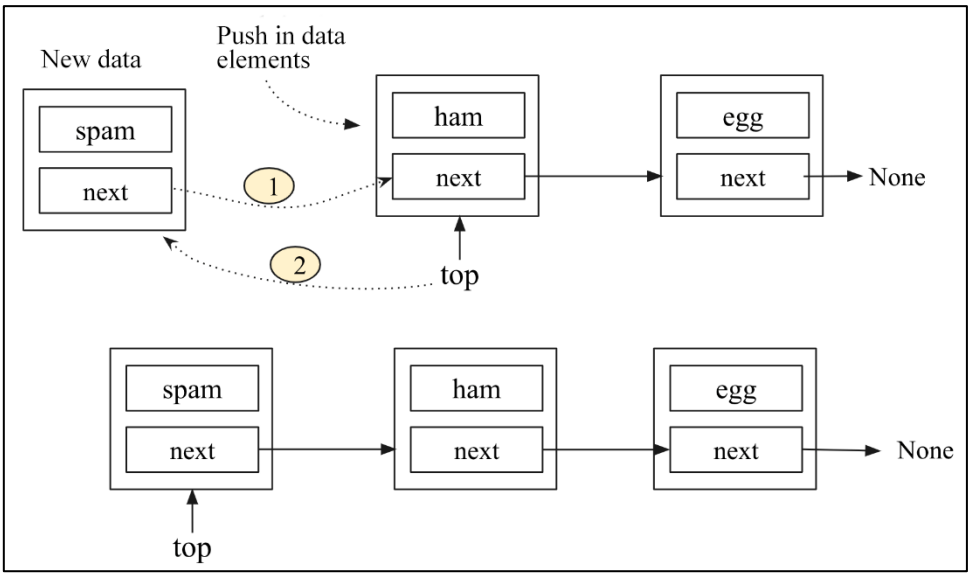


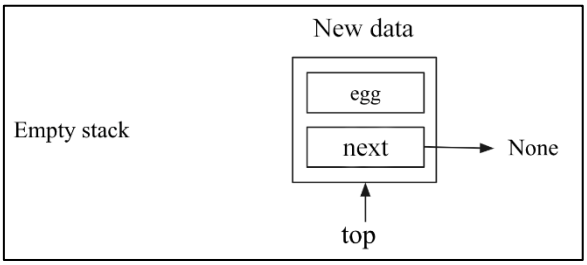


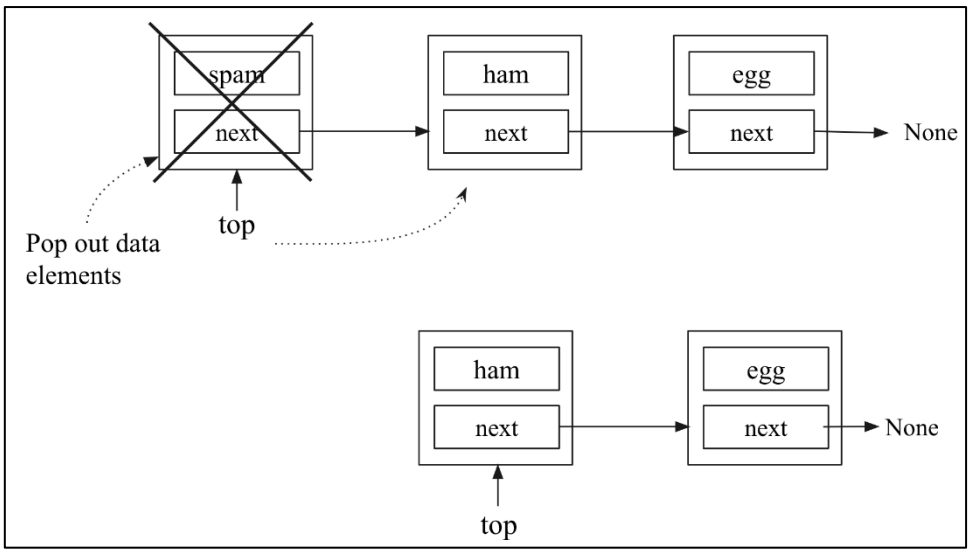


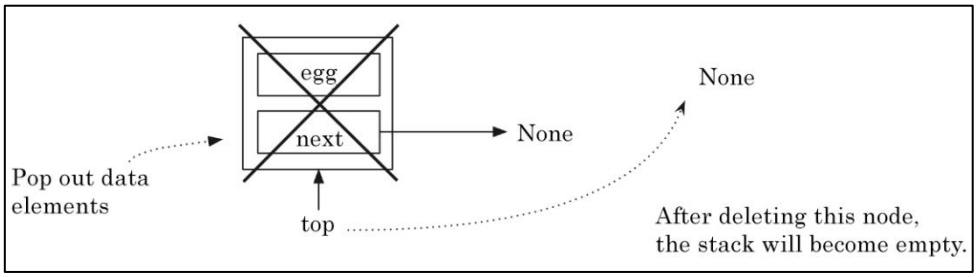

pop operation
pop operation
pop operation
pop operation
top
2
1
0
spam
ham
egg
stack
stack
stack
Stack empty
top -1
Stack underflow

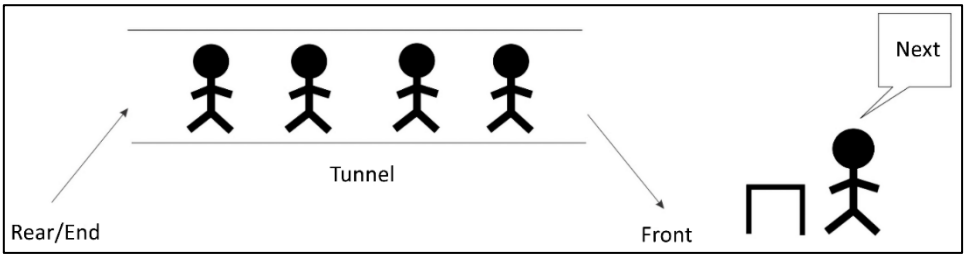

Push in data elements
Data
Next
None
Pop out data elements
Top
Stack using linked list

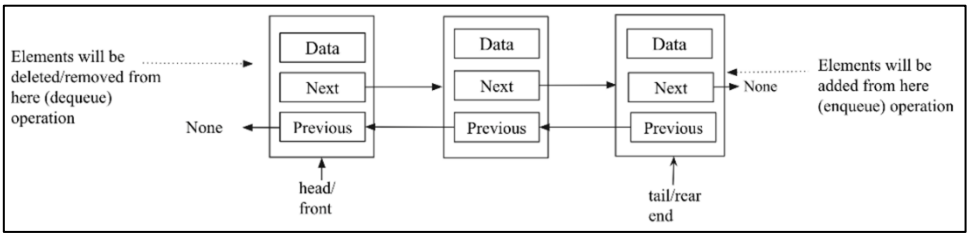

New data
Push in data elements
spam
next
ham
egg
None
1
2
top
spam
next
ham
egg
None
top

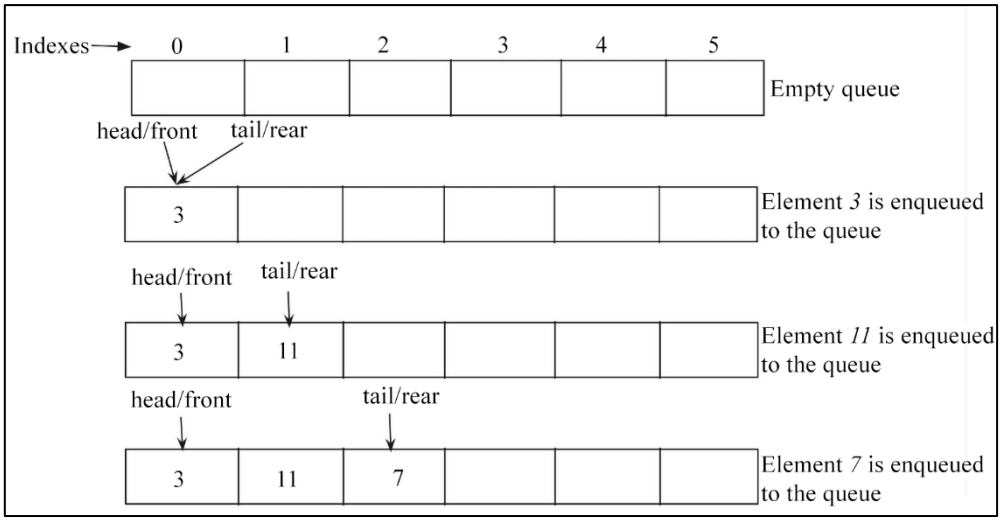

New data
Empty stack
egg
next
None
top

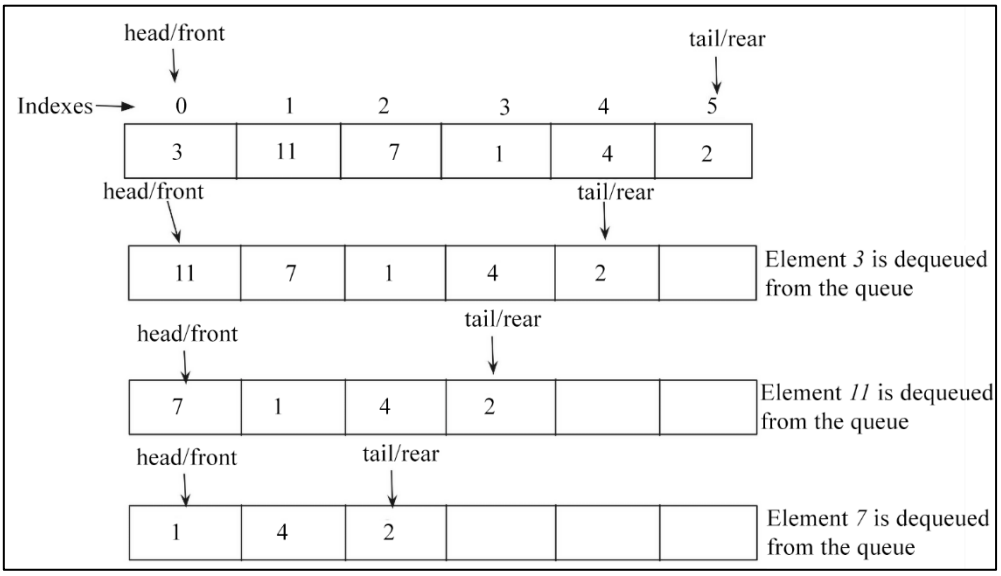

spam
next
ham
next
egg
next
None
top
Pop out data
elements
ham
next
egg
next
None
top

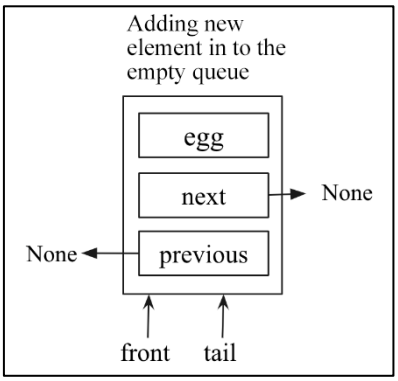

egg
next
None
Pop out data
elements
top
None
After deleting this node,
the stack will become empty.

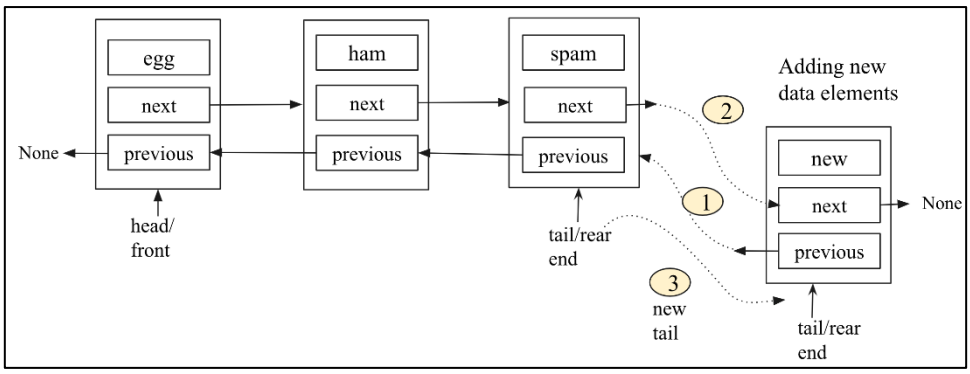

Next
Tunnel
Rear/End
Front

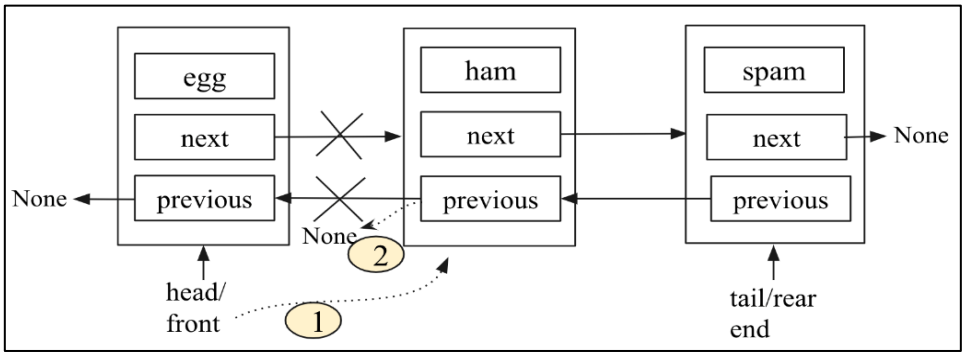

Elements will be
deleted/removed from
here (dequeue)
operation
Data
Next
Previous
None
head/
front
Data
Next
Previous
Data
Next
Previous
None
tail/rear
end
Elements will be
added from here
(enqueue) operation

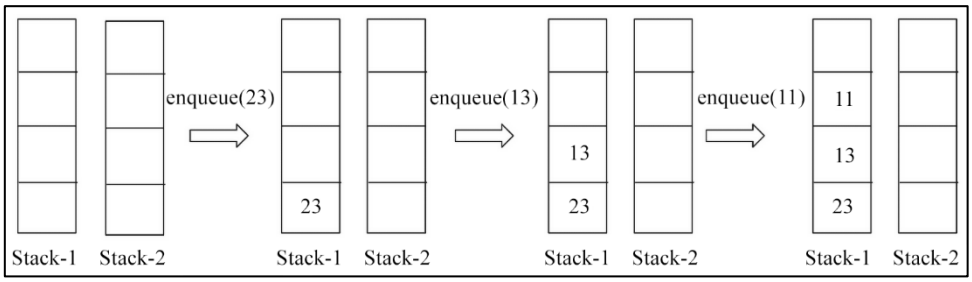

Indexes
0
1
2
3
4
5
Empty queue
head/front
tail/rear
3
Element 3 is enqueued to the queue
head/front
tail/rear
3
11
Element 11 is enqueued to the queue
head/front
tail/rear
3
11
7
Element 7 is enqueued to the queue

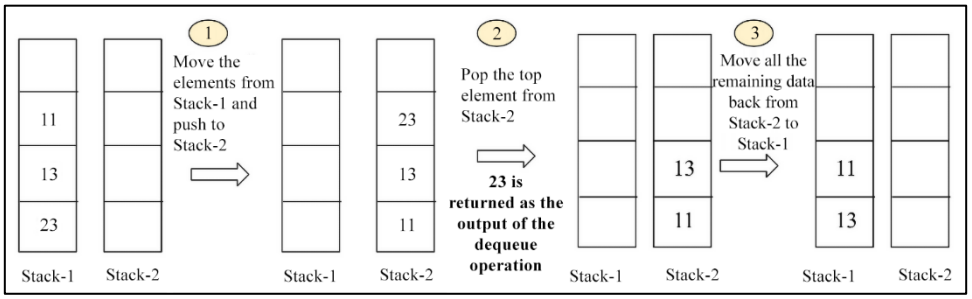

head/front
tail/rear
Indexes
0
1
2
3
4
5
3
11
7
1
4
2
head/front
tail/rear
11
7
1
4
2
Element 3 is dequeued from the queue
head/front
tail/rear
7
1
4
2
Element 11 is dequeued from the queue
head/front
tail/rear
1
4
2
Element 7 is dequeued from the queue

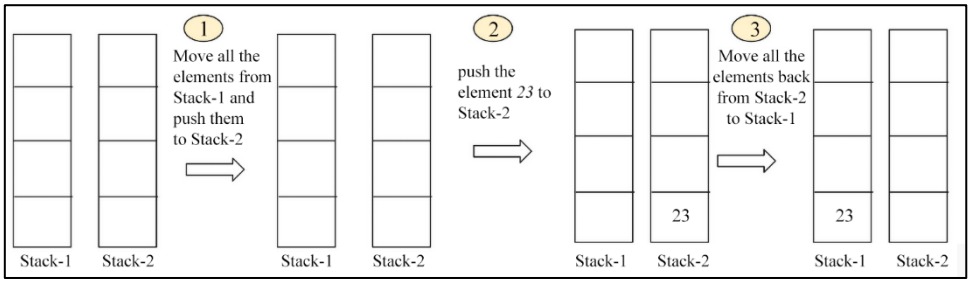

Adding new element in to the empty queue
egg
next
None
None
previous
front
tail

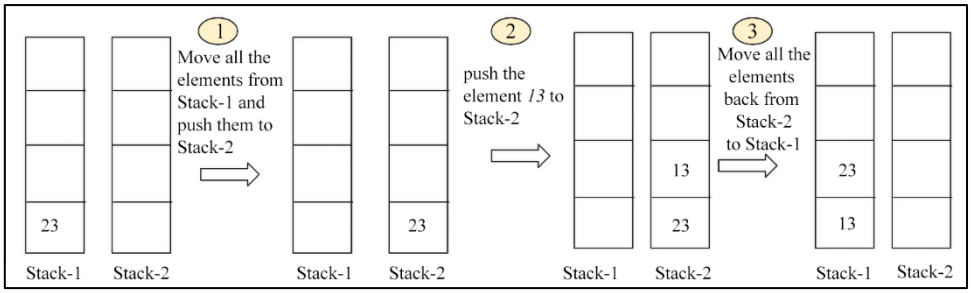

egg
next
previous
None
head/
front
ham
next
previous
spam
next
previous
tail/rear
end
Adding new
data elements
2
new
next
previous
None
1
3
new
tail
tail/rear
end

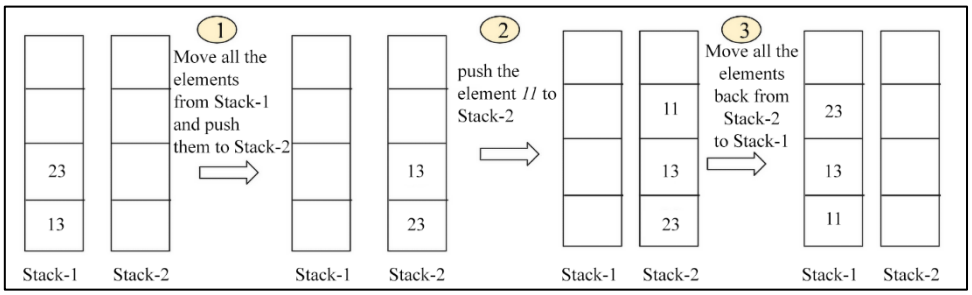

egg
next
previous
None
head/
front
1
ham
next
previous
None
2
spam
next
previous
None
tail/rear
end

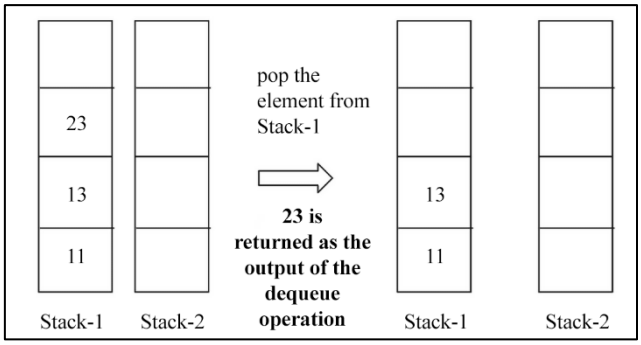

enqueue(23)
enqueue(13)
enqueue(11)
23
13
23
11
13
23
Stack-1
Stack-2

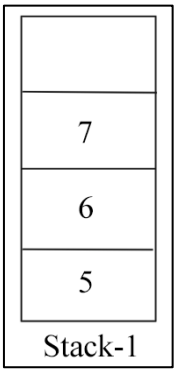

1
Move the
elements from
Stack-1 and
push to
Stack-2
2
Pop the top
element from
Stack-2
23 is
returned as the
output of the
dequeue
operation
3
Move all the
remaining data
back from
Stack-2 to
Stack-1
11
13
23
23
13
11
13
11
Stack-1
Stack-2

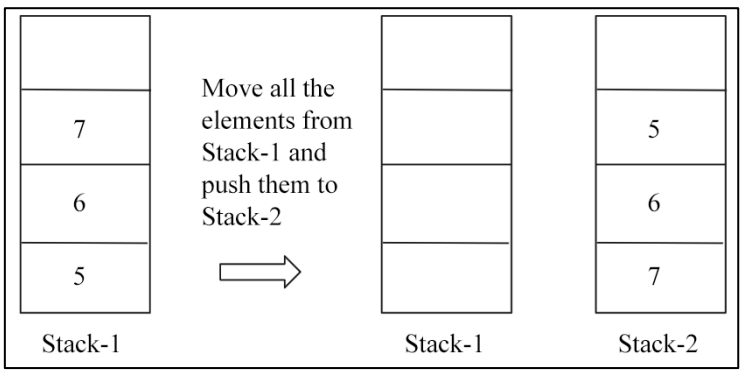

1
Move all the
elements from
Stack-1 and
push them
to Stack-2
2
push the
element 23 to
Stack-2
3
Move all the
elements back
from Stack-2
to Stack-1
23
23
Stack-1
Stack-2


1
Move all the elements from Stack-1 and push them to Stack-2
2
push the element 13 to Stack-2
3
Move all the elements back from Stack-2 to Stack-1
23
13
23
23
13
Stack-1
Stack-2
Stack-1
Stack-2
Stack-1
Stack-2
Stack-1
Stack-2

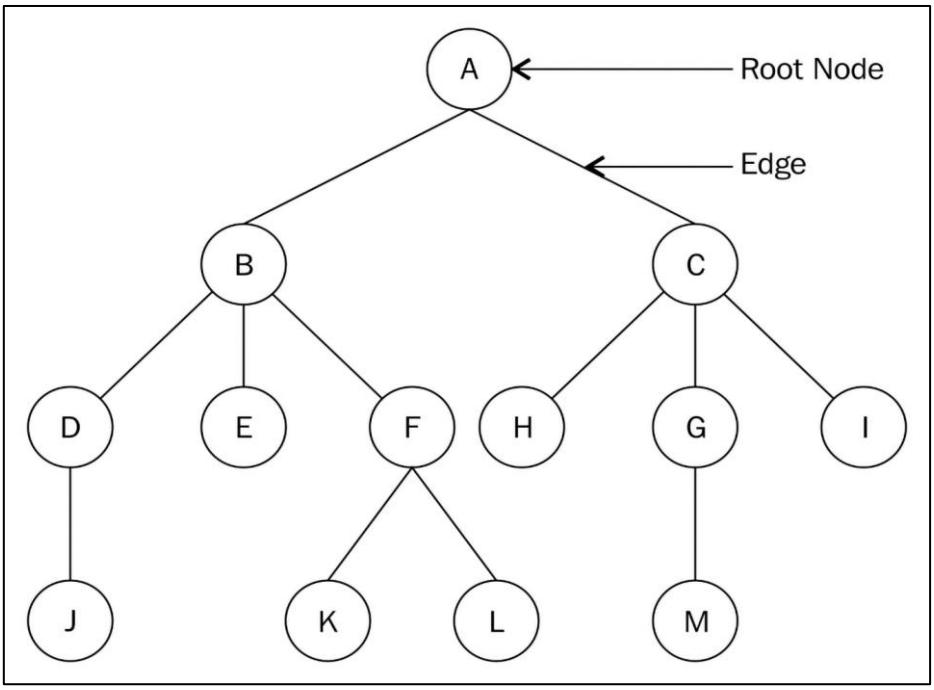

1
Move all the elements from Stack-1 and push them to Stack-2
2
push the element 11 to Stack-2
3
Move all the elements back from Stack-2 to Stack-1
23
13
13
23
11
13
23
23
13
11
Stack-1
Stack-2
Stack-1
Stack-2
Stack-1
Stack-2
Stack-1
Stack-2

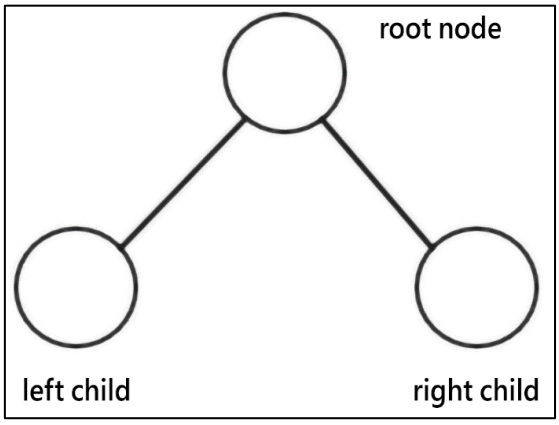

23
13
11
pop the element from Stack-1
23 is returned as the output of the dequeue operation
13
11
Stack-1
Stack-2
Stack-1
Stack-2

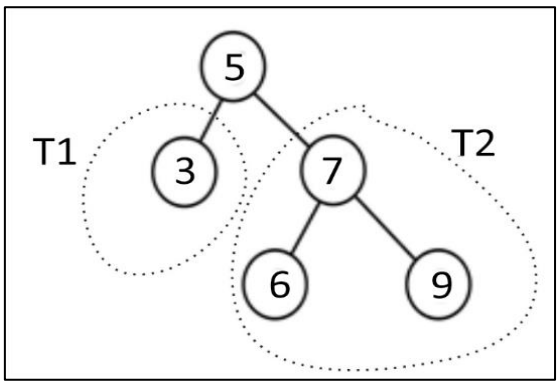

7
6
5
Stack-1

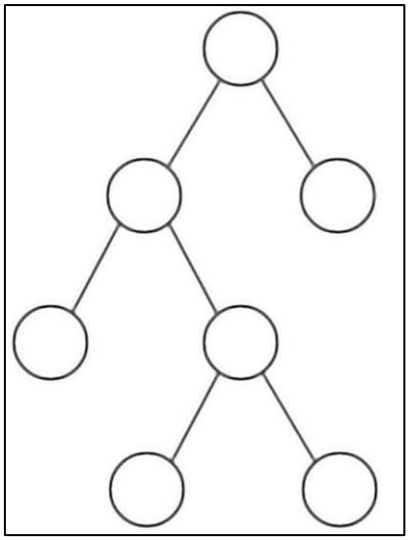

7
6
5
Move all the elements from Stack-1 and push them to Stack-2
5
6
7
Stack-1
Stack-1
Stack-2

