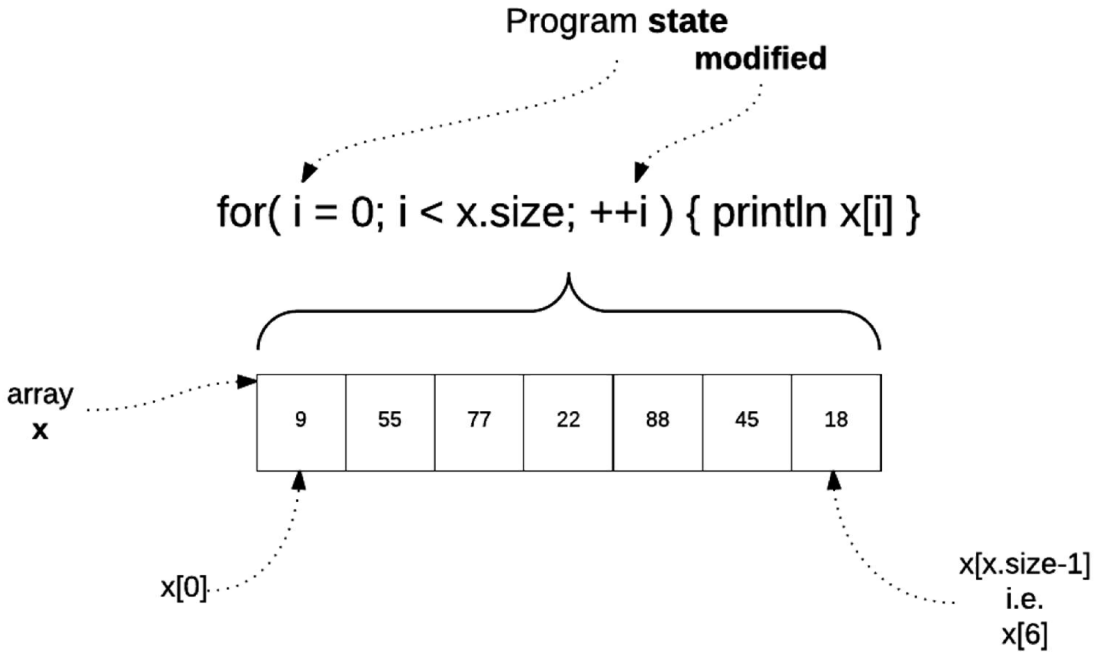
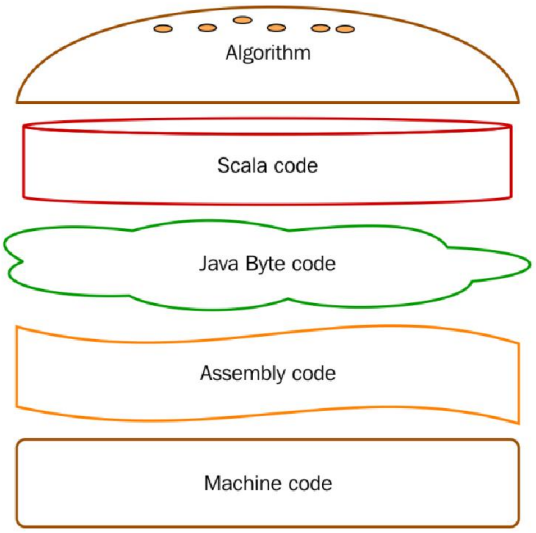
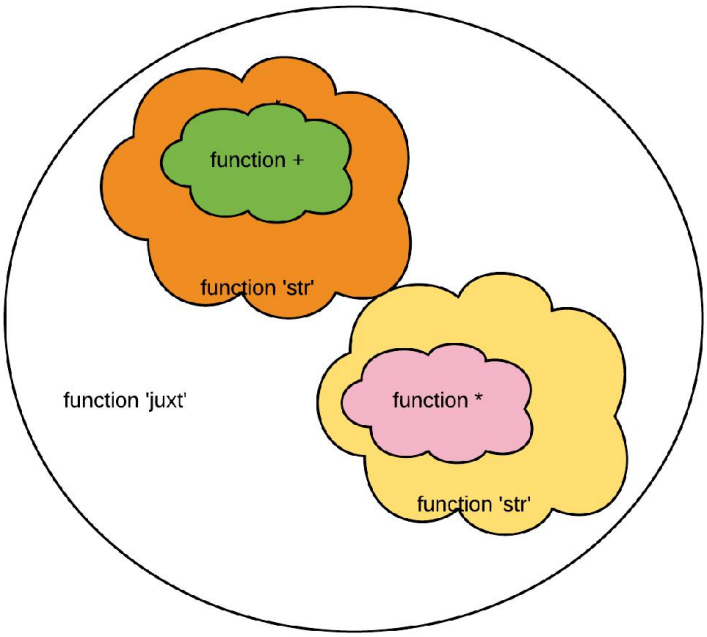
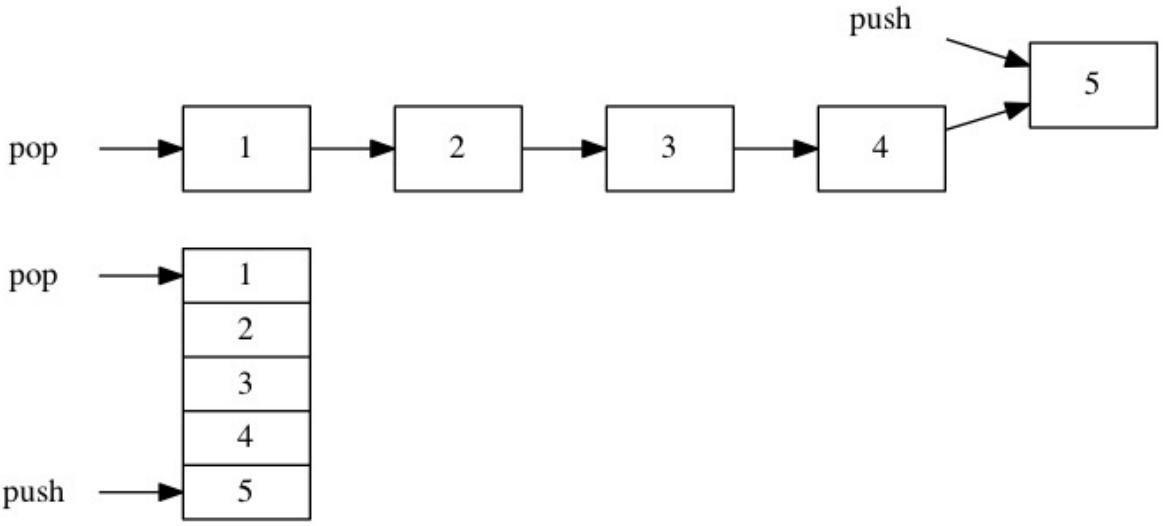
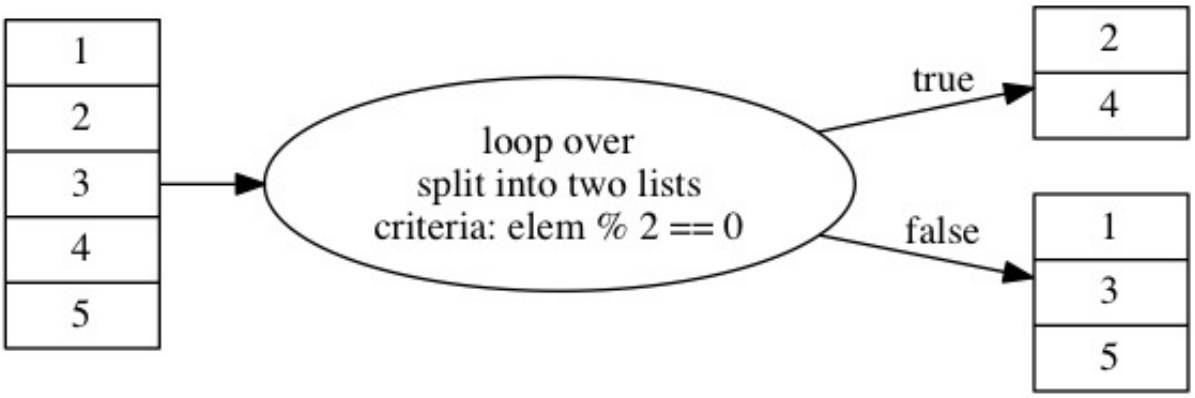
Chapter 1: Why Functional Programming?