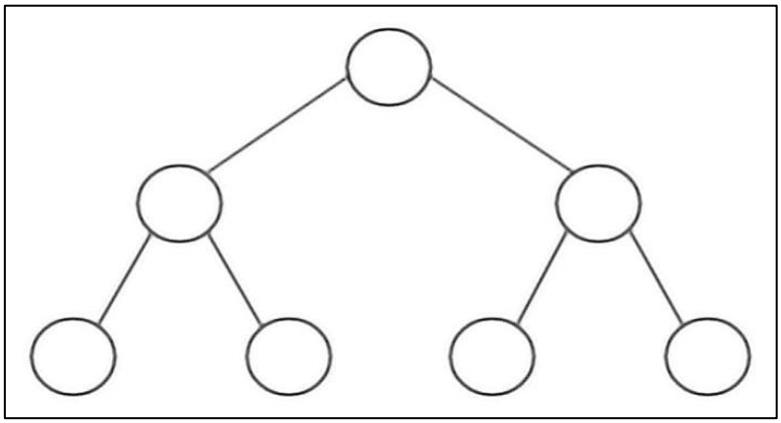
Chapter 6: Trees

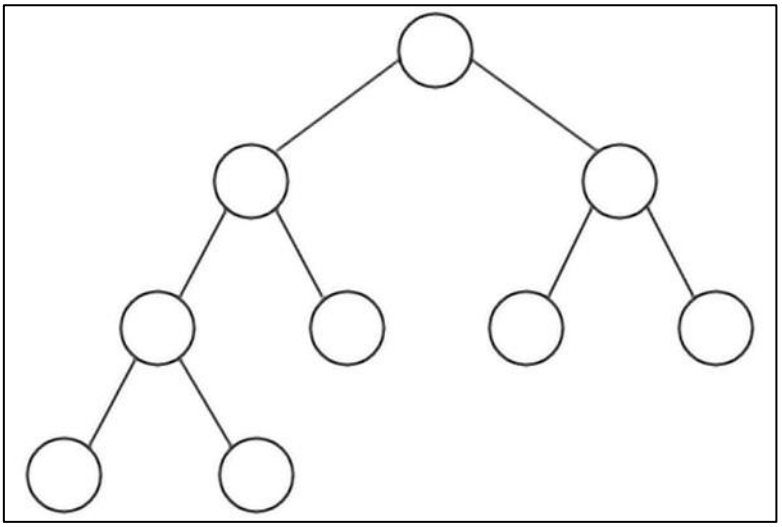


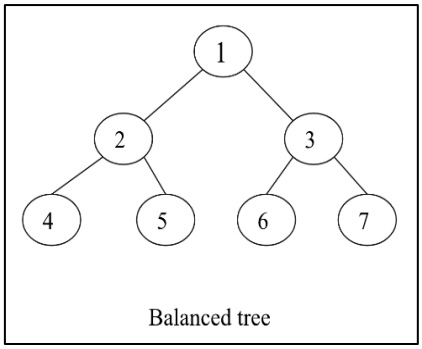




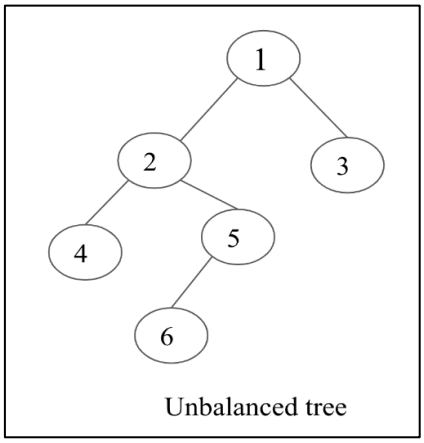




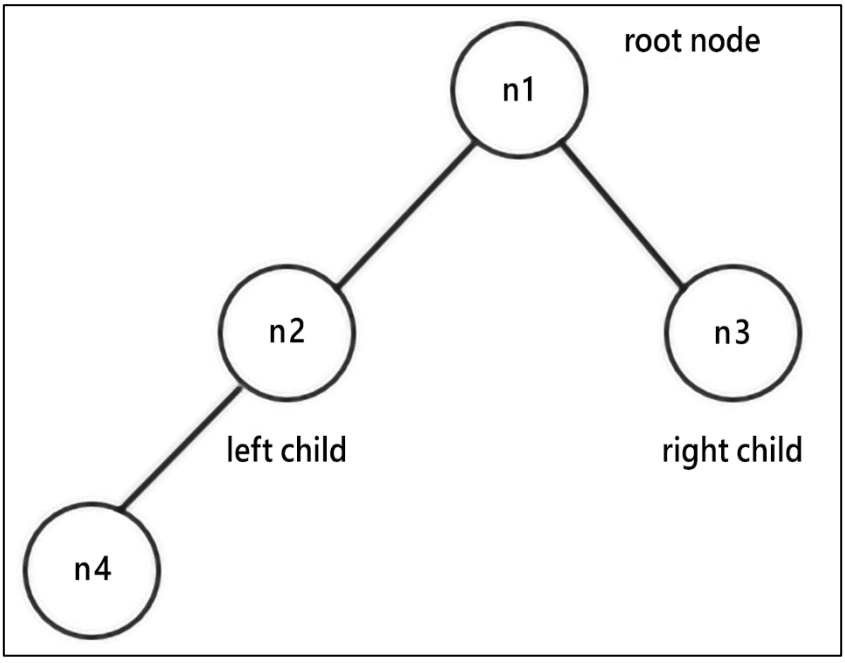


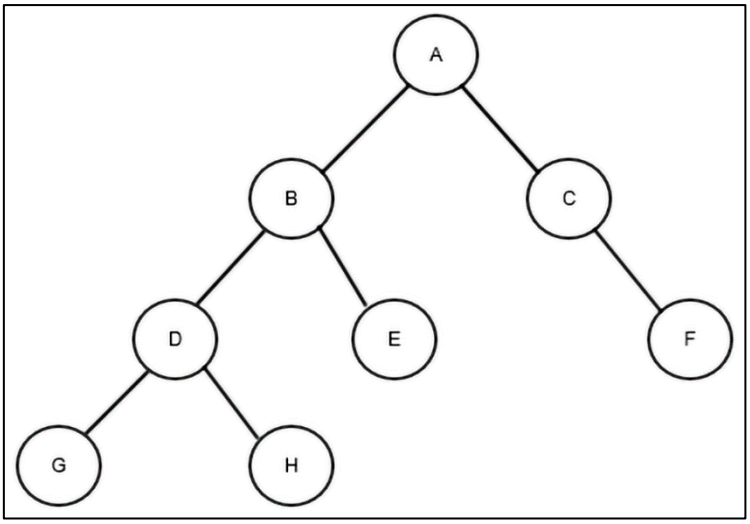


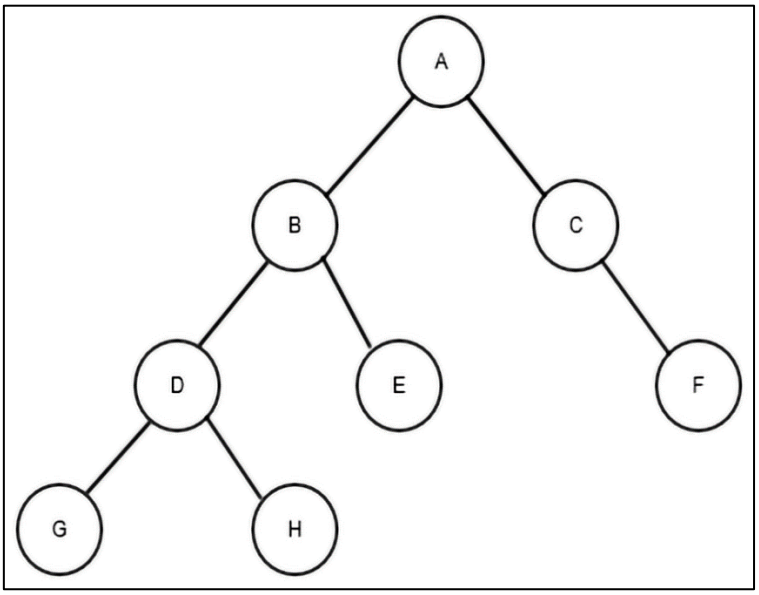
Balanced tree

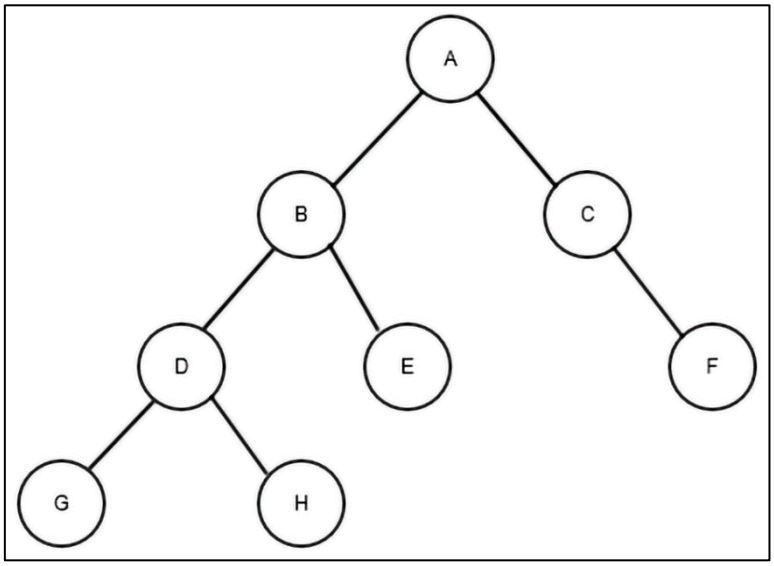


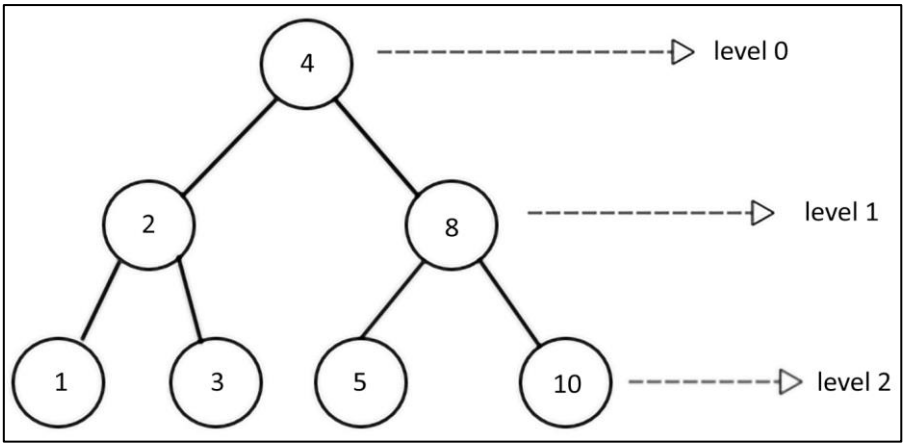
Unbalanced tree

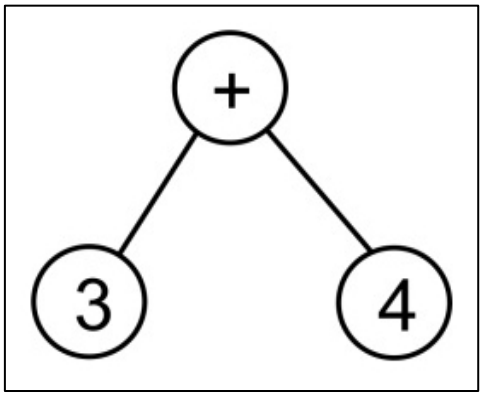


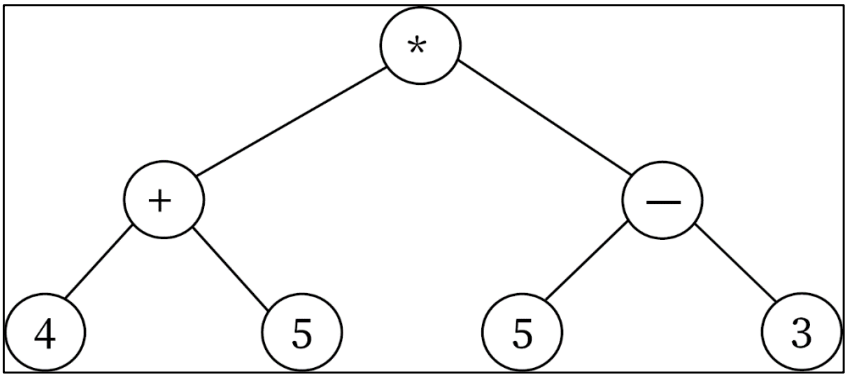

A
B
C
D
E
F
G
H

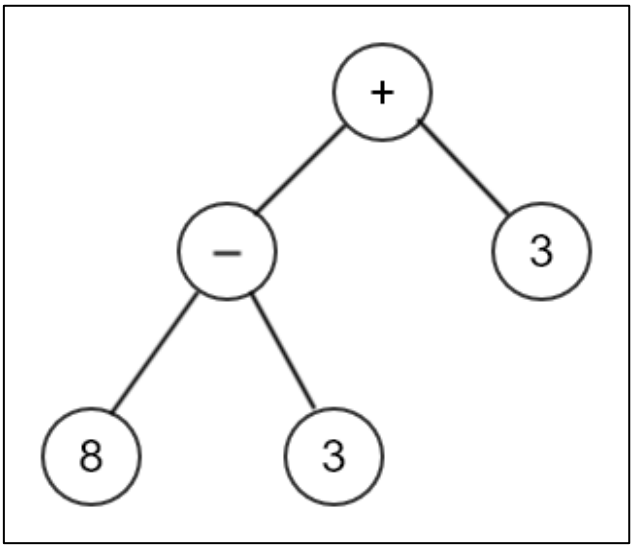

A
B
C
D
E
F
G
H

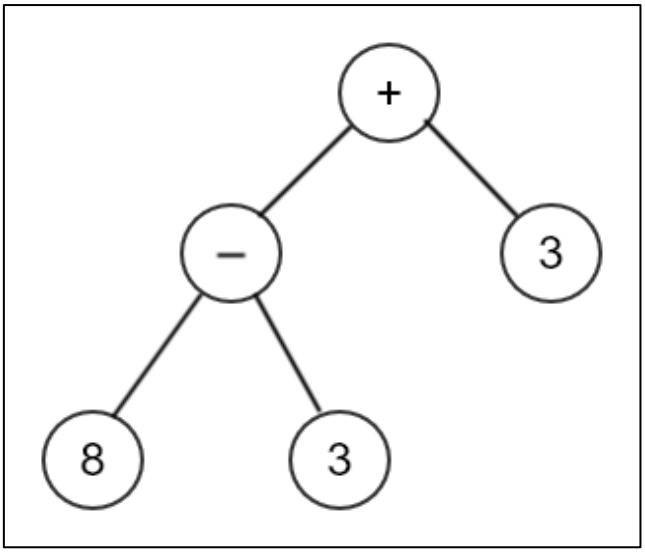

A
B
C
D
E
F
G
H

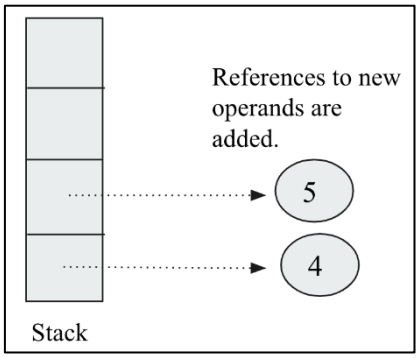

4
level 0
2
8
level 1
1
3
5
10
level 2

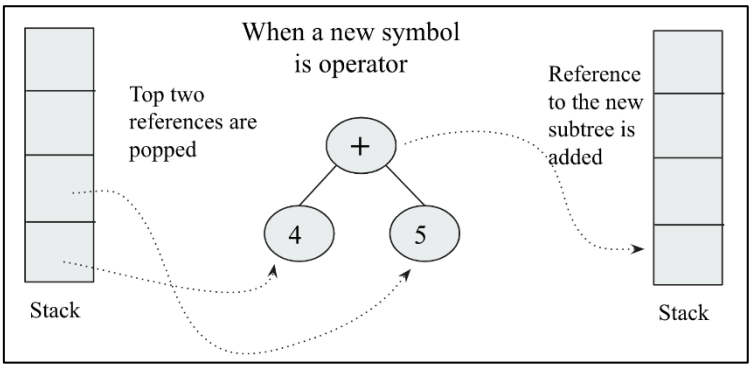

+
3
4

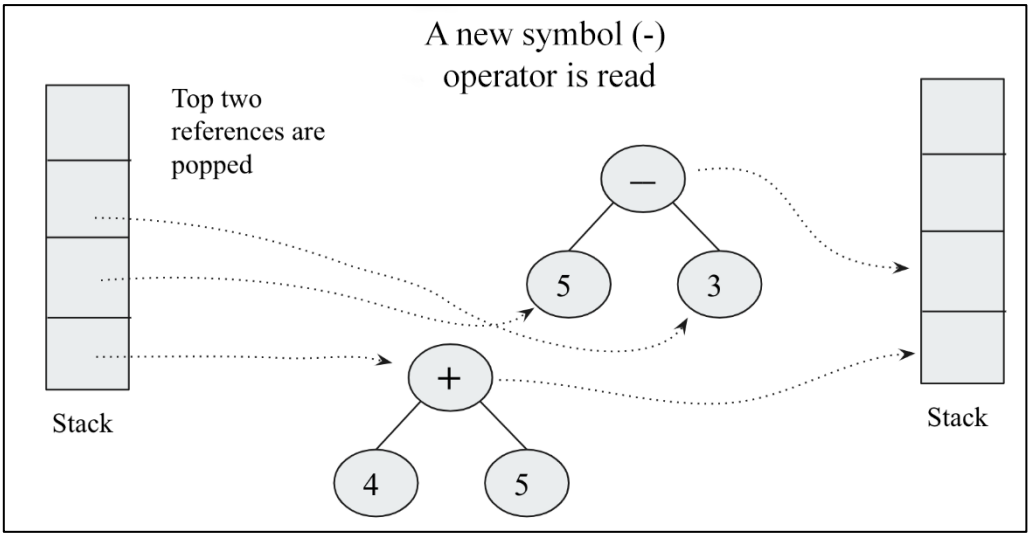

*
+
—
4
5
5
3

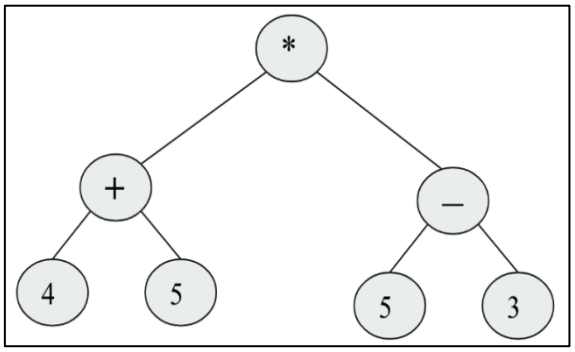

+
–
3
8
3

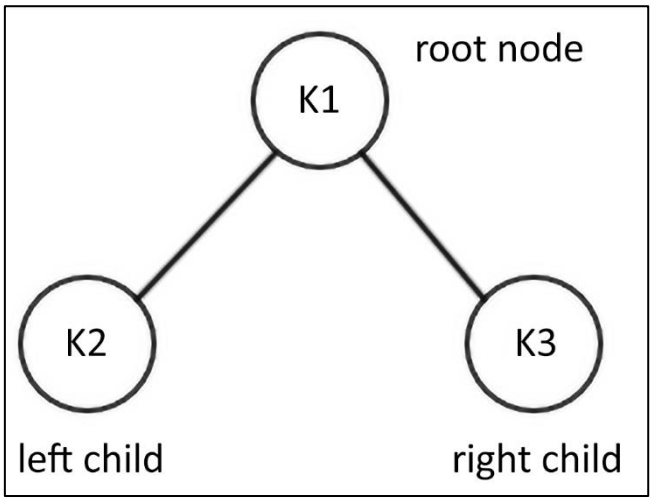

+
−
3
8
3

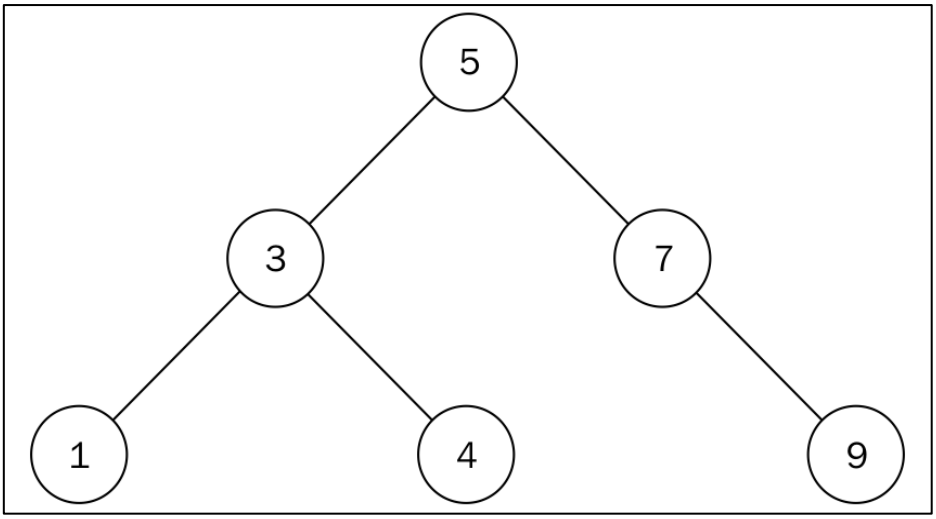

References to new
operands are
added.
5
4
Stack

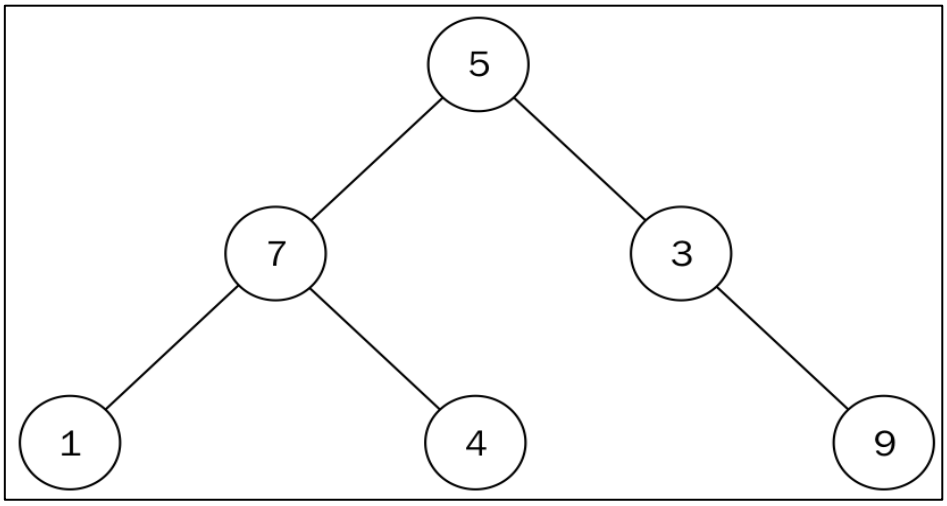

When a new symbol
is operator
Top two
references are
popped
Reference
to the new
subtree is
added
+
4
5
Stack
Stack

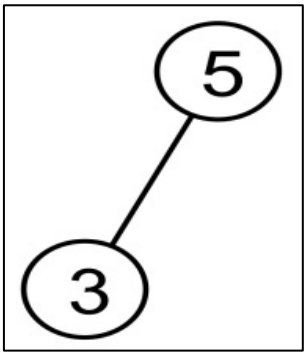

A new symbol (-)
operator is read
Top two
references are
popped
Stack
Stack
–
5
3
+
4
5

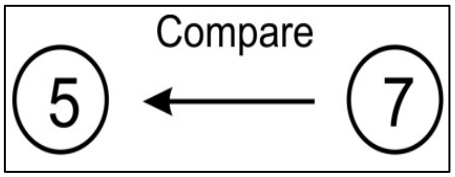

*
+
–
4
5
5
3

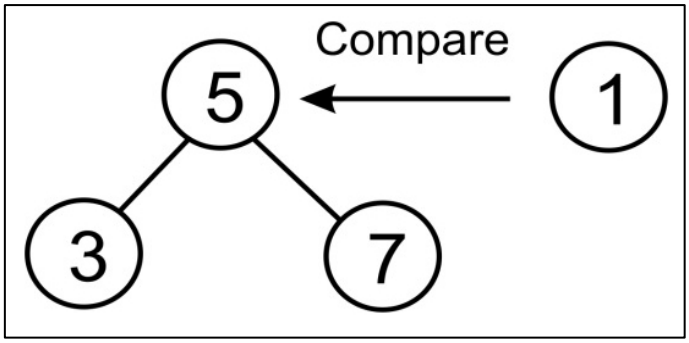

root node
K1
K2
K3
left child
right child

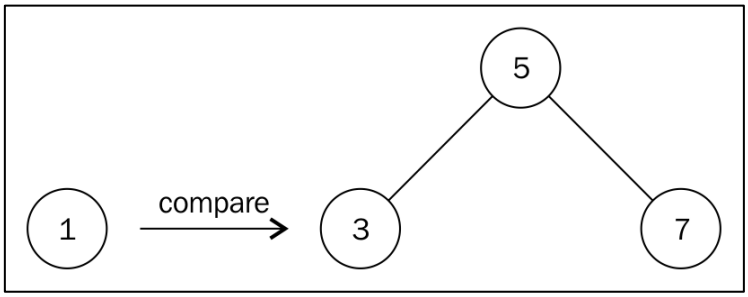

5
3
7
1
4
9

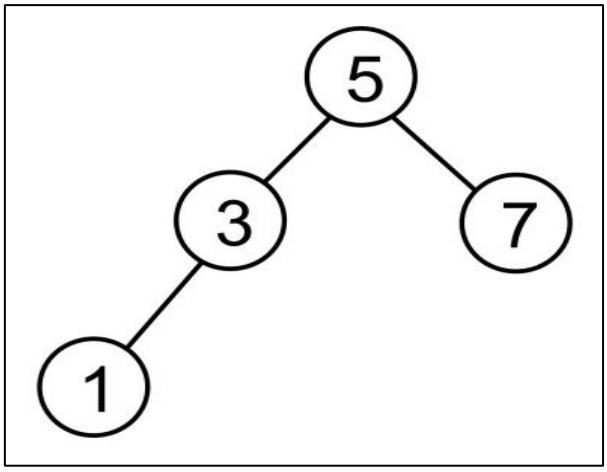

5
7
3
1
4
9

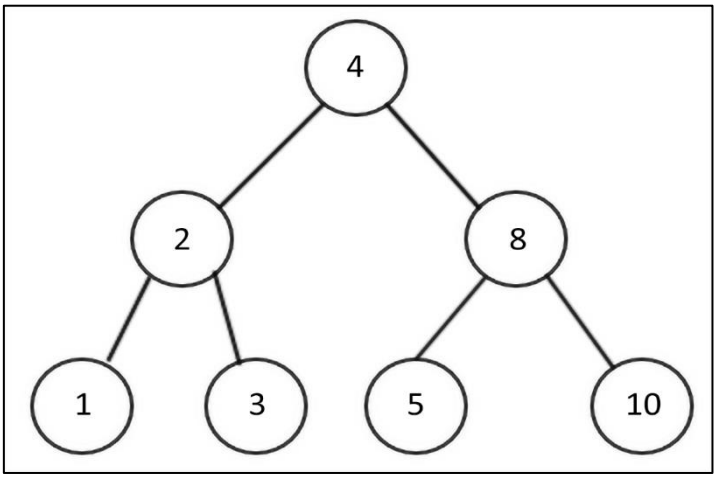

5
3

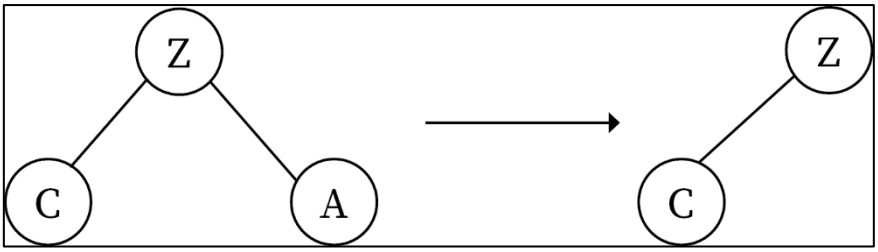

Compare
5
7

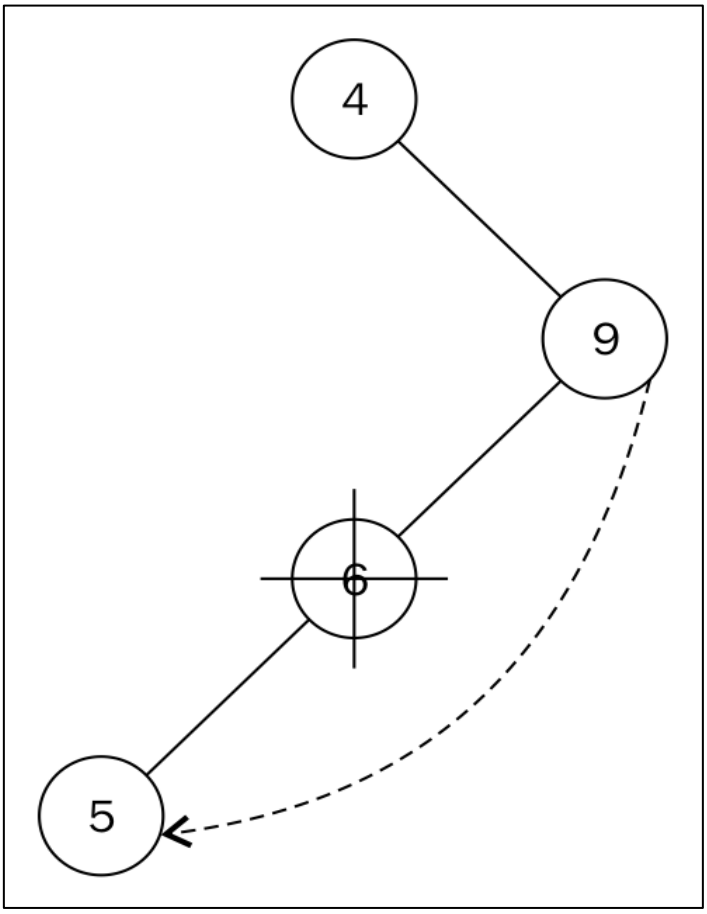

Compare
5
1
3
7

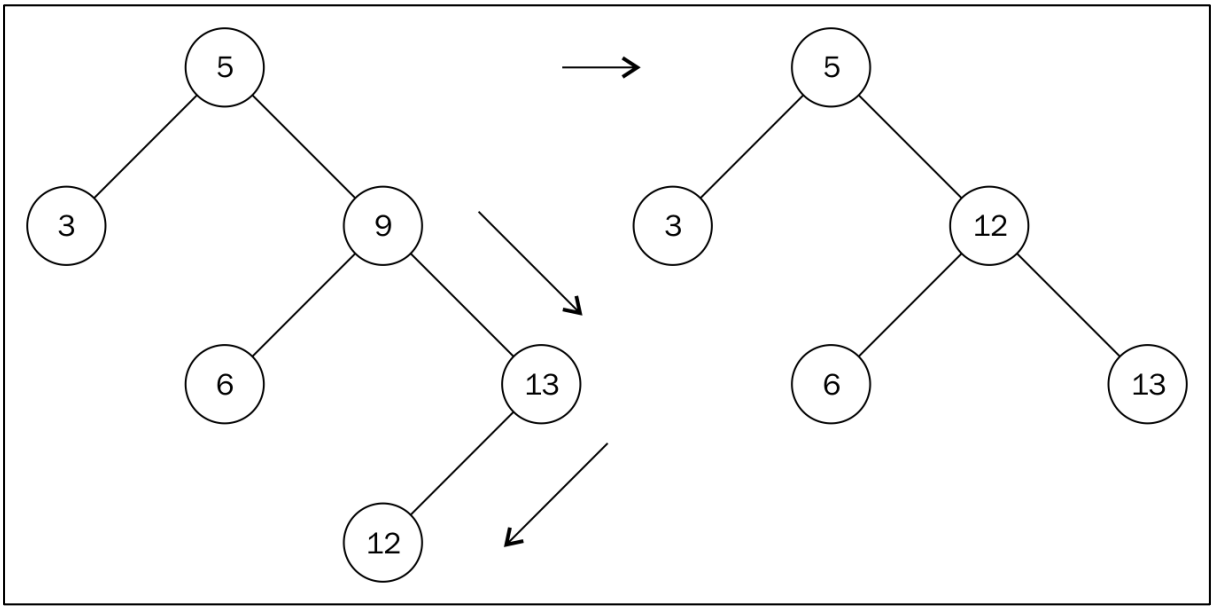

5
1
compare
3
7

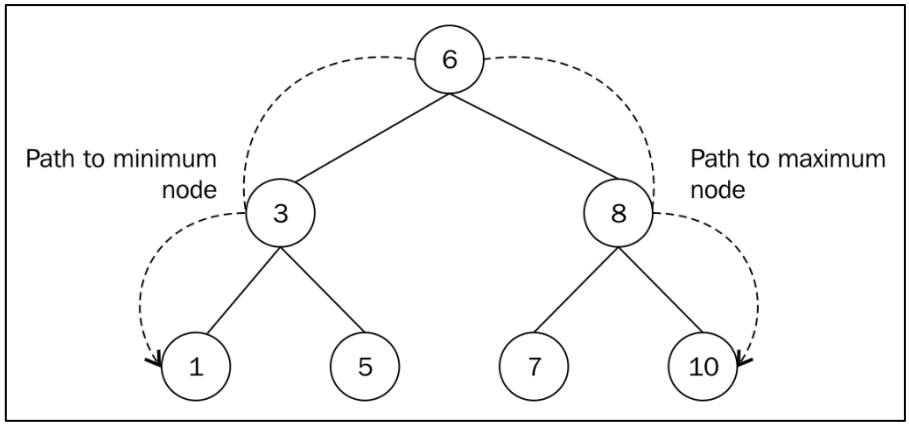

5
3
7
1

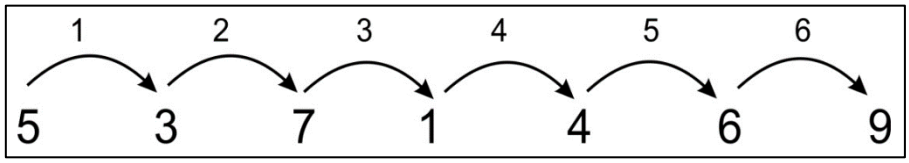

4
2
8
1
3
5
10

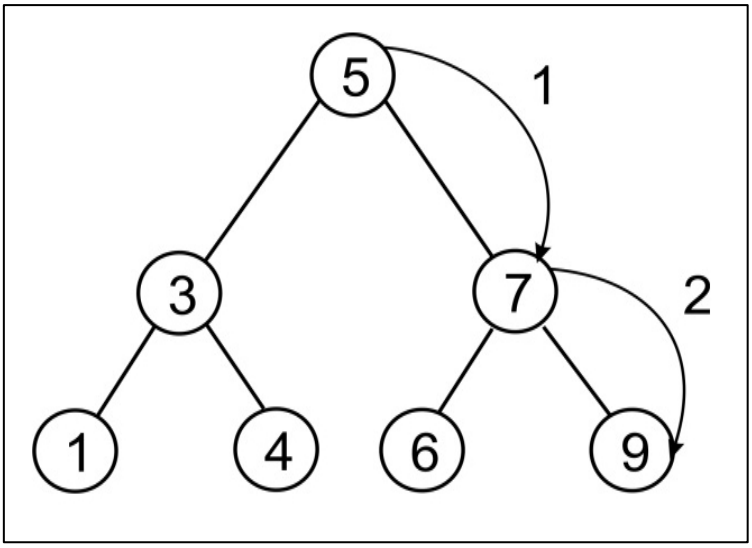

Z
C
A
Z
C

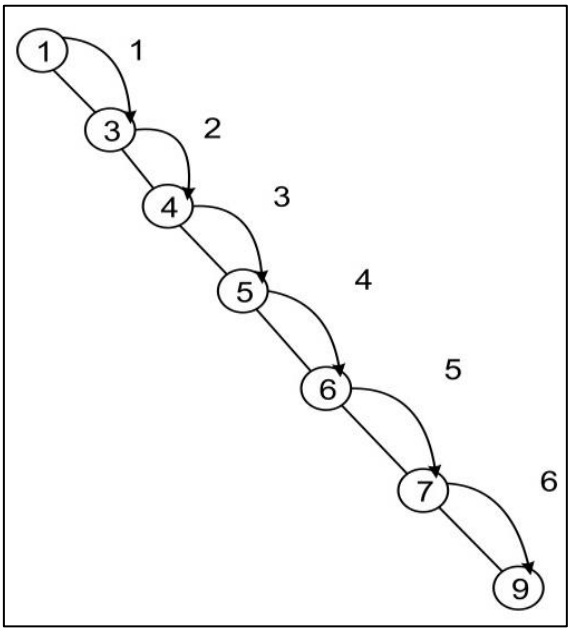

4
9
6
5

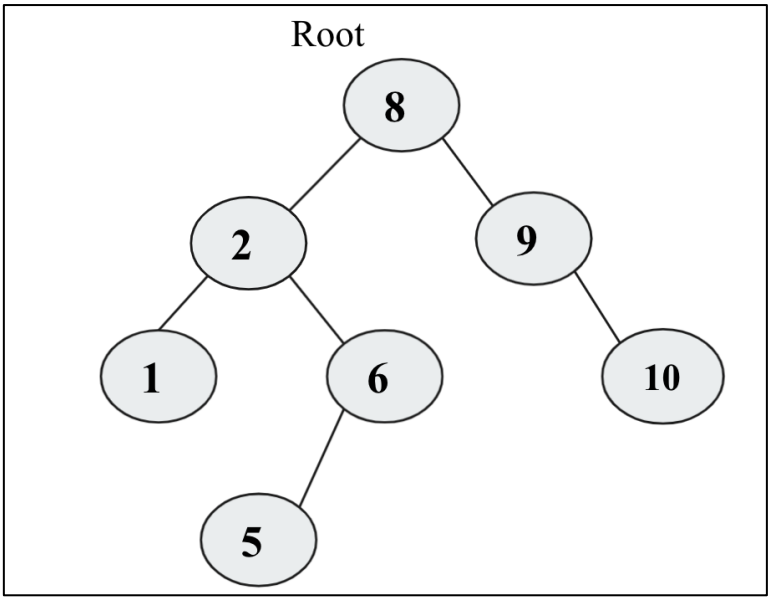

5
3
9
6
13
12
5
3
12
6
13

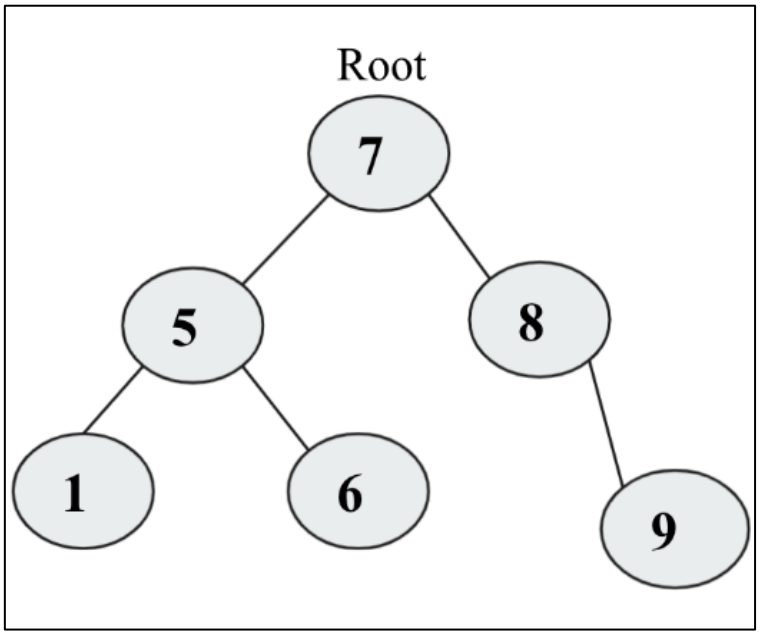

6
Path to minimum node
Path to maximum node
3
8
1
5
7
10


1
2
3
4
5
6
5
3
7
1
4
6
9

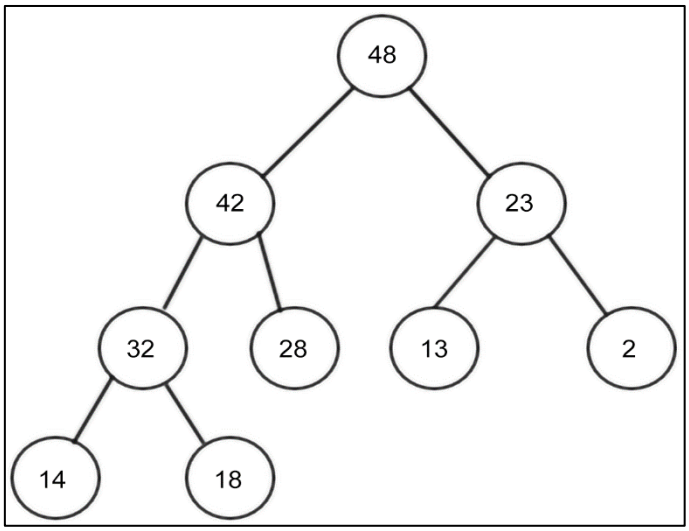

5
1
3
7
2
1
4
6
9

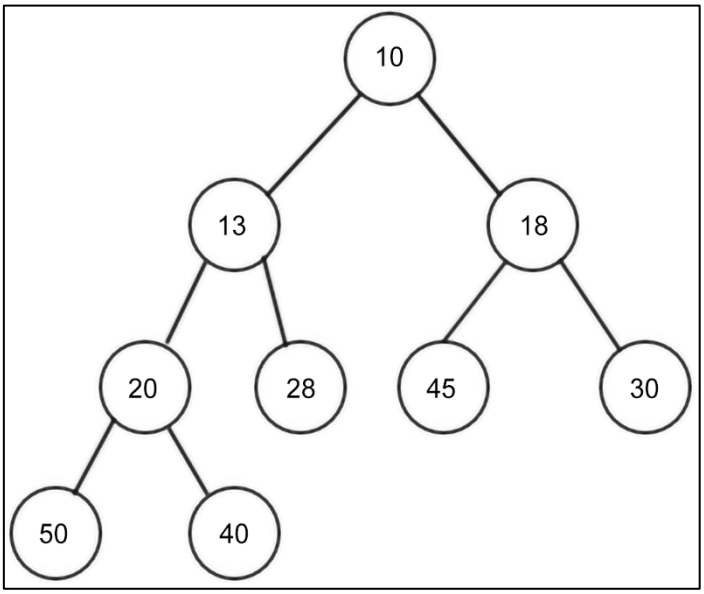

1
1
3
2
4
3
5
4
6
5
7
6
9

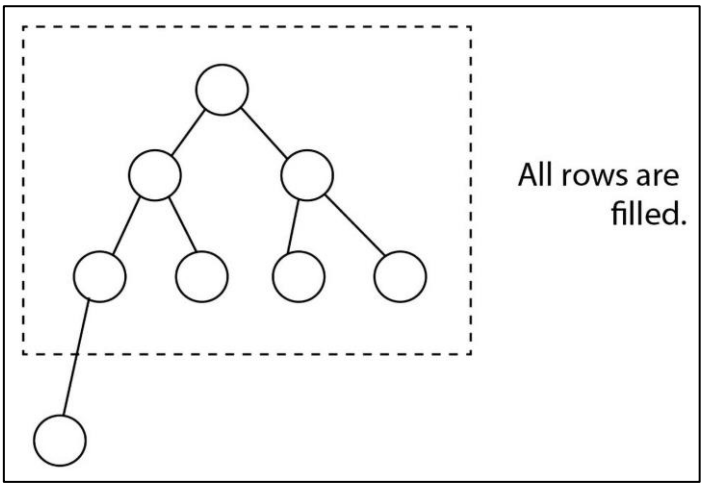

Root
8
2
9
1
6
10
5

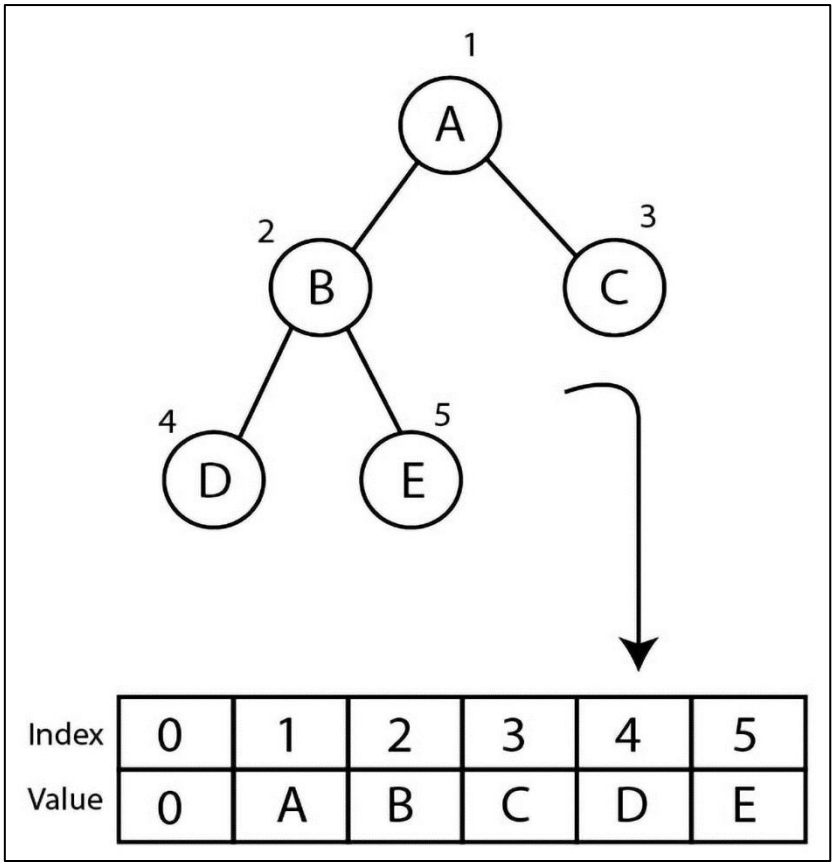

Root
7
5
8
1
6
9

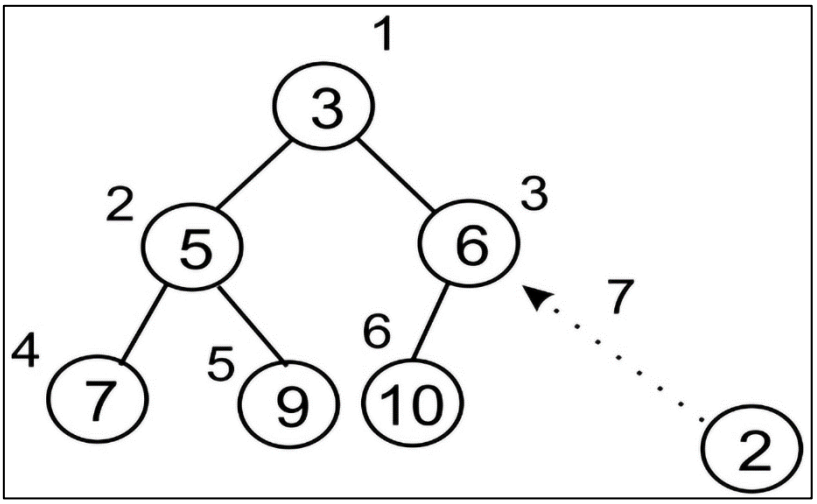
Chapter 7: Heaps and Priority Queues

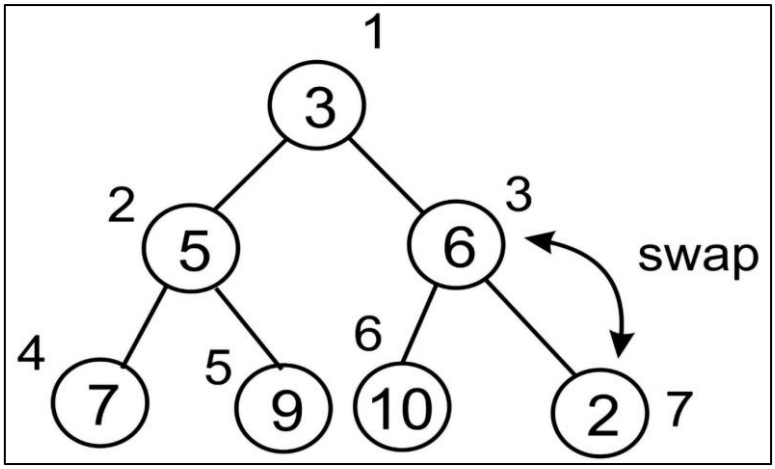


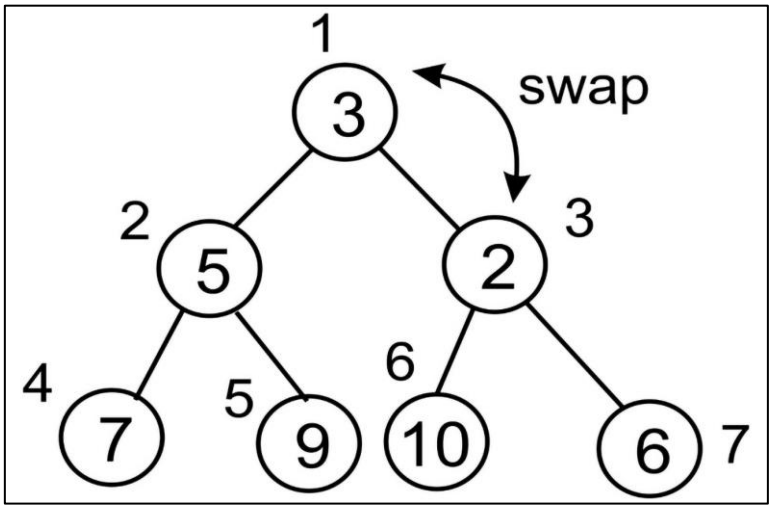


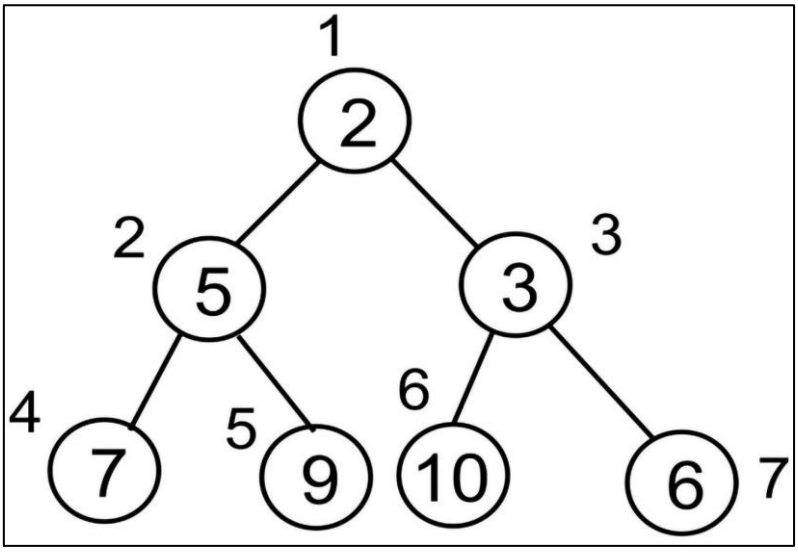


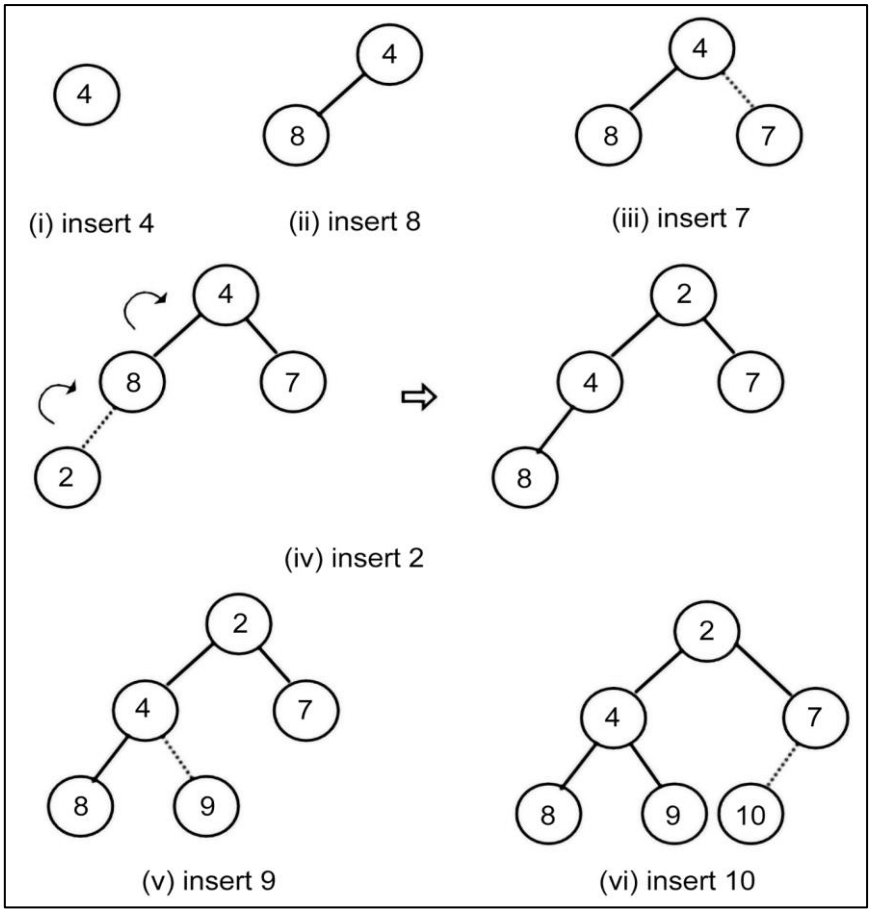

1
A
2
B
3
C
4
D
5
E
Index
Value
0
1
2
3
4
5
0
A
B
C
D
E

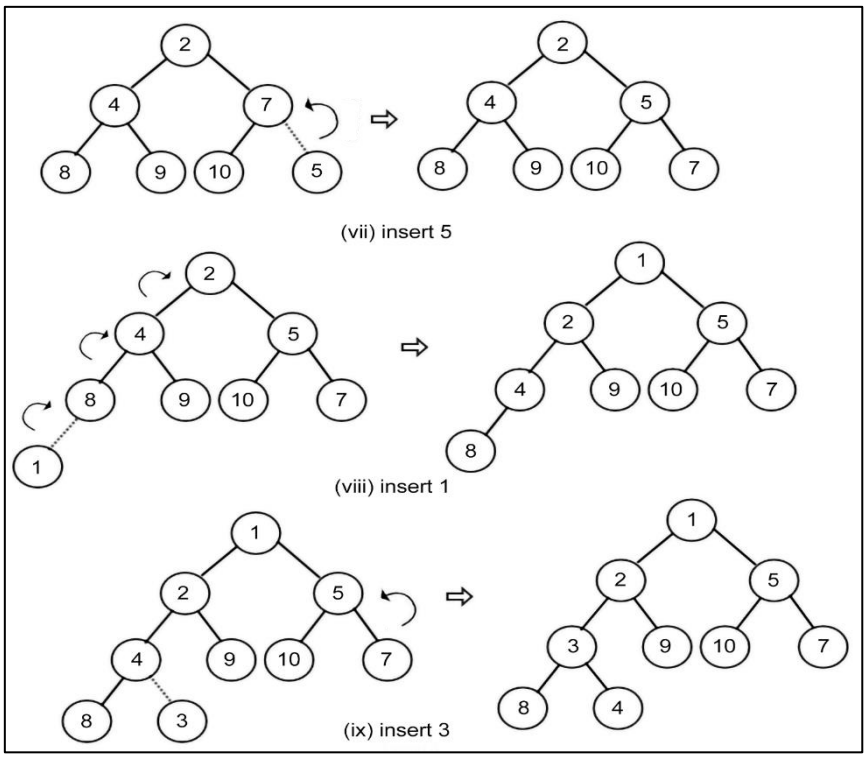

1
3
2
5
3
6
7
4
7
5
9
6
10
2

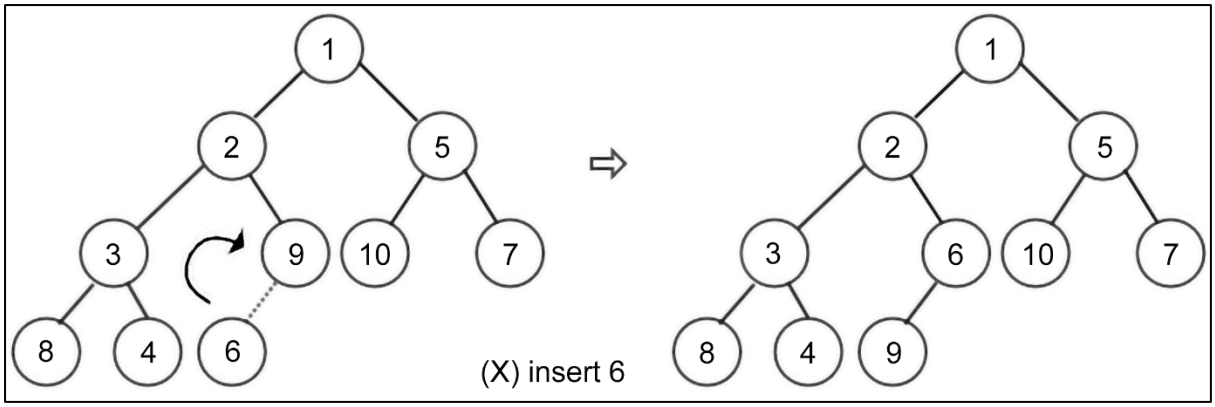

1
3
2
5
3
6
swap
4
7
5
9
6
10
2
7

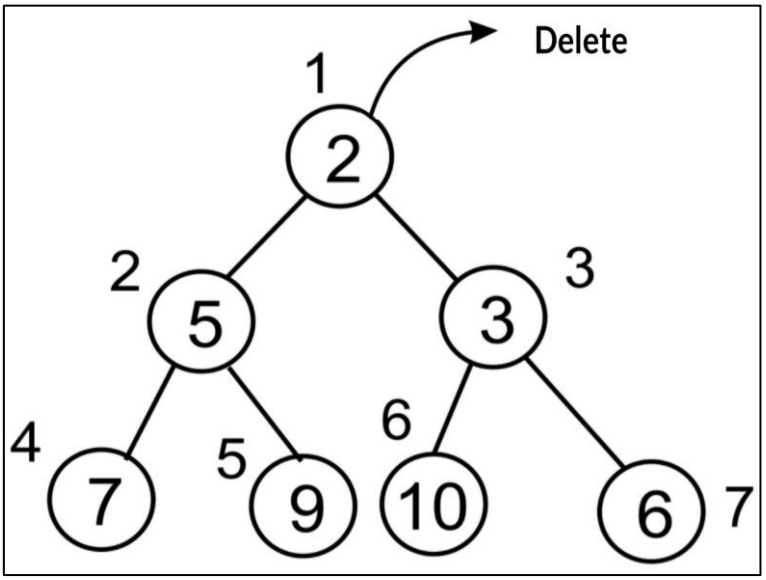

1
3
swap
2
5
3
2
4
7
5
9
6
10
6
7

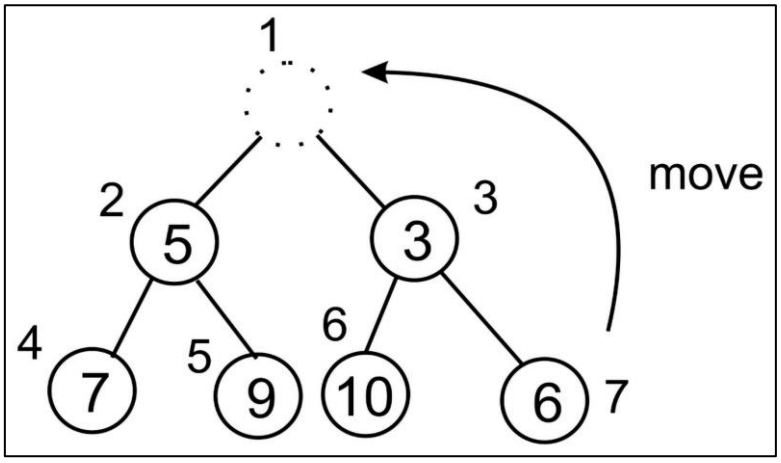

1
2
2
5
3
3
4
7
5
9
6
10
6
7

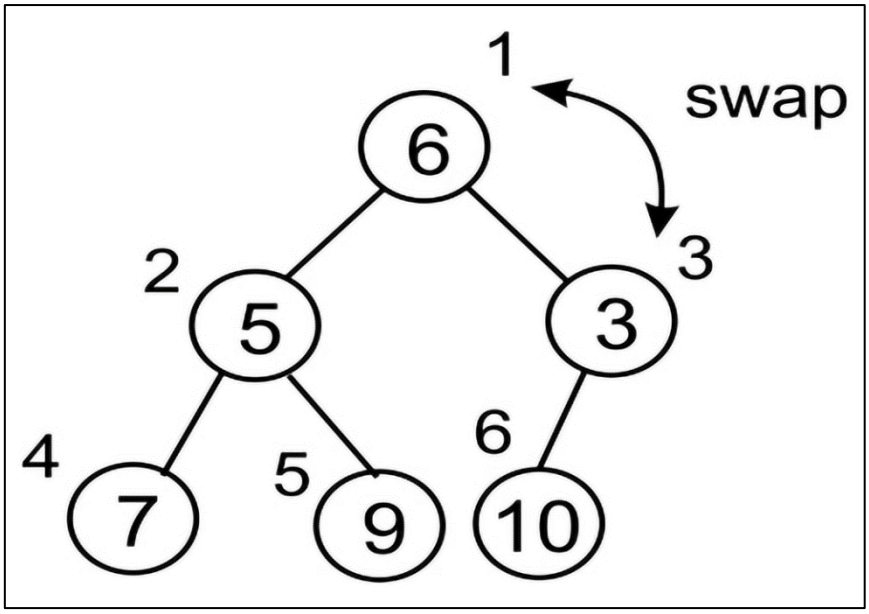

4
4
8
4
8
7
(i) insert 4
(ii) insert 8
(iii) insert 7
4
8
7
2
2
4
7
8
(iv) insert 2
2
4
7
8
9
2
4
7
8
9
10
(v) insert 9
(vi) insert 10

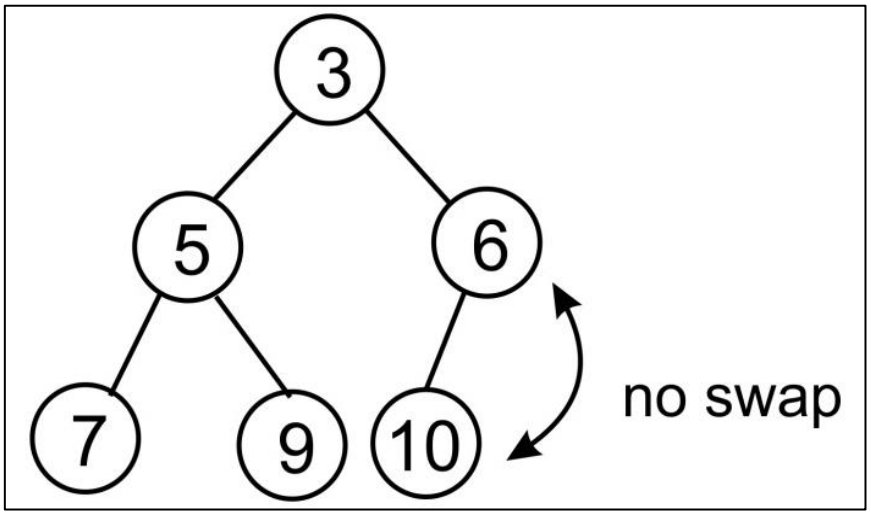

2
4
7
8
9
10
5
2
4
5
8
9
10
7
(vii) insert 5
2
4
5
8
9
10
7
1
1
2
5
4
9
10
7
8
(viii) insert 1
1
2
5
4
9
10
7
8
3
1
2
5
3
9
10
7
8
4
(ix) insert 3

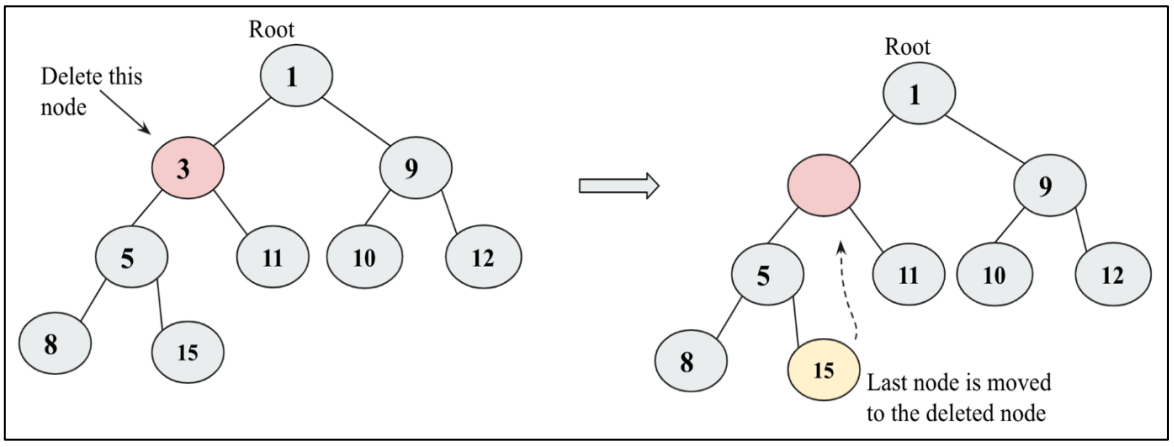

1
2
5
3
9
10
7
8
4
6
(X) insert 6
1
2
5
3
6
10
7
8
4
9

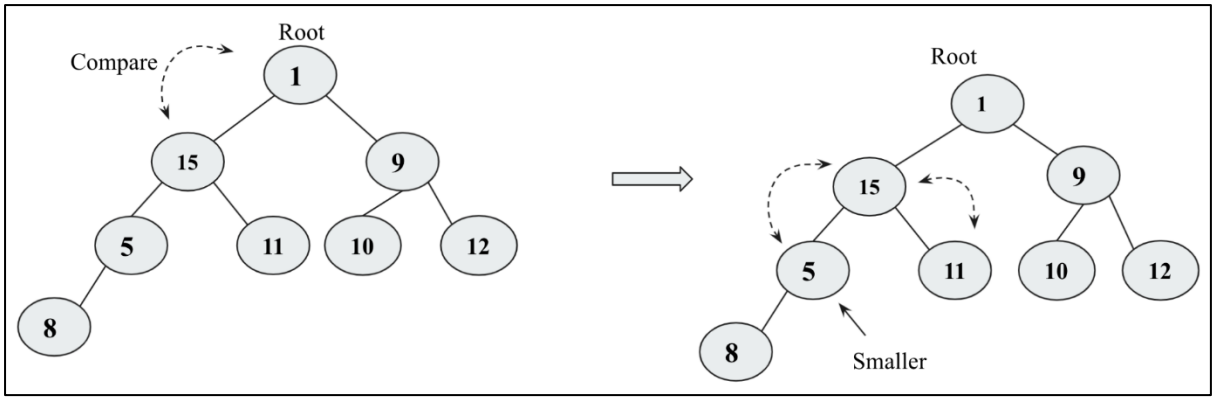

Delete
1
2
2
5
3
3
4
7
5
9
6
10
6
7

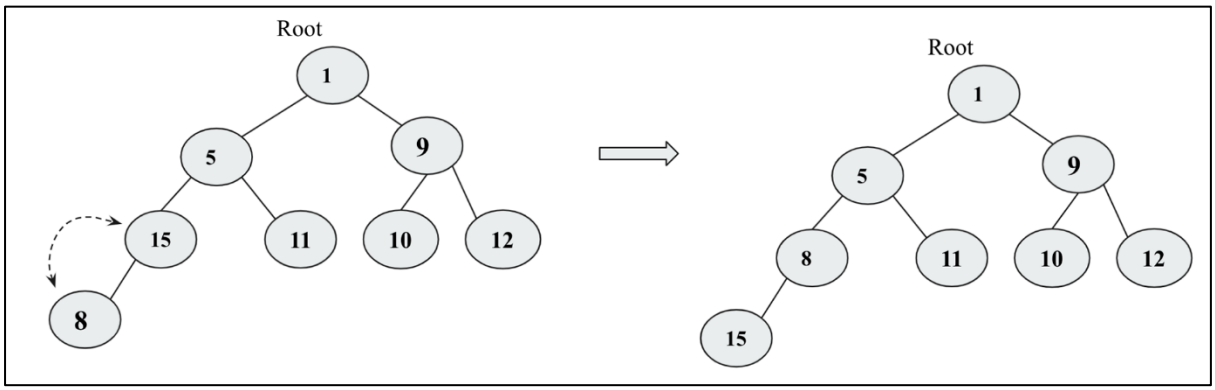

1
move
2
5
3
3
4
7
5
9
6
10
6
7

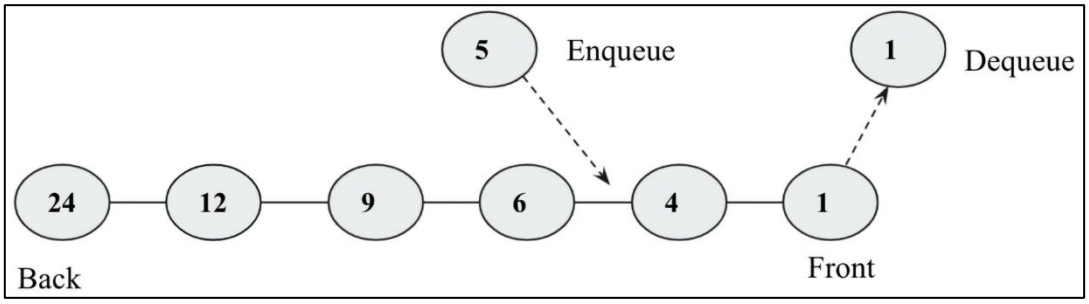

1
swap
6
2
5
3
3
4
7
5
9
6
10

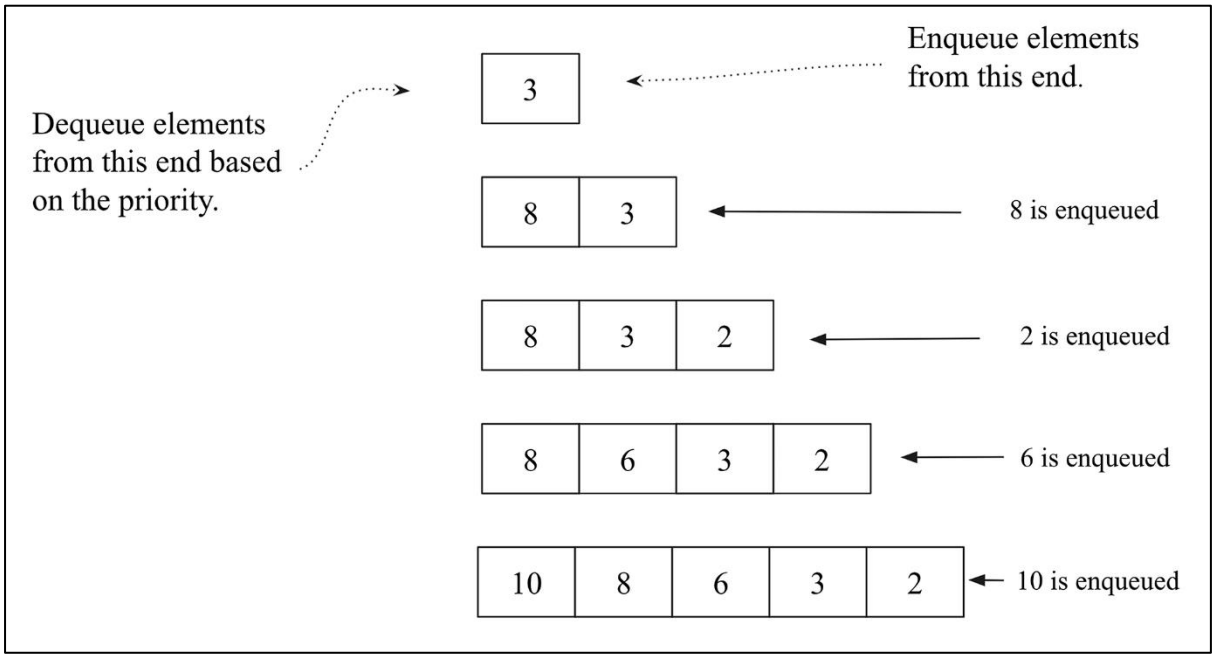

3
5
6
7
9
10
no swap


Root
Delete this node
1
3
9
5
11
10
12
8
15
Root
1
9
5
11
10
12
8
15
Last node is moved to the deleted node

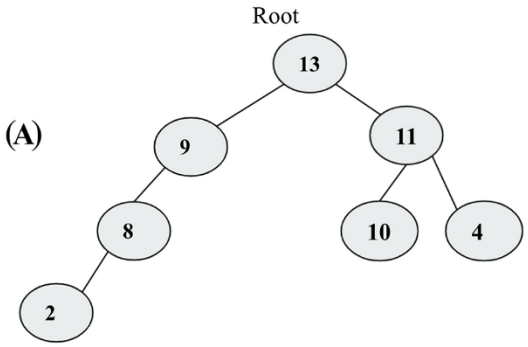

Root
Compare
1
15
9
5
11
10
12
8
Root
1
15
9
5
11
10
12
8
Smaller


Root
1
5
9
15
11
10
12
8
Root
1
5
9
8
11
10
12
15

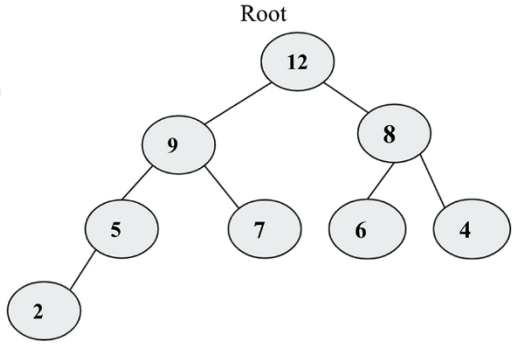

5
Enqueue
1
Dequeue
24
12
9
6
4
1
Back
Front


Dequeue elements from this end based on the priority.
Enqueue elements from this end.
3
8
3
8 is enqueued
8
3
2
2 is enqueued
8
6
3
2
6 is enqueued
10
8
6
3
2
10 is enqueued

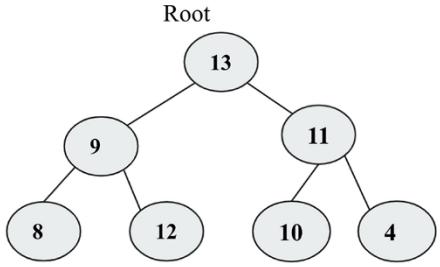
(A)


Root
13
9
11
8
10
4
2

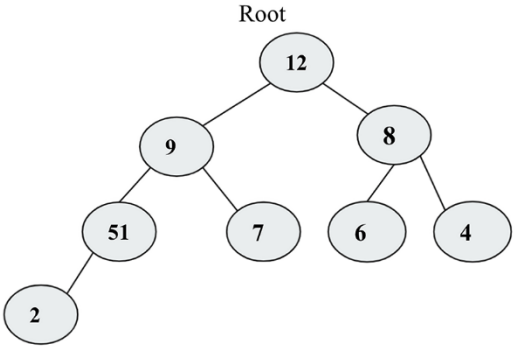
(B)


Root
12
9
8
5
7
6
4
2

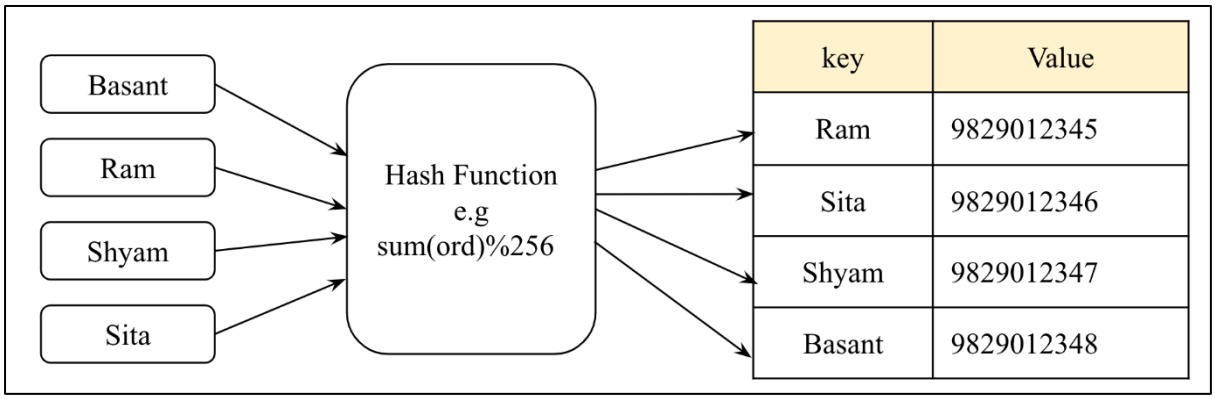
(C)


Root
13
9
11
8
12
10
4

(D)


Root
12
9
8
51
7
6
4
2

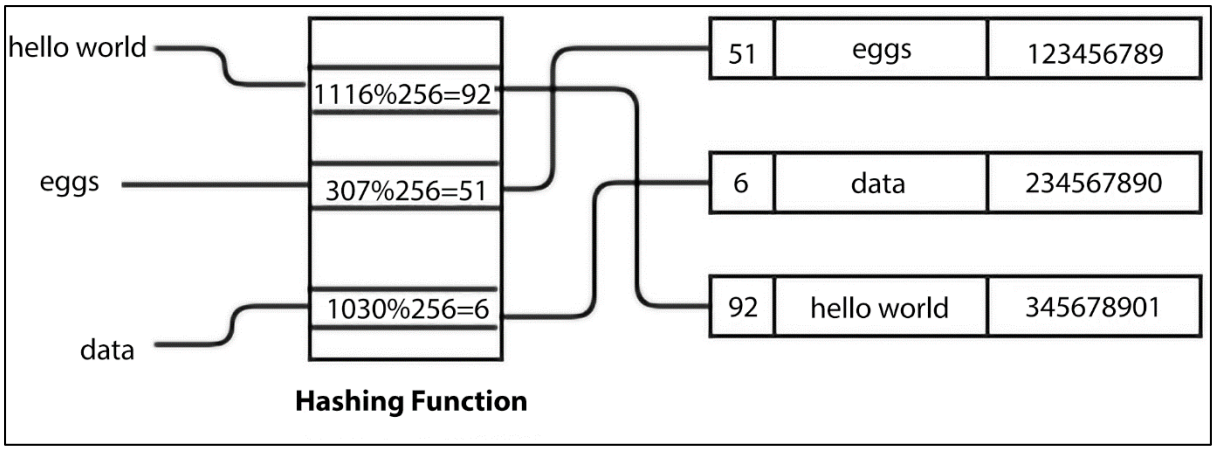
Chapter 8: Hash Tables

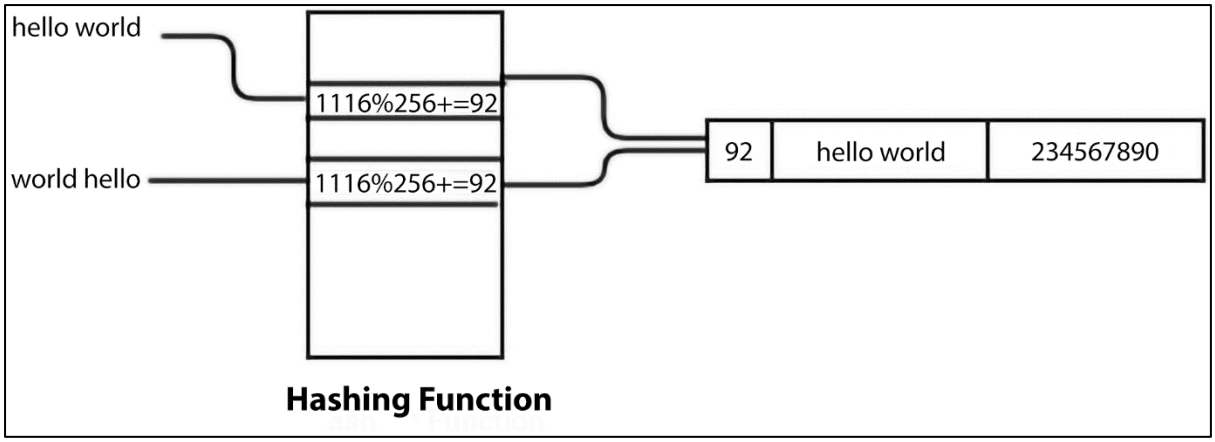


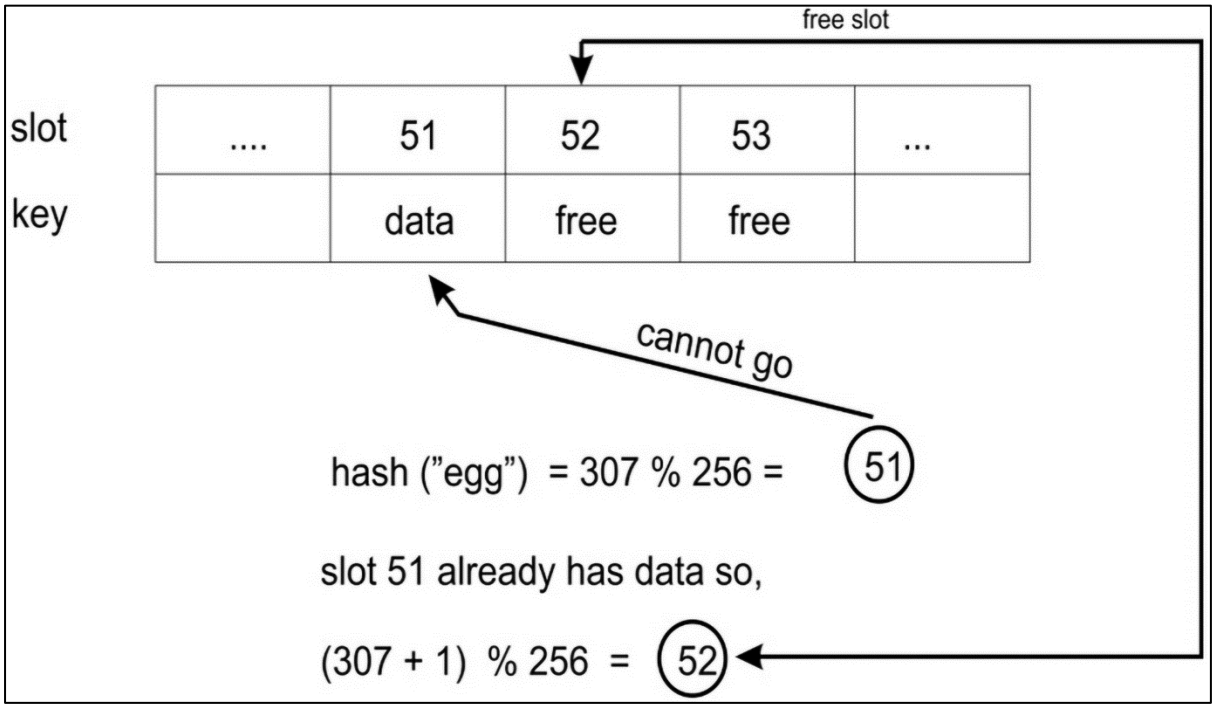
h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116

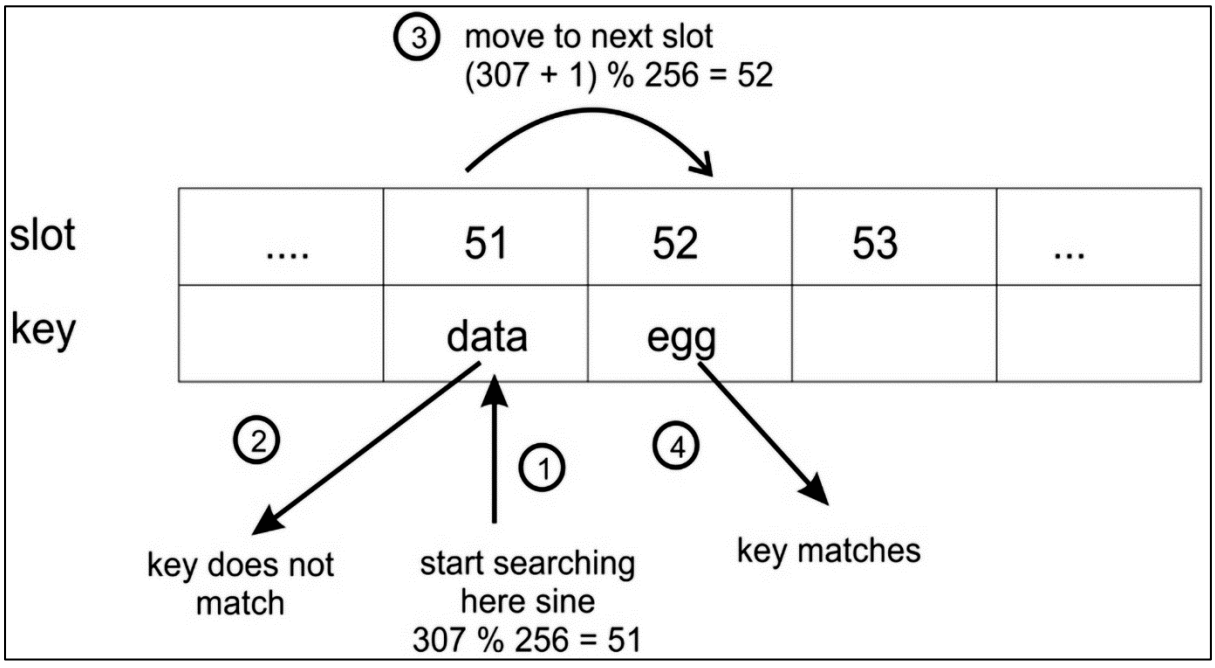
g	e	l	l	o		x	o	r	l	d	
103	101	108	108	111	32	120	111	114	108	100	= 1116
-1						+1					

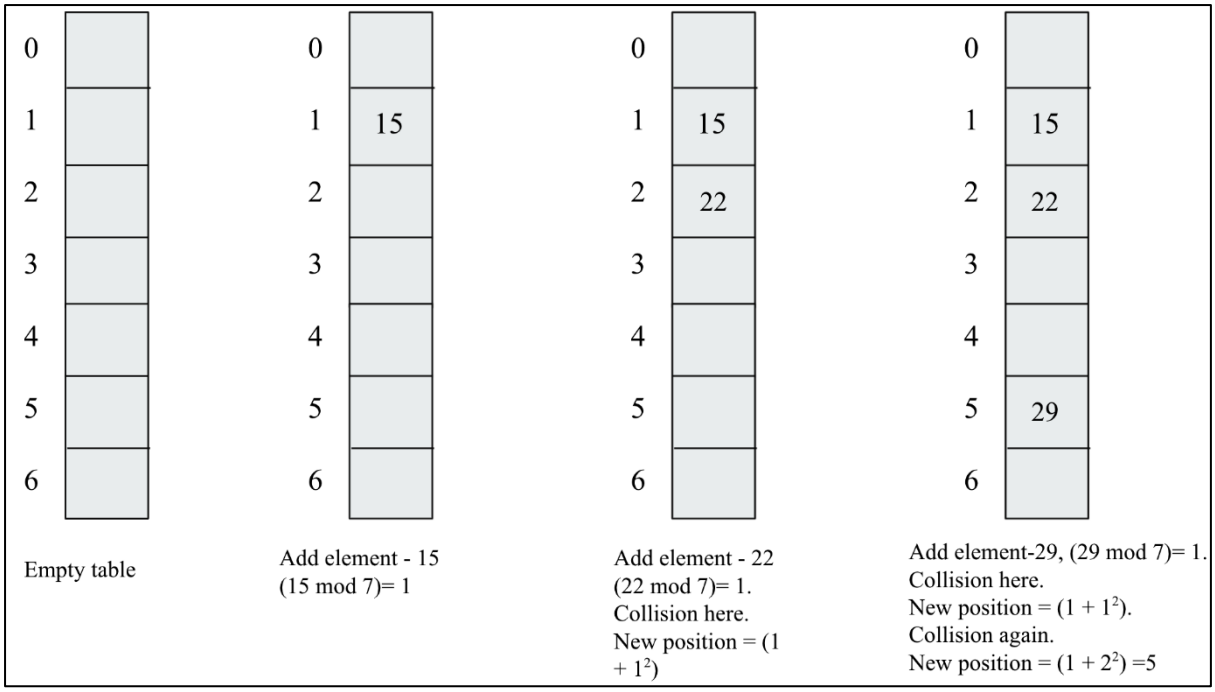
h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116
1	2	3	4	5	6	7	8	9	10	11	
104	202	324	432	555	192	833	888	1026	1080	1100	= 6736

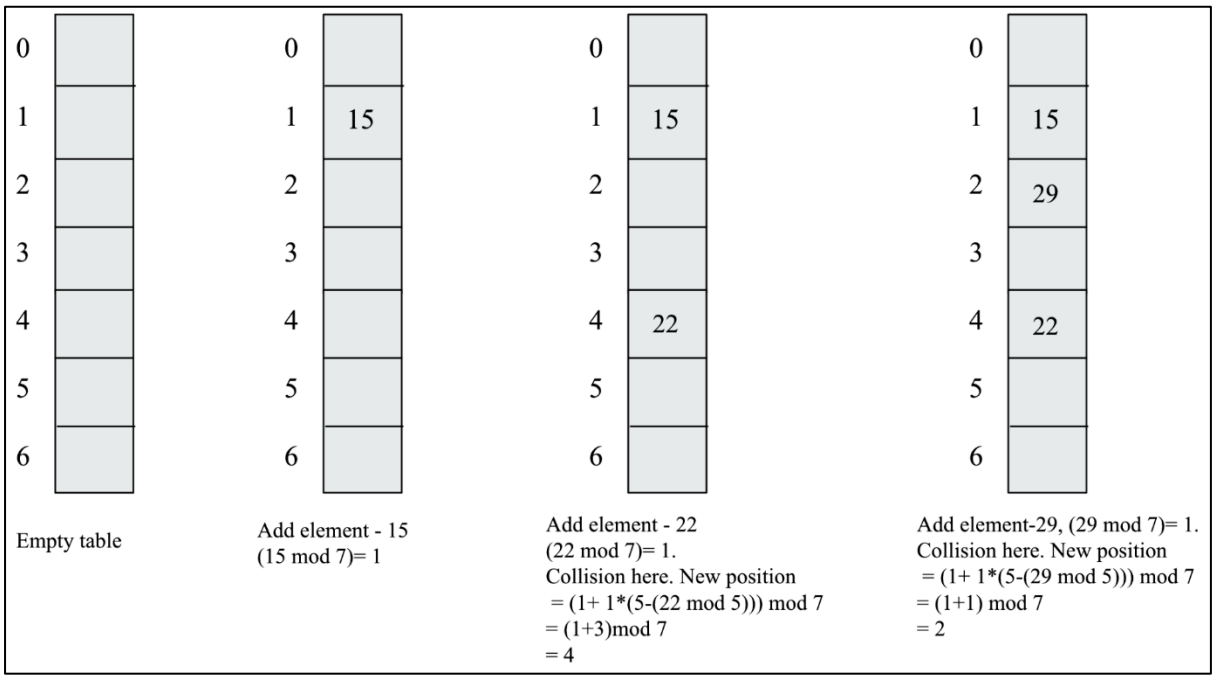


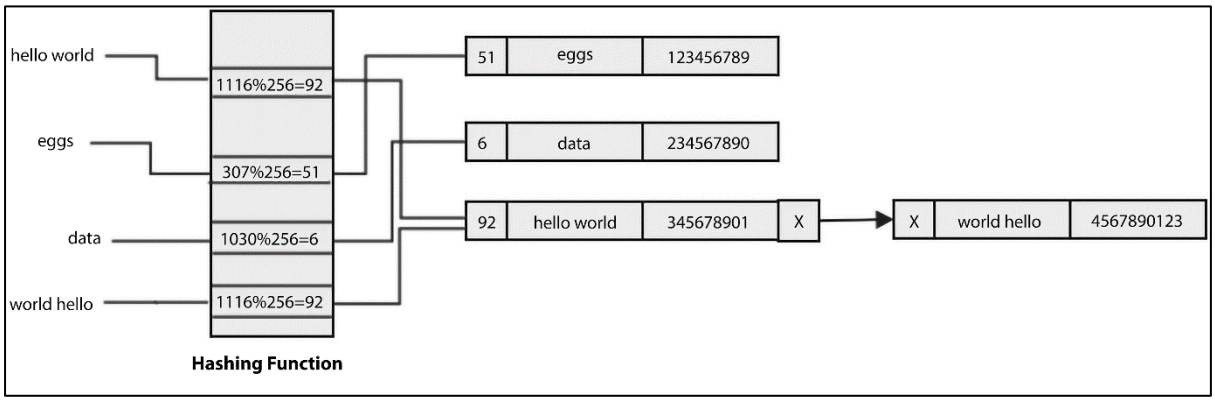

hello world
1116%256+=92
world hello
1116%256+=92
92
hello world
234567890
Hashing Function

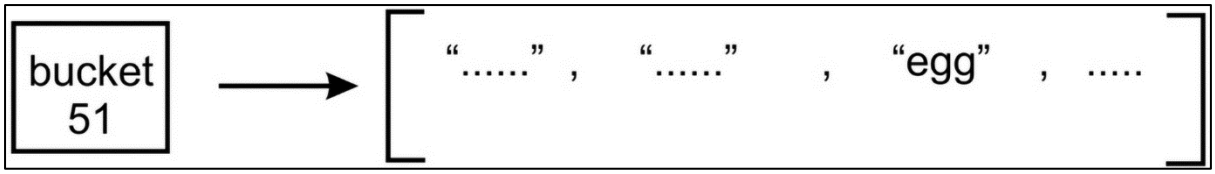

free slot
slot
key
....
51
52
53
...
data
free
free
cannot go
hash ("egg") = 307 % 256 =
51
slot 51 already has data so,
(307 + 1) % 256 =
52

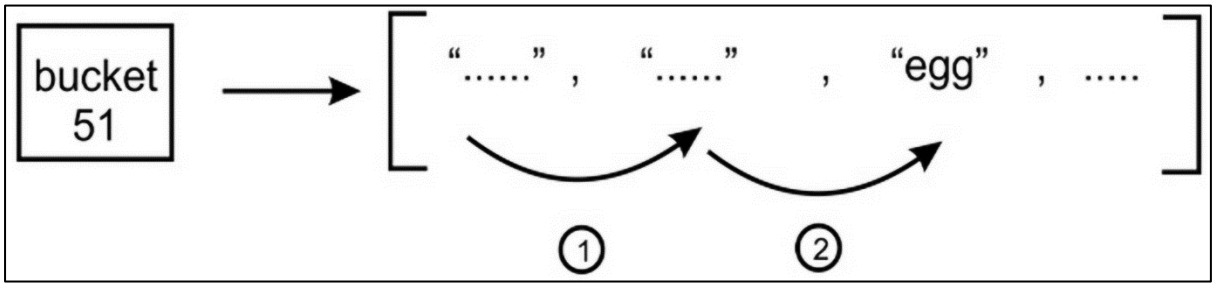

3
move to next slot
(307 + 1) % 256 = 52
slot
key
....
51
52
53
...
data
egg
2
1
4
key does not match
start searching here sine 307 % 256 = 51
key matches

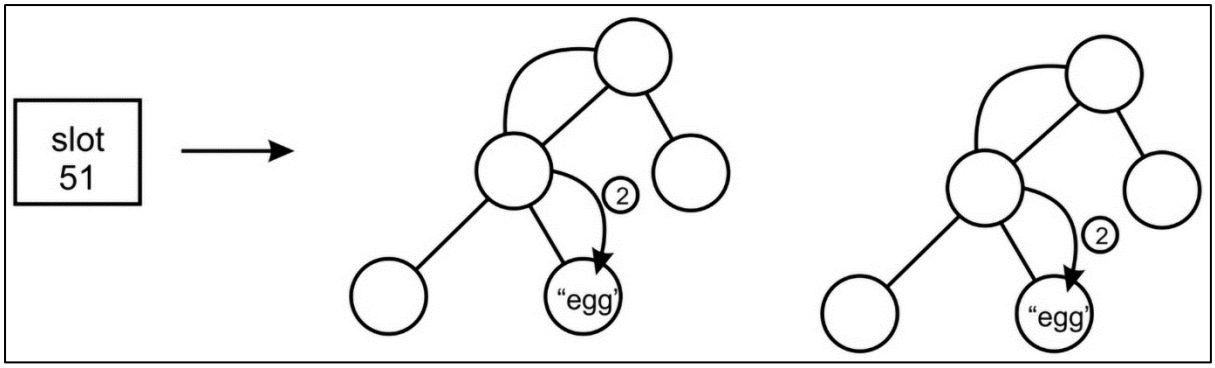

0
1
2
3
4
5
6
15
22
29
Empty table
Add element - 15
(15 mod 7)= 1
Add element - 22
(22 mod 7)= 1.
Collision here.
New position = (1 + 1²)
Add element-29, (29 mod 7)= 1.
Collision here.
New position = (1 + 1²).
Collision again.
New position = (1 + 2²) =5

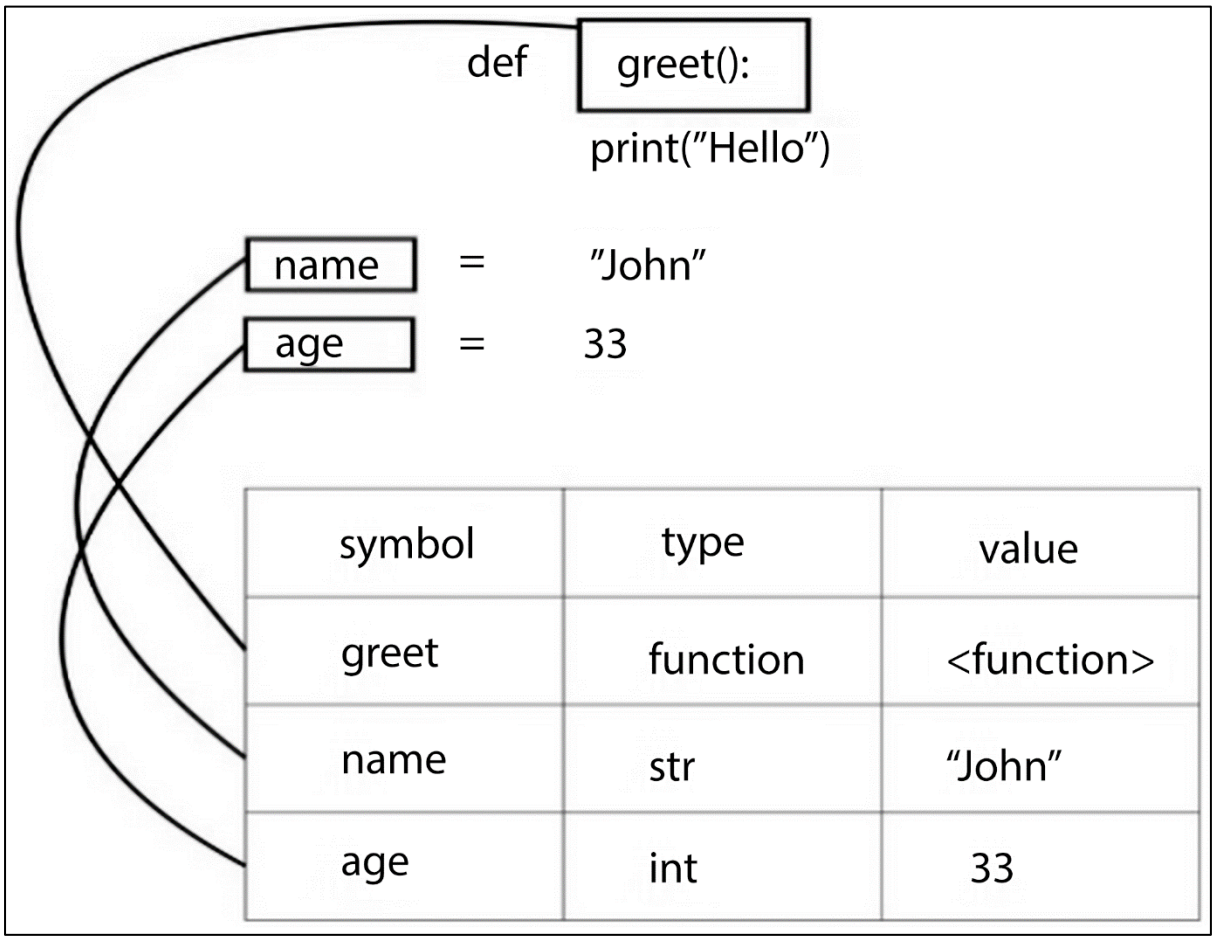

0
1
2
3
4
5
6
15
22
29
Empty table
Add element - 15
(15 mod 7)= 1
Add element - 22
(22 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(22 mod 5))) mod 7
= (1+3)mod 7
= 4
Add element-29, (29 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(29 mod 5))) mod 7
= (1+1) mod 7
= 2


hello world
eggs
data
world hello
1116%256=92
307%256=51
1030%256=6
1116%256=92
Hashing Function
51
eggs
123456789
6
data
234567890
92
hello world
345678901
X
X
world hello
4567890123