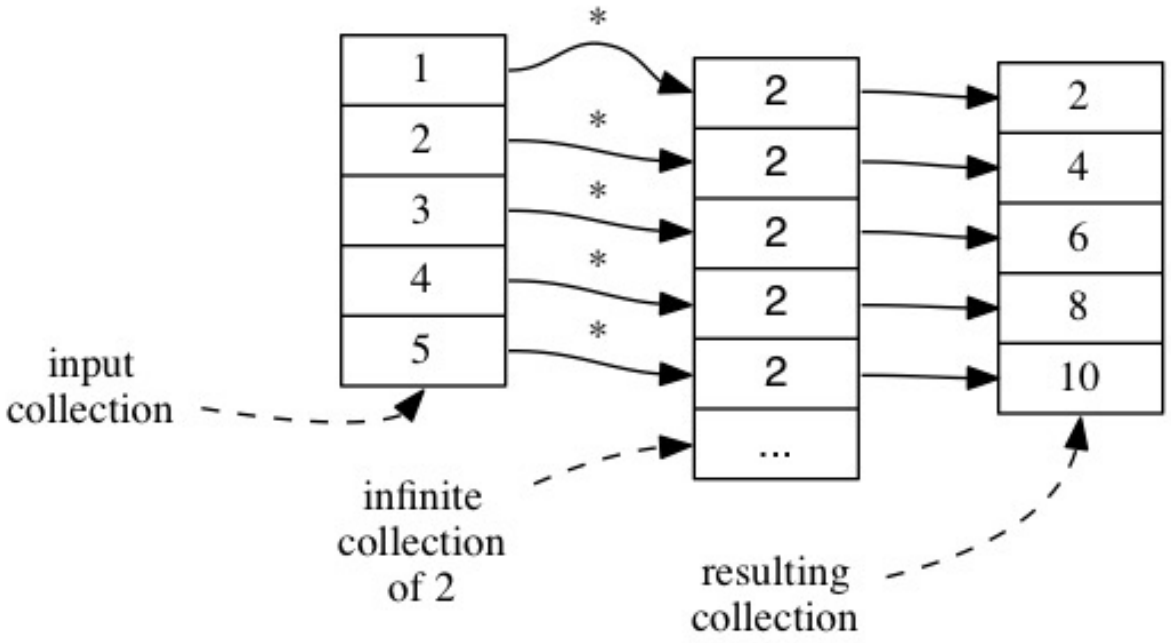


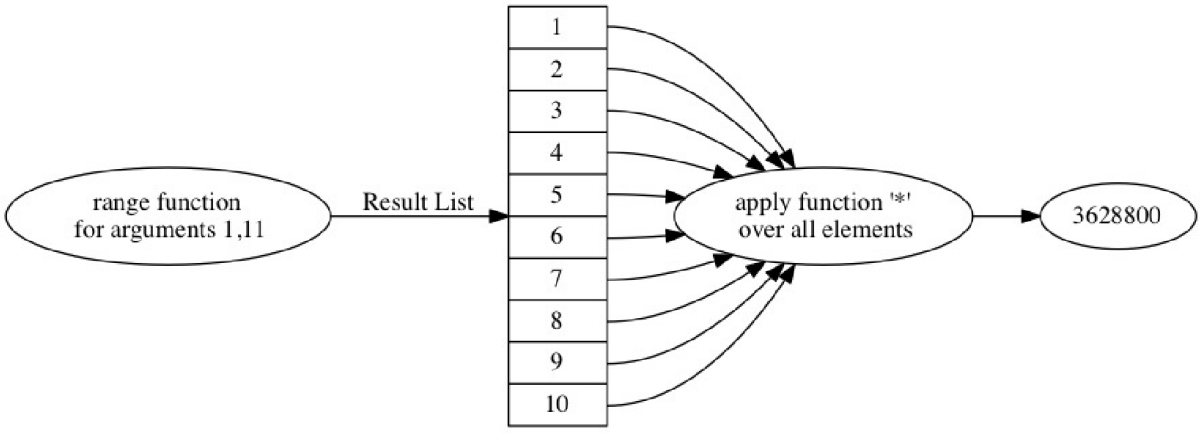


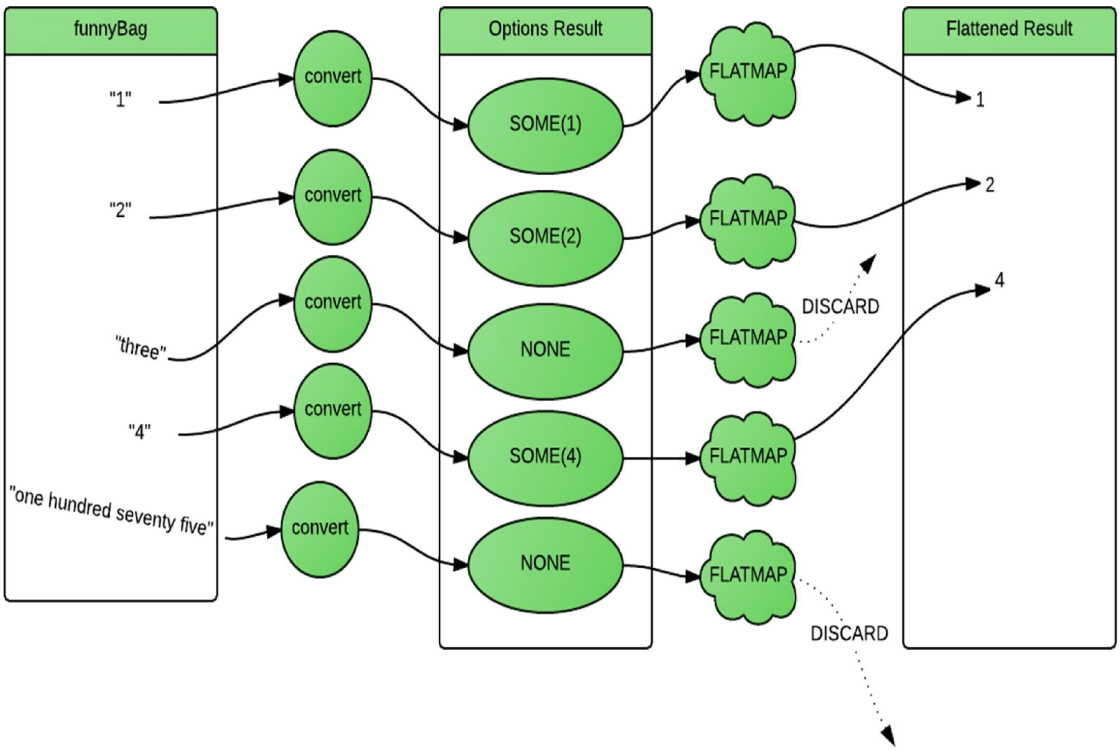

function +
function 'str'
function 'juxt'
function *
function 'str'

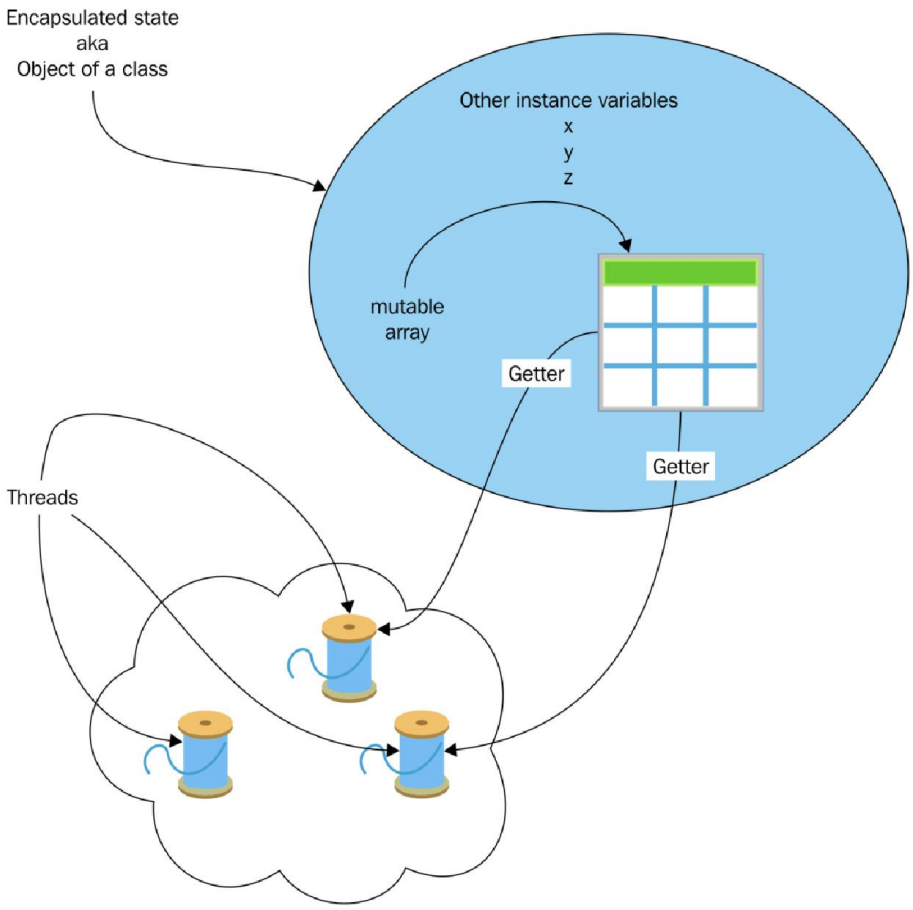

push
5
pop
1
2
3
4
pop
1
2
3
4
push
5

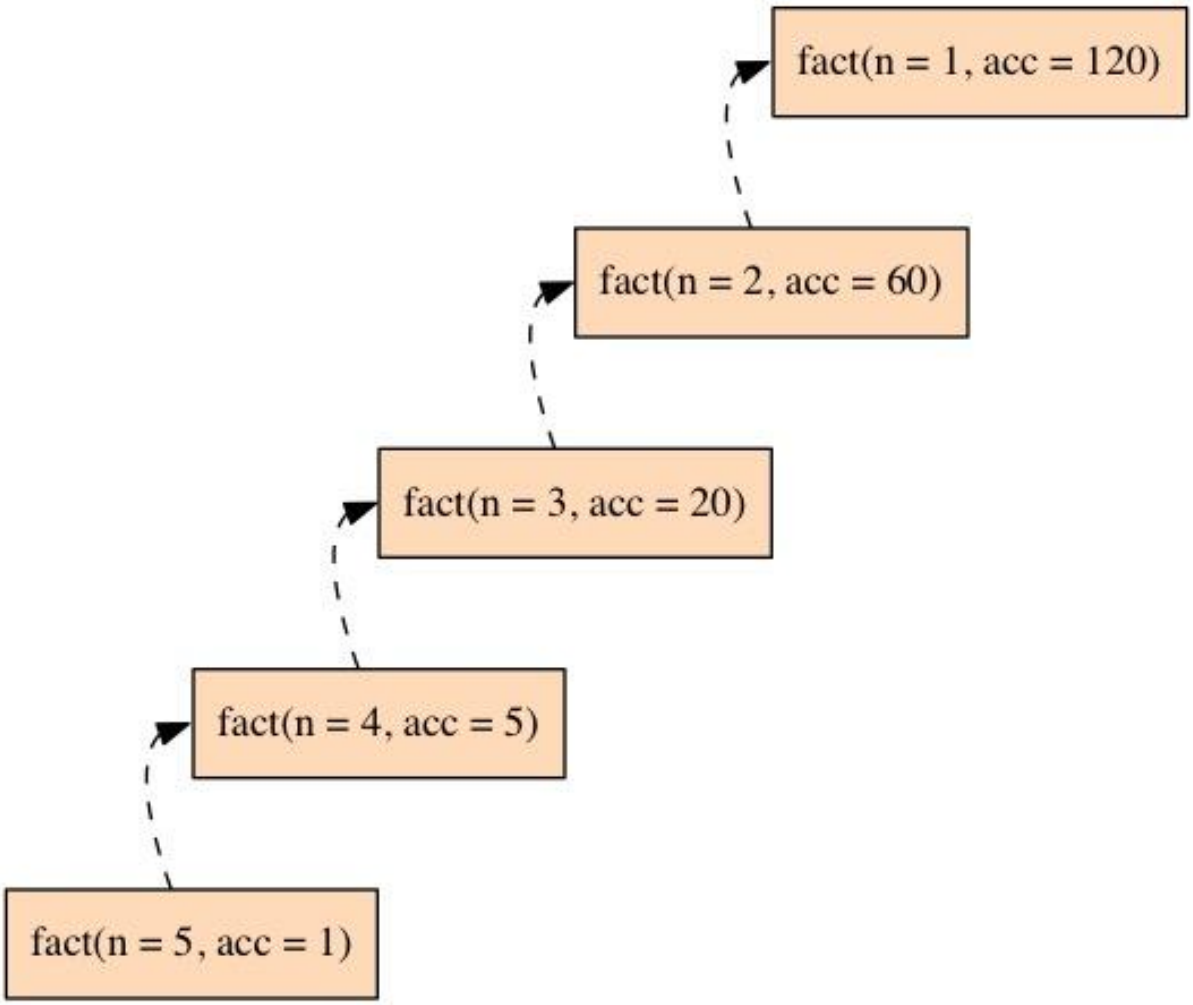

1
2
3
4
5
loop over
split into two lists
criteria: elem % 2 == 0
true
false
2
4
1
3
5