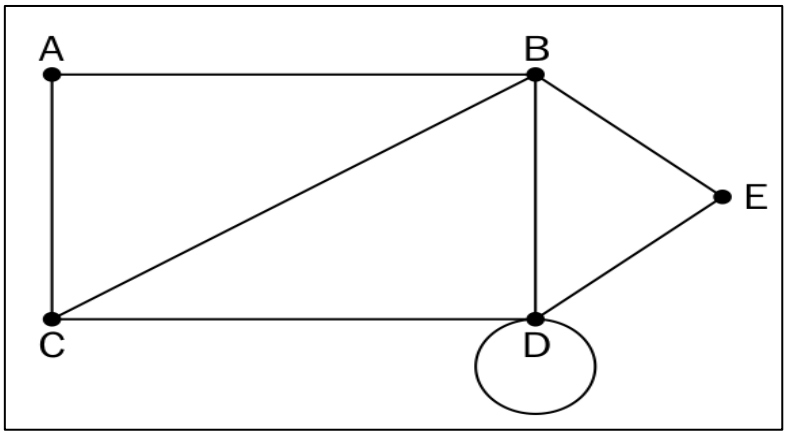


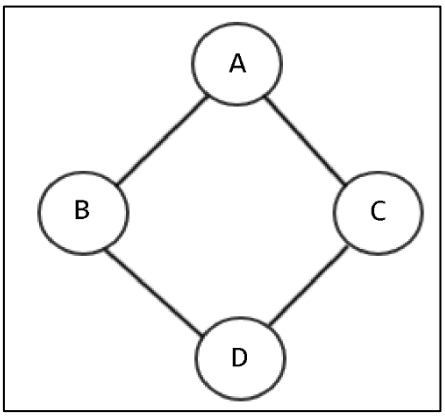


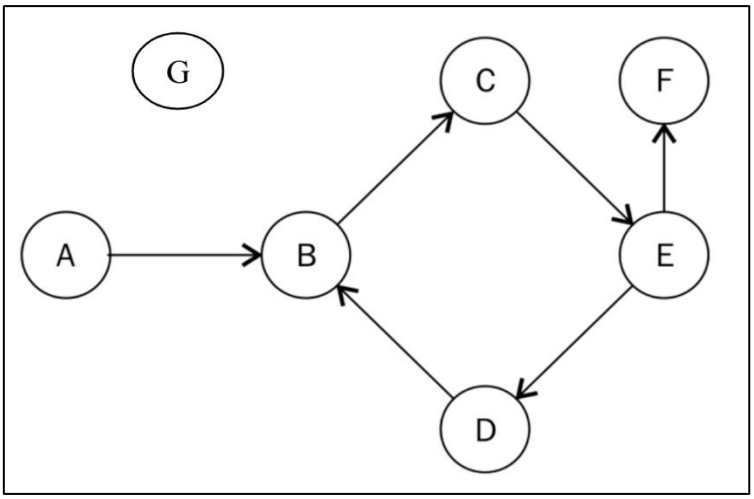


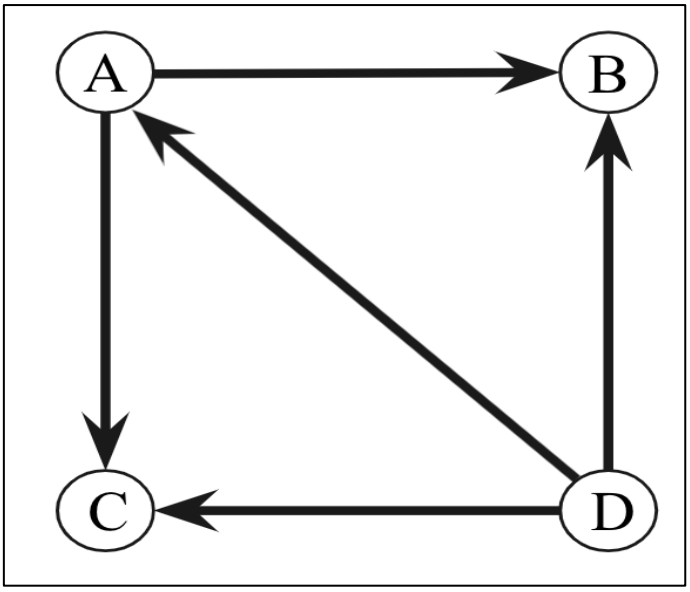
symbol	type	value
greet	function	<function>
name	str	“John”
age	int	33

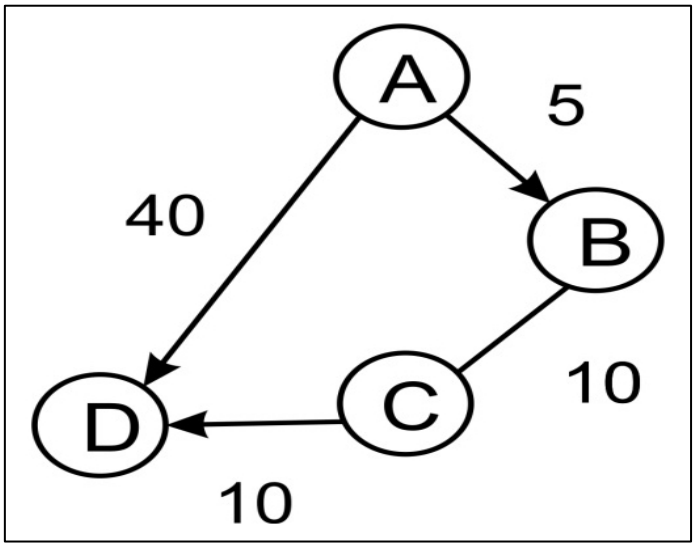
Chapter 9: Graphs and Algorithms

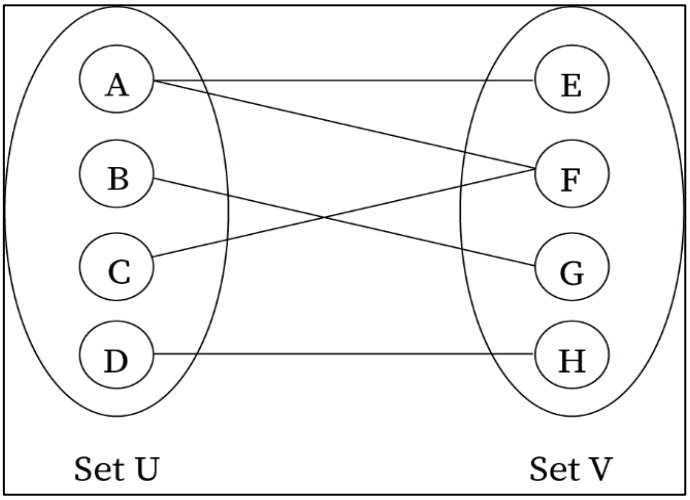


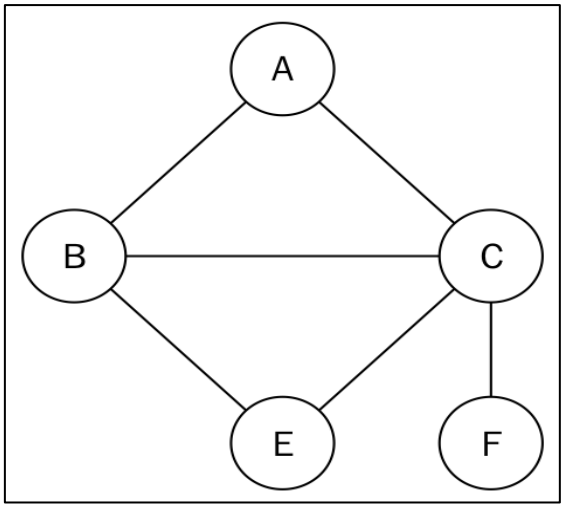


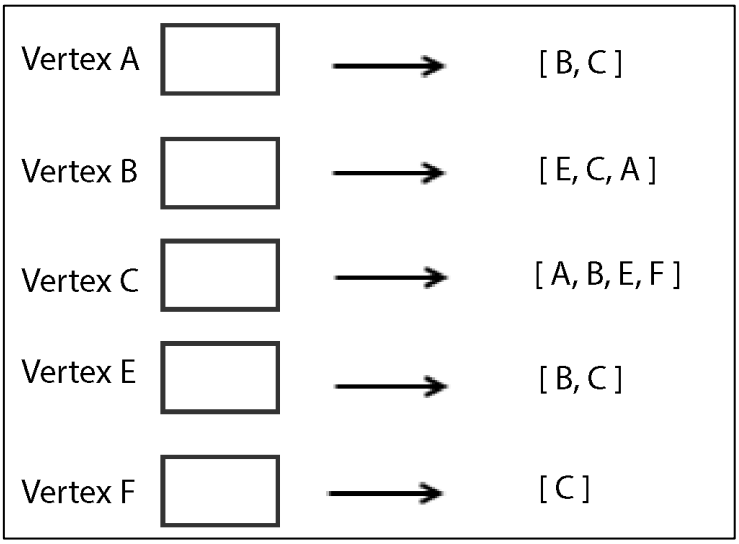


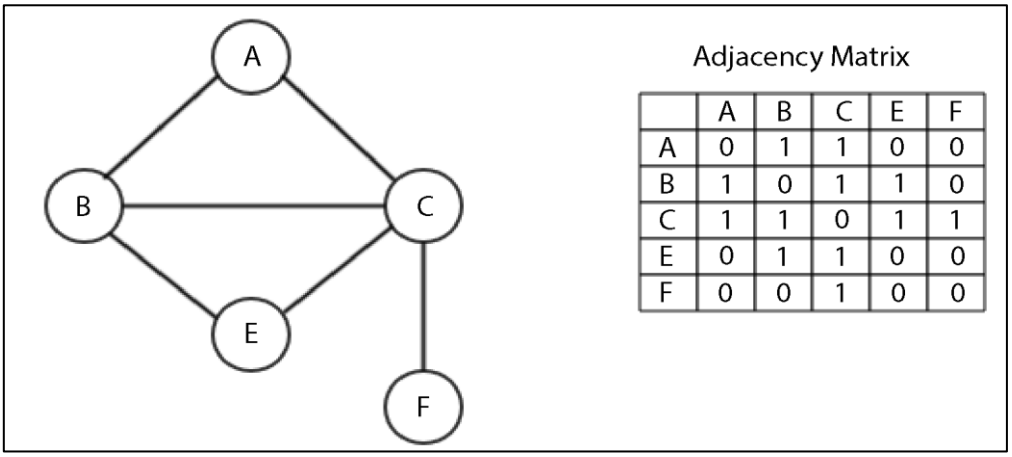

A
B
C
D


A
5
40
B
10
C
D
10

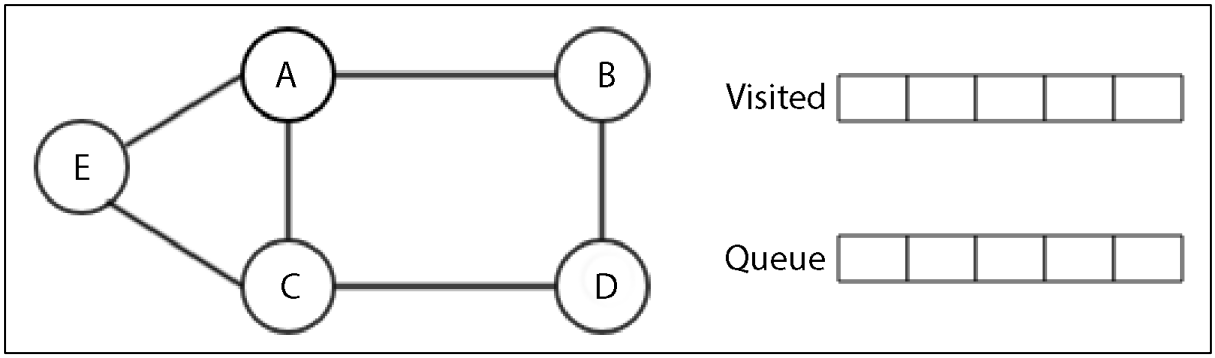

A
B
C
D
E
F
G
H
Set U
Set V

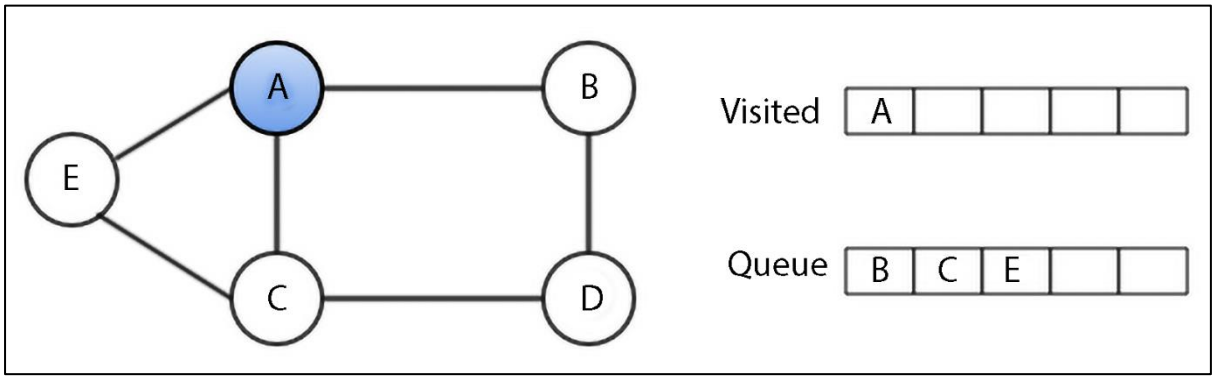


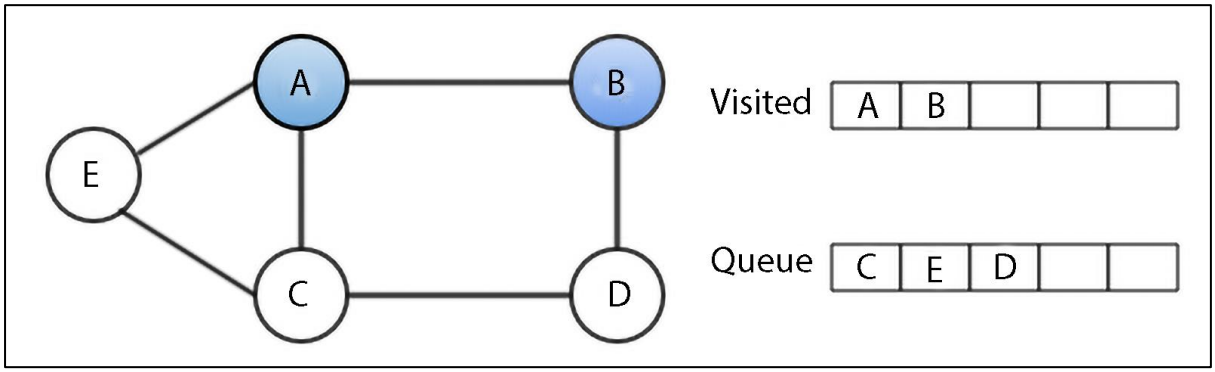


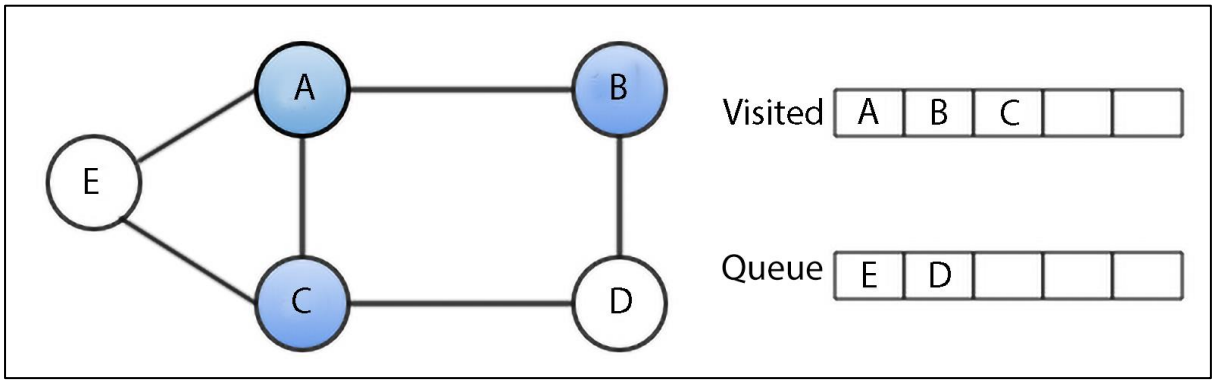


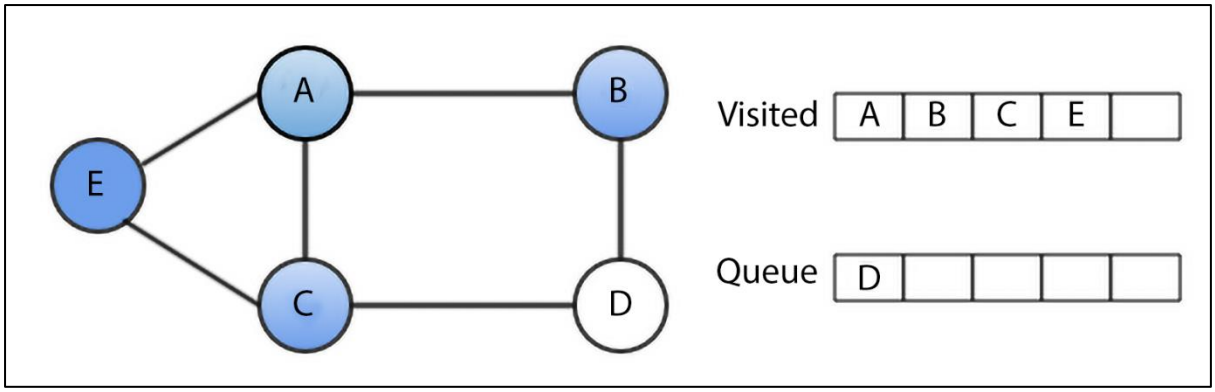
	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

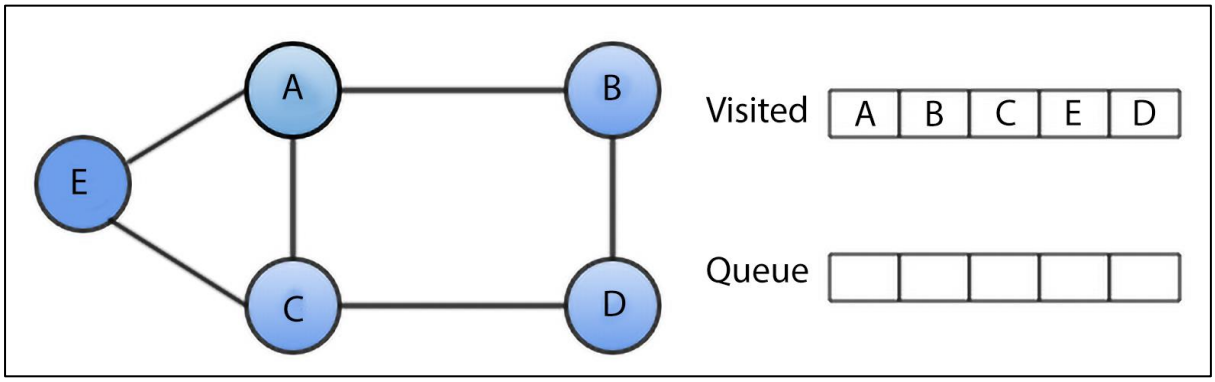

A
B
E
C
D
Visited
Queue

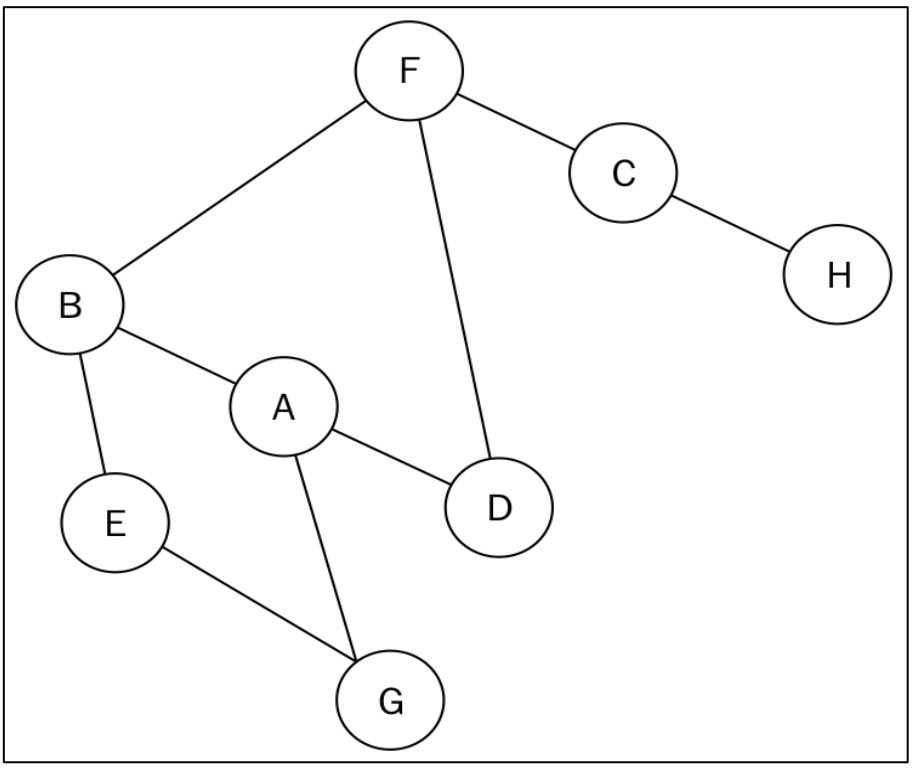

A
B
E
C
D
Visited
A
Queue
B
C
E

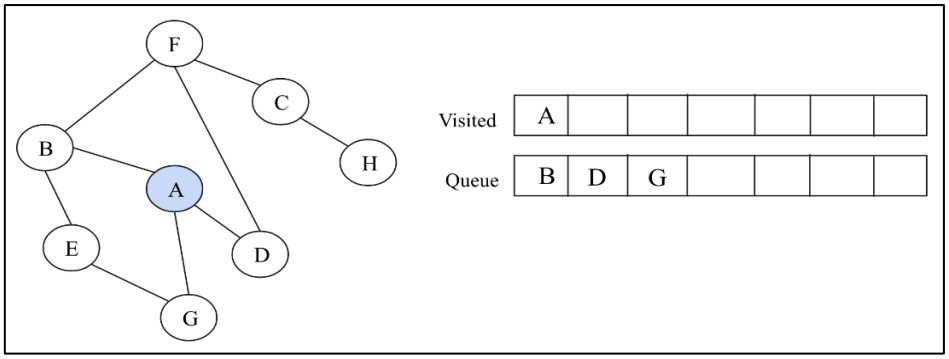

A
B
E
C
D
Visited
A
B
Queue
C
E
D

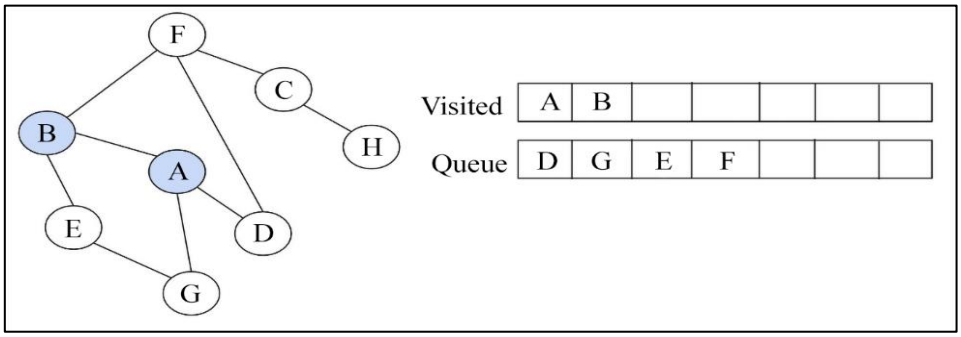

A
B
E
C
D
Visited
A
B
C
Queue
E
D

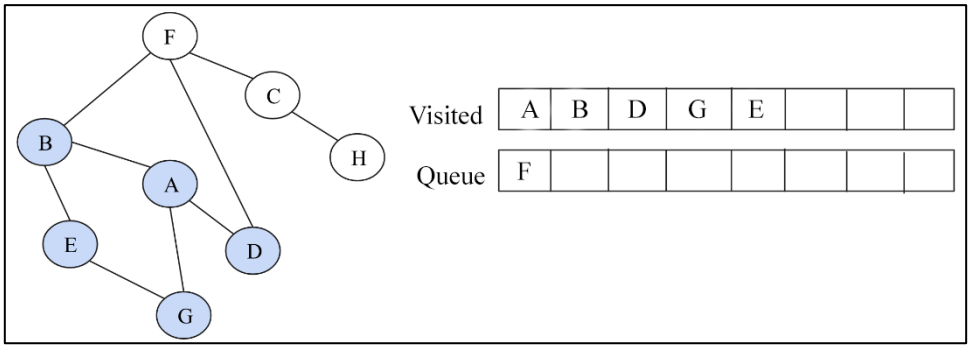

A
B
E
C
D
Visited
A
B
C
E
Queue
D

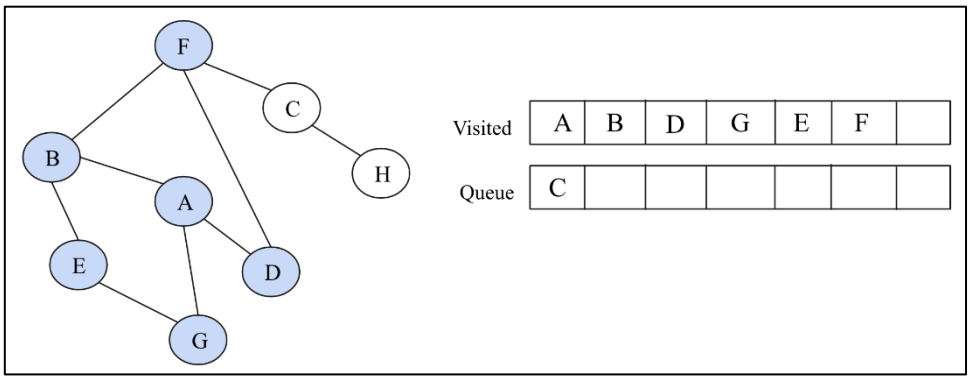

A
B
E
C
D
Visited
A
B
C
E
D
Queue

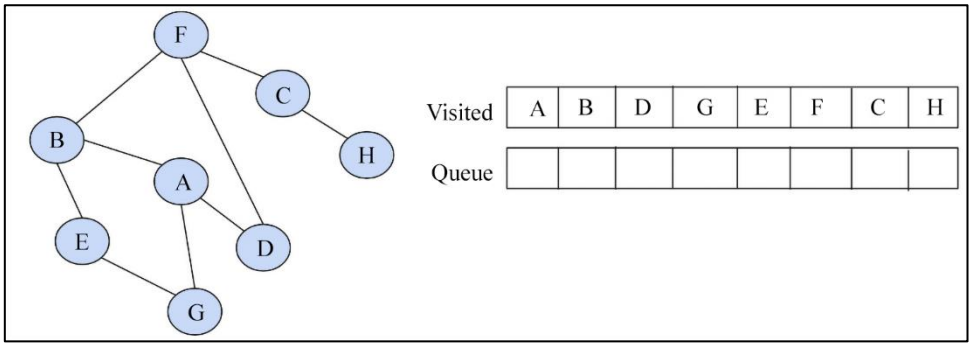

F
C
H
B
A
E
D
G

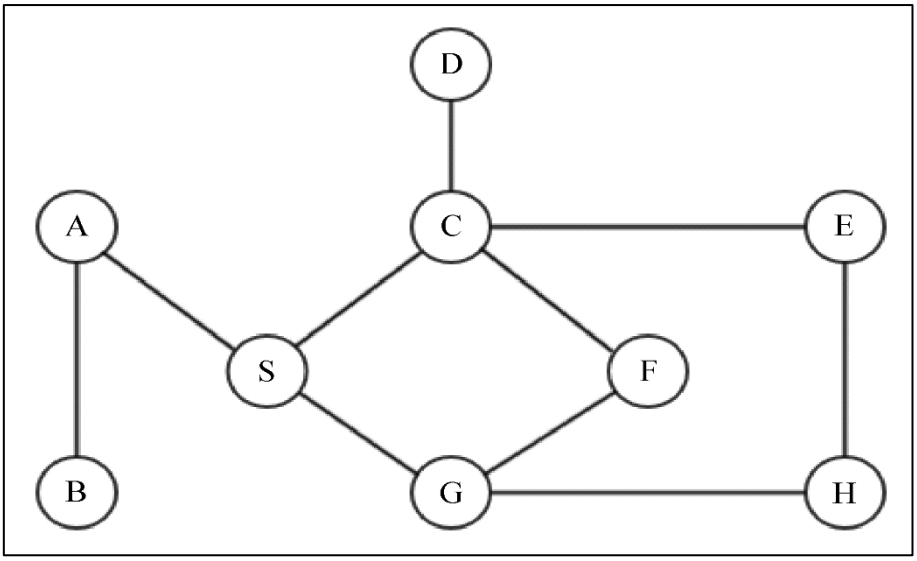

F
C
H
B
A
E
D
G
Visited
A
Queue
B
D
G

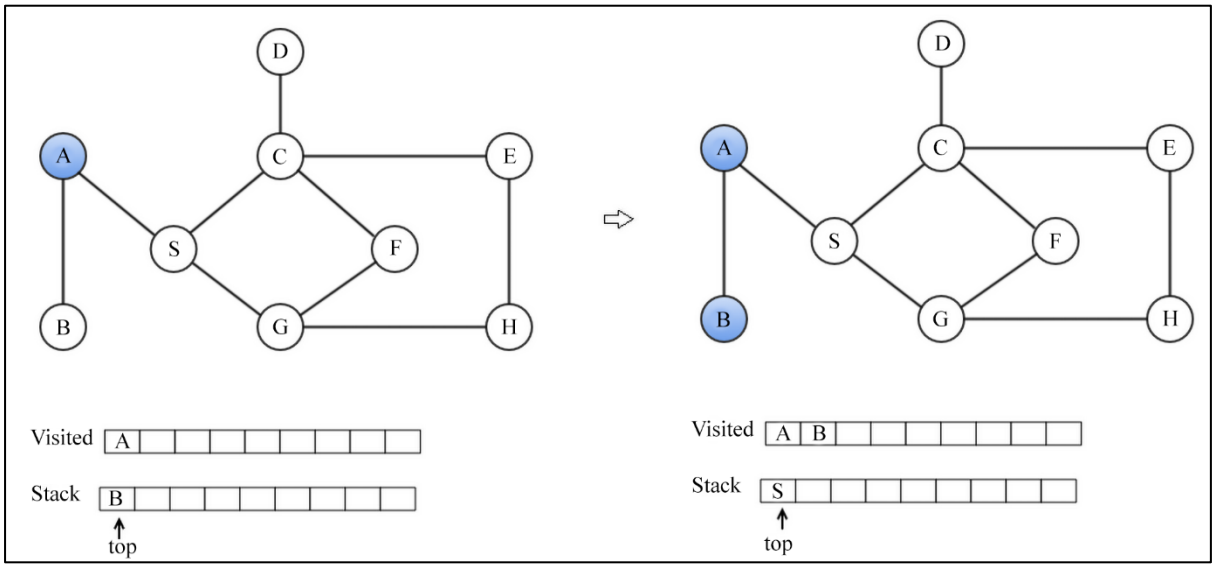

F
C
H
B
A
E
D
G
Visited
A
B
Queue
D
G
E
F

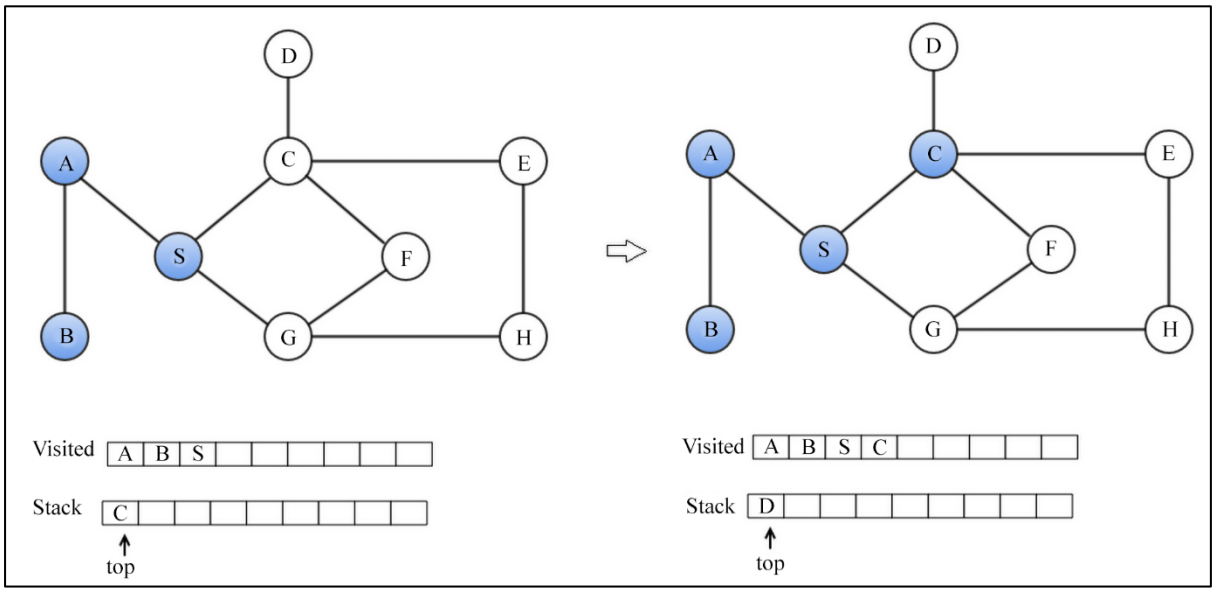

F
C
H
B
A
E
D
G
Visited
A
B
D
G
E
Queue
F

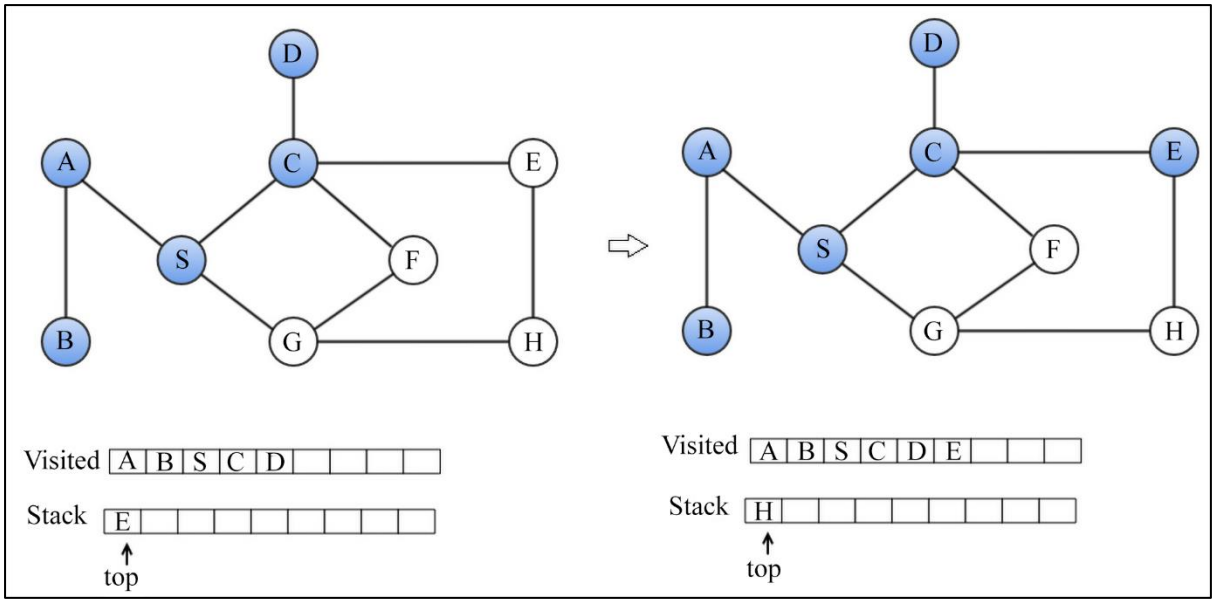

F
C
H
B
A
E
D
G
Visited
A
B
D
G
E
F
Queue
C

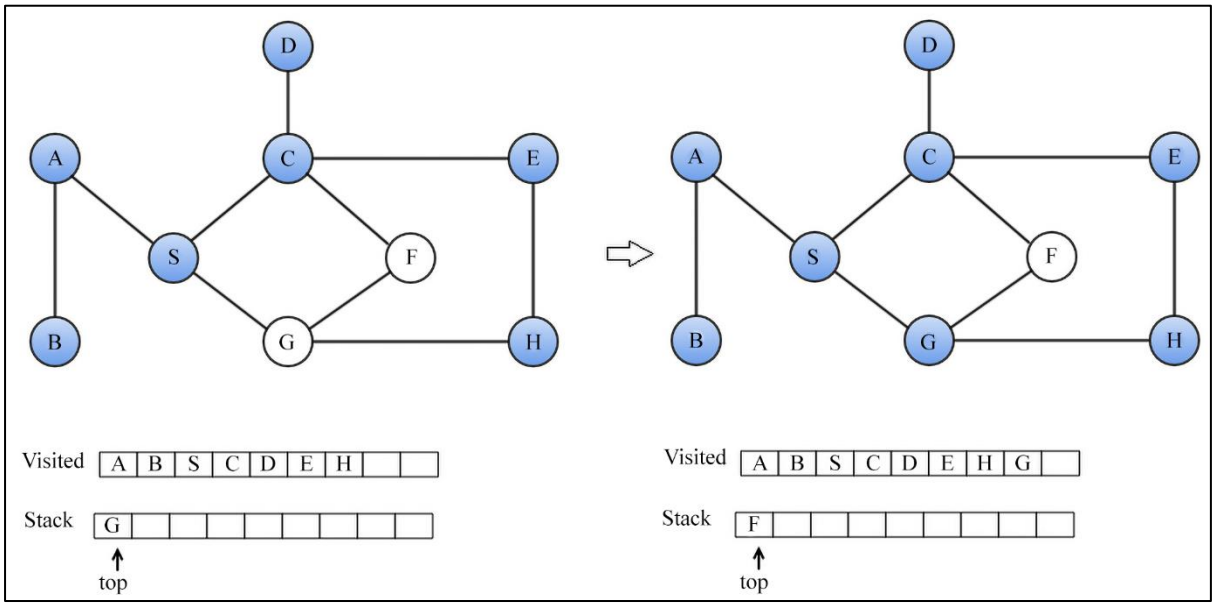

F
C
H
B
A
E
D
G
Visited
A
B
D
G
E
F
C
H
Queue

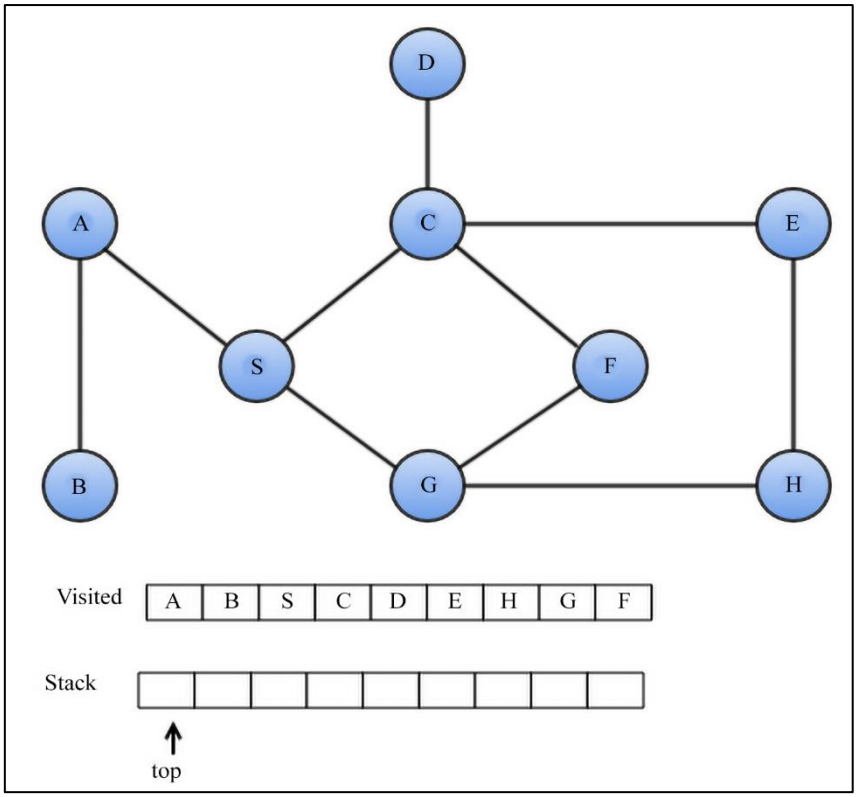

D
A
C
E
S
F
B
G
H

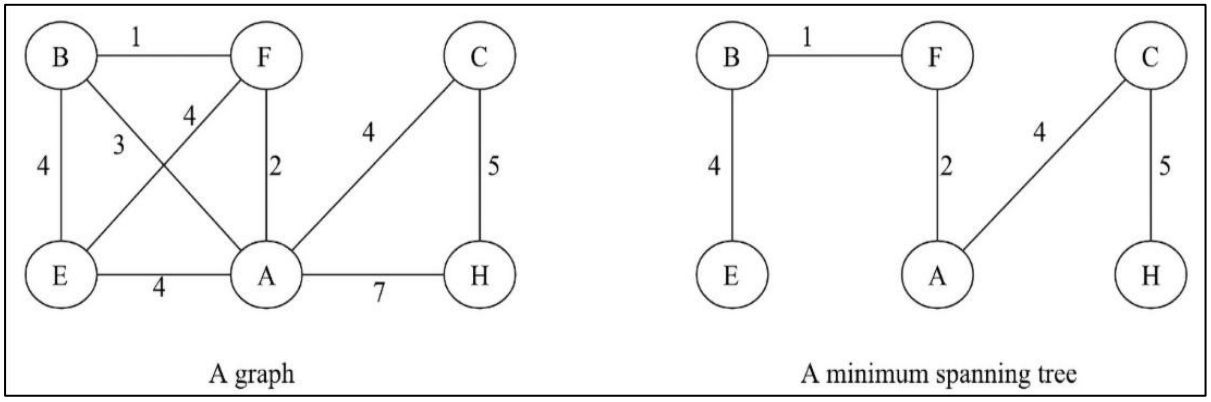

D
A
C
E
S
F
B
G
H
Visited
A
Stack
B
top
D
A
C
E
S
F
B
G
H
Visited
A
B
Stack
S
top

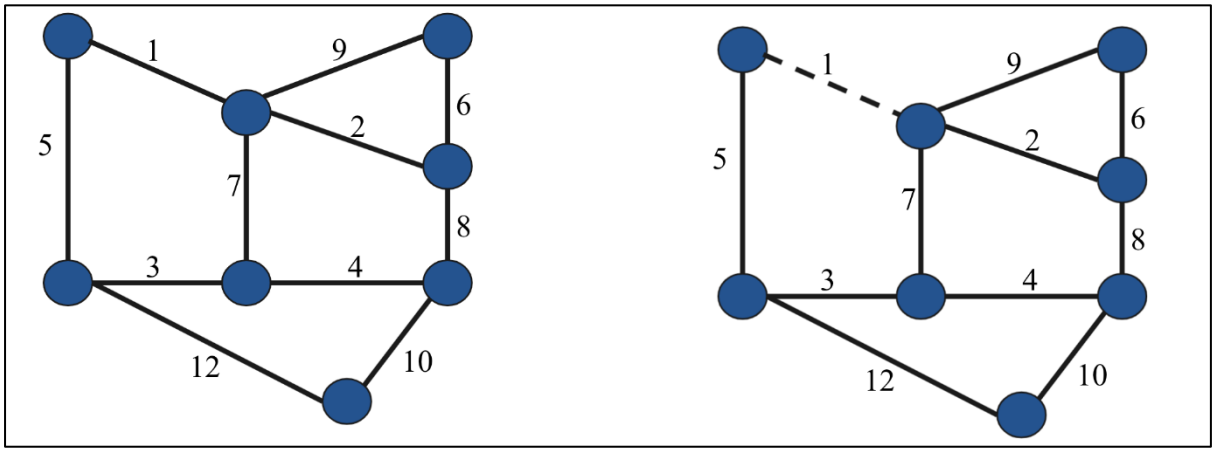

D
A
C
E
S
F
B
G
H
Visited
A
B
S
Stack
C
top
D
A
C
E
S
F
B
G
H
Visited
A
B
S
C
Stack
D
top