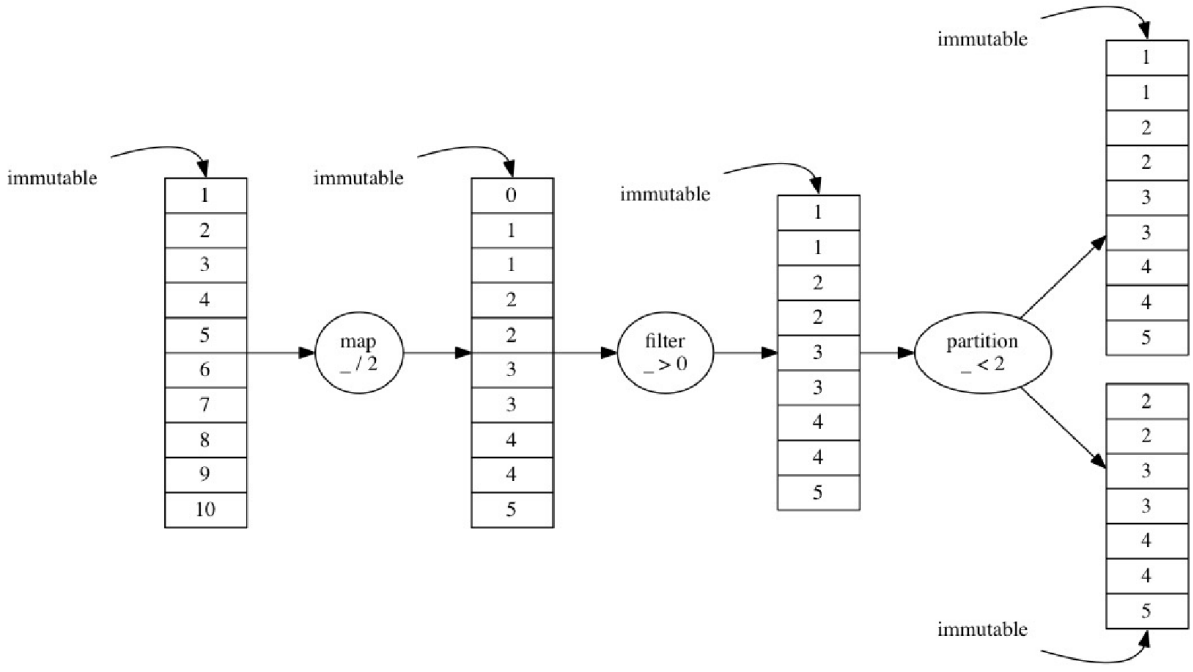

input
collection
infinite
collection
of 2
resulting
collection
*
1
2
3
4
5
2
...
6
8
10

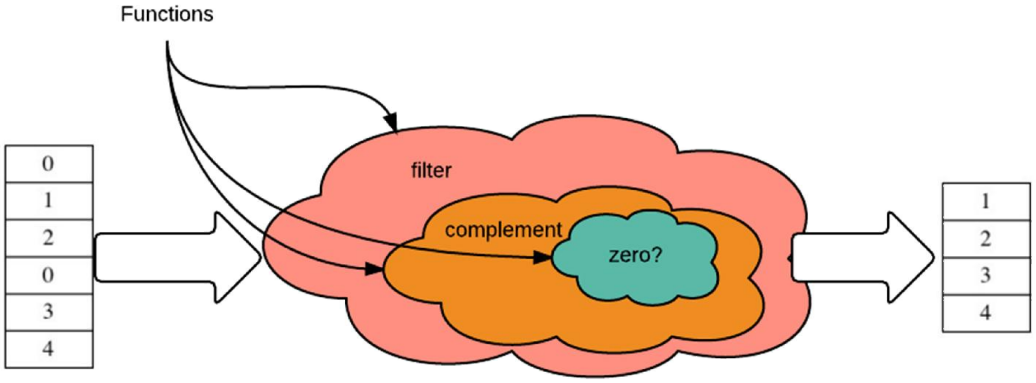

range function
for arguments 1,11
Result List
apply function '*'
over all elements
3628800
1
2
3
4
5
6
7
8
9
10

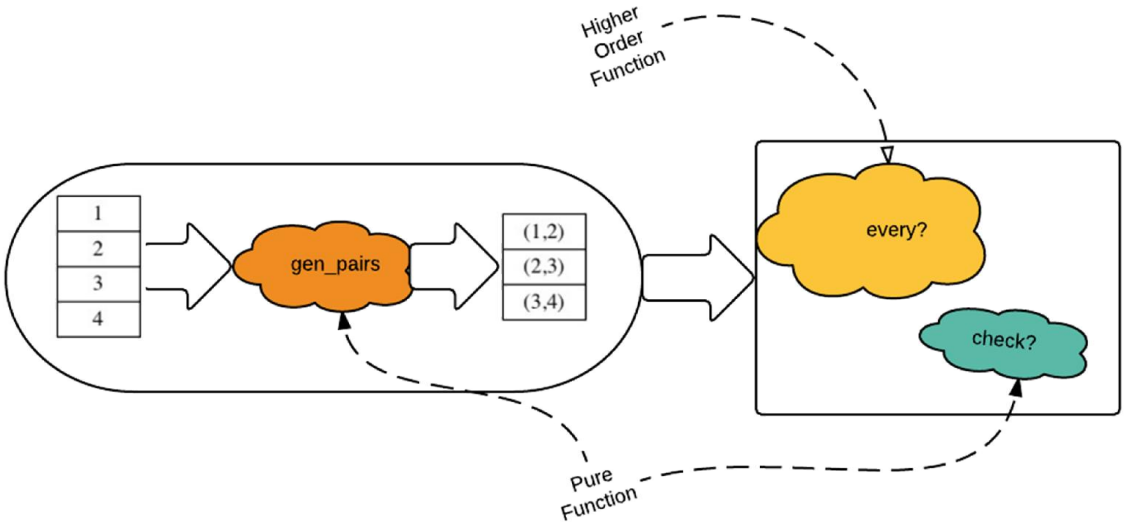

funnyBag
"1"
"2"
"three"
"4"
"one hundred seventy five"
convert
convert
convert
convert
convert
Options Result
SOME(1)
SOME(2)
NONE
SOME(4)
NONE
FLATMAP
FLATMAP
FLATMAP
FLATMAP
FLATMAP
DISCARD
DISCARD
Flattened Result
1
2
4


Encapsulated state
aka
Object of a class
Other instance variables
x
y
z
mutable
array
Getter
Getter
Threads

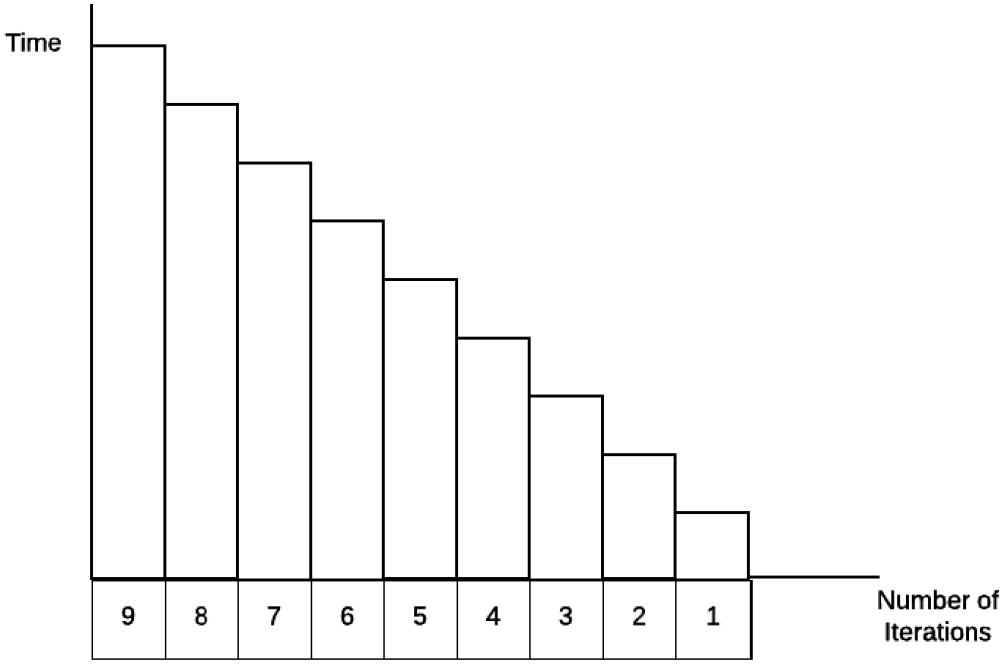

fact(n = 1, acc = 120)
fact(n = 2, acc = 60)
fact(n = 3, acc = 20)
fact(n = 4, acc = 5)
fact(n = 5, acc = 1)


immutable
1
2
3
4
5
6
7
8
9
10
map
_ / 2
immutable
0
1
1
2
2
3
3
4
4
5
filter
_ > 0
immutable
1
1
2
2
3
3
4
4
5
partition
_ < 2
immutable
1
1
2
2
3
3
4
4
5
immutable
2
2
3
3
4
4
5


Functions
0
1
2
0
3
4
filter
complement
zero?
1
2
3
4

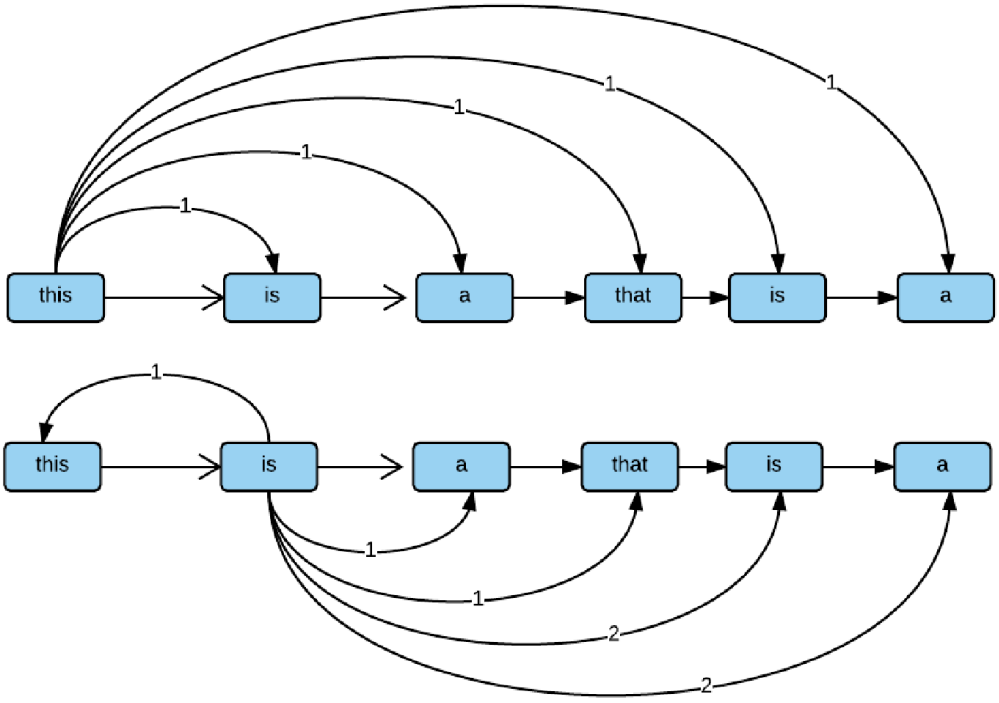

Higher
Order
Function
1
2
3
4
gen_pairs
(1,2)
(2,3)
(3,4)
every?
check?
Pure
Function