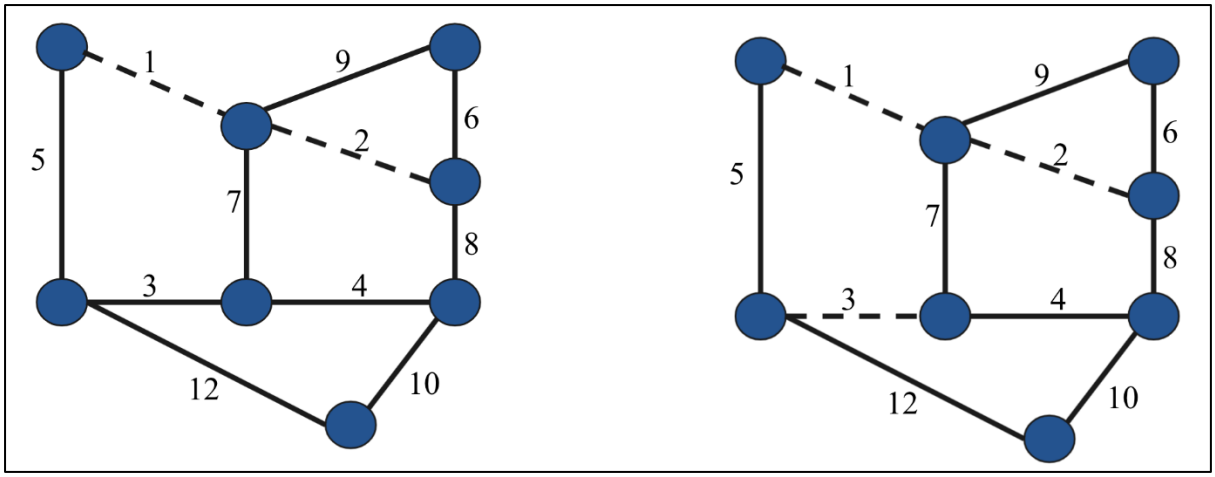
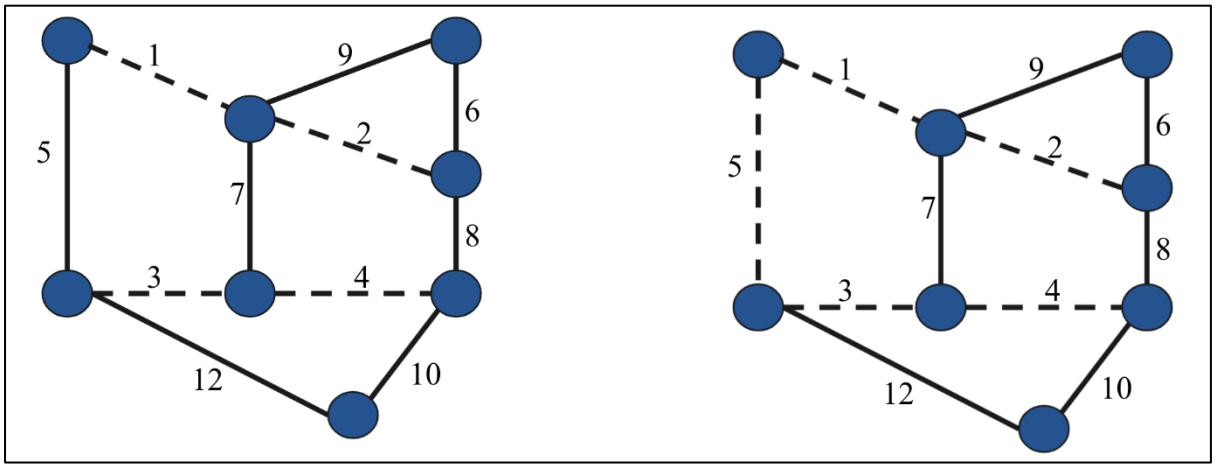
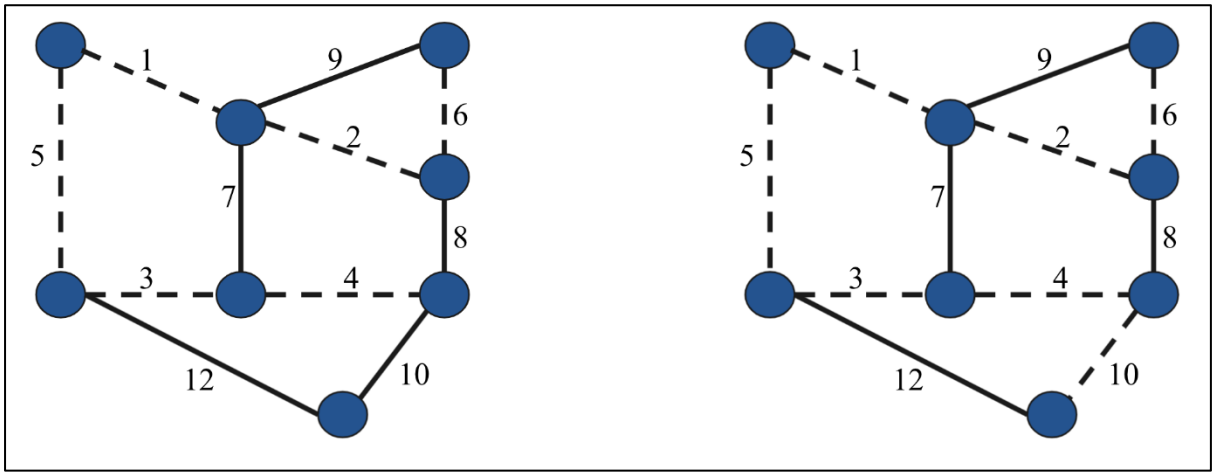

D
A
C
E
S
F
B
G
H
Visited
A B S C D
Stack
E
top
D
A
C
E
S
F
B
G
H
Visited
A B S C D E
Stack
H
top

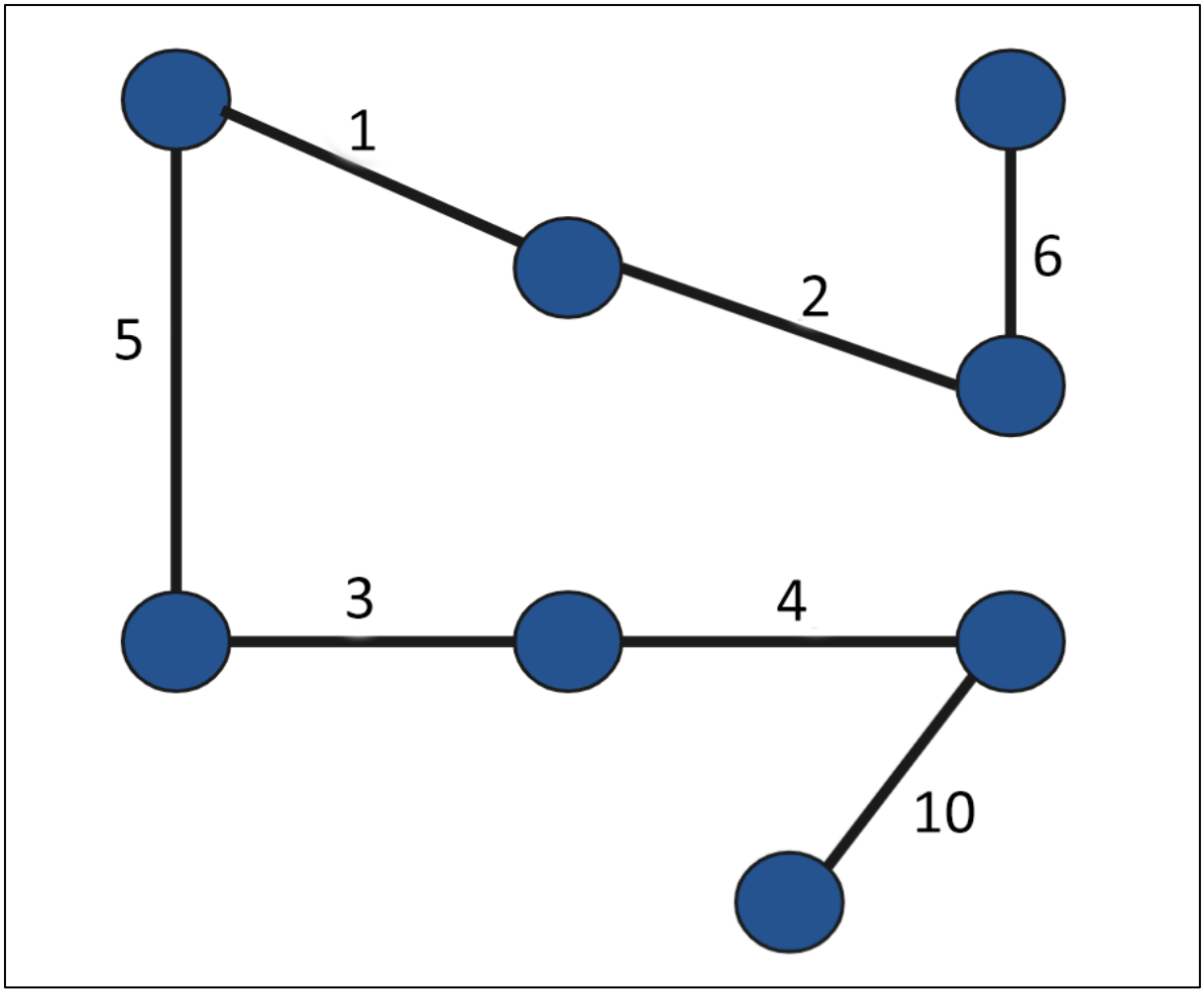

D
A
C
E
S
F
B
G
H
Visited
A B S C D E H
Stack
G
top
D
A
C
E
S
F
B
G
H
Visited
A B S C D E H G
Stack
F
top

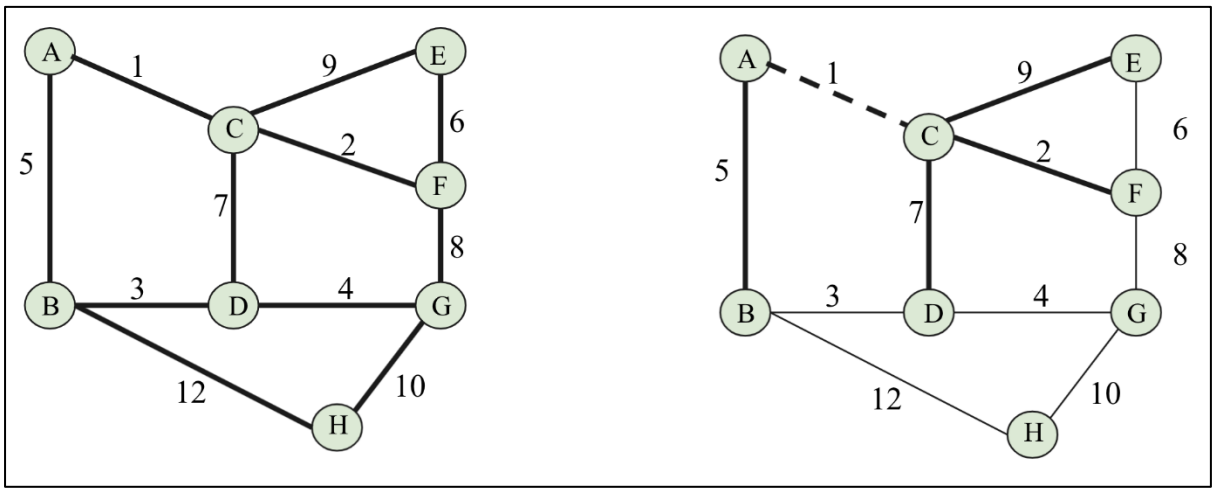

D
A
C
E
S
F
B
G
H
Visited
A B S C D E H G F
Stack
top

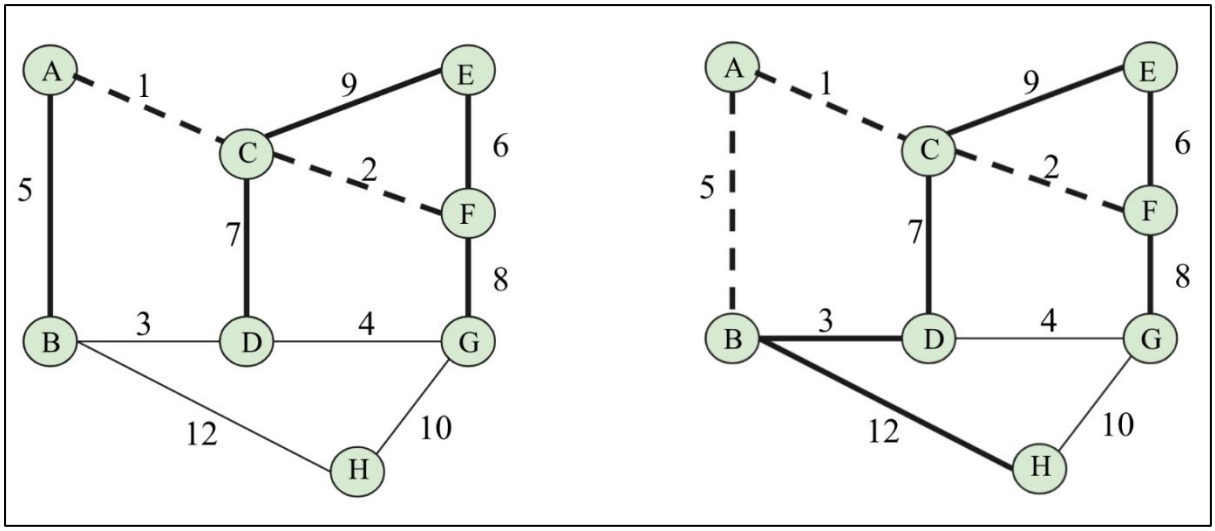

B
F
C
E
A
H
1
4
3
4
2
4
5
4
7
A graph
A minimum spanning tree

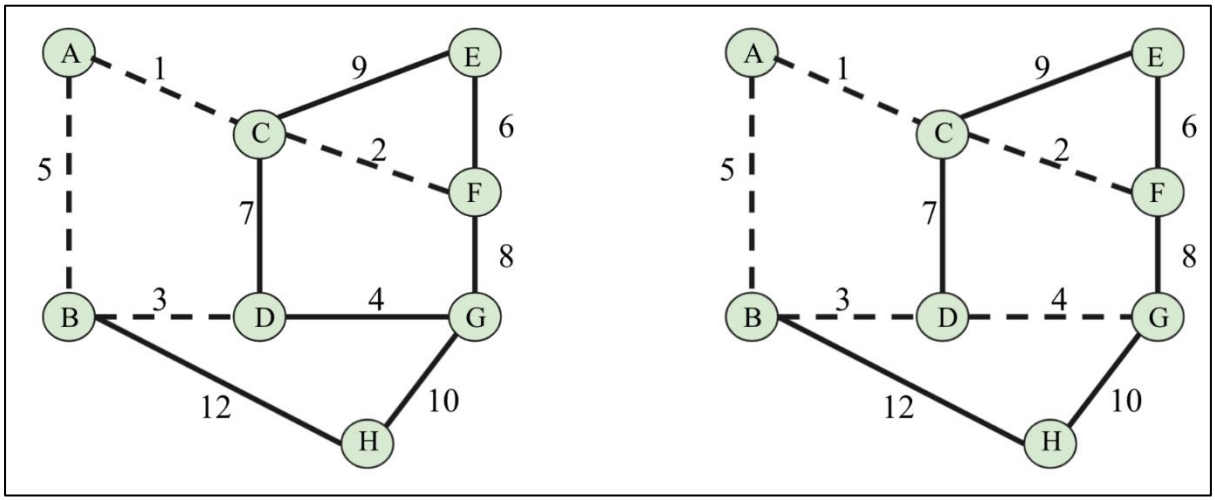

1
9
6
2
5
7
8
3
4
12
10

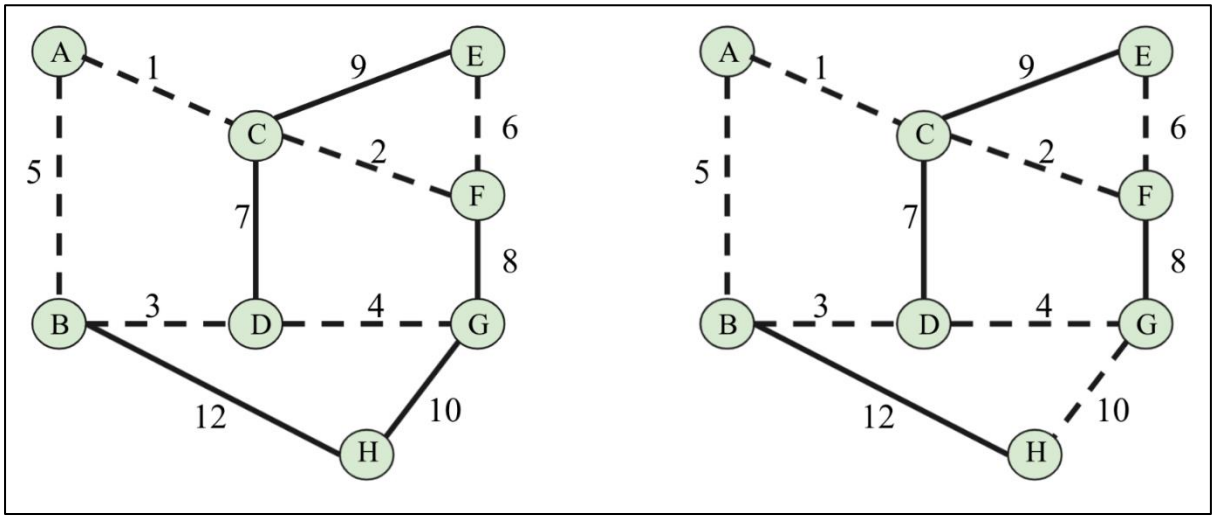

1
9
5
6
2
7
8
3
4
12
10

1
9
5
6
2
7
8
3
4
12
10

1
9
5
6
2
7
8
3
4
12
10

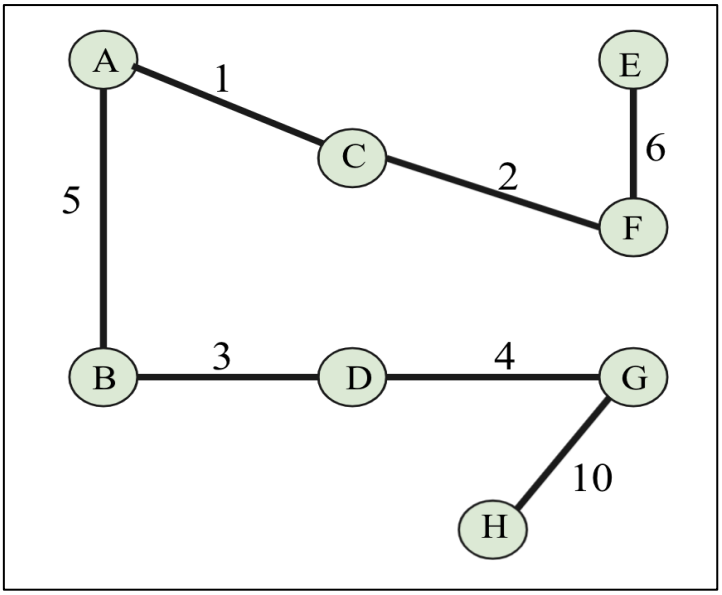

1
2
3
4
5
6
10

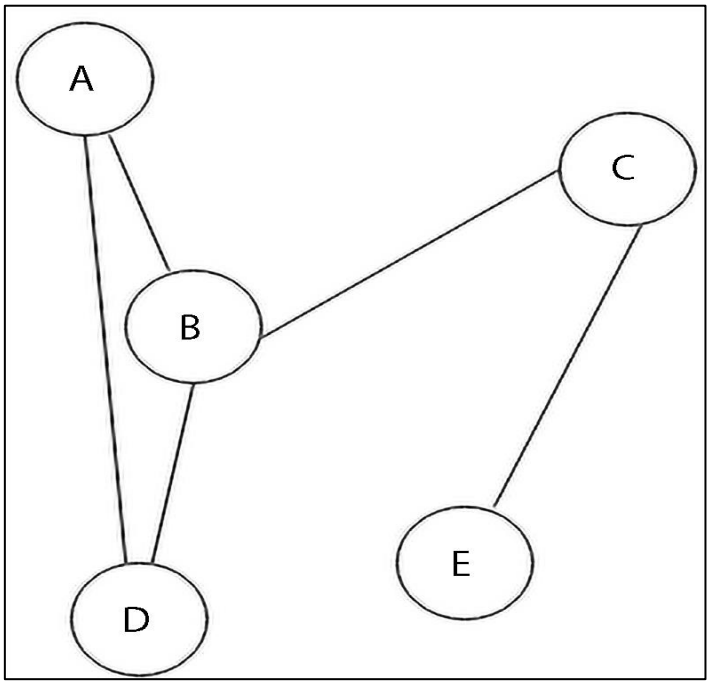

A
B
C
D
E
F
G
H
1
2
3
4
5
6
7
8
9
10
12


A
E
C
F
B
D
G
H
1
9
6
5
2
7
8
3
4
12
10

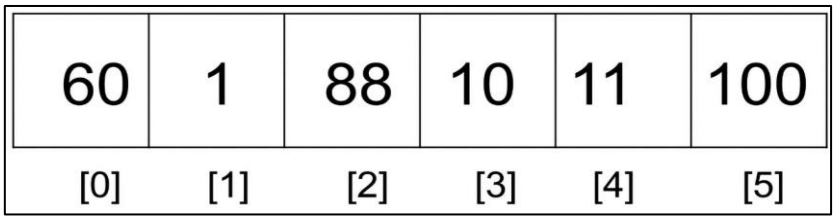

A
E
C
F
B
D
G
H
1
9
6
5
2
7
8
3
4
12
10

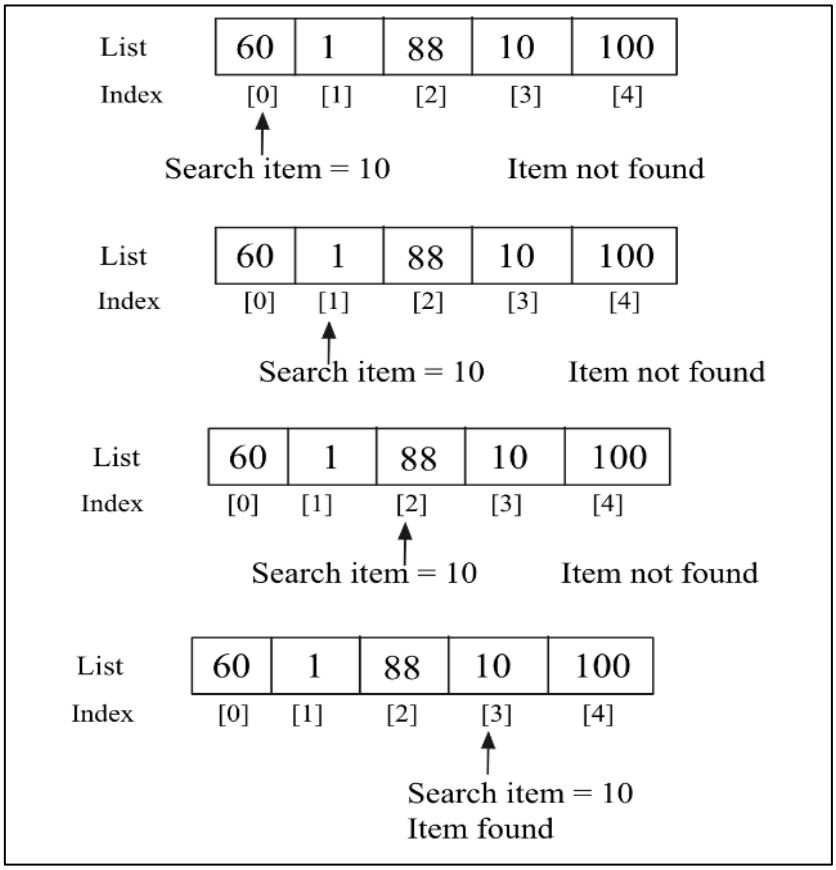

A
E
C
F
B
D
G
H
1
9
6
5
2
7
8
3
4
12
10

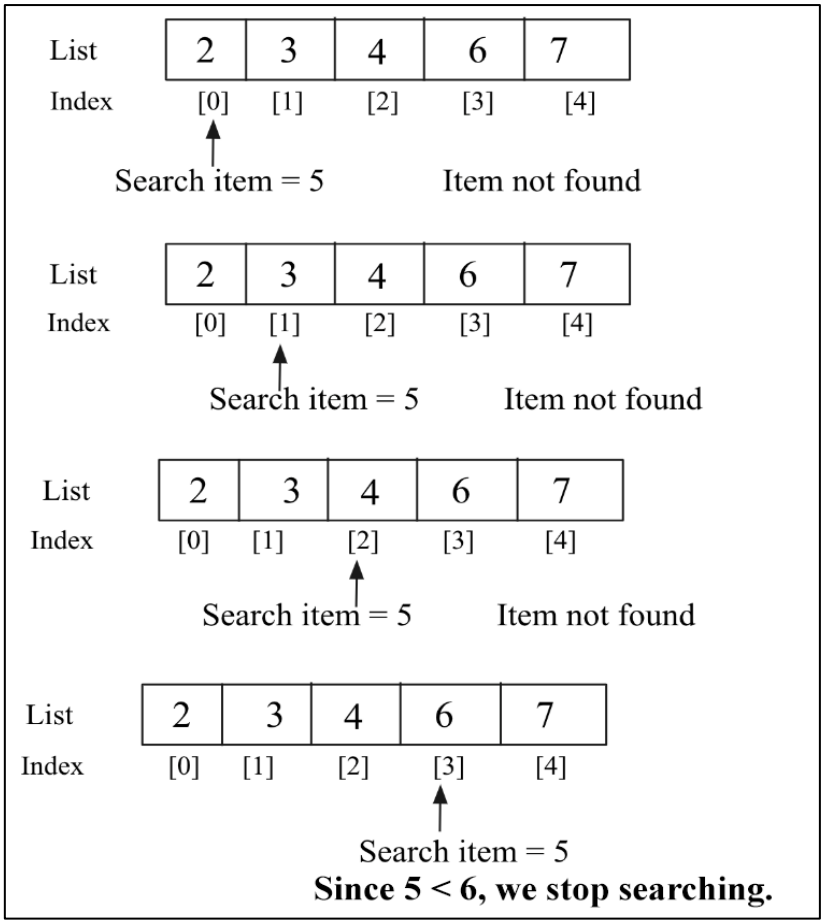

A
1
E
C
6
2
5
F
B
3
D
4
G
10
H

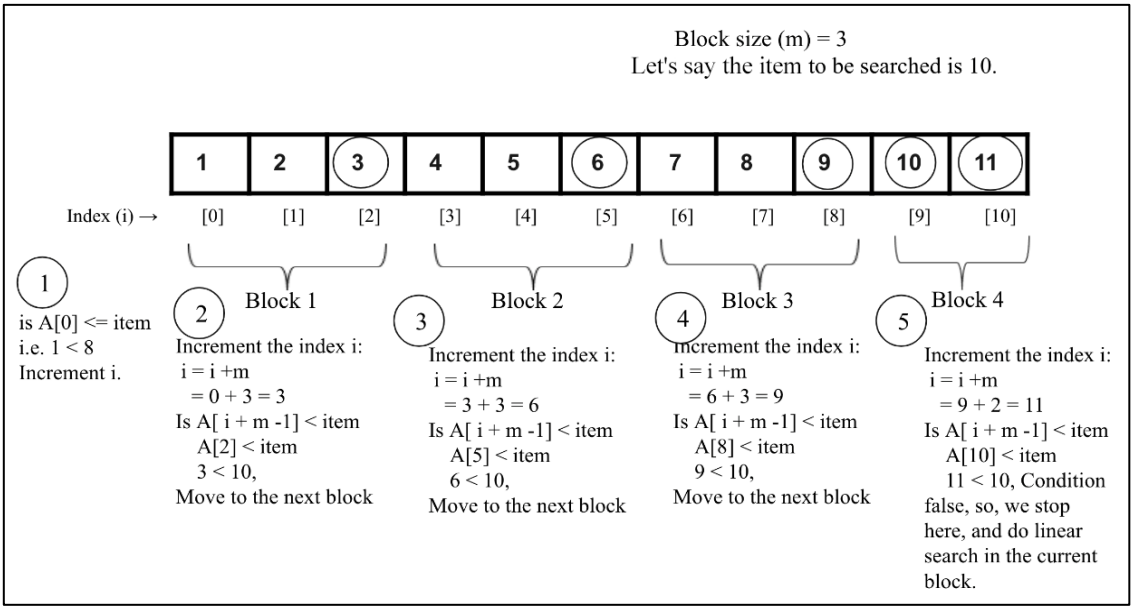

A
C
B
E
D

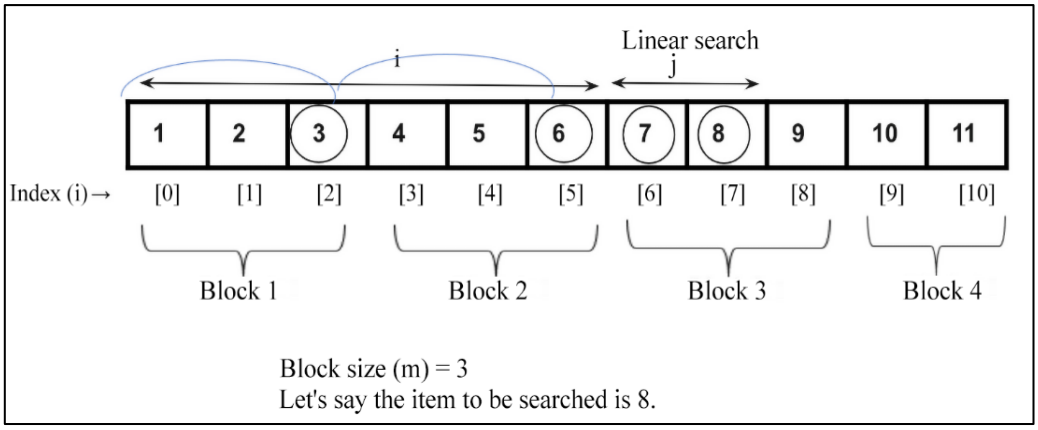
Chapter 10: Searching

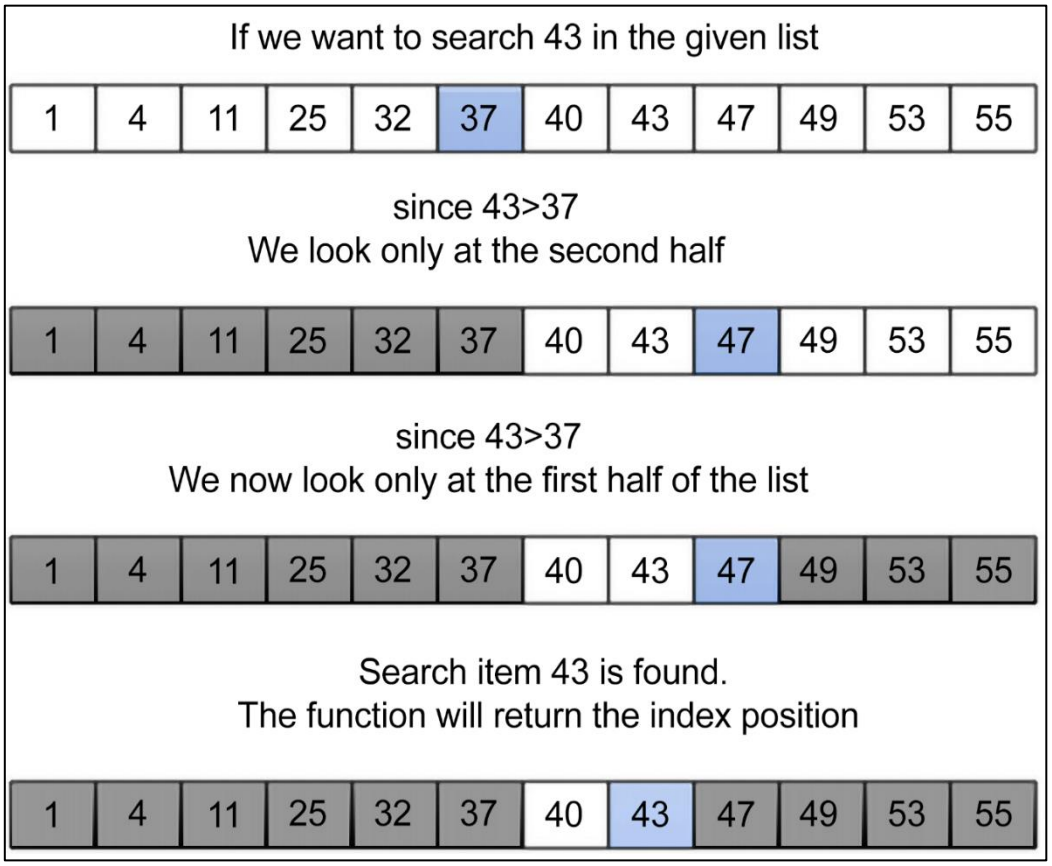


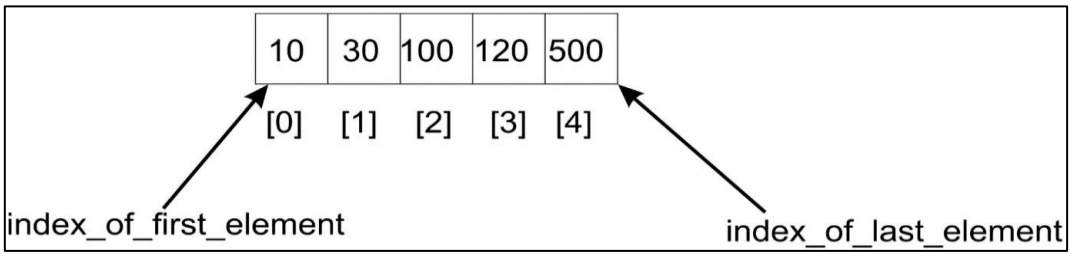


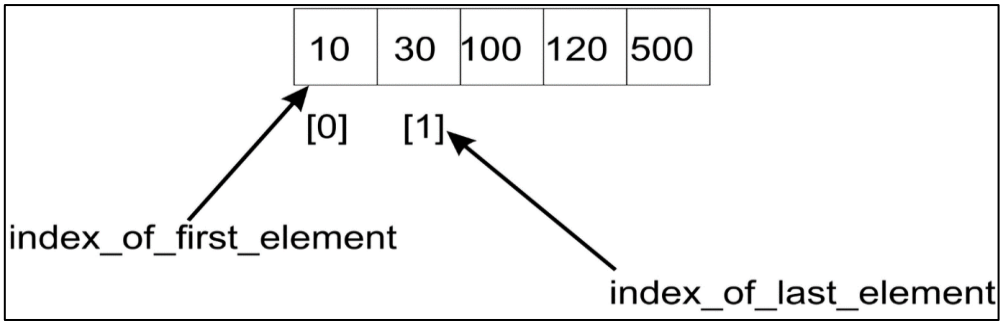

List
2 3 4 6 7
Index
[0] [1] [2] [3] [4]
Search item = 5
Item not found
List
2 3 4 6 7
Index
[0] [1] [2] [3] [4]
Search item = 5
Item not found
List
2 3 4 6 7
Index
[0] [1] [2] [3] [4]
Search item = 5
Item not found
List
2 3 4 6 7
Index
[0] [1] [2] [3] [4]
Search item = 5
Since 5 < 6, we stop searching.

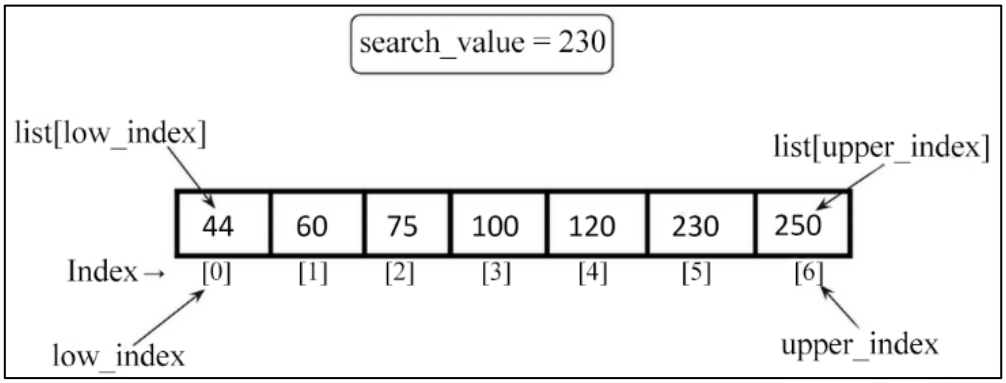

Block size (m) = 3
Let's say the item to be searched is 10.
1 2 3 4 5 6 7 8 9 10 11
Index (i) →
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
1
is A[0] <= item
i.e. 1 < 8
Increment i.
2
Block 1
Increment the index i:
i = i +m
= 0 + 3 = 3
Is A[i + m -1] < item
A[2] < item
3 < 10,
Move to the next block
3
Block 2
Increment the index i:
i = i +m
= 3 + 3 = 6
Is A[i + m -1] < item
A[5] < item
6 < 10,
Move to the next block
4
Block 3
Increment the index i:
i = i +m
= 6 + 3 = 9
Is A[i + m -1] < item
A[8] < item
9 < 10,
Move to the next block
5
Block 4
Increment the index i:
i = i +m
= 9 + 2 = 11
Is A[i + m -1] < item
A[10] < item
11 < 10, Condition false, so, we stop here, and do linear search in the current block.

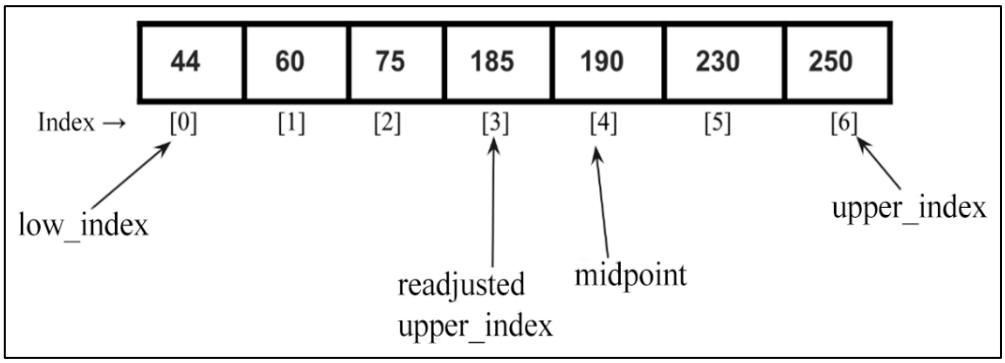

Linear search
i
j
1
2
3
4
5
6
7
8
9
10
11
Index (i)→
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
Block 1
Block 2
Block 3
Block 4
Block size (m) = 3
Let's say the item to be searched is 8.

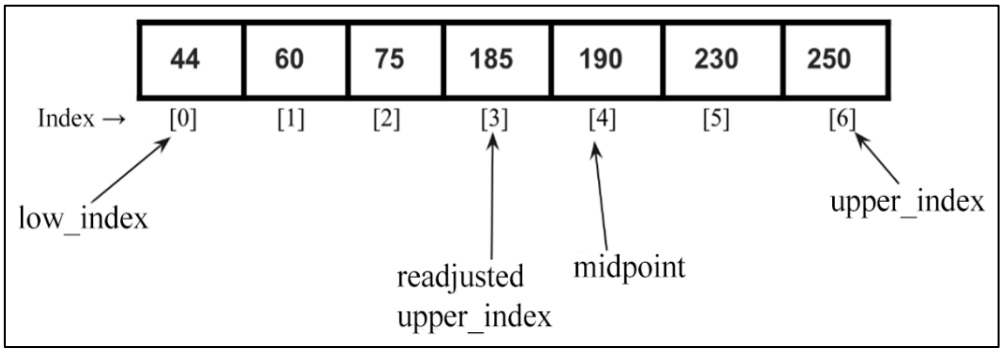

If we want to search 43 in the given list
1 4 11 25 32 37 40 43 47 49 53 55
since 43>37
We look only at the second half
1 4 11 25 32 37 40 43 47 49 53 55
since 43>37
We now look only at the first half of the list
1 4 11 25 32 37 40 43 47 49 53 55
Search item 43 is found.
The function will return the index position
1 4 11 25 32 37 40 43 47 49 53 55

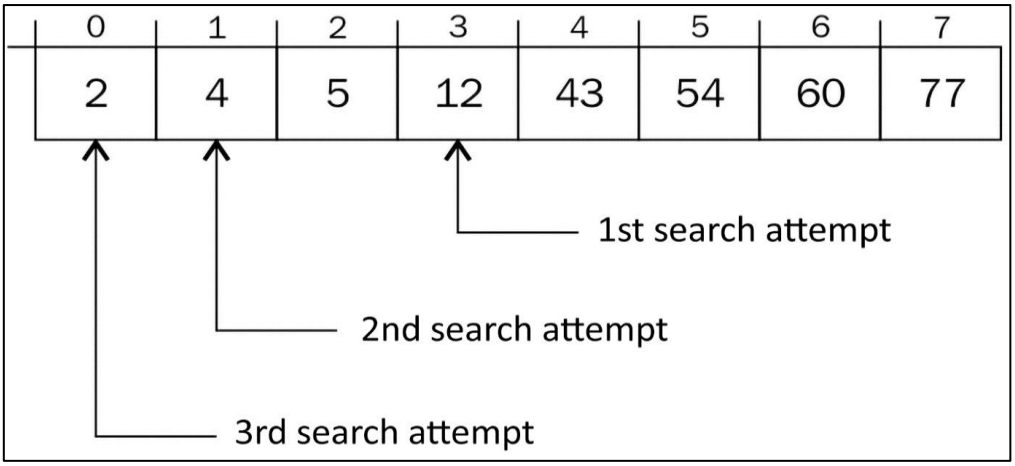

10
30
100
120
500
[0]
[1]
[2]
[3]
[4]
index_of_first_element
index_of_last_element

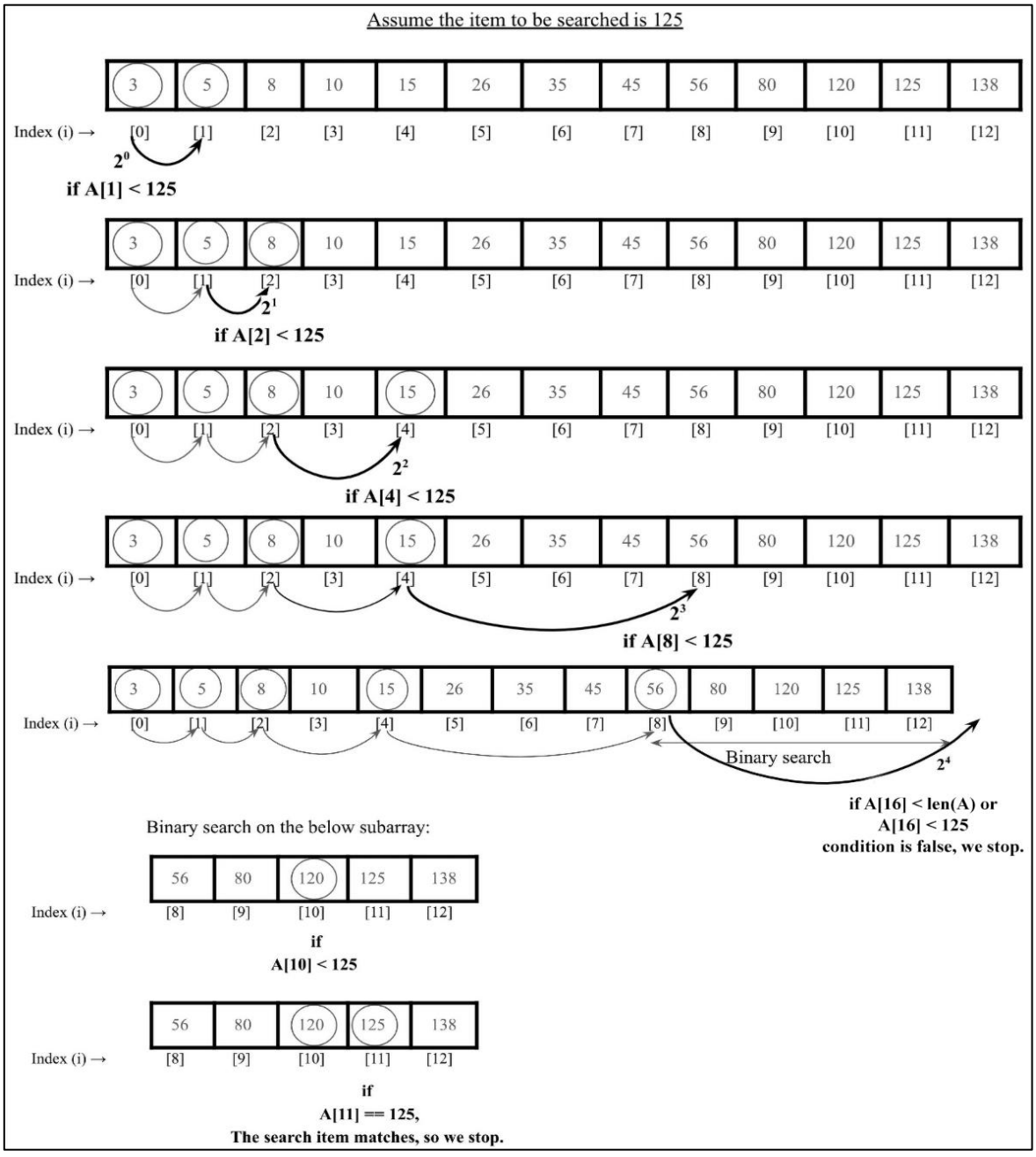

10
30
100
120
500
[0]
[1]
index_of_first_element
index_of_last_element


search_value = 230
list[low_index]
list[upper_index]
44
60
75
100
120
230
250
Index →
[0]
[1]
[2]
[3]
[4]
[5]
[6]
low_index
upper_index

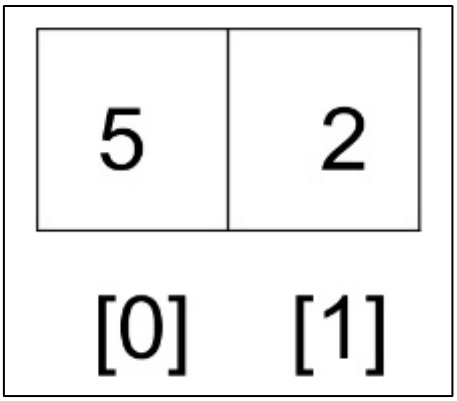

44
60
75
185
190
230
250
Index →
[0]
[1]
[2]
[3]
[4]
[5]
[6]
low_index
upper_index
readjusted
upper_index
midpoint

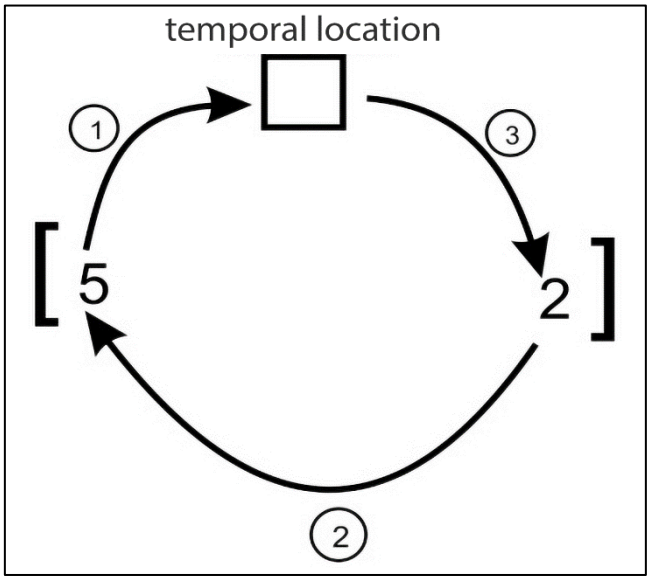

44
60
75
185
190
230
250
Index →
[0]
[1]
[2]
[3]
[4]
[5]
[6]
low_index
upper_index
readjusted
upper_index
midpoint

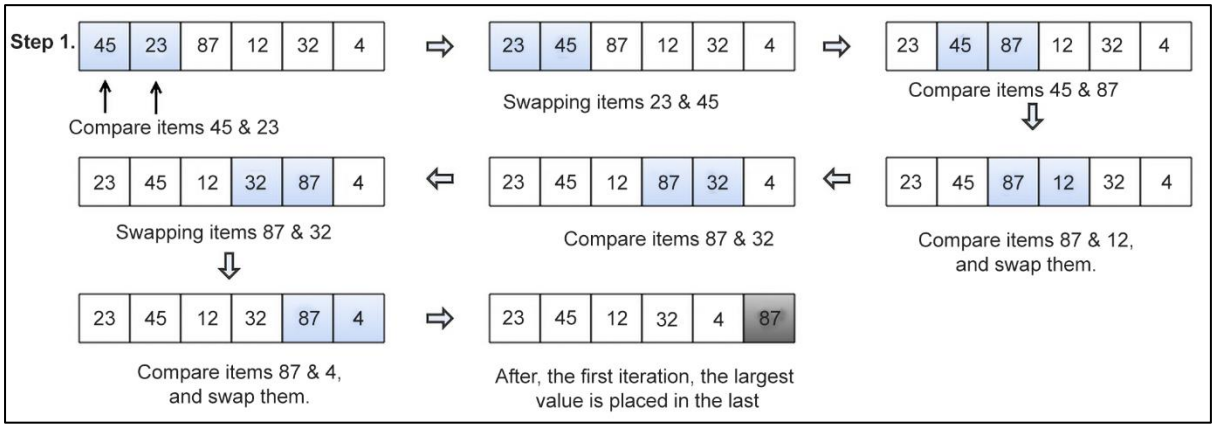

0 1 2 3 4 5 6 7
2 4 5 12 43 54 60 77
1st search attempt
2nd search attempt
3rd search attempt

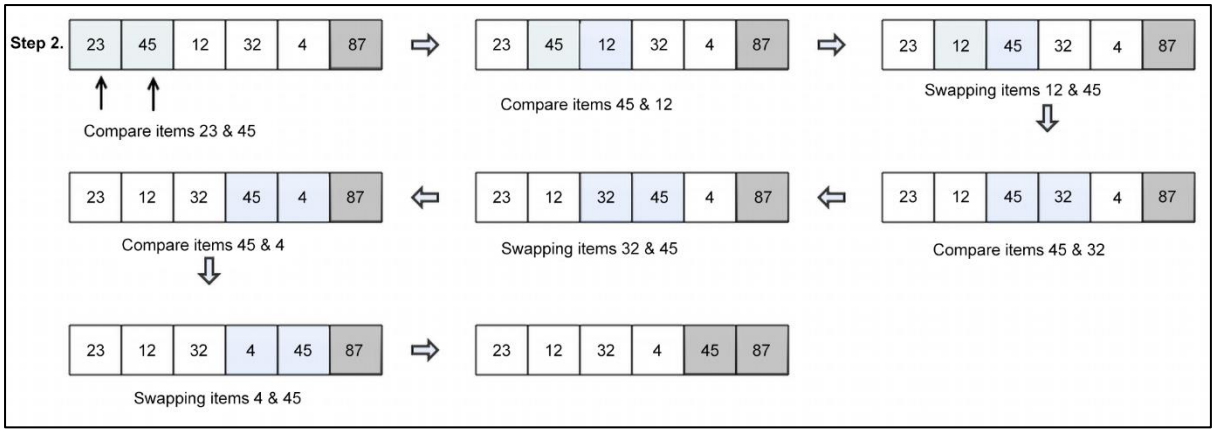

Assume the item to be searched is 125
3 5 8 10 15 26 35 45 56 80 120 125 138
Index (i) → [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]
2^0
if A[1] < 125
2^1
if A[2] < 125
2^2
if A[4] < 125
2^3
if A[8] < 125
Binary search
2^4
if A[16] < len(A) or A[16] < 125 condition is false, we stop.
Binary search on the below subarray:
56 80 120 125 138
[8] [9] [10] [11] [12]
if A[10] < 125
if A[11] == 125, The search item matches, so we stop.

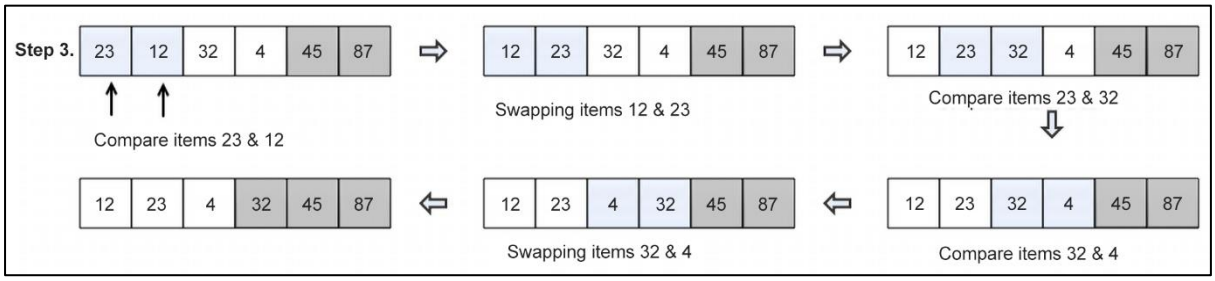
Chapter 11: Sorting

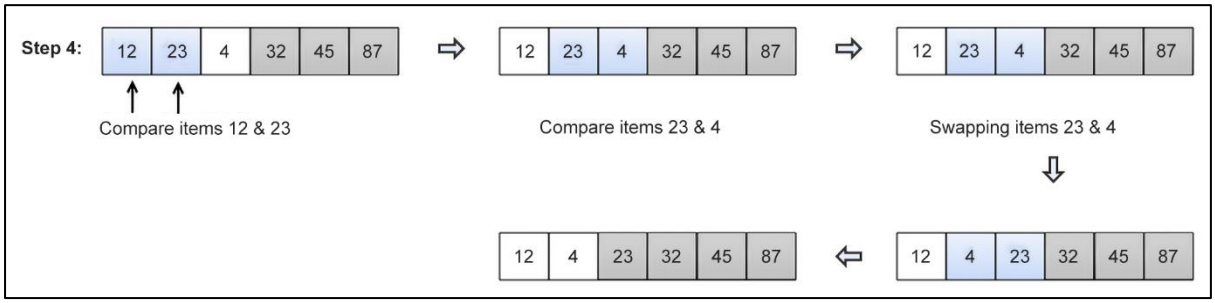


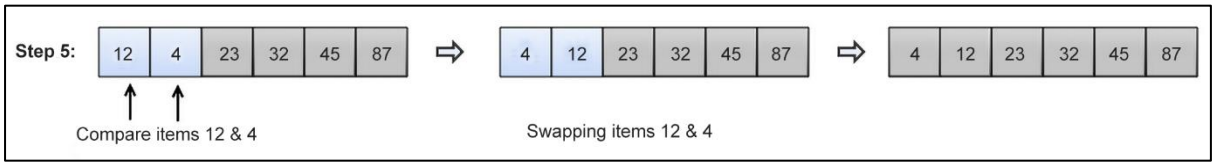


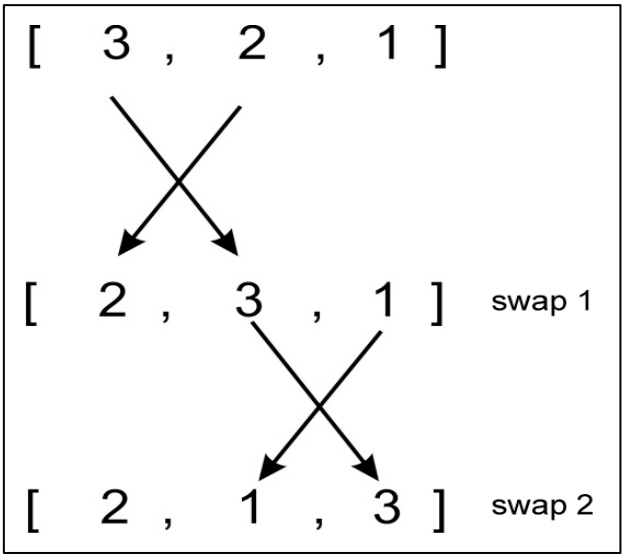


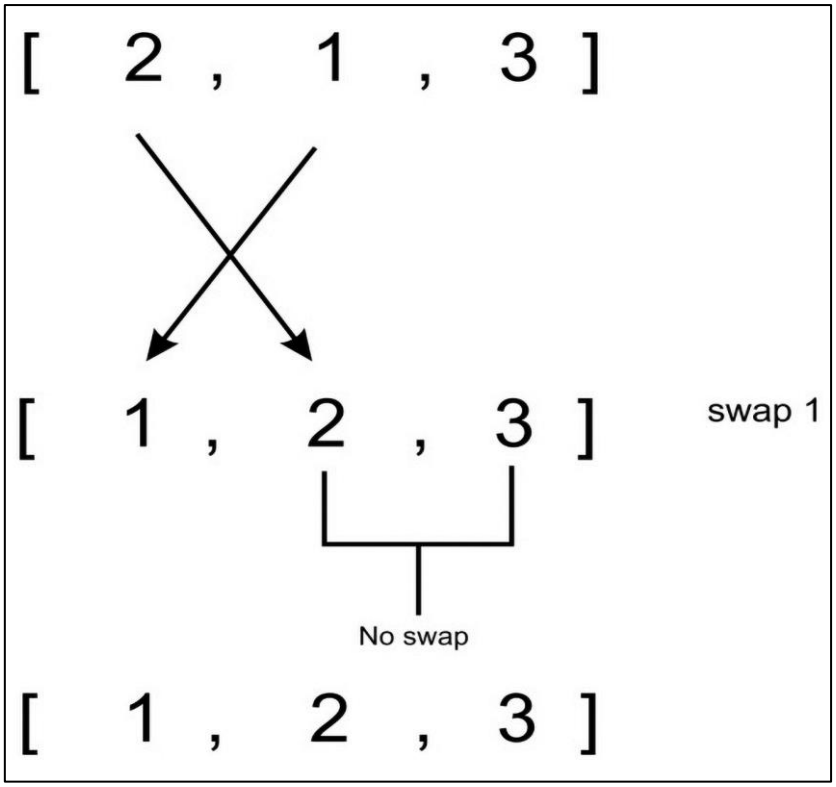

Step 2.
23 45 12 32 4 87
Compare items 23 & 45
23 45 12 32 4 87
Compare items 45 & 12
23 12 45 32 4 87
Swapping items 12 & 45
23 12 45 32 4 87
Compare items 45 & 32
23 12 32 45 4 87
Swapping items 32 & 45
23 12 32 45 4 87
Compare items 45 & 4
23 12 32 4 45 87
Swapping items 4 & 45
23 12 32 4 45 87

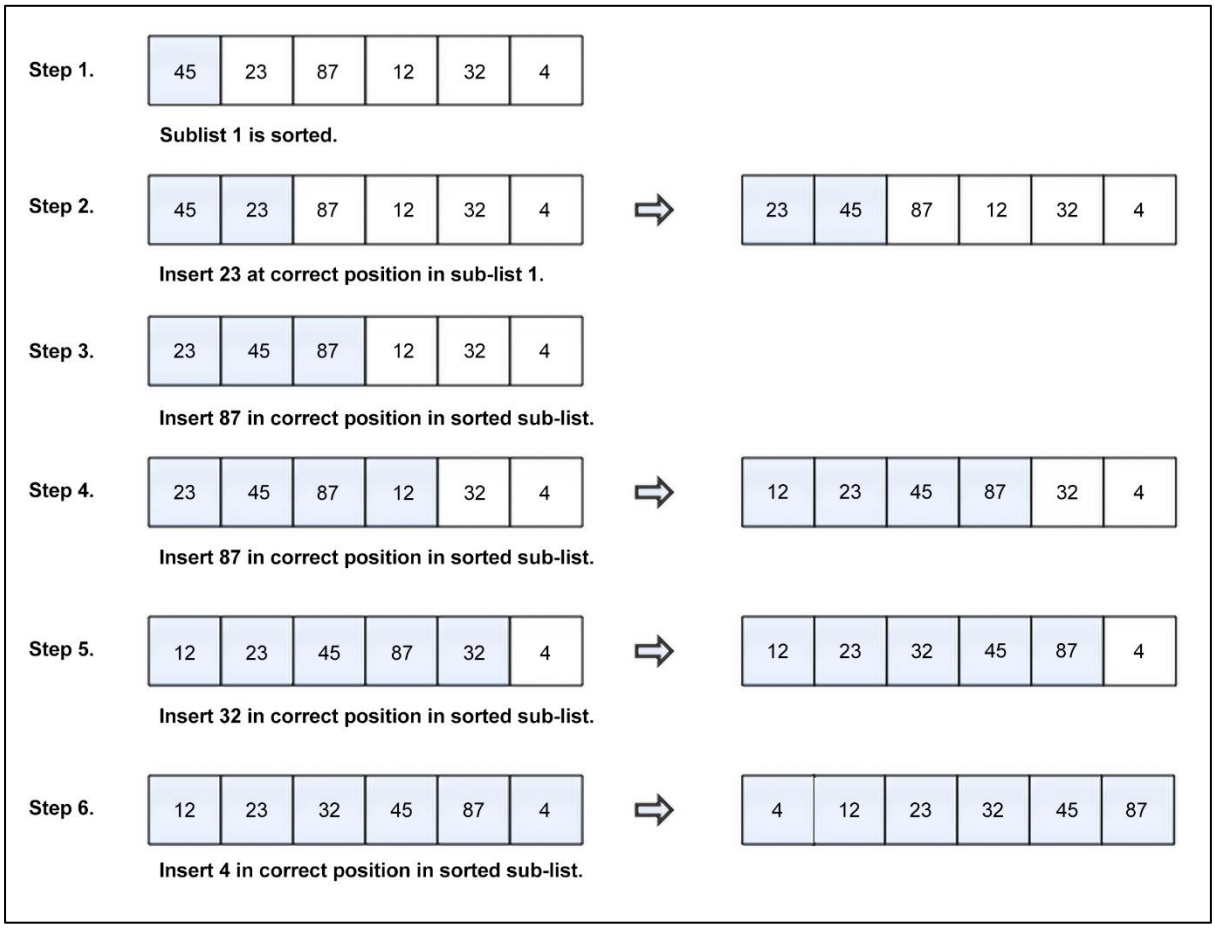

Step 3.
23 12 32 4 45 87
Compare items 23 & 12
12 23 32 4 45 87
Swapping items 12 & 23
12 23 32 4 45 87
Compare items 23 & 32
12 23 4 32 45 87
Swapping items 32 & 4
12 23 4 32 45 87
Compare items 32 & 4
12 23 32 4 45 87

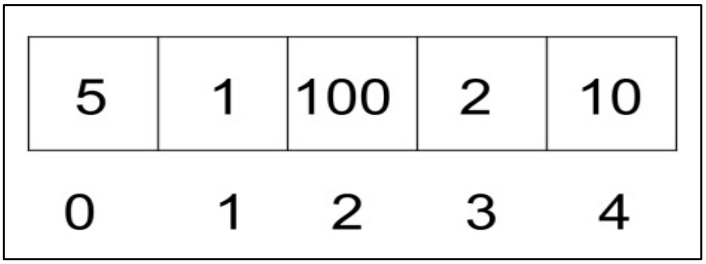

Step 4:
12 23 4 32 45 87
Compare items 12 & 23
12 23 4 32 45 87
Compare items 23 & 4
12 23 4 32 45 87
Swapping items 23 & 4
12 4 23 32 45 87
12 4 23 32 45 87

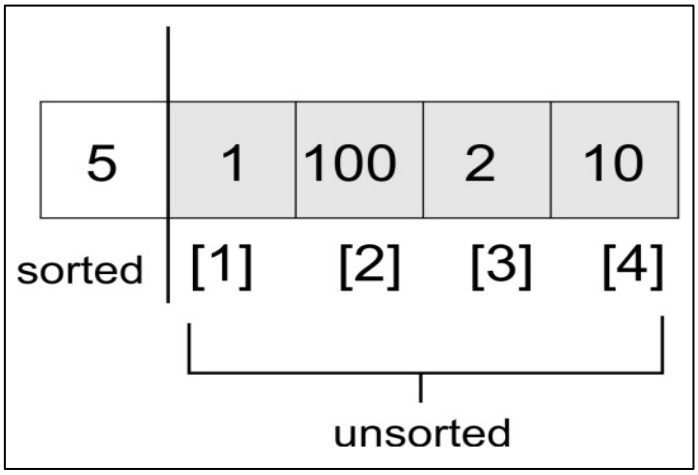

Step 5:
12 4 23 32 45 87
Compare items 12 & 4
4 12 23 32 45 87
Swapping items 12 & 4
4 12 23 32 45 87

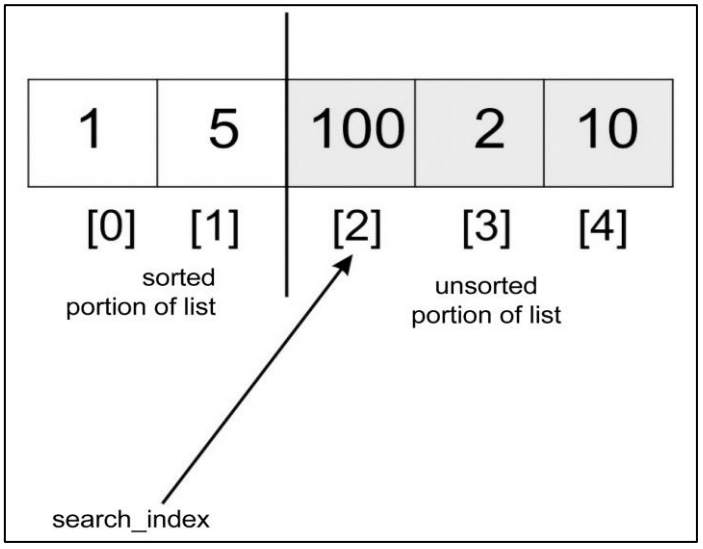

[3 , 2 , 1]
[2 , 3 , 1]
swap 1
[2 , 1 , 3]
swap 2

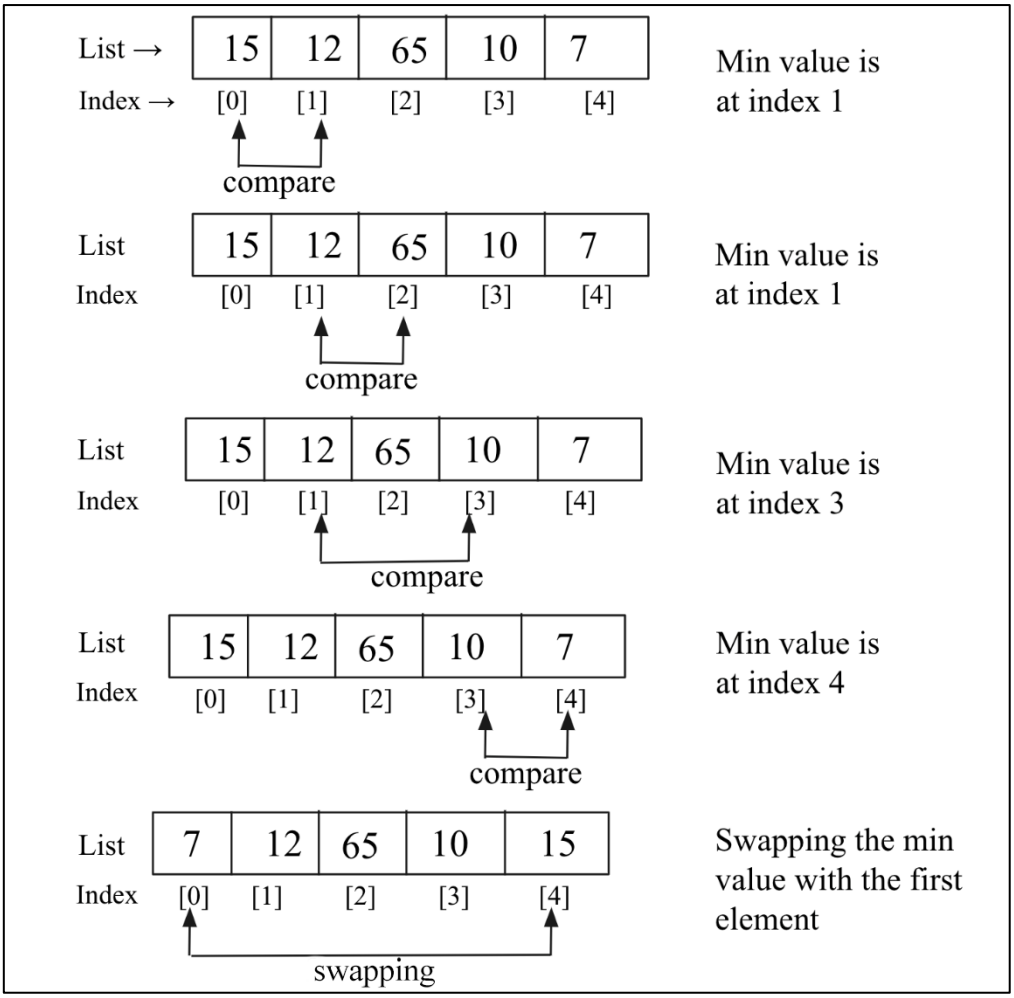

[2 , 1 , 3]
[1 , 2 , 3]
swap 1
No swap
[1 , 2 , 3]

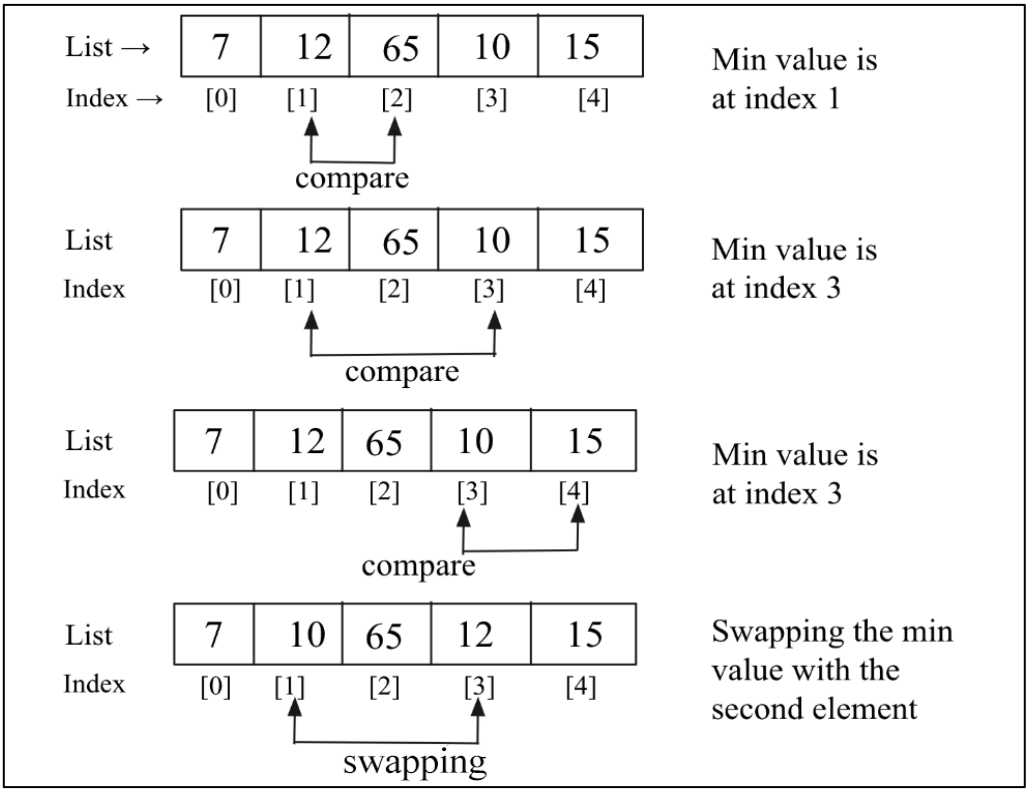

Step 1.
45 23 87 12 32 4
Sublist 1 is sorted.
Step 2.
45 23 87 12 32 4
23 45 87 12 32 4
Insert 23 at correct position in sub-list 1.
Step 3.
23 45 87 12 32 4
Insert 87 in correct position in sorted sub-list.
Step 4.
23 45 87 12 32 4
12 23 45 87 32 4
Insert 87 in correct position in sorted sub-list.
Step 5.
12 23 45 87 32 4
12 23 32 45 87 4
Insert 32 in correct position in sorted sub-list.
Step 6.
12 23 32 45 87 4
4 12 23 32 45 87
Insert 4 in correct position in sorted sub-list.

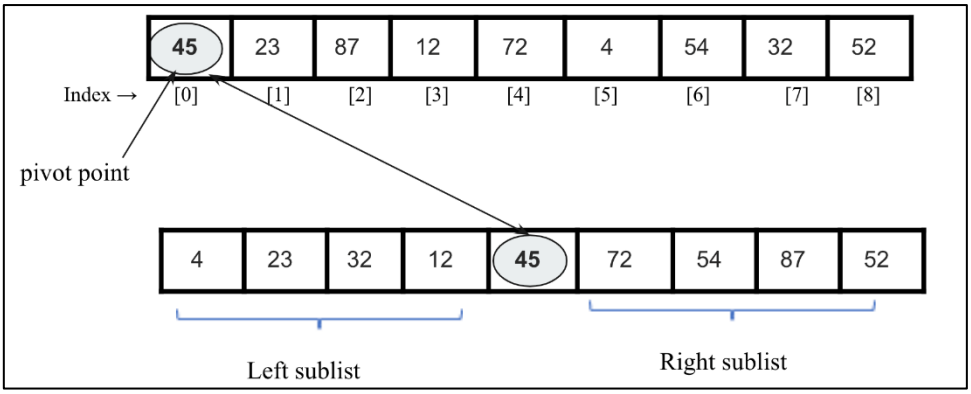

5
1
100
2
10
0
1
2
3
4

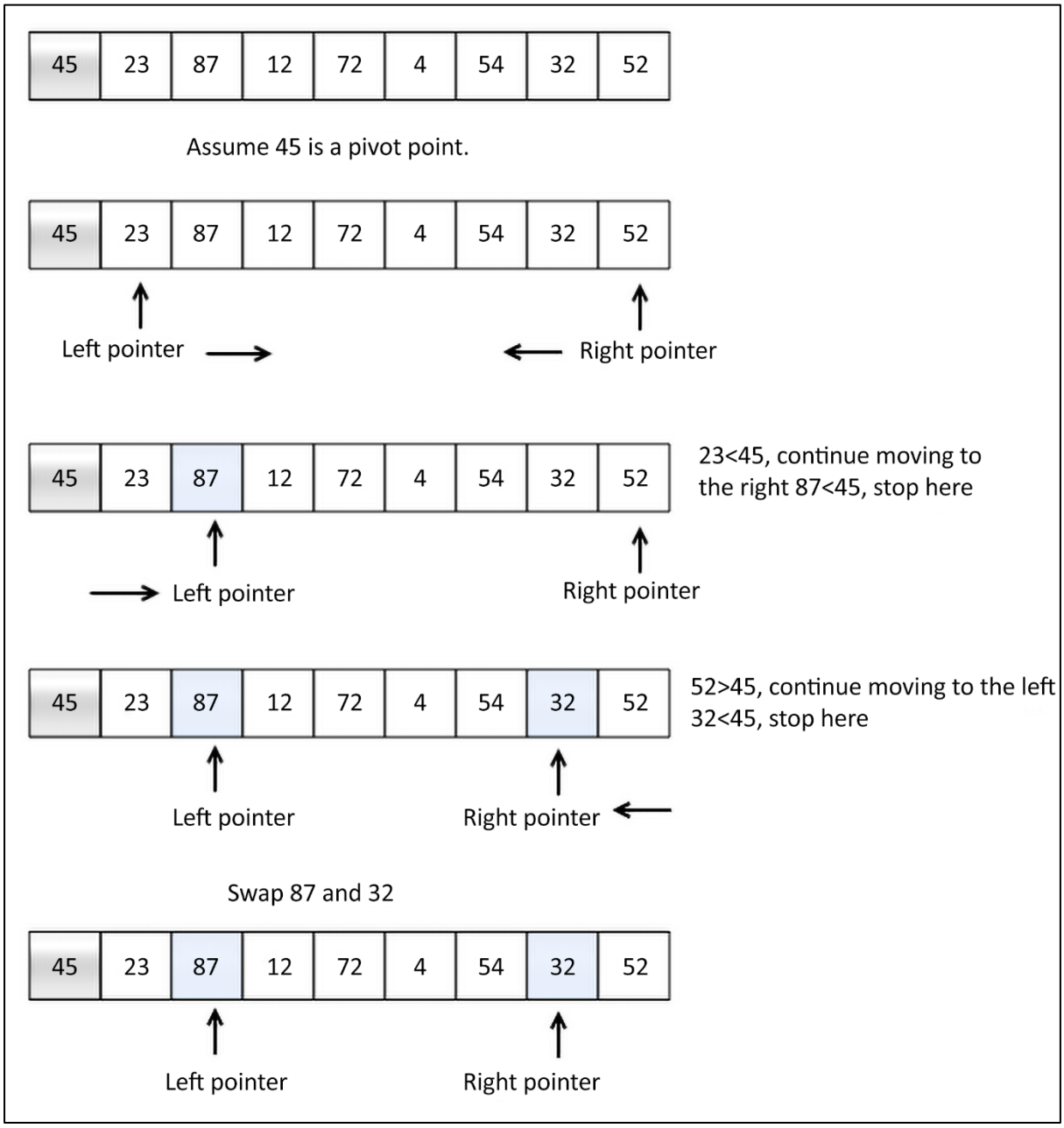

5
1
100
2
10
sorted
[1]
[2]
[3]
[4]
unsorted

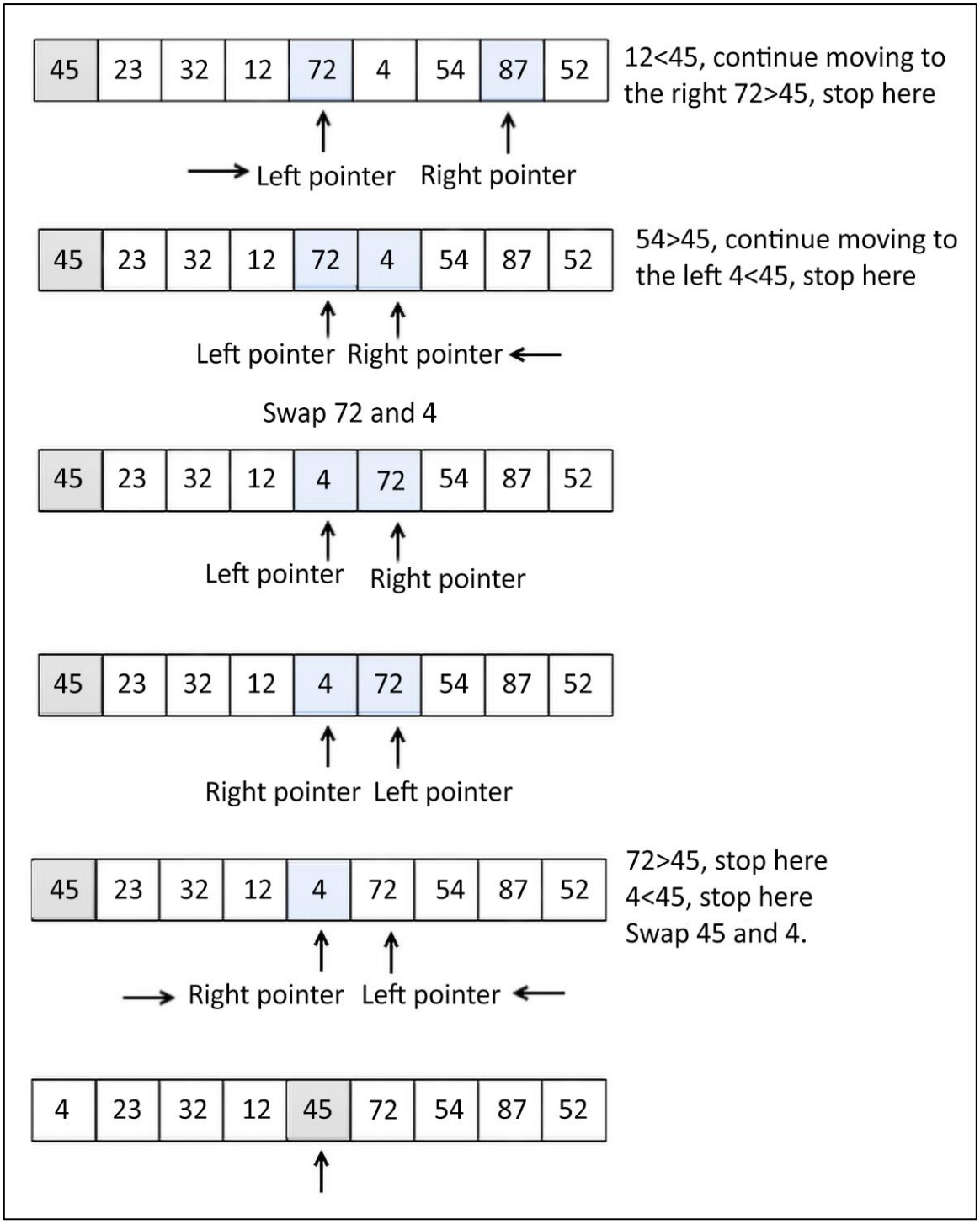

1
5
100
2
10
[0]
[1]
[2]
[3]
[4]
sorted
portion of list
unsorted
portion of list
search_index

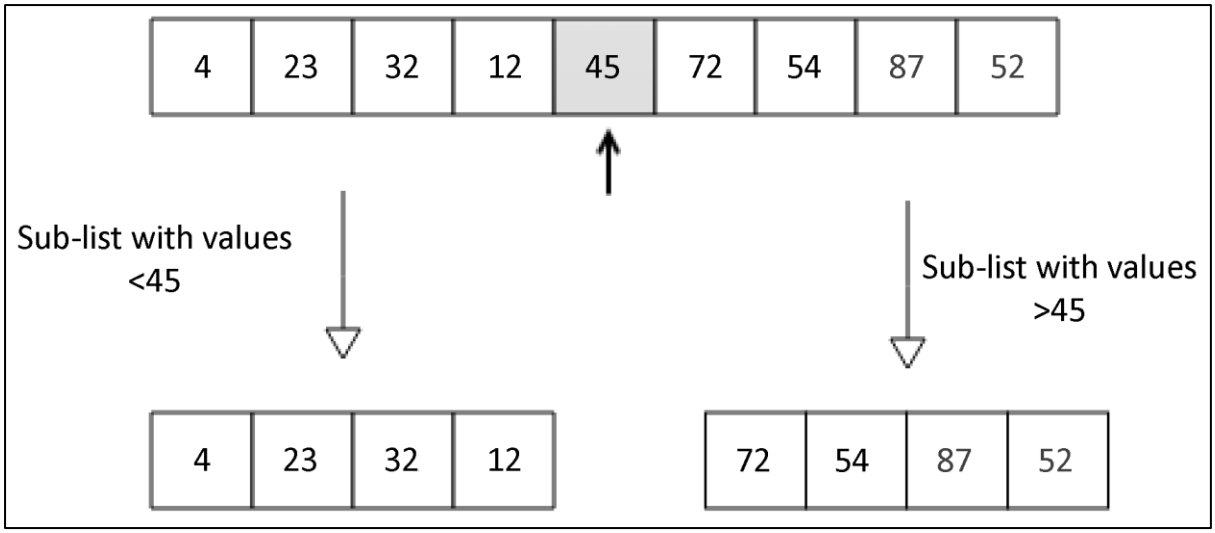

List →
Index →
15 12 65 10 7
[0] [1] [2] [3] [4]
compare
Min value is at index 1
List
Index
15 12 65 10 7
[0] [1] [2] [3] [4]
compare
Min value is at index 1
List
Index
15 12 65 10 7
[0] [1] [2] [3] [4]
compare
Min value is at index 3
List
Index
15 12 65 10 7
[0] [1] [2] [3] [4]
compare
Min value is at index 4
List
Index
7 12 65 10 15
[0] [1] [2] [3] [4]
swapping
Swapping the min value with the first element

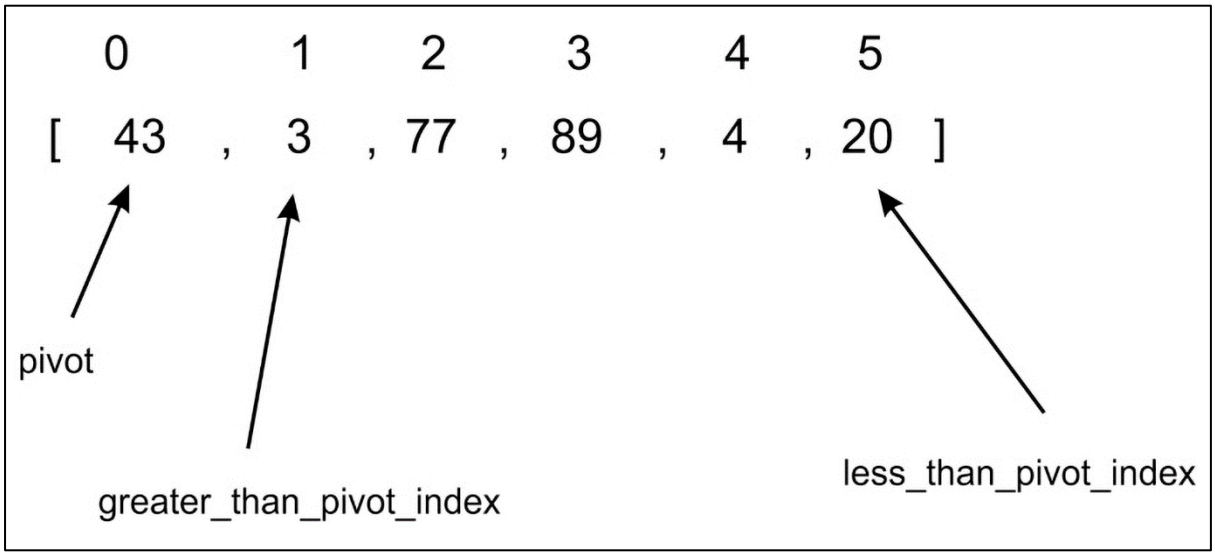

List →
Index →
7 12 65 10 15
[0] [1] [2] [3] [4]
compare
Min value is at index 1
List
Index
7 12 65 10 15
[0] [1] [2] [3] [4]
compare
Min value is at index 3
List
Index
7 12 65 10 15
[0] [1] [2] [3] [4]
compare
Min value is at index 3
List
Index
7 10 65 12 15
[0] [1] [2] [3] [4]
swapping
Swapping the min value with the second element