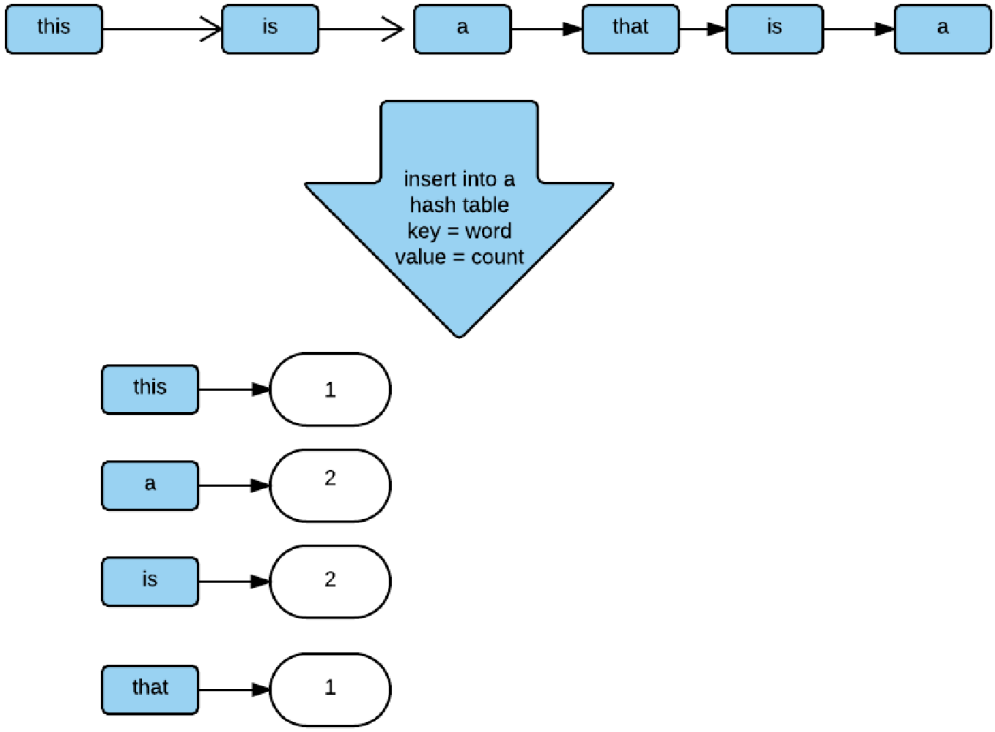
Chapter 2: Building Blocks

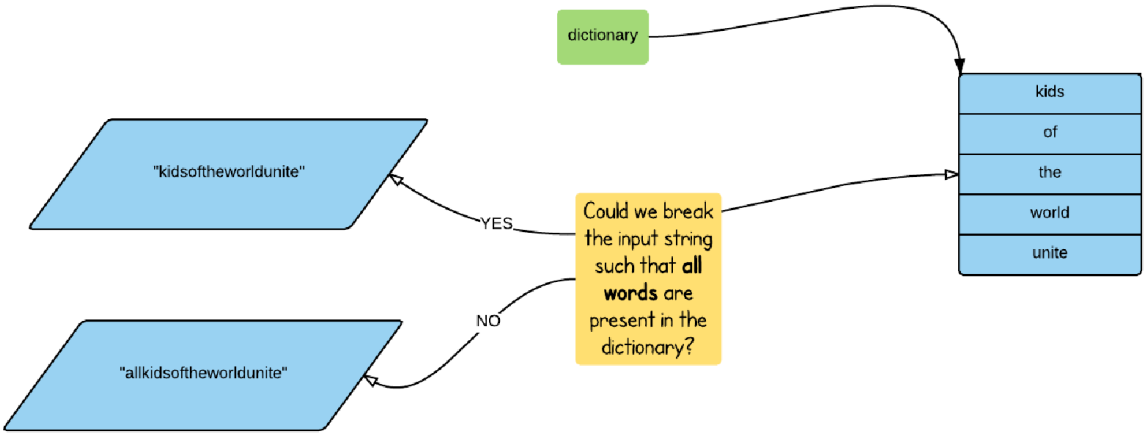


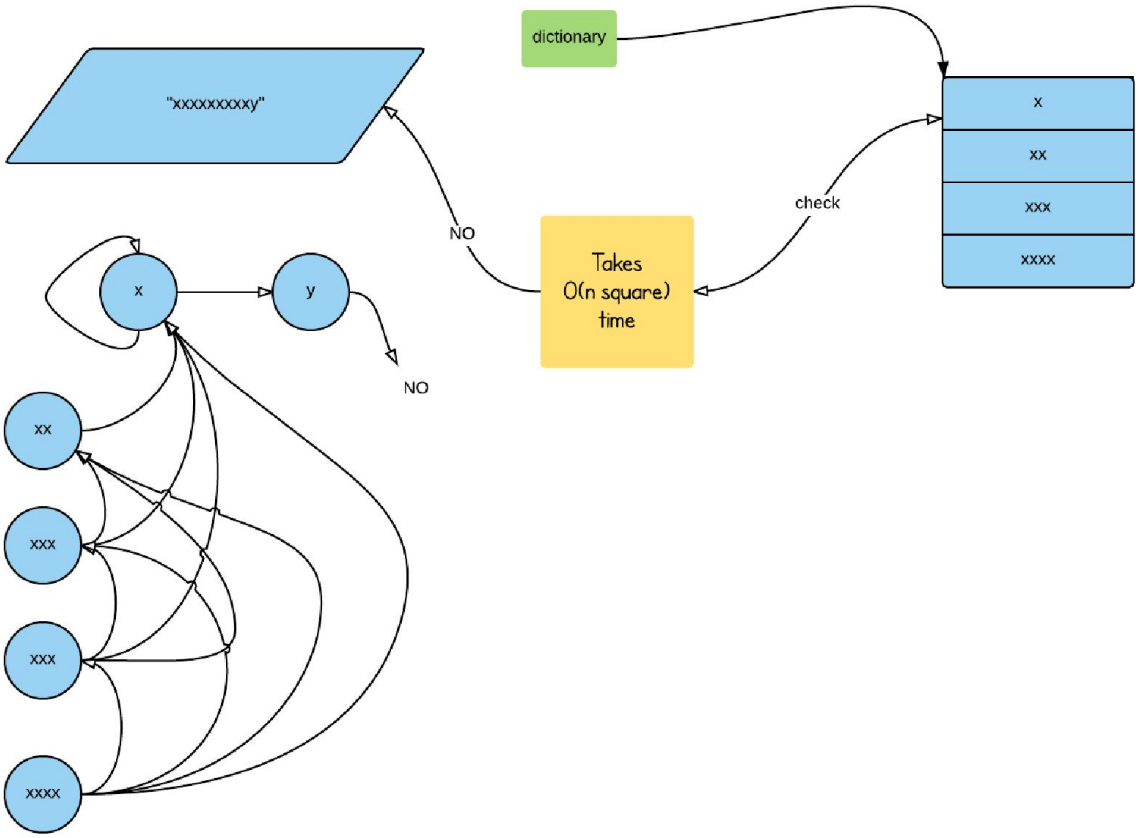
Num Elems	Num Iterations	(Num*Num)/2
9	55	40
20	210	200
30	465	450
100	5050	5000
1000	500500	500000

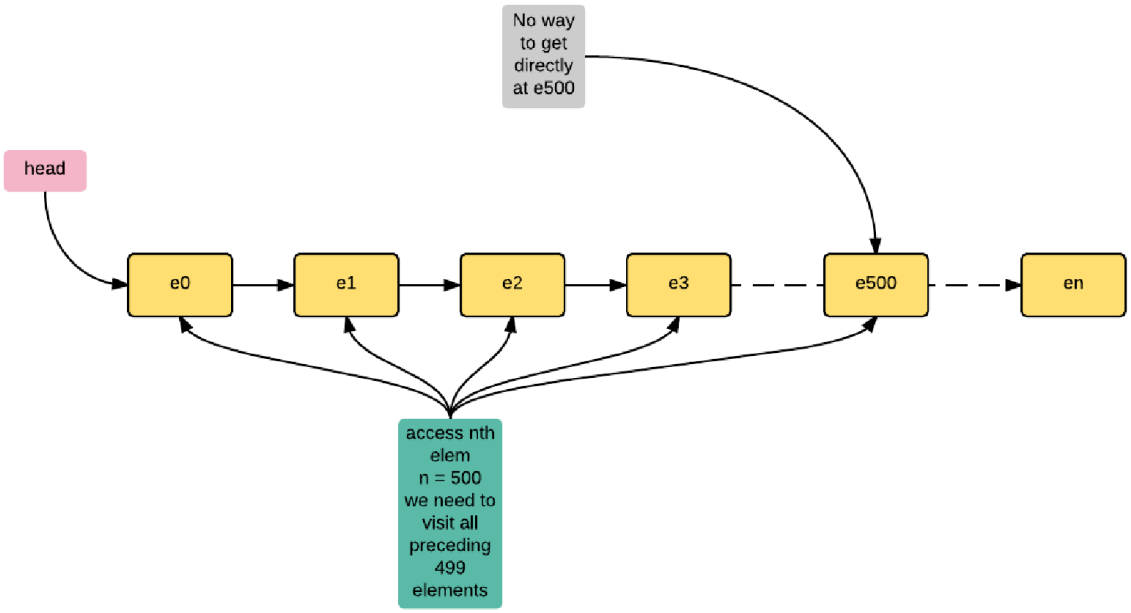
Num Elems	logN
256	8
4096	12
16384	14
65536	16
1048576	20

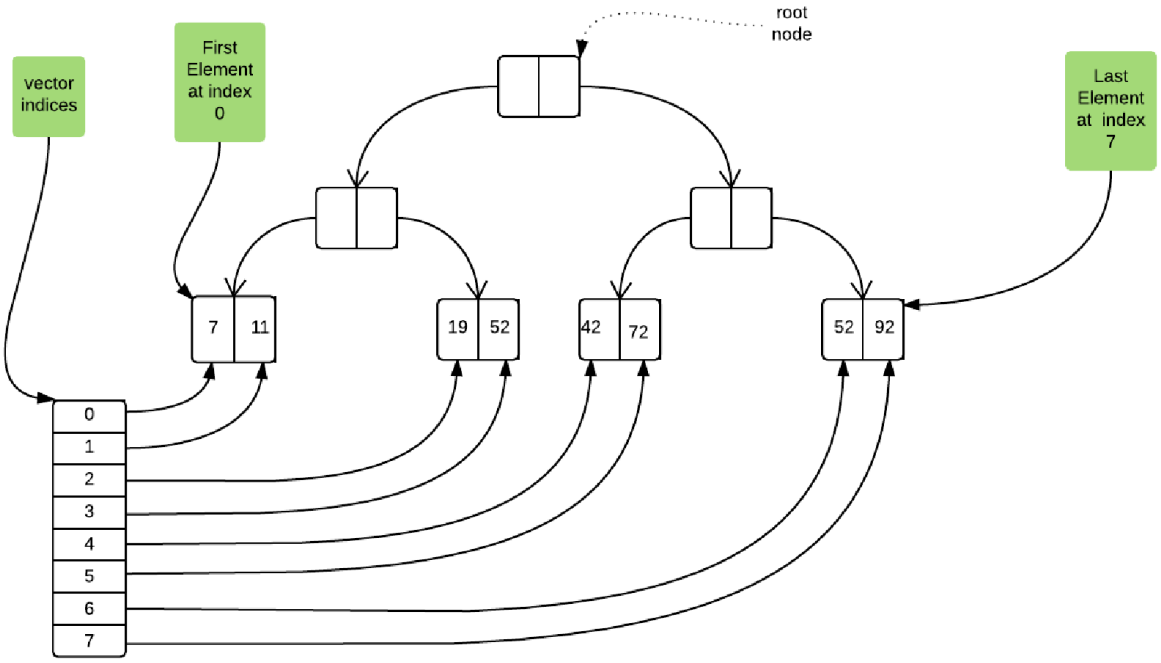


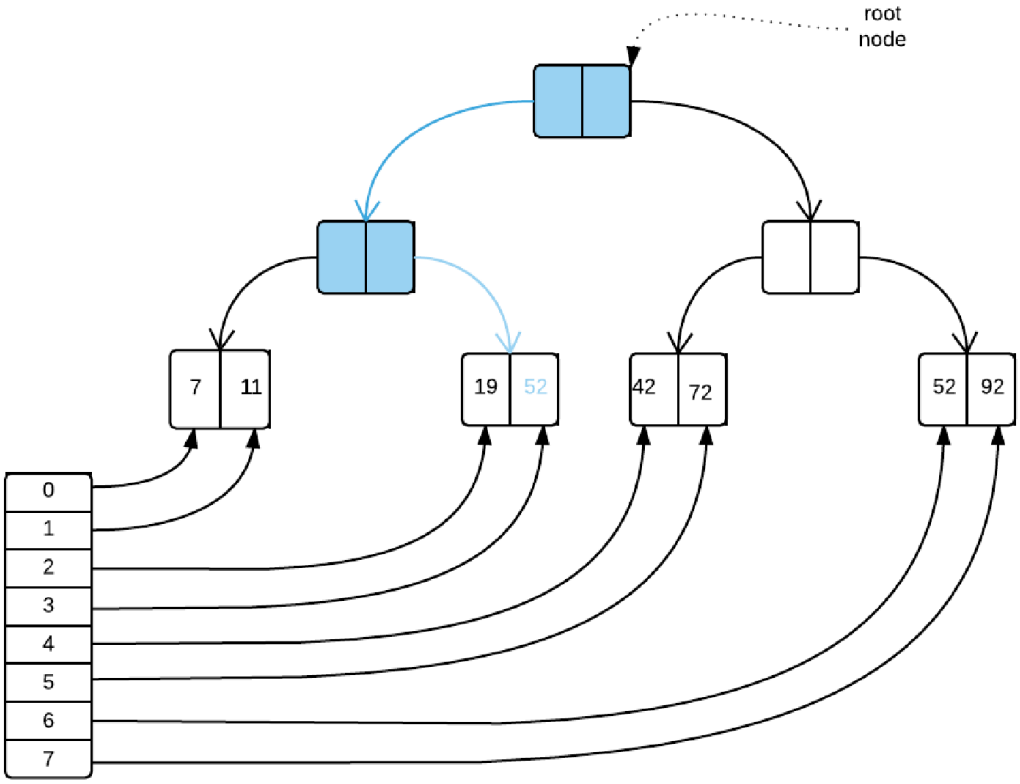

this
is
a
that
is
a
insert into a
hash table
key = word
value = count
this
1
a
2
is
2
that
1

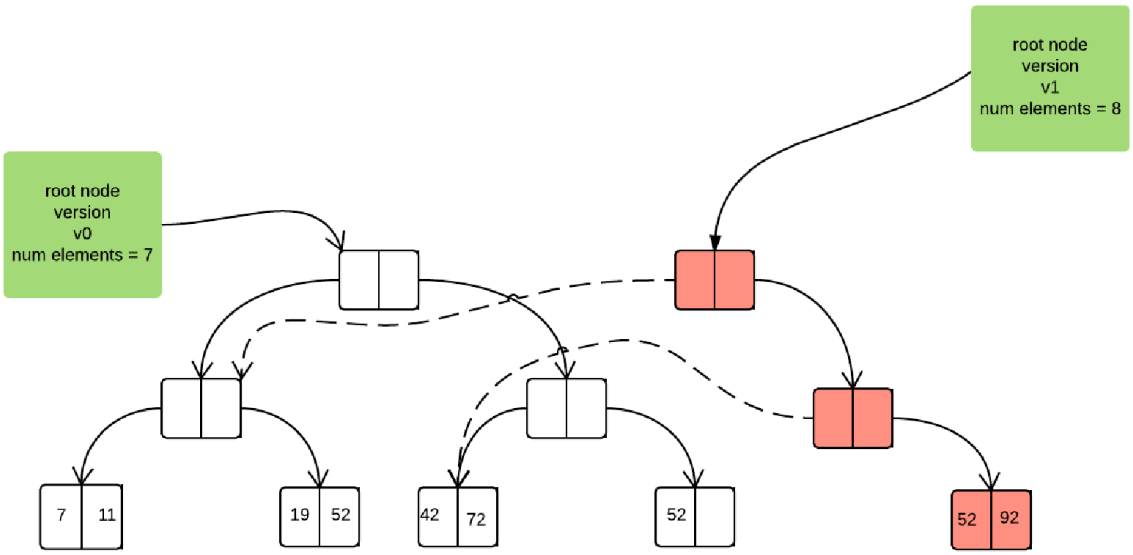

dictionary
kids
of
the
world
unite
"kidsoftheworldunite"
YES
Could we break the input string such that all words are present in the dictionary?
NO
"allkidsoftheworldunite"

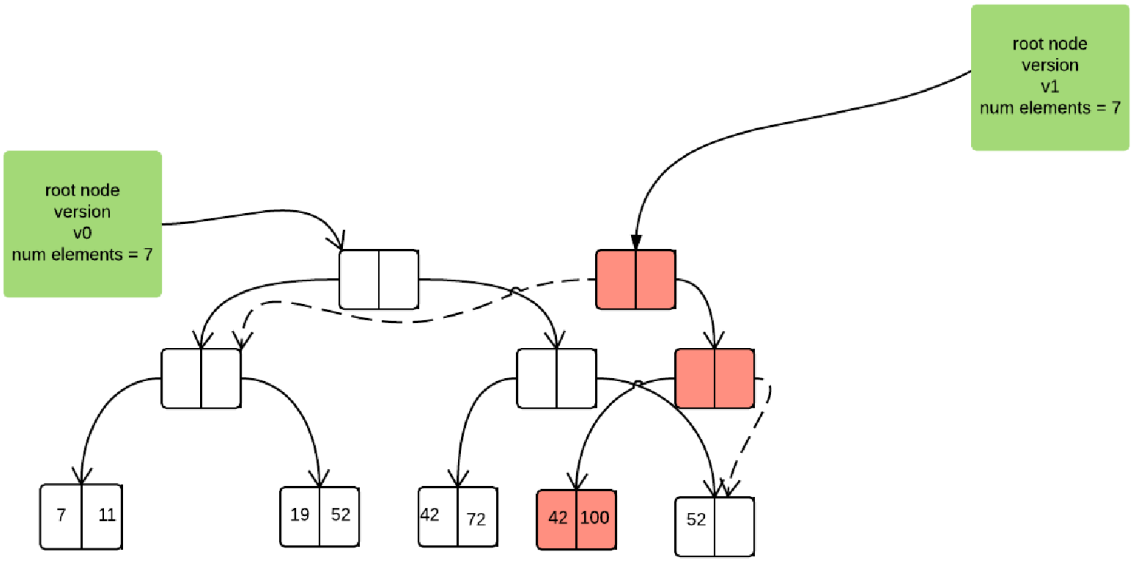

dictionary
"xxxxxxxxy"
x
xx
xxx
xxxx
check
NO
Takes
O(n square)
time
x
y
NO
xx
xxx
xxx
xxxx

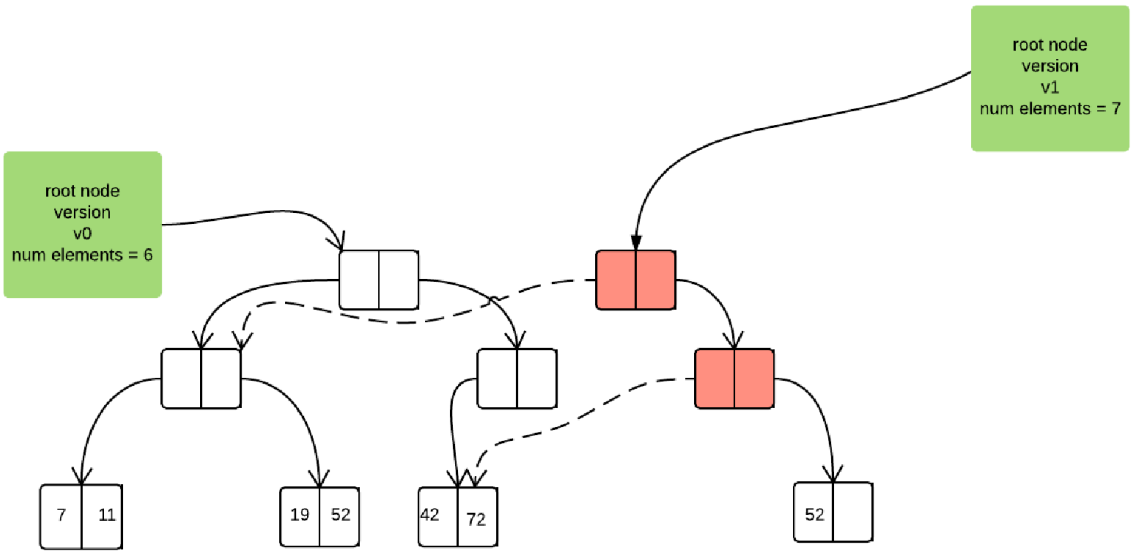

No way
to get
directly
at e500
head
e0
e1
e2
e3
e500
en
access nth
elem
n = 500
we need to
visit all
preceding
499
elements

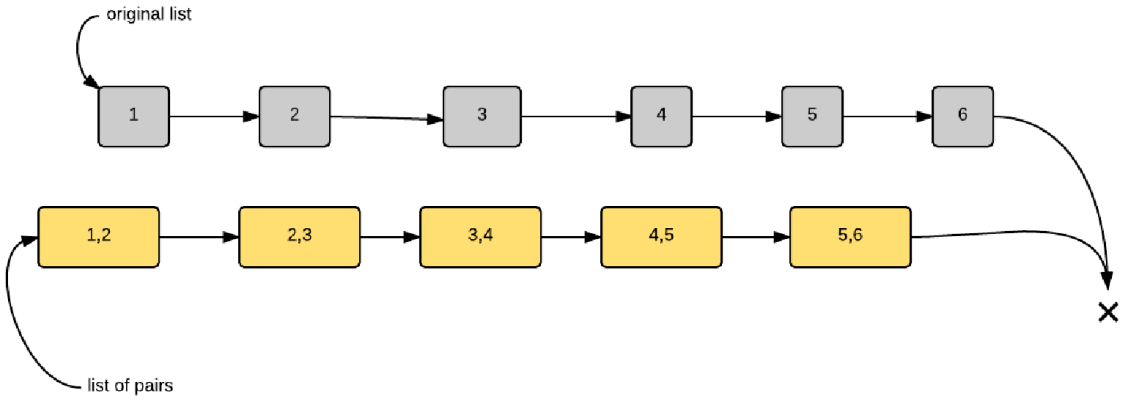

root
node
vector
indices
First
Element
at index
0
Last
Element
at index
7
7
11
19
52
42
72
52
92
0
1
2
3
4
5
6
7