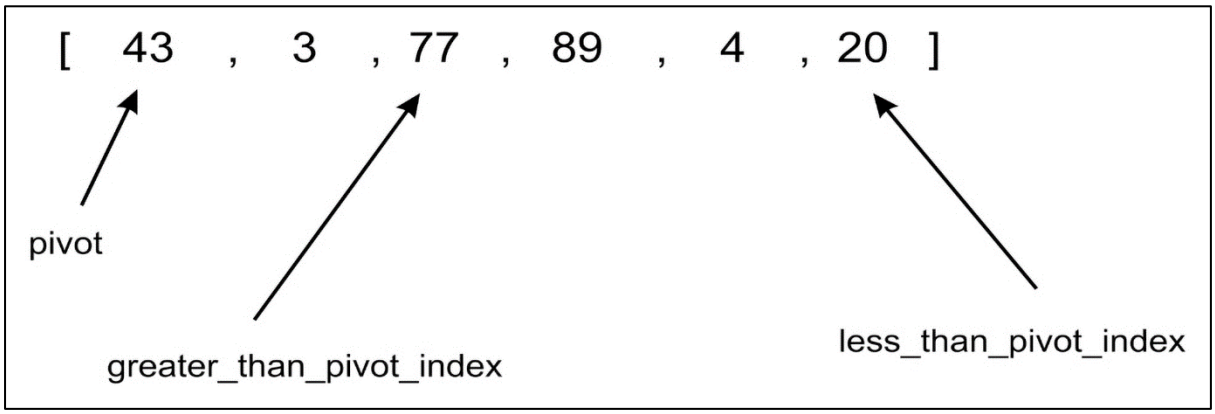

45
23
87
12
72
4
54
32
52
Index →
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
pivot point
4
23
32
12
45
72
54
87
52
Left sublist
Right sublist

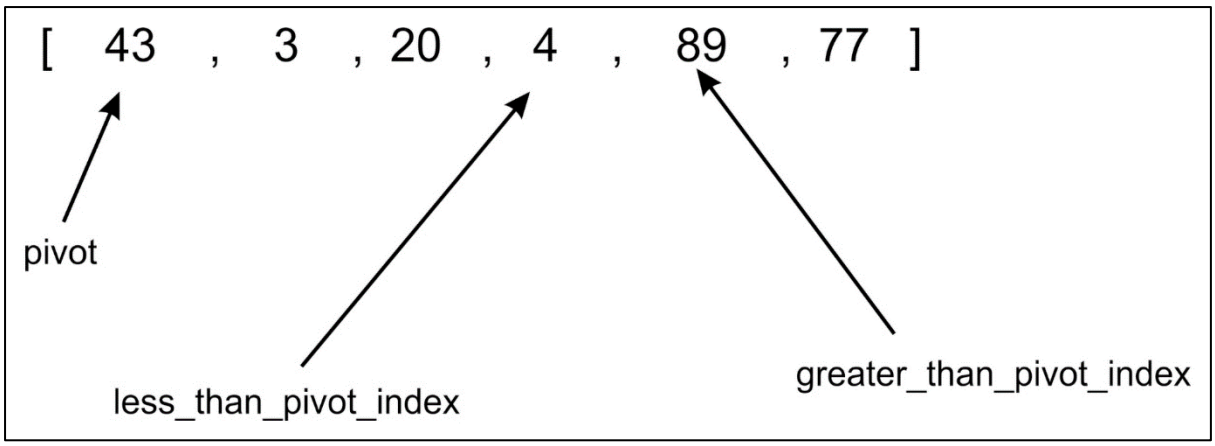

45 23 87 12 72 4 54 32 52
Assume 45 is a pivot point.
45 23 87 12 72 4 54 32 52
Left pointer
Right pointer
45 23 87 12 72 4 54 32 52
23<45, continue moving to the right 87<45, stop here
Left pointer
Right pointer
45 23 87 12 72 4 54 32 52
52>45, continue moving to the left
32<45, stop here
Left pointer
Right pointer
Swap 87 and 32
45 23 87 12 72 4 54 32 52
Left pointer
Right pointer

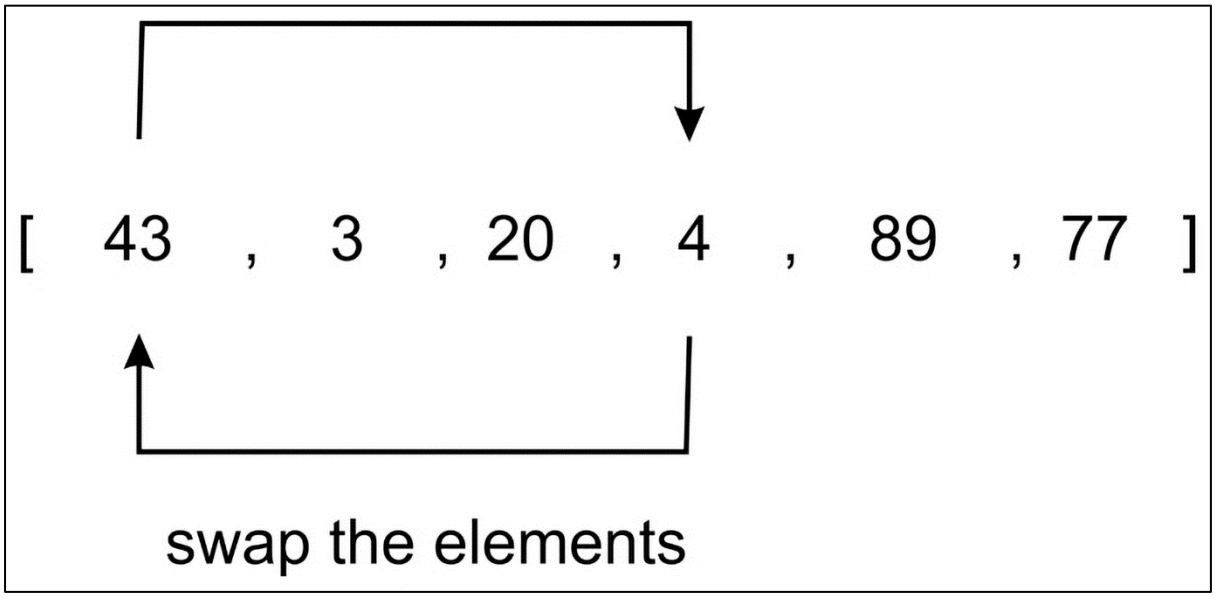

45 23 32 12 72 4 54 87 52
12<45, continue moving to the right 72>45, stop here
Left pointer
Right pointer
45 23 32 12 72 4 54 87 52
54>45, continue moving to the left 4<45, stop here
Left pointer
Right pointer
Swap 72 and 4
45 23 32 12 4 72 54 87 52
Left pointer
Right pointer
45 23 32 12 4 72 54 87 52
Right pointer
Left pointer
45 23 32 12 4 72 54 87 52
72>45, stop here
4<45, stop here
Swap 45 and 4.
Right pointer
Left pointer
4 23 32 12 45 72 54 87 52

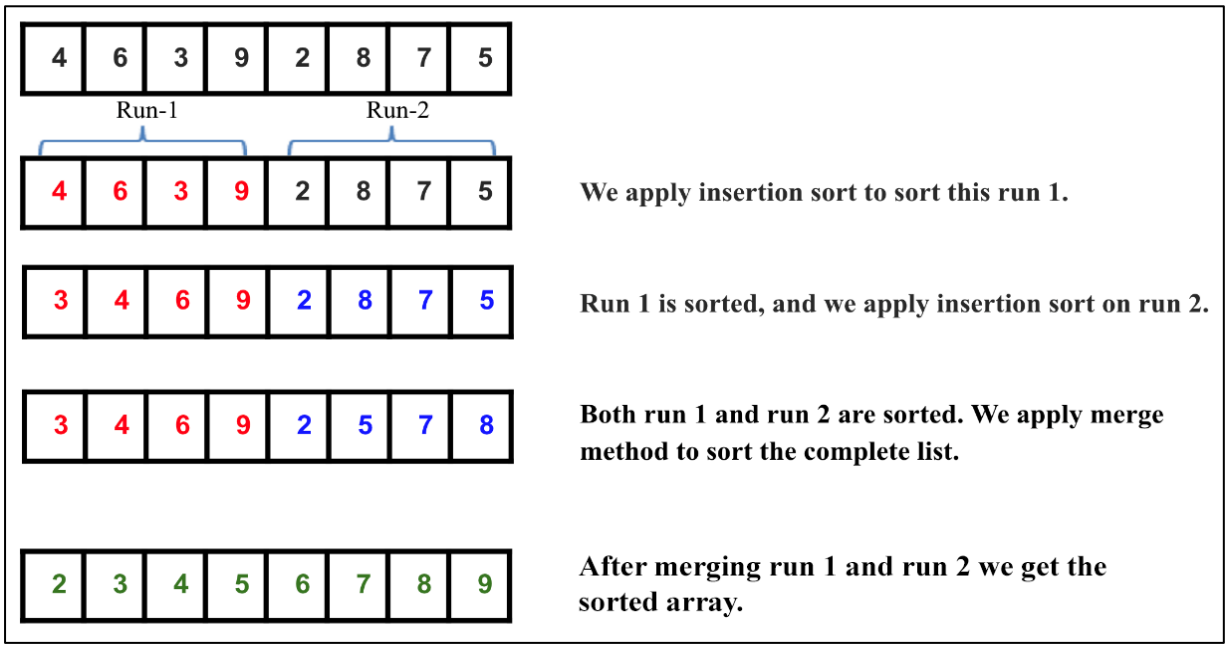

4 23 32 12 45 72 54 87 52
Sub-list with values <45
Sub-list with values >45
4 23 32 12
72 54 87 52


0 1 2 3 4 5
[43 , 3 , 77 , 89 , 4 , 20]
pivot
greater_than_pivot_index
less_than_pivot_index

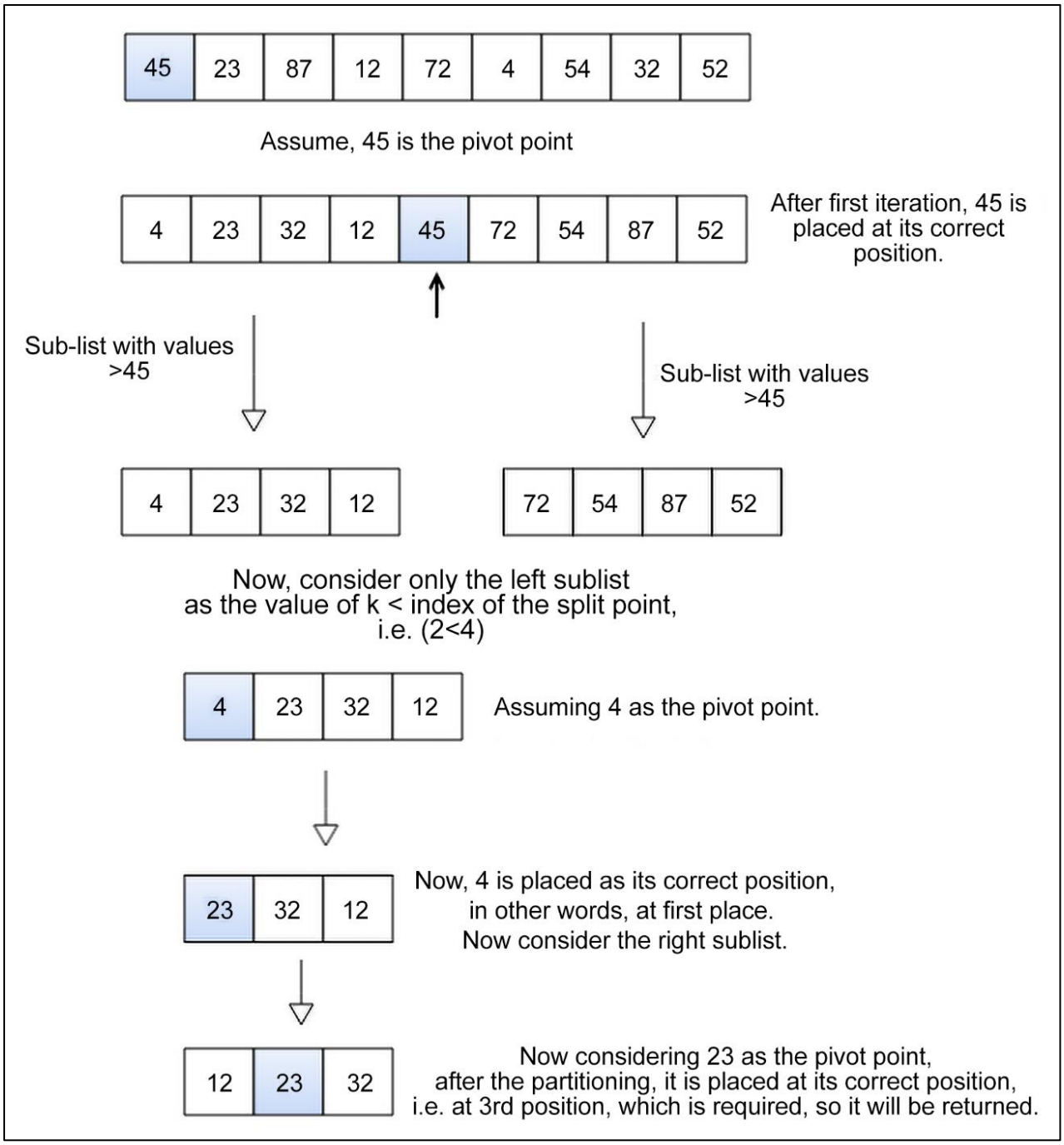

[43 , 3 , 77 , 89 , 4 , 20]
pivot
greater_than_pivot_index
less_than_pivot_index

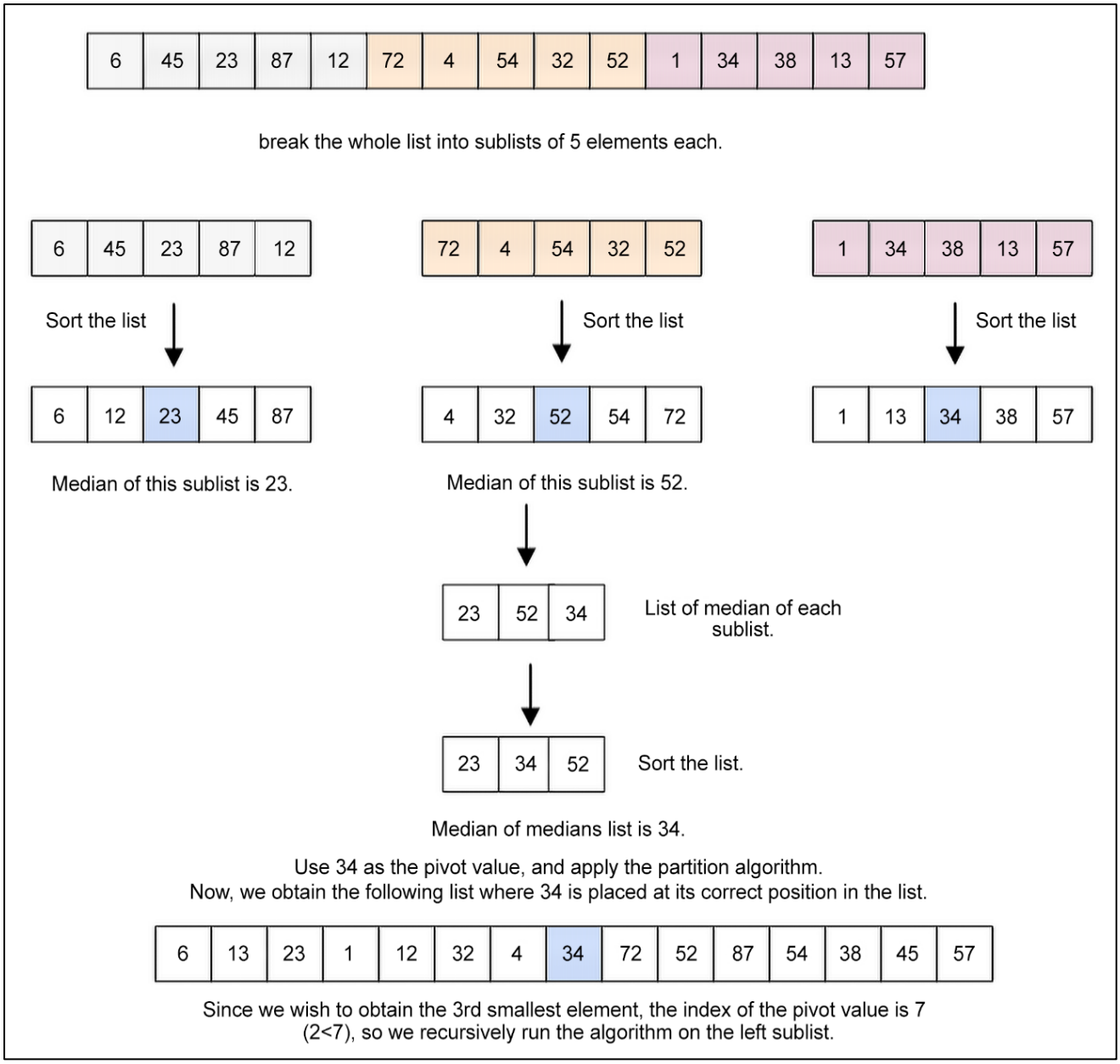

[43 , 3 , 20 , 4 , 89 , 77]
pivot
less_than_pivot_index
greater_than_pivot_index


[43 , 3 , 20 , 4 , 89 , 77]
swap the elements

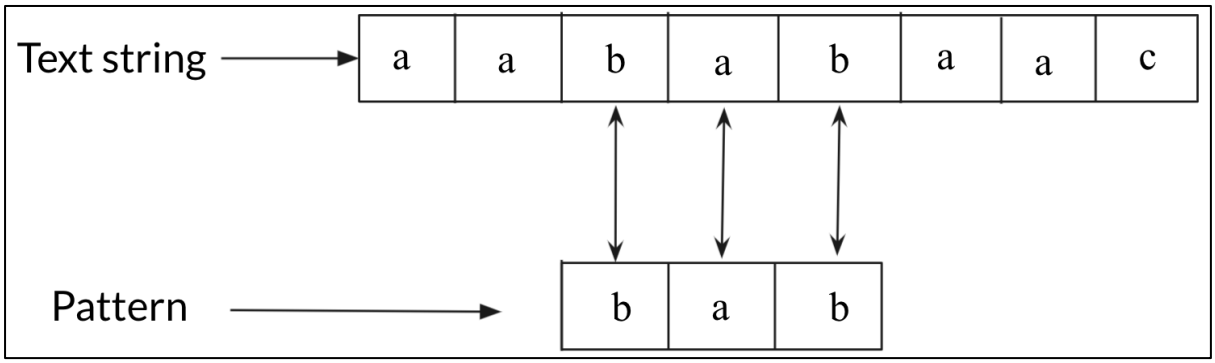

4 6 3 9 2 8 7 5
Run-1
Run-2
4 6 3 9 2 8 7 5
We apply insertion sort to sort this run 1.
3 4 6 9 2 8 7 5
Run 1 is sorted, and we apply insertion sort on run 2.
3 4 6 9 2 5 7 8
Both run 1 and run 2 are sorted. We apply merge method to sort the complete list.
2 3 4 5 6 7 8 9
After merging run 1 and run 2 we get the sorted array.

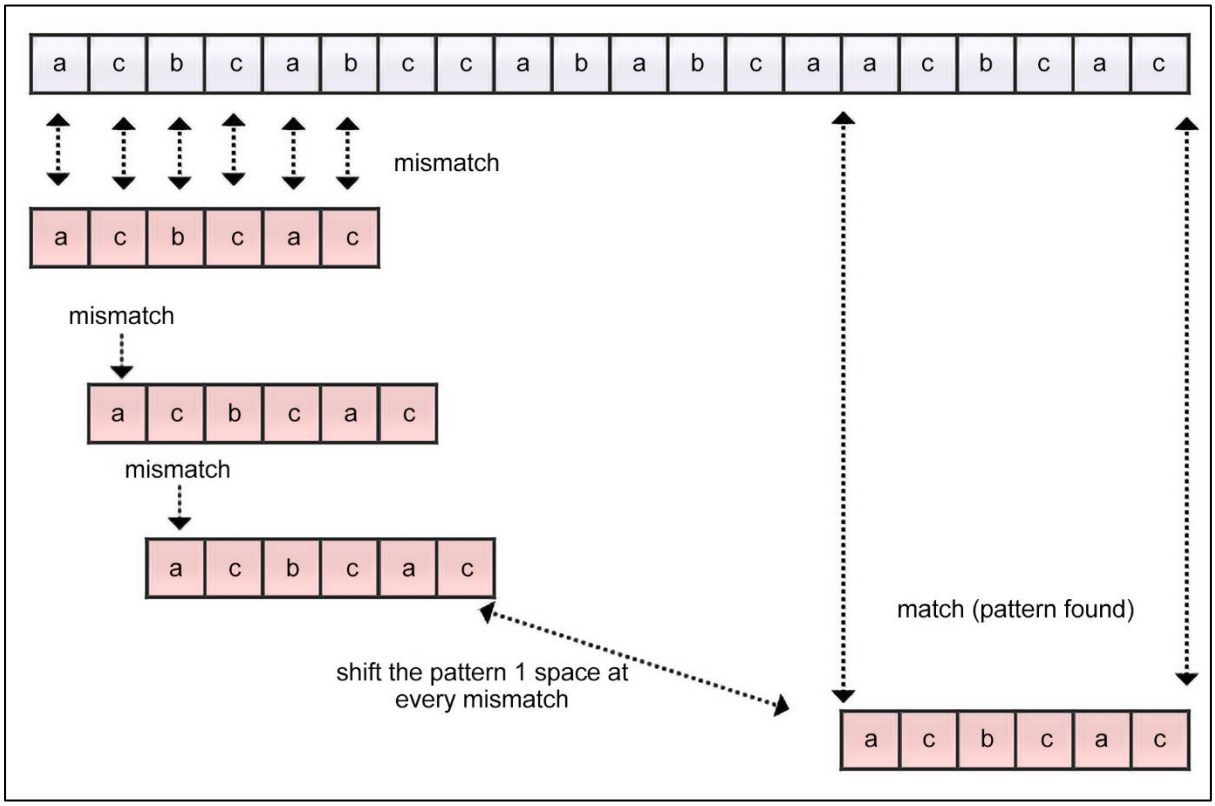
Chapter 12: Selection Algorithms

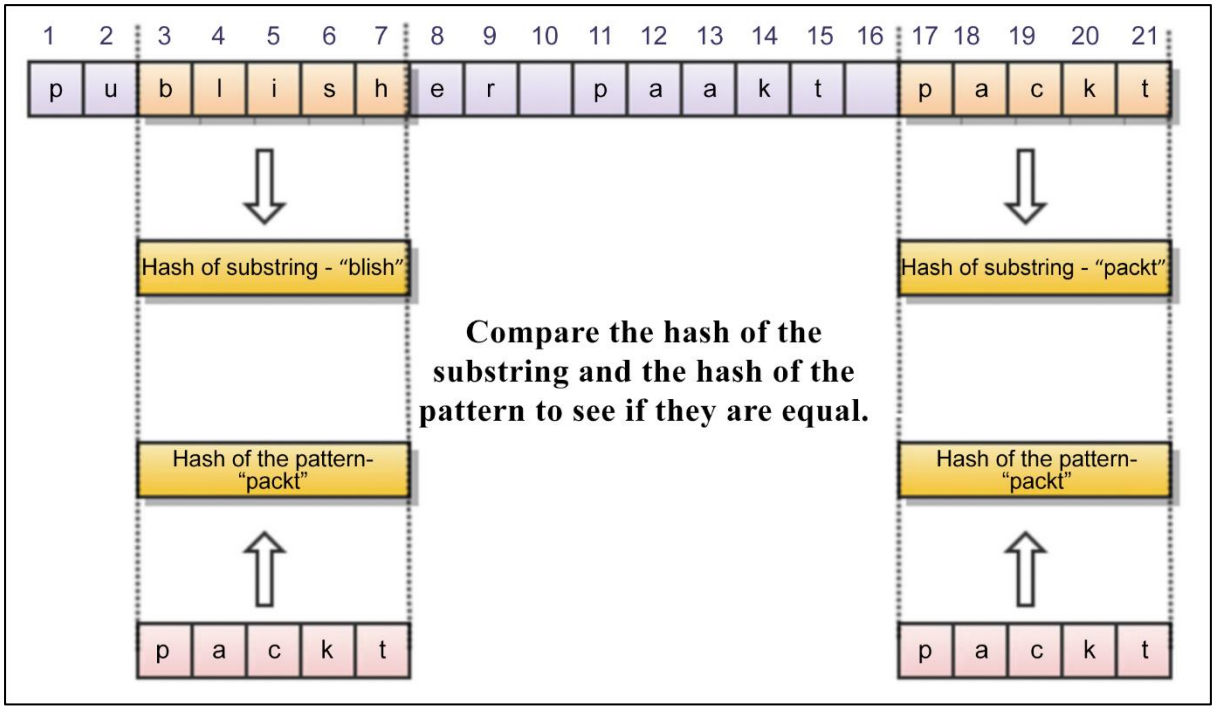


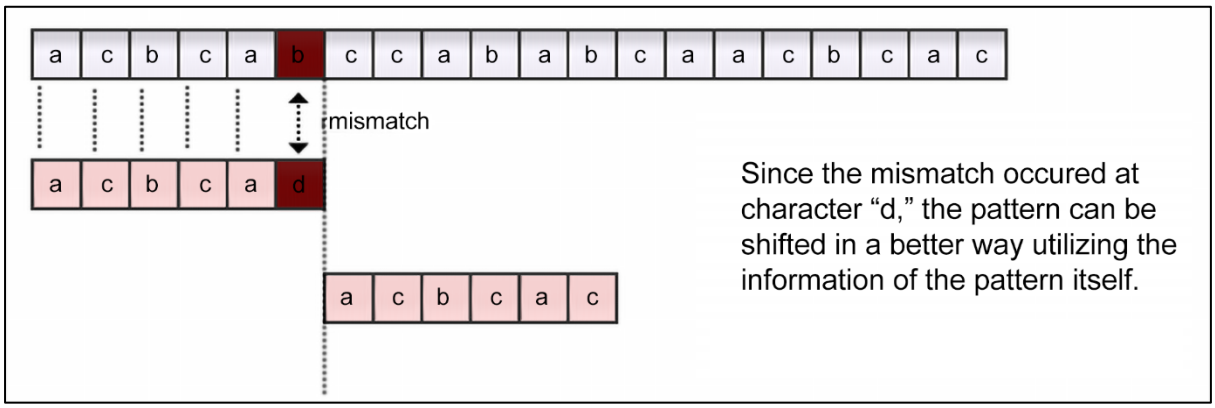

6 45 23 87 12 72 4 54 32 52 1 34 38 13 57
break the whole list into sublists of 5 elements each.
6 45 23 87 12
72 4 54 32 52
1 34 38 13 57
Sort the list
Sort the list
Sort the list
6 12 23 45 87
4 32 52 54 72
1 13 34 38 57
Median of this sublist is 23.
Median of this sublist is 52.
23 52 34
List of median of each sublist.
23 34 52
Sort the list.
Median of medians list is 34.
Use 34 as the pivot value, and apply the partition algorithm.
Now, we obtain the following list where 34 is placed at its correct position in the list.
6 13 23 1 12 32 4 34 72 52 87 54 38 45 57
Since we wish to obtain the 3rd smallest element, the index of the pivot value is 7
(2<7), so we recursively run the algorithm on the left sublist.

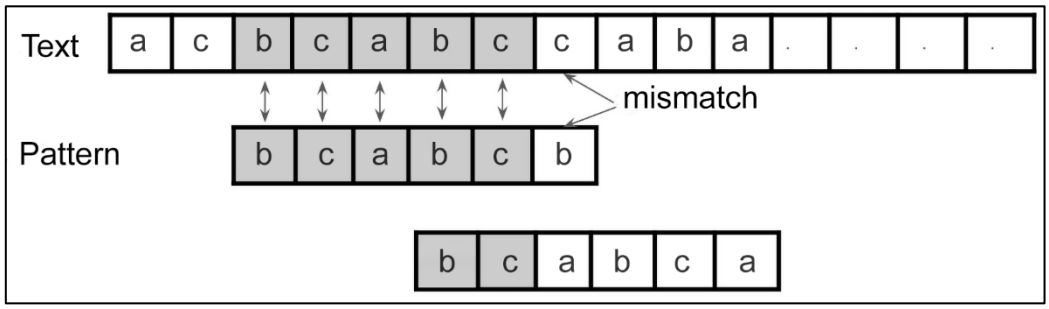
Chapter 13: String Matching Algorithms



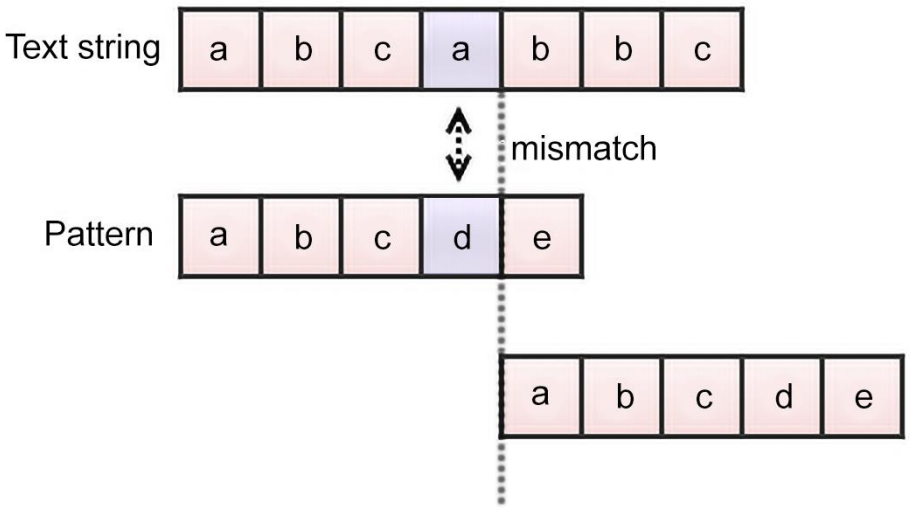



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
p u b l i s h e r p a a k t p a c k t
Hash of substring - "blish"
Hash of substring - "packt"
Compare the hash of the substring and the hash of the pattern to see if they are equal.
Hash of the pattern- "packt"
Hash of the pattern- "packt"
p a c k t
p a c k t


a c b c a b c c a b a b c a a c b c a c
mismatch
a c b c a d
a c b c a c
Since the mismatch occured at character "d," the pattern can be shifted in a better way utilizing the information of the pattern itself.


Text
a c b c a b c c a b a . . . .
mismatch
Pattern
b c a b c b
b c a b c a

Index	1	2	3	4	5
Pattern	a	b	c	d	e
Prefix_function	0	0	0	0	0



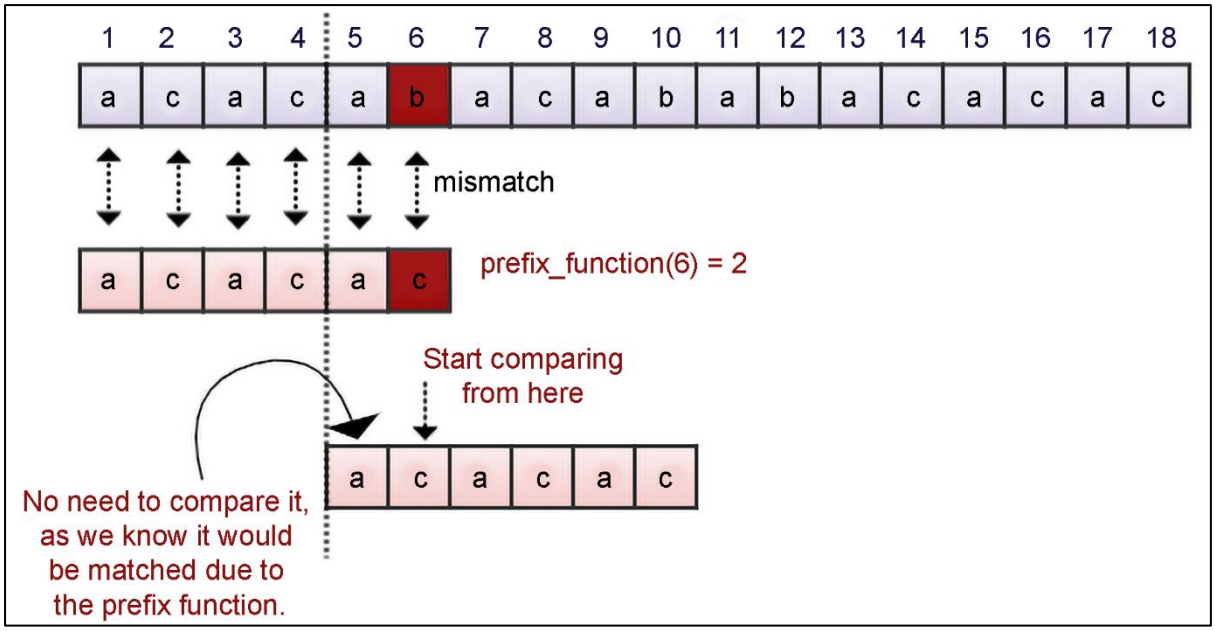
Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0						

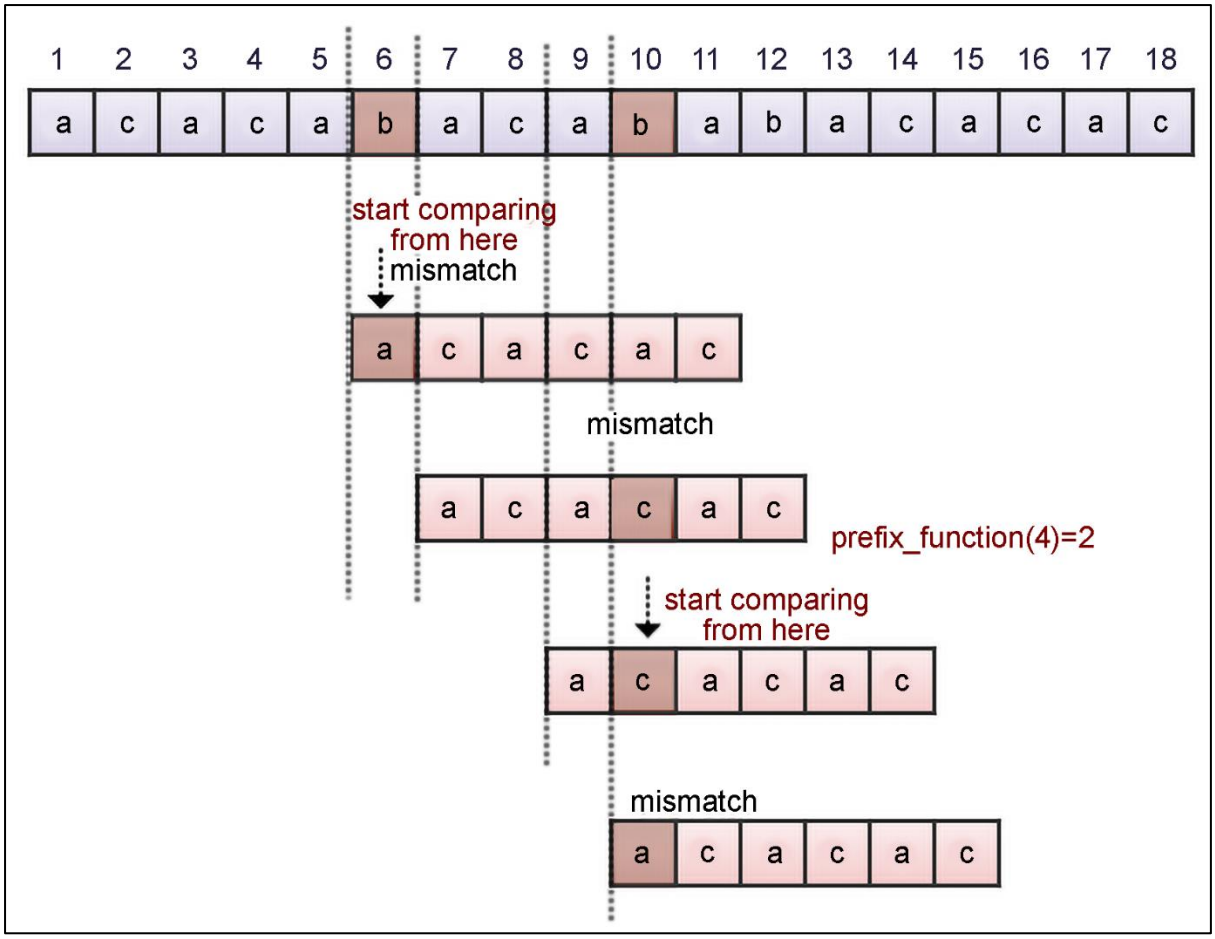
Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1					

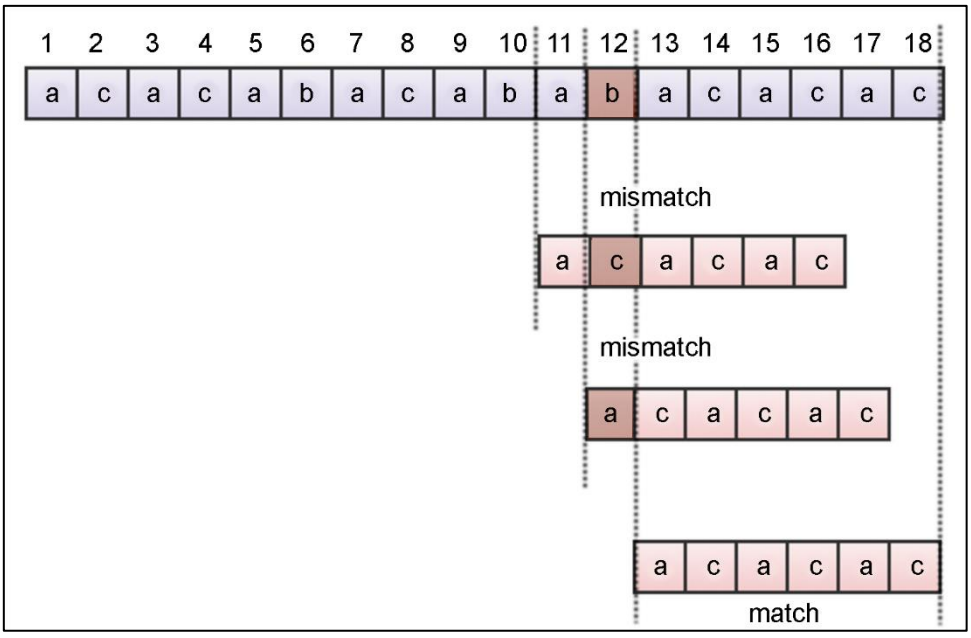
Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2				

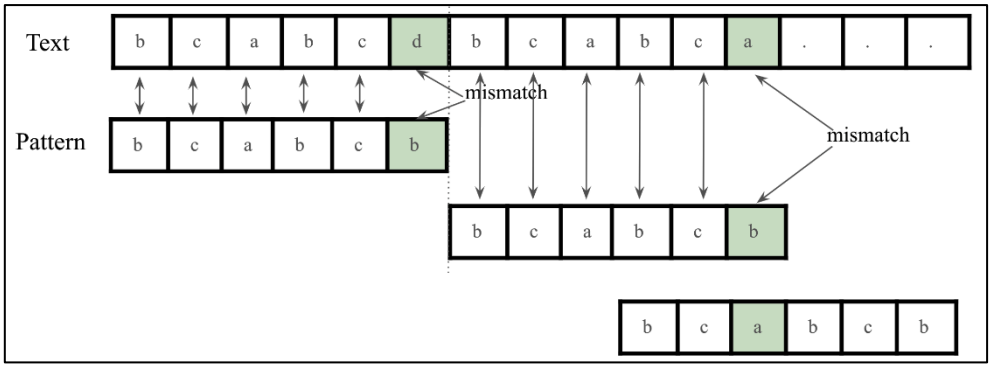
Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2	0	0	1	2

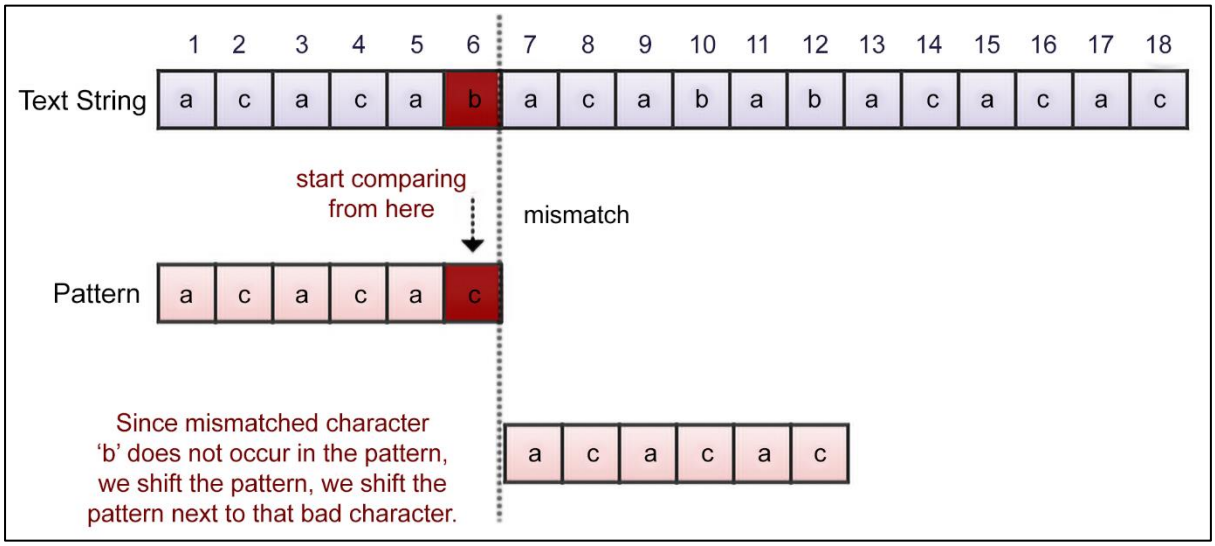
Index	1	2	3	4	5	6
Pattern	a	c	a	c	a	c
Prefix_function	0	0	1	2	1	2

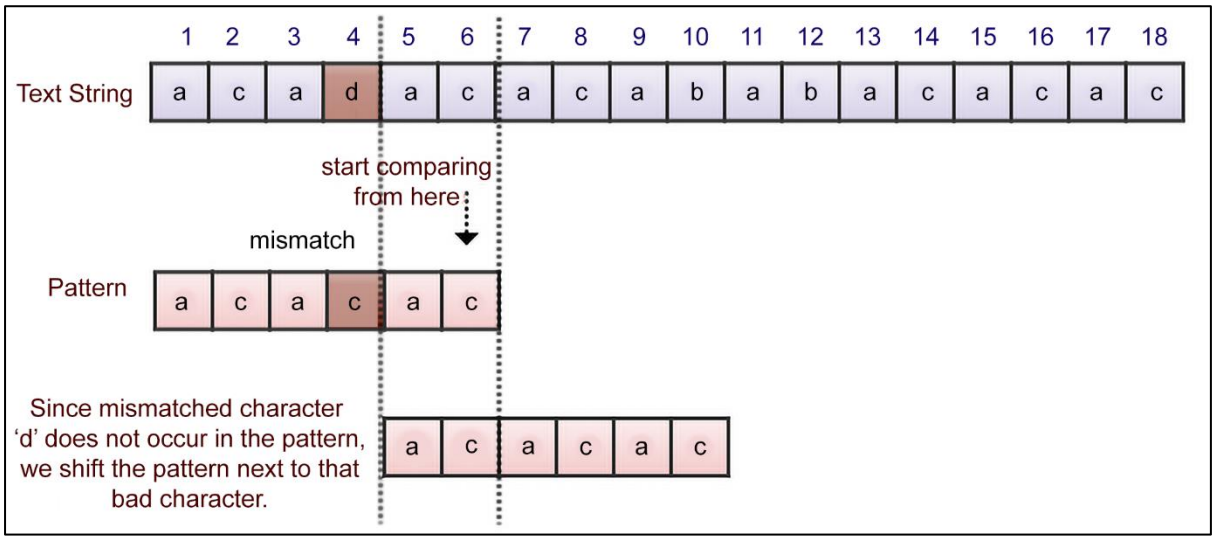


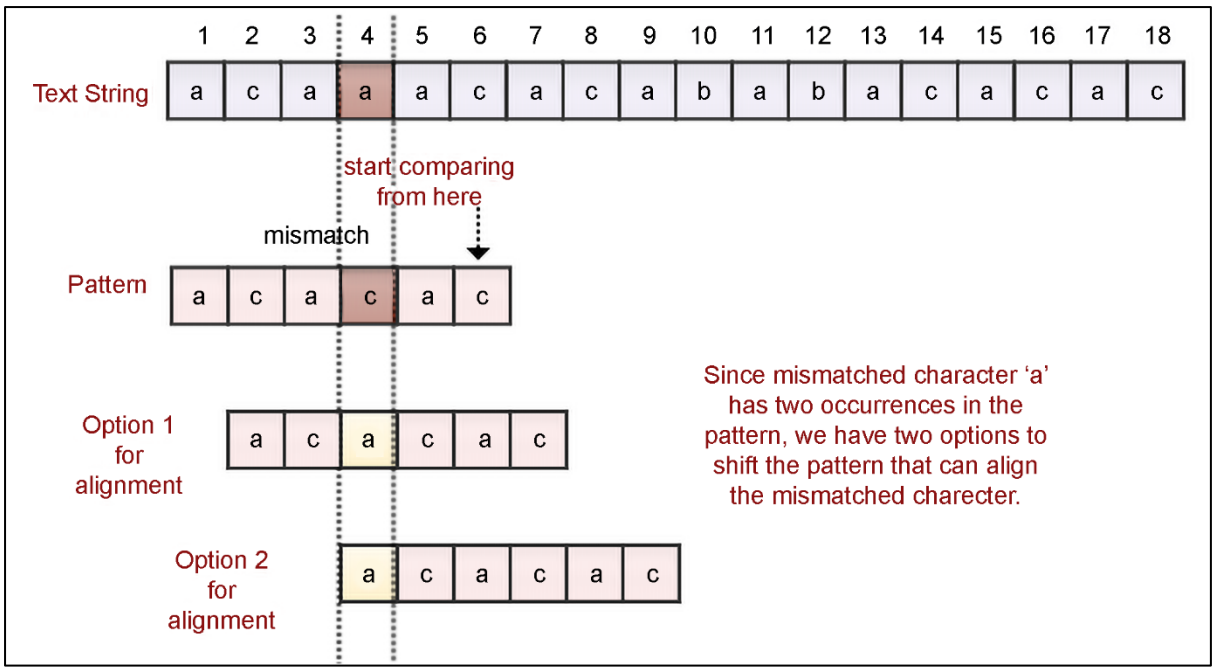

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
a c a c a b a c a b a b a c a c a c
start comparing
from here
mismatch
a c a c a c
mismatch
a c a c a c
prefix_function(4)=2
start comparing
from here
a c a c a c
mismatch
a c a c a c

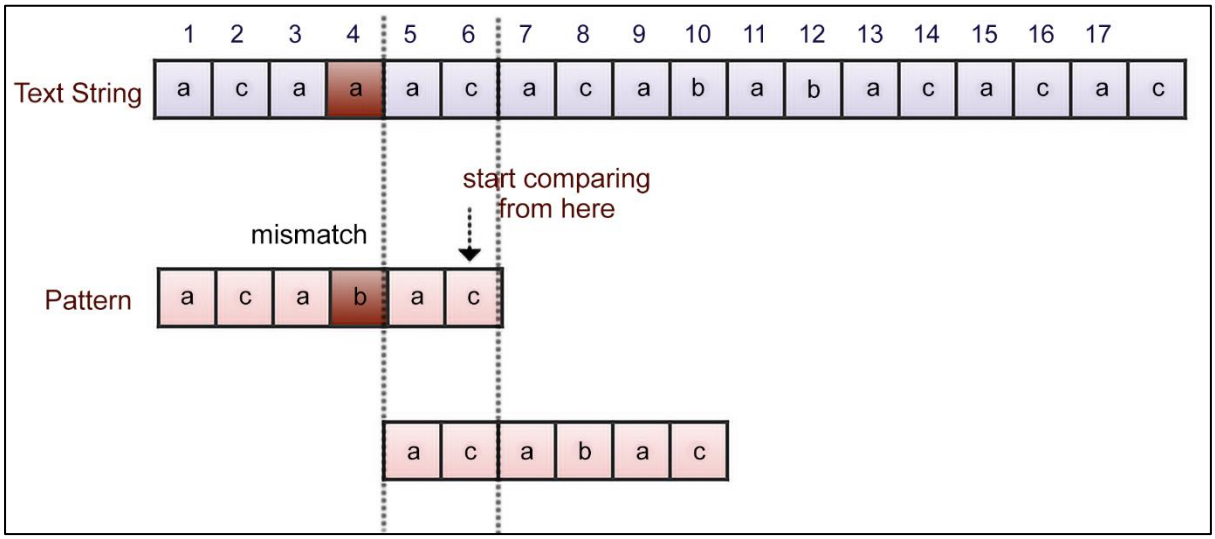

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
a c a c a b a c a b a b a c a c a c
mismatch
a c a c a c
mismatch
a c a c a c
a c a c a c
match

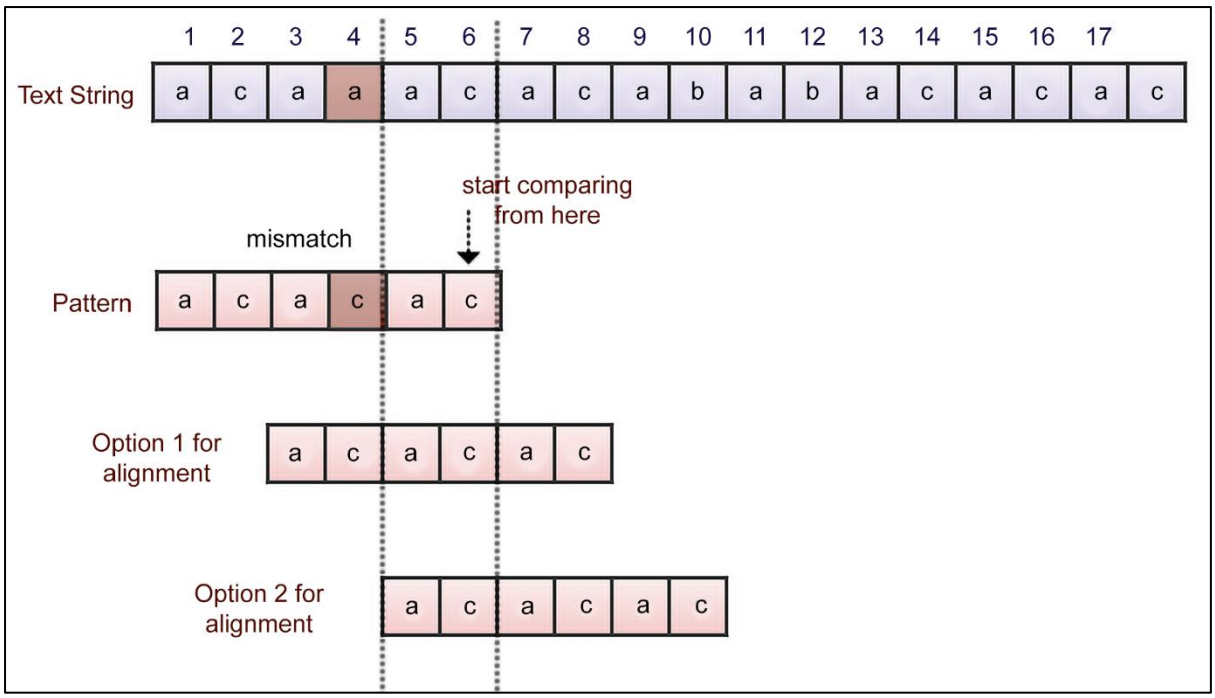

Text
b c a b c d b c a b c a . . .
mismatch
Pattern
b c a b c b
mismatch
b c a b c b
b c a b c b

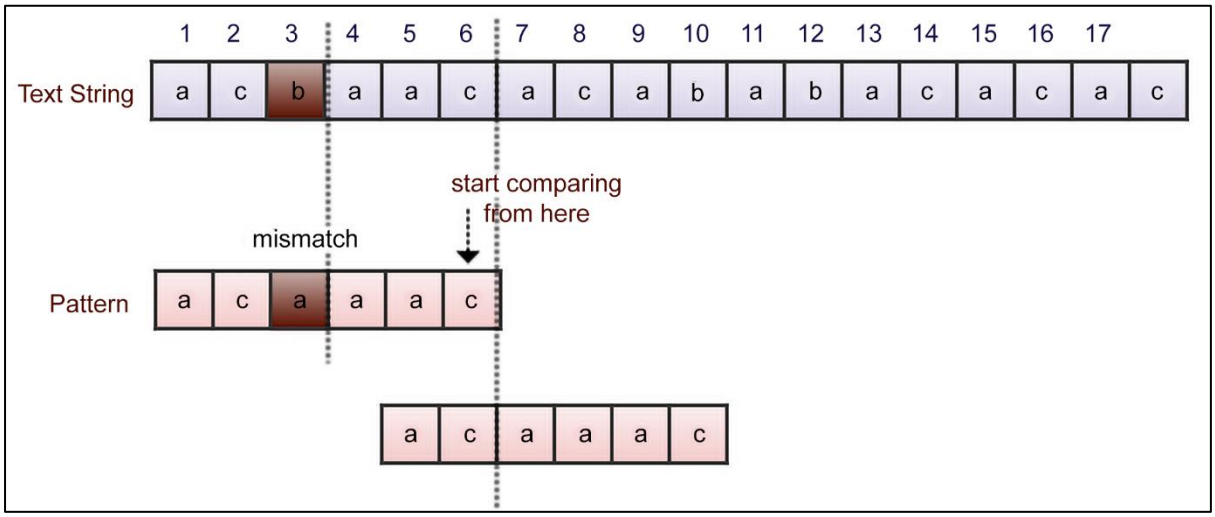

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Text String
a c a c a b a c a b a b a c a c a c
start comparing
from here
mismatch
Pattern
a c a c a c
Since mismatched character
'b' does not occur in the pattern,
we shift the pattern, we shift the
pattern next to that bad character.
a c a c a c

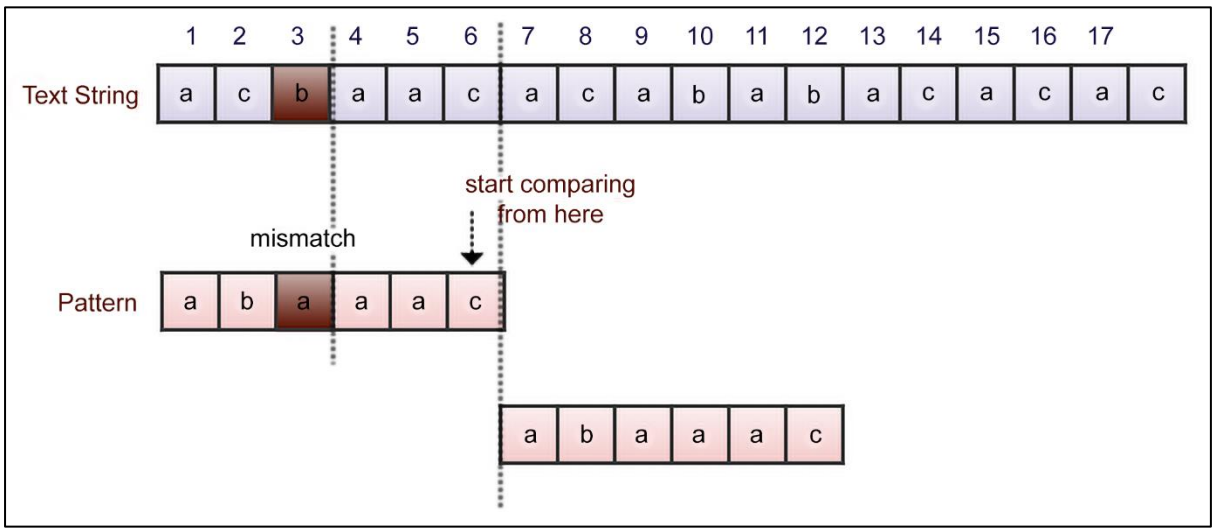

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Text String
a c a d a c a c a b a b a c a c a c
start comparing
from here
mismatch
Pattern
a c a c a c
Since mismatched character
'd' does not occur in the pattern,
we shift the pattern next to that
bad character.
a c a c a c


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Text String
a c a a a c a c a b a b a c a c a c
start comparing from here
mismatch
Pattern
a c a c a c
Option 1 for alignment
a c a c a c
Option 2 for alignment
a c a c a c
Since mismatched character 'a' has two occurrences in the pattern, we have two options to shift the pattern that can align the mismatched charecter.

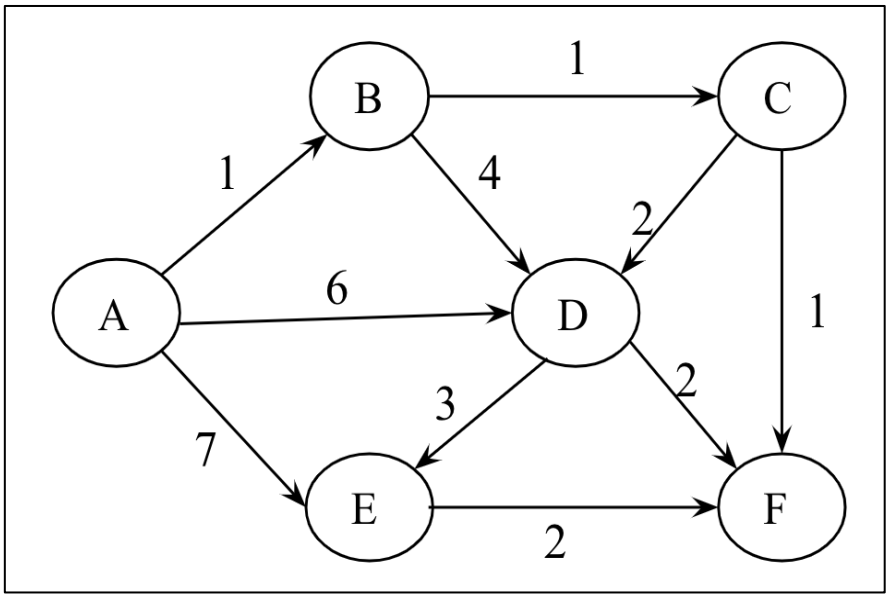

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Text String
a c a a a c a c a b a b a c a c a c
start comparing from here
mismatch
Pattern
a c a b a c
a c a b a c

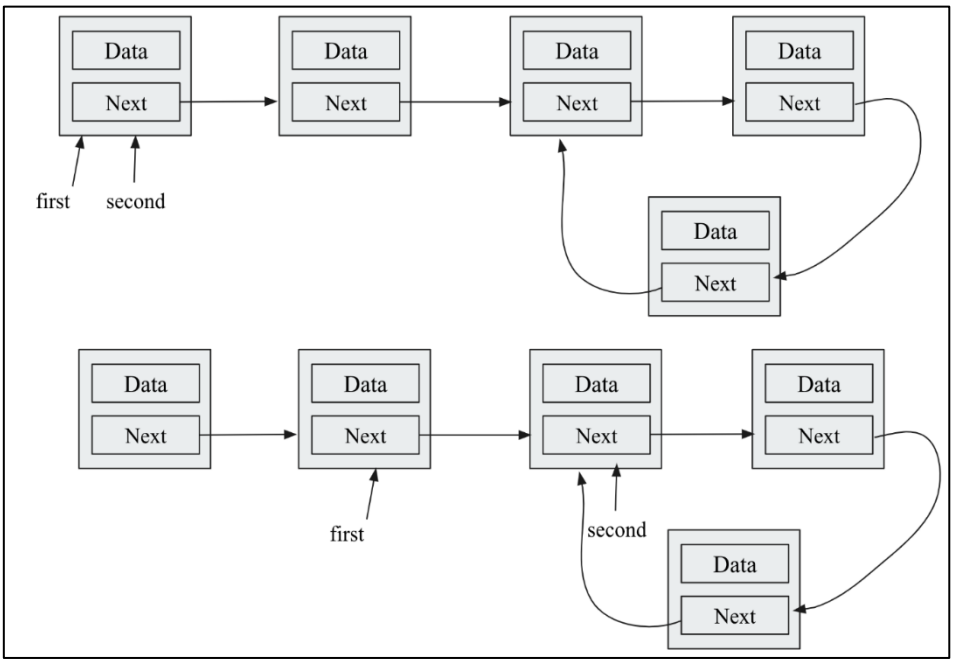

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Text String
a c a a a c a c a b a b a c a c a c
start comparing from here
mismatch
Pattern
a c a c a c
Option 1 for alignment
a c a c a c
Option 2 for alignment
a c a c a c

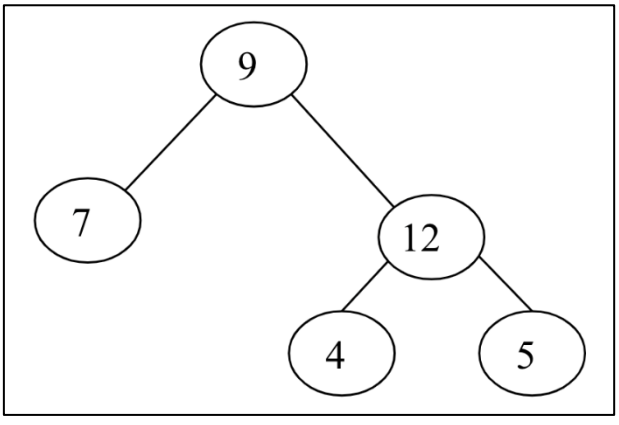

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Text String
a c b a a c a c a b a b a c a c a c
start comparing from here
mismatch
Pattern
a c a a a c
a c a a a c

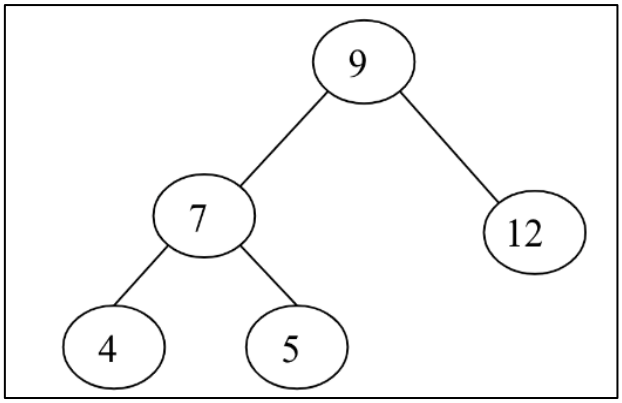

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Text String
a c b a a c a c a b a b a c a c a c
start comparing from here
mismatch
Pattern
a b a a a c
a b a a a c

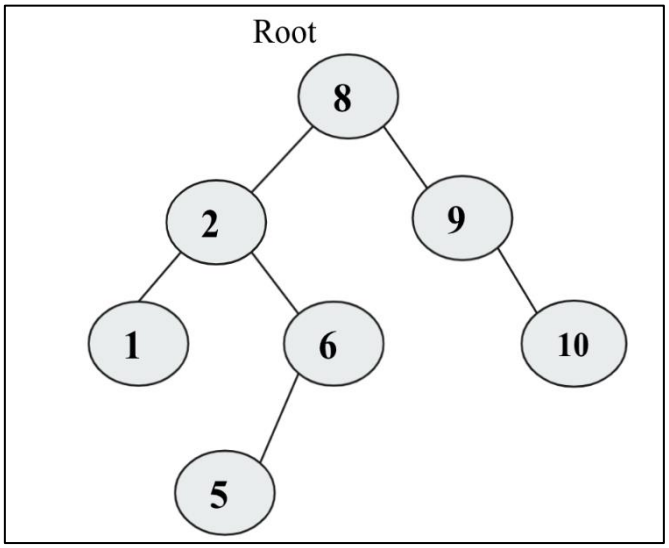
Appendix: Answers to the Questions

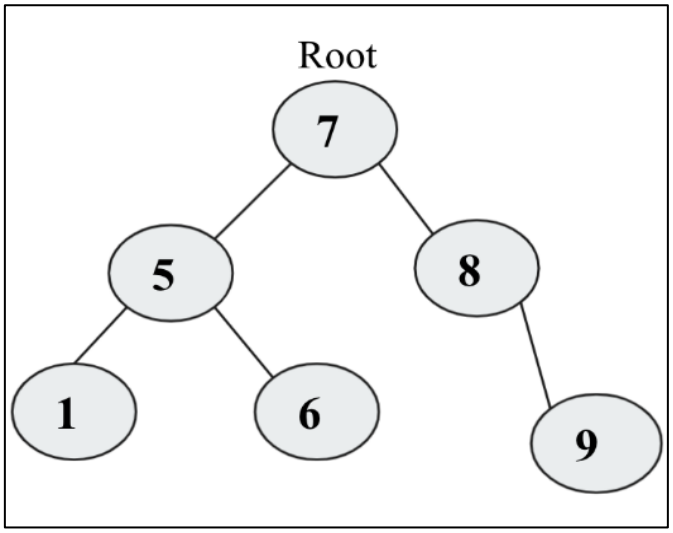


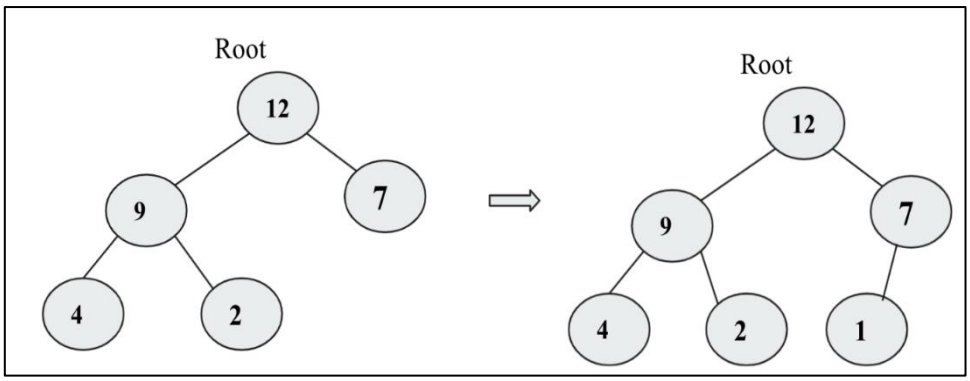


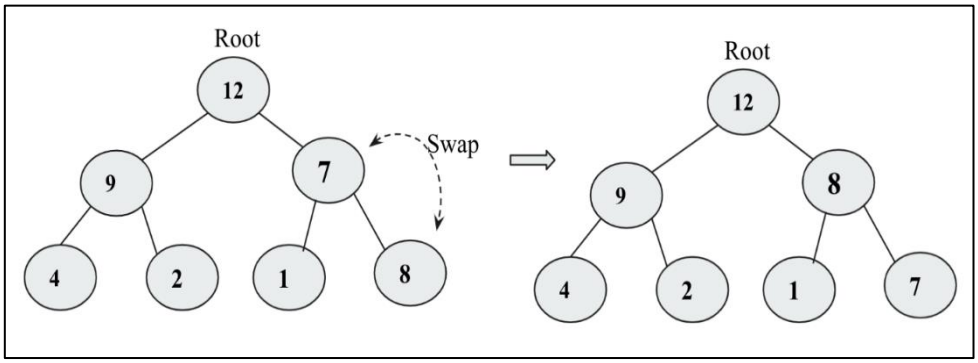


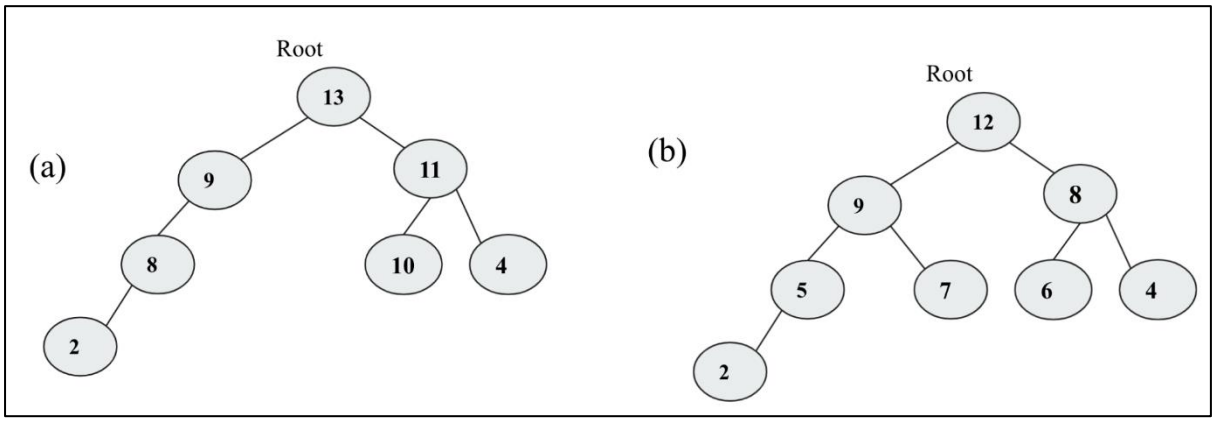

9
7
12
4
5

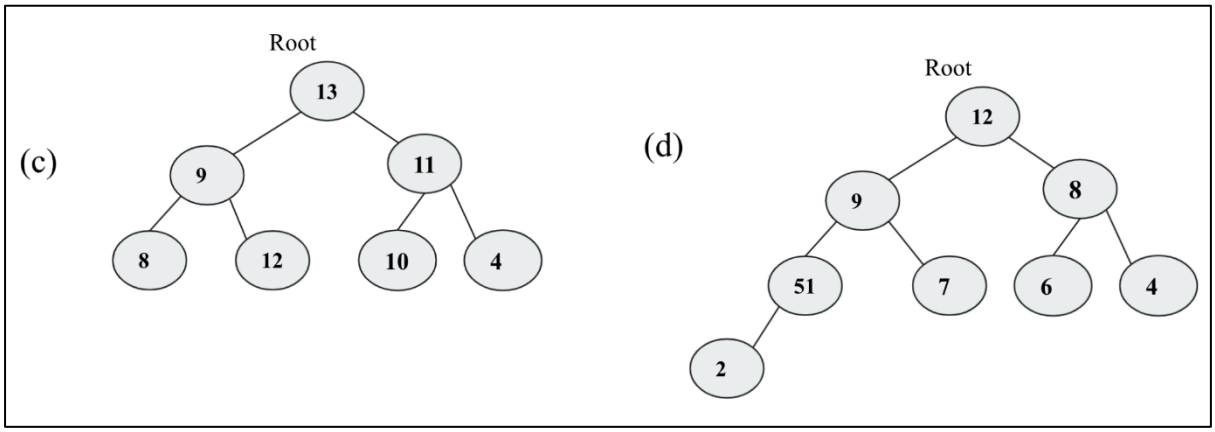

Root
8
2
9
1
6
10
5

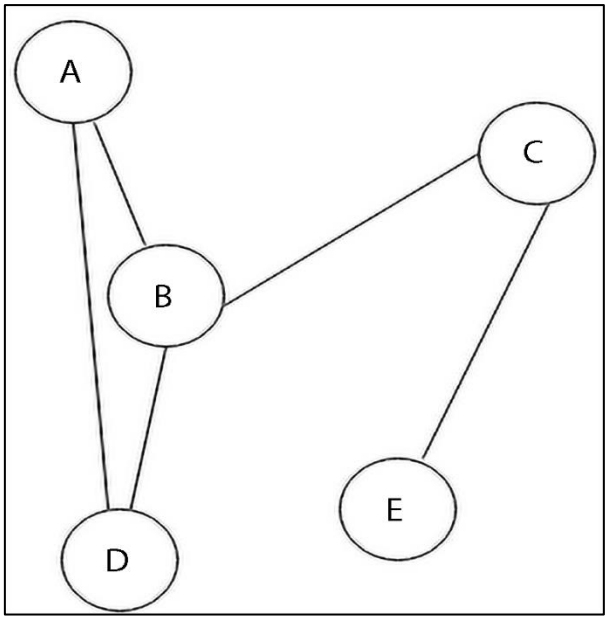

Root
7
5
8
1
6
9

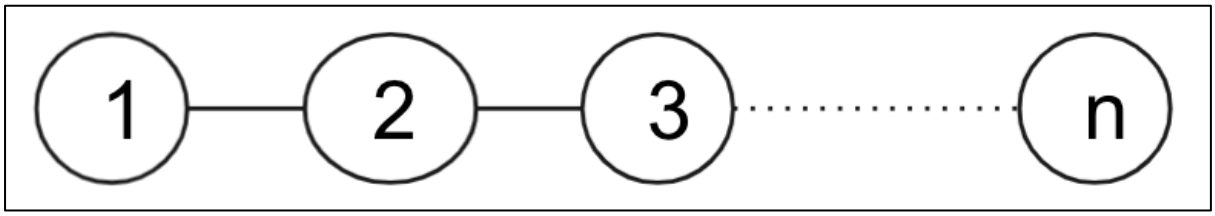

Root
12
9
7
4
2
Root
12
9
7
4
2
1

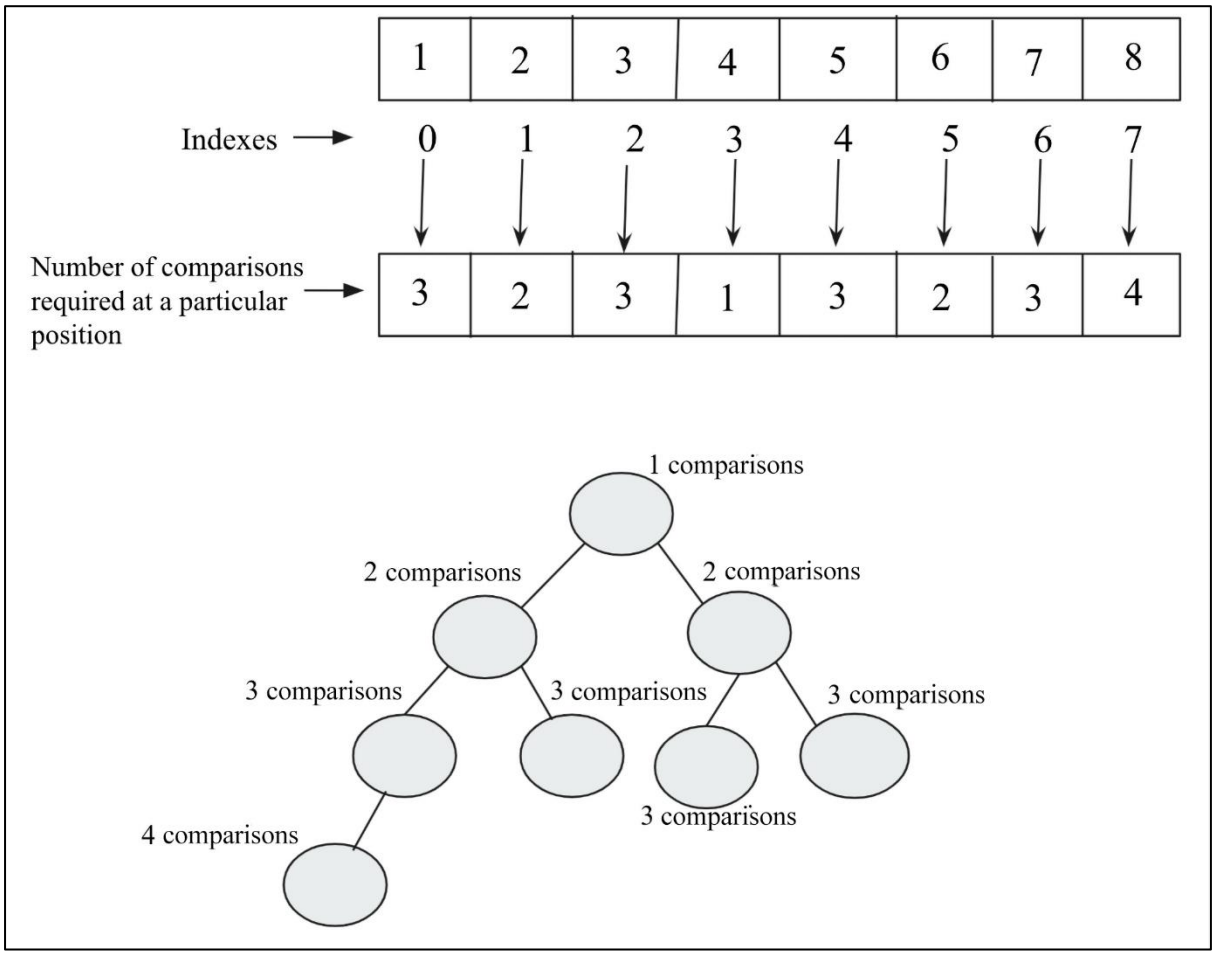

Root
12
9
7
4
2
1
8
Swap
Root
12
9
8
4
2
1
7

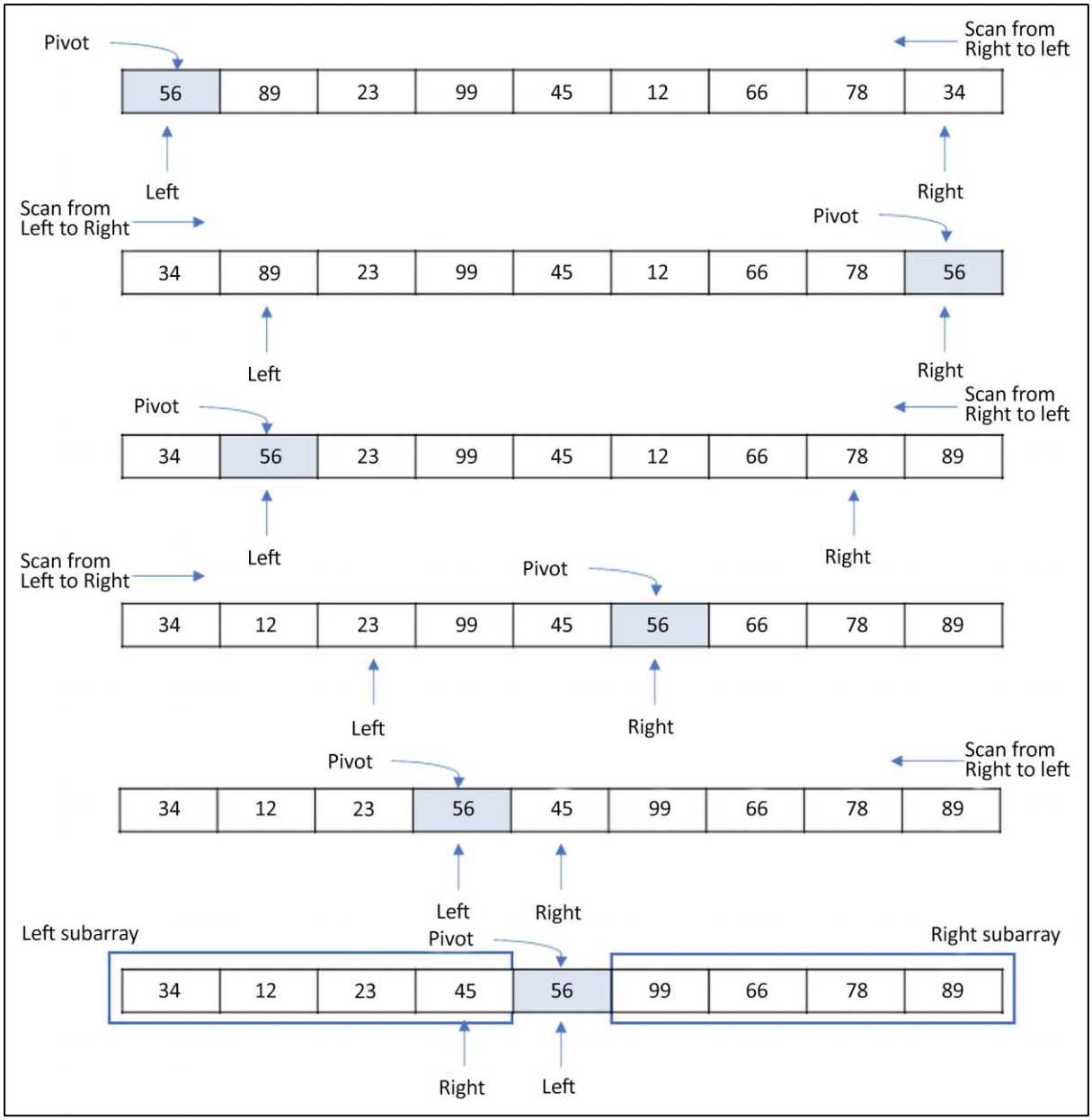

(a)
Root
13
9
11
8
10
4
2
(b)
Root
12
9
8
5
7
6
4
2

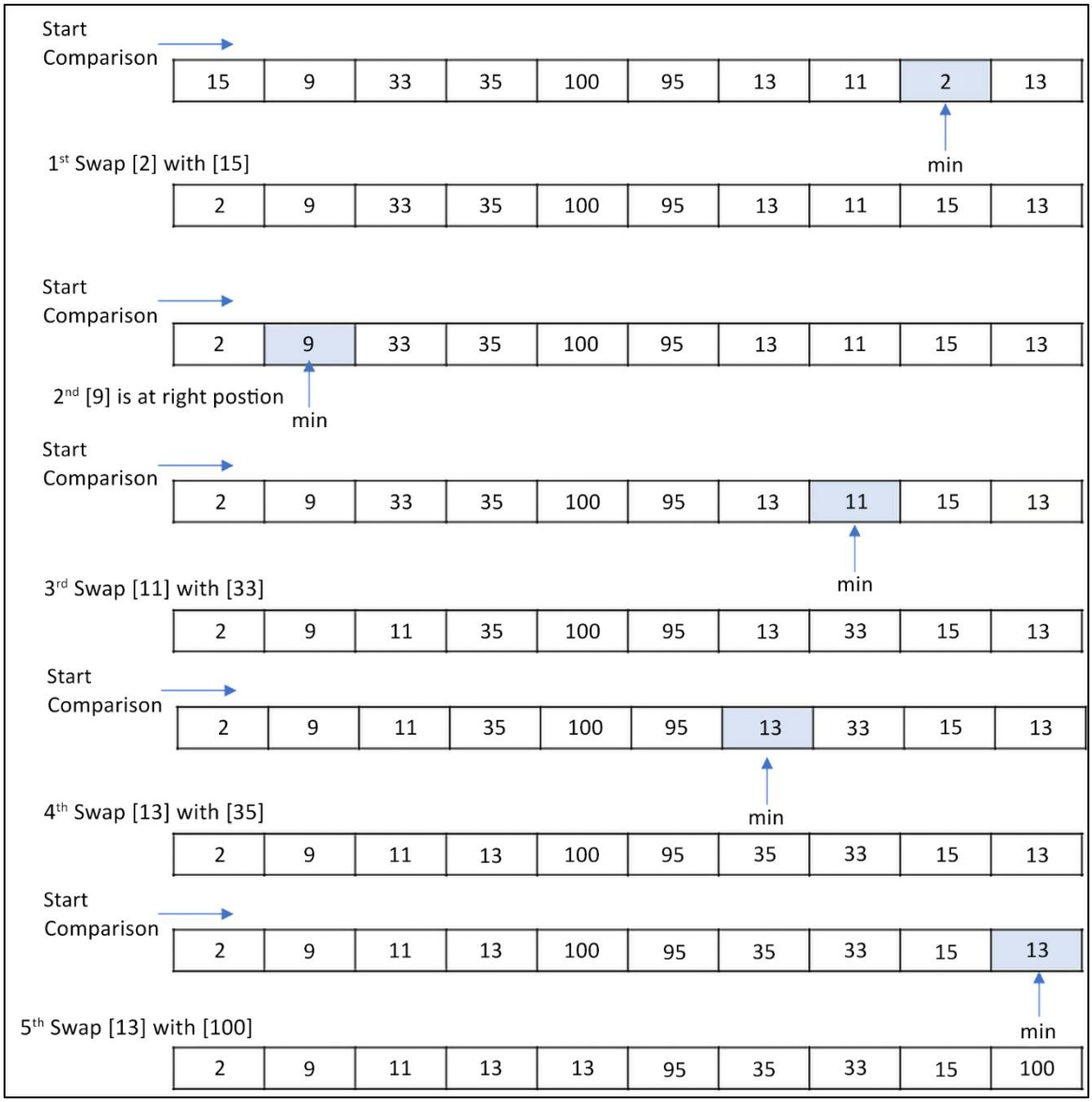

(c)
Root
13
9
11
8
12
10
4
(d)
Root
12
9
8
51
7
6
4
2


A
C
B
E
D


1
2
3
n


1 2 3 4 5 6 7 8
Indexes
0 1 2 3 4 5 6 7
Number of comparisons required at a particular position
3 2 3 1 3 2 3 4
1 comparisons
2 comparisons
2 comparisons
3 comparisons
3 comparisons
3 comparisons
3 comparisons
4 comparisons


Pivot
Scan from Right to left
56 89 23 99 45 12 66 78 34
Left
Right
Scan from Left to Right
Pivot
34 89 23 99 45 12 66 78 56
Left
Right
Pivot
Scan from Right to left
34 56 23 99 45 12 66 78 89
Left
Right
Scan from Left to Right
Pivot
34 12 23 99 45 56 66 78 89
Left
Right
Pivot
Scan from Right to left
34 12 23 56 45 99 66 78 89
Left
Right
Left subarray
Pivot
Right subarray
34 12 23 45 56 99 66 78 89
Right
Left


Start
Comparison
15 9 33 35 100 95 13 11 2 13
min
1st Swap [2] with [15]
2 9 33 35 100 95 13 11 15 13
Start
Comparison
2 9 33 35 100 95 13 11 15 13
2nd [9] is at right postion
min
Start
Comparison
2 9 33 35 100 95 13 11 15 13
min
3rd Swap [11] with [33]
2 9 11 35 100 95 13 33 15 13
Start
Comparison
2 9 11 35 100 95 13 33 15 13
min
4th Swap [13] with [35]
2 9 11 13 100 95 35 33 15 13
Start
Comparison
2 9 11 13 100 95 35 33 15 13
min
5th Swap [13] with [100]
2 9 11 13 13 95 35 33 15 100