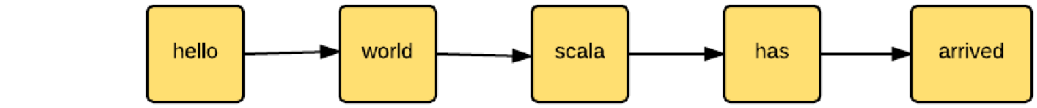

root
node
7
11
19
52
42
72
52
92
0
1
2
3
4
5
6
7

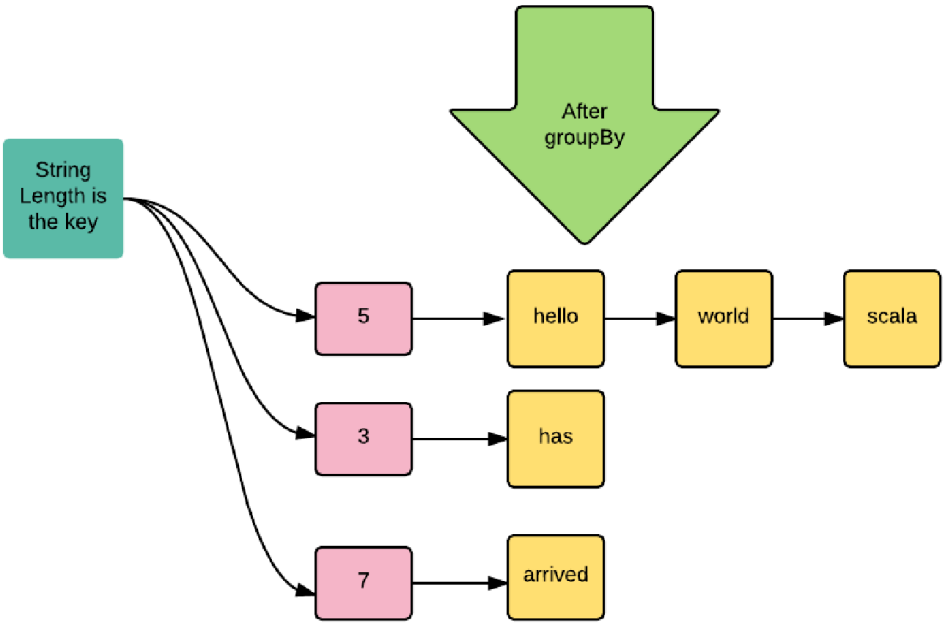

root node
version
v1
num elements = 8
root node
version
v0
num elements = 7
7
11
19
52
42
72
52
52
92

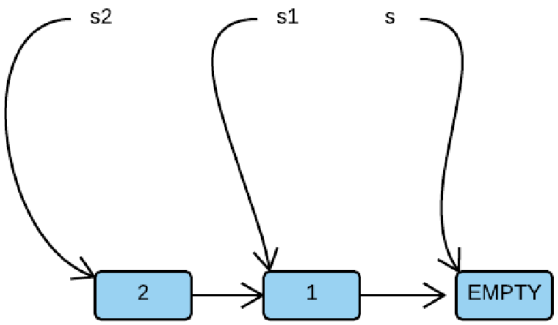

root node
version
v1
num elements = 7
root node
version
v0
num elements = 7
7
11
19
52
42
72
42
100
52


root node
version
v1
num elements = 7
root node
version
v0
num elements = 6
7
11
19
52
42
72
52

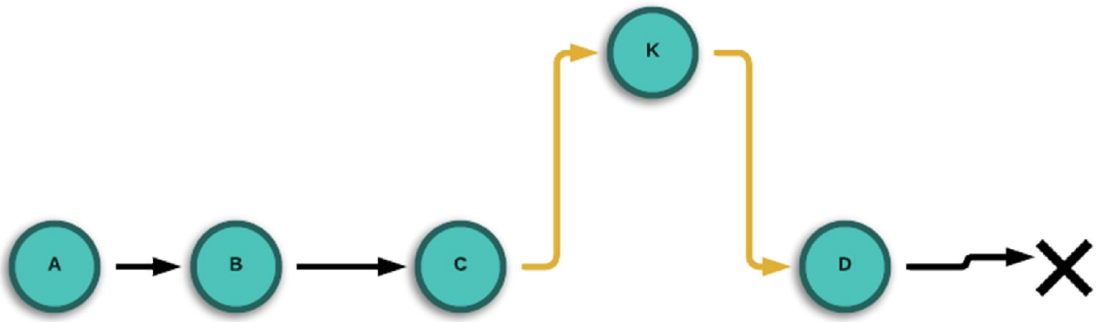

original list
1
2
3
4
5
6
1,2
2,3
3,4
4,5
5,6
list of pairs

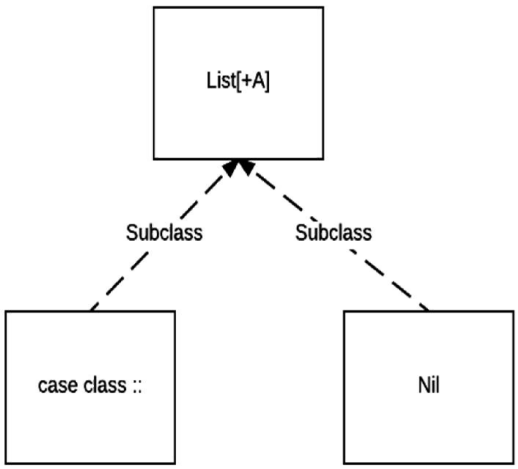

hello
world
scala
has
arrived

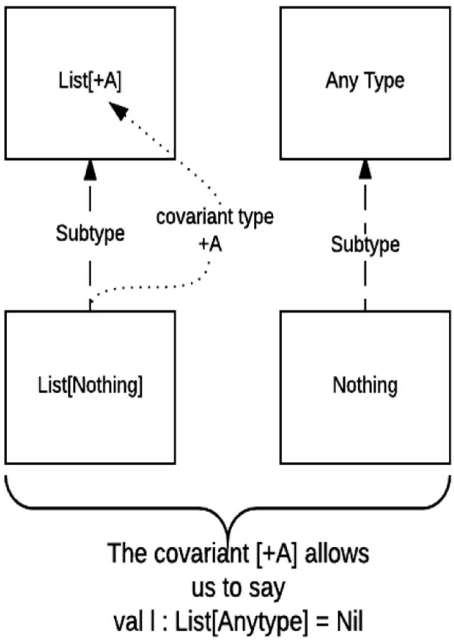

After
groupBy
String
Length is
the key
5
hello
world
scala
3
has
7
arrived

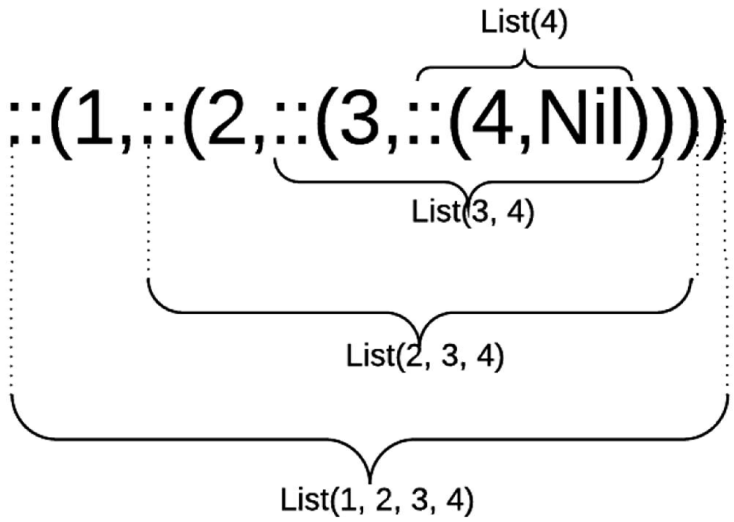

s2
s1
s
2
1
EMPTY

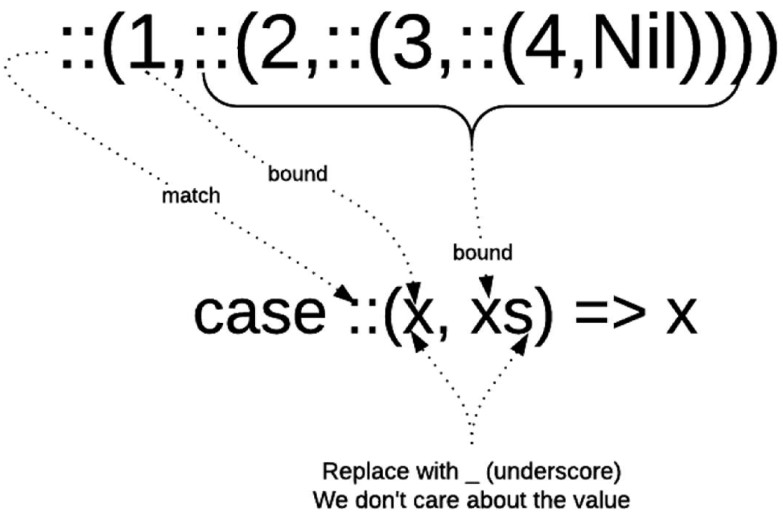
Chapter 3: Lists

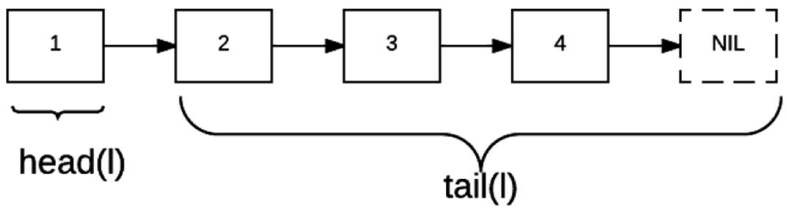


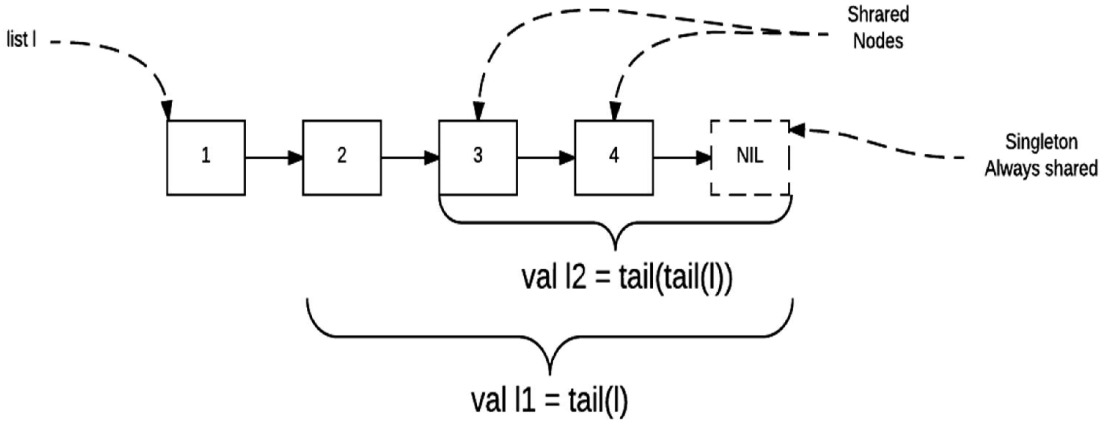


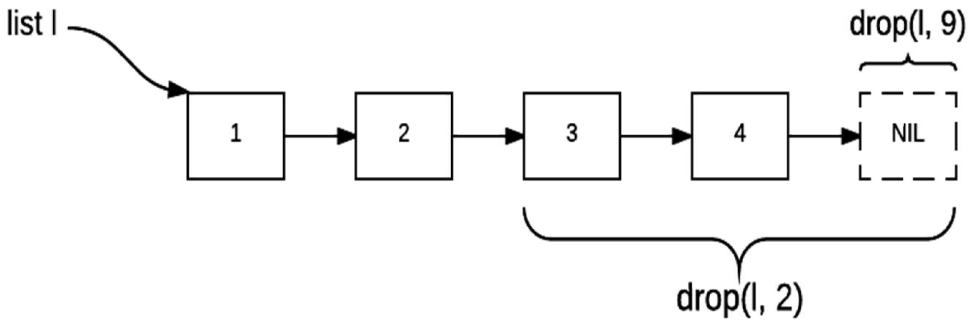


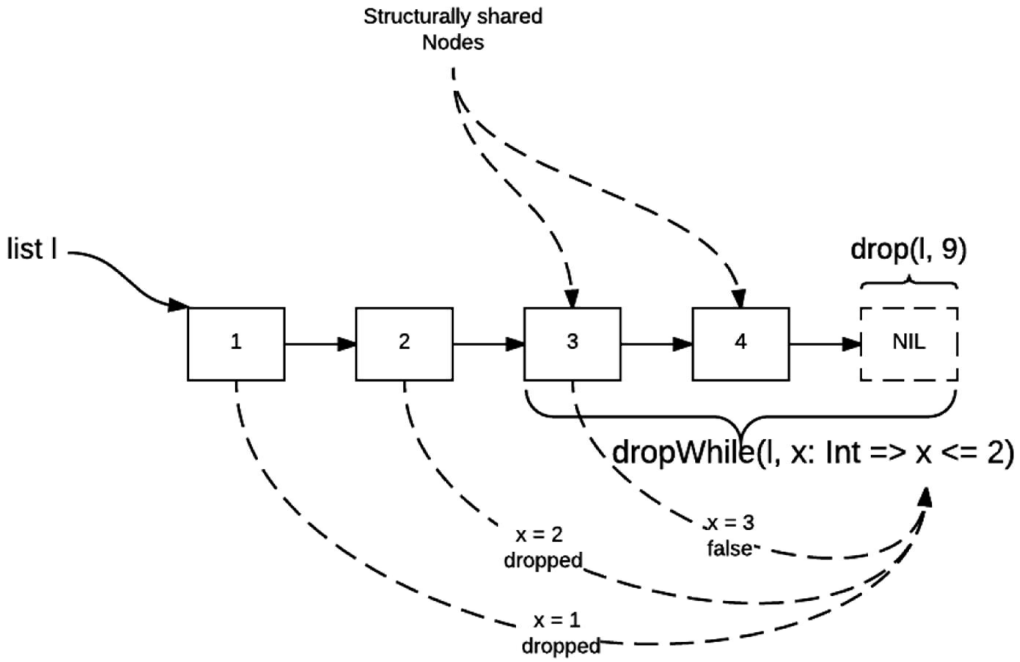

List(4)
::(1,::(2,::(3,::(4,Nil))))
List(3, 4)
List(2, 3, 4)
List(1, 2, 3, 4)

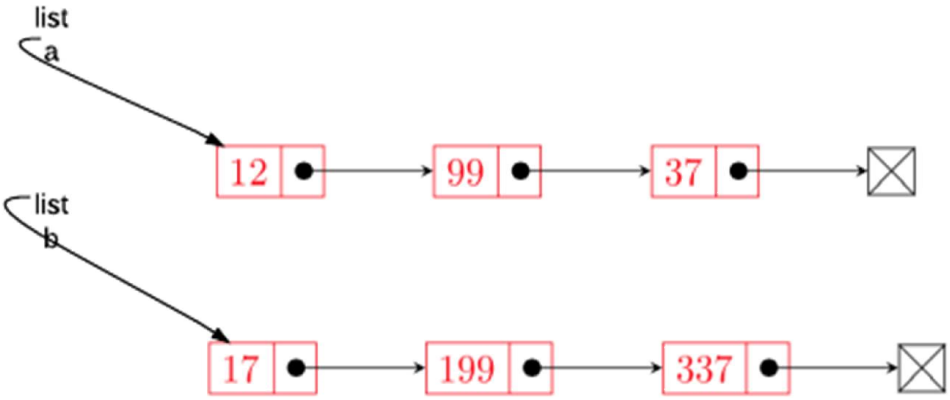

::(1,::(2,::(3,::(4,Nil))))
bound
match
bound
case ::(x, xs) => x
Replace with _ (underscore)
We don't care about the value

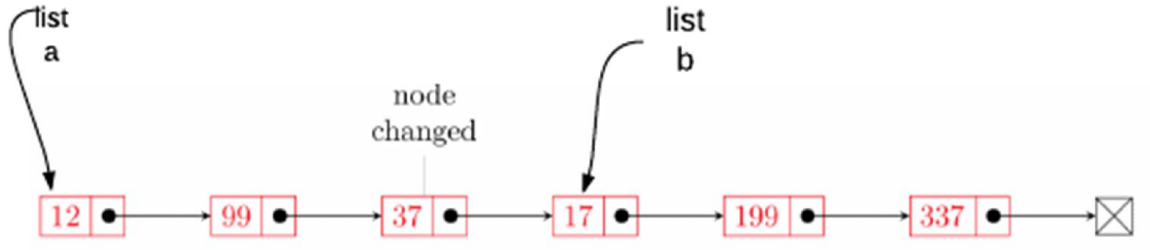

1
2
3
4
NIL
head(l)
tail(l)

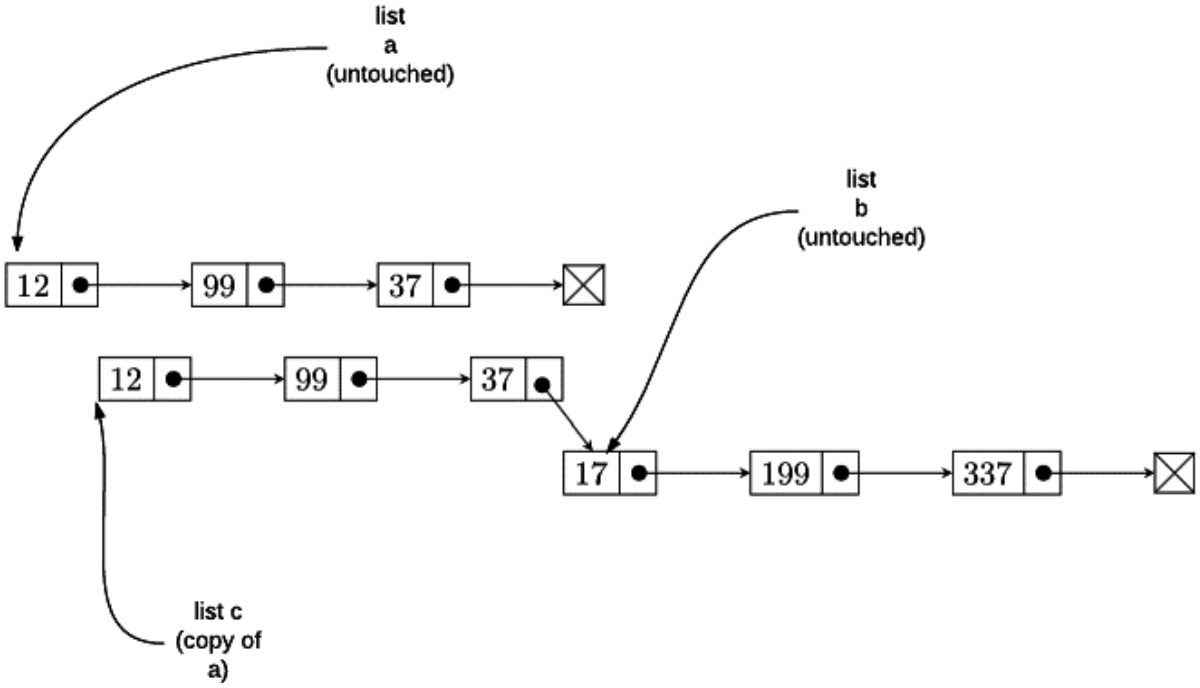

list l
Shrared
Nodes
1
2
3
4
NIL
Singleton
Always shared
val l2 = tail(tail(l))
val l1 = tail(l)

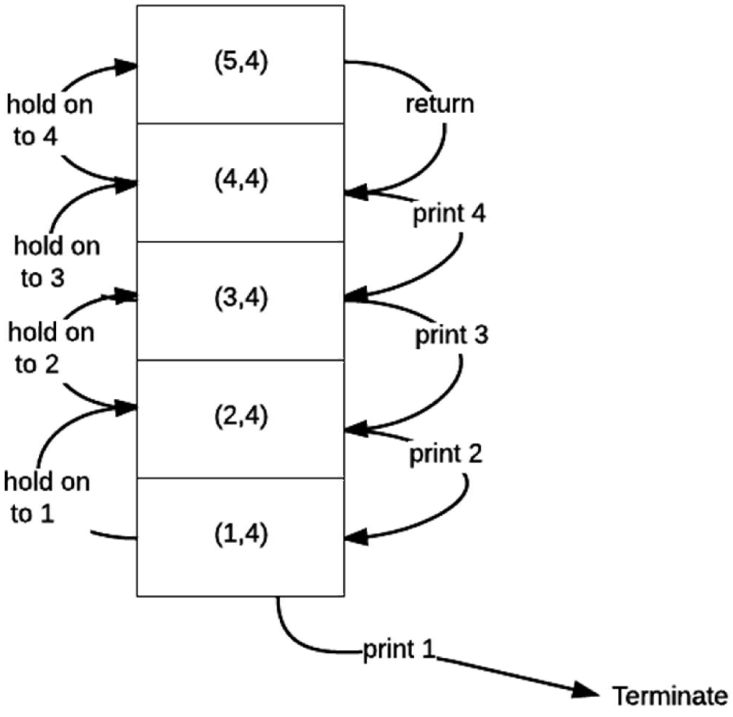

list l
drop(l, 9)
1
2
3
4
NIL
drop(l, 2)

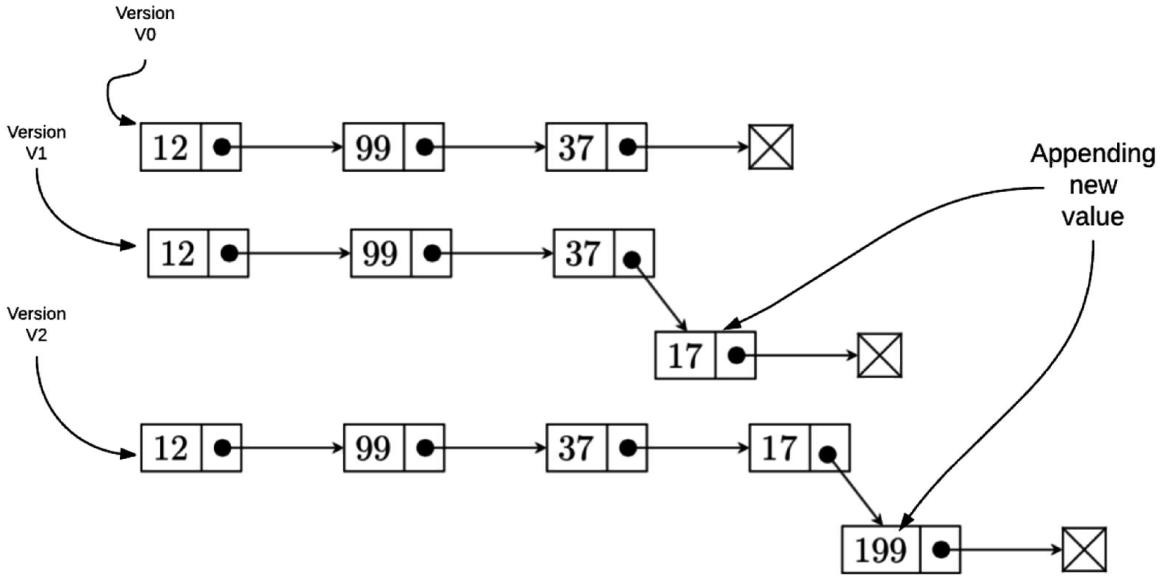

Structurally shared
Nodes
list l
drop(l, 9)
1
2
3
4
NIL
dropWhile(l, x: Int => x <= 2)
x = 2
dropped
x = 3
false
x = 1
dropped

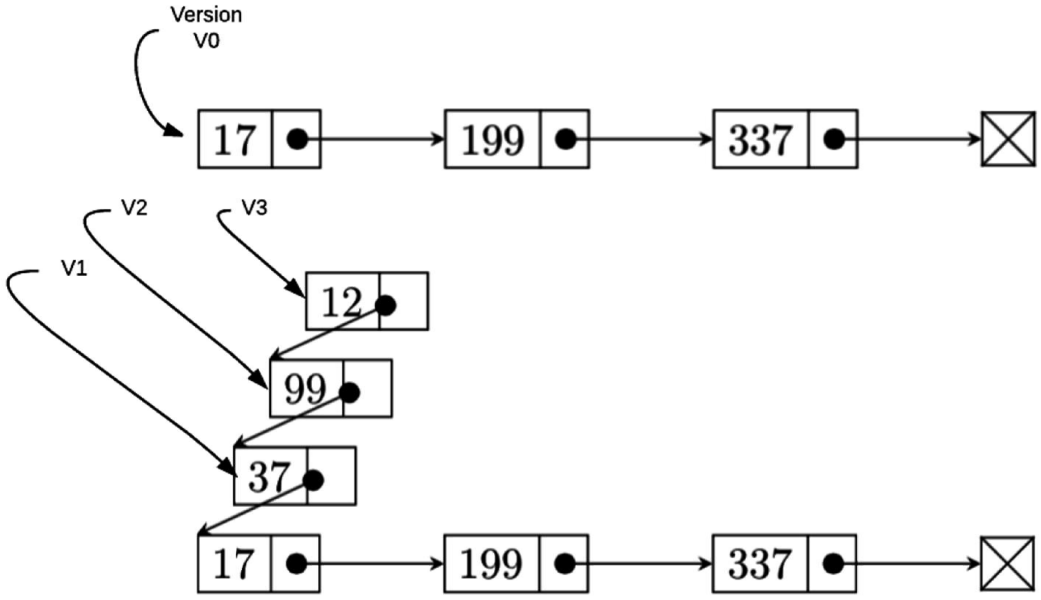

list
a
12
99
37
list
b
17
199
337

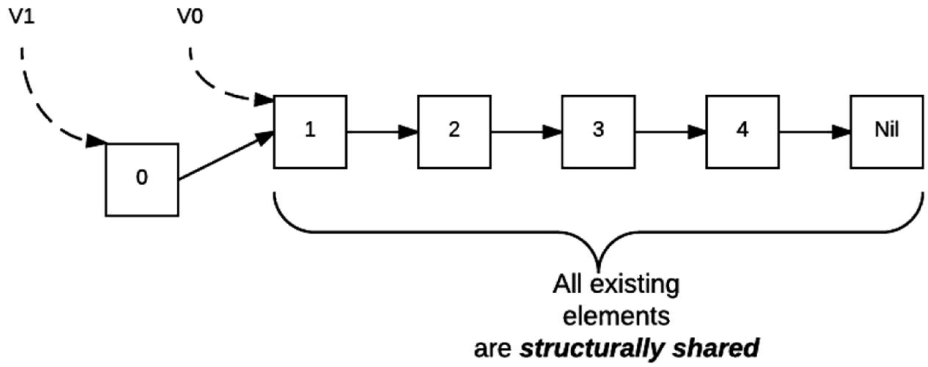

list
a
node
changed
list
b
12
99
37
17
199
337

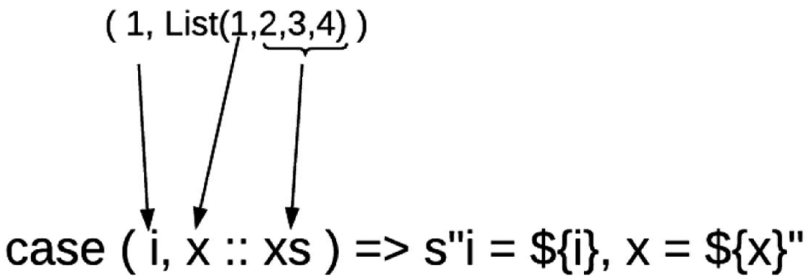

list
a
(untouched)
list
b
(untouched)
12
99
37
12
99
37
17
199
337
list c
(copy of
a)

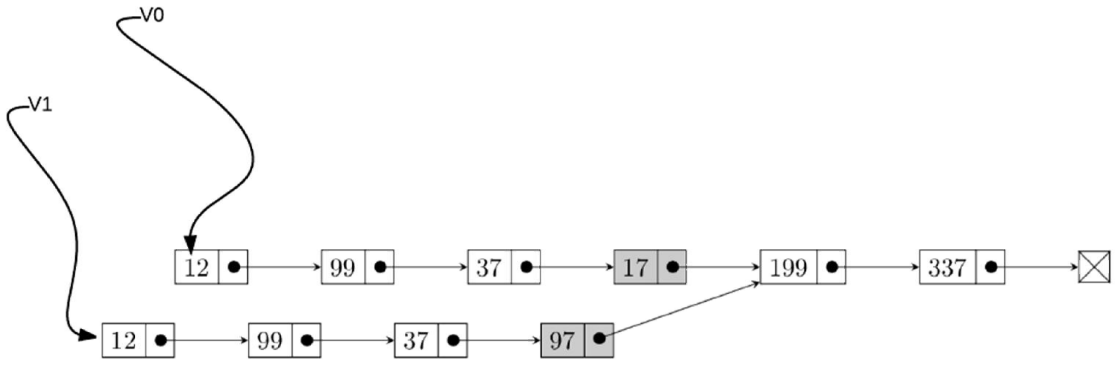

(5,4)
(4,4)
(3,4)
(2,4)
(1,4)
hold on
to 4
hold on
to 3
hold on
to 2
hold on
to 1
return
print 4
print 3
print 2
print 1
Terminate

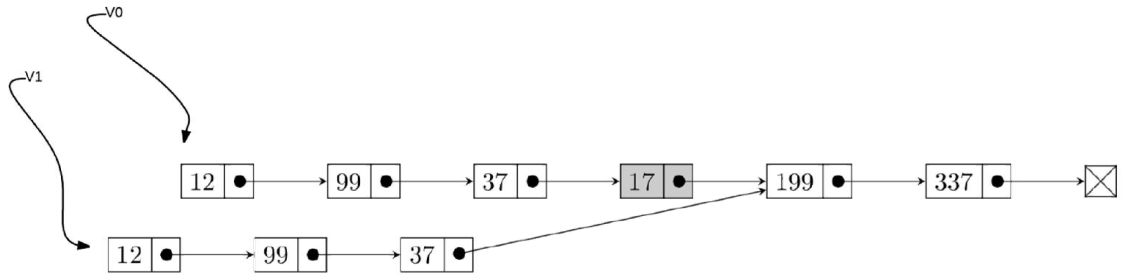

Version
V0
Version
V1
Version
V2
12
99
37
17
199
Appending
new
value


Version
V0
V1
V2
V3
17
199
337
12
99
37

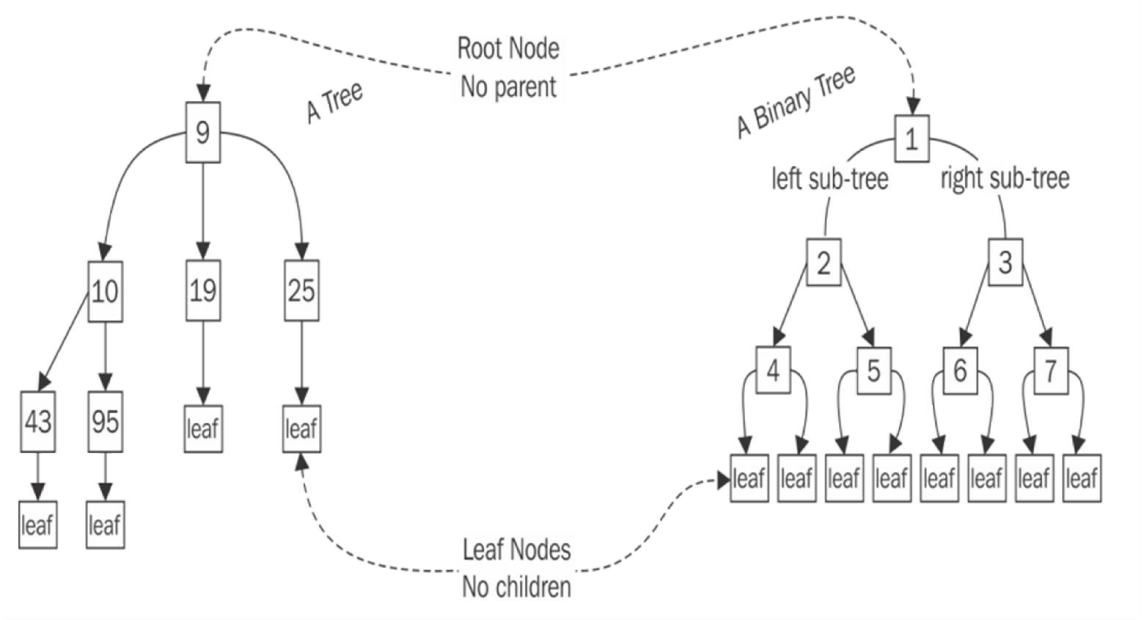

V1
V0
0
1
2
3
4
Nil
All existing
elements
are ***structurally shared***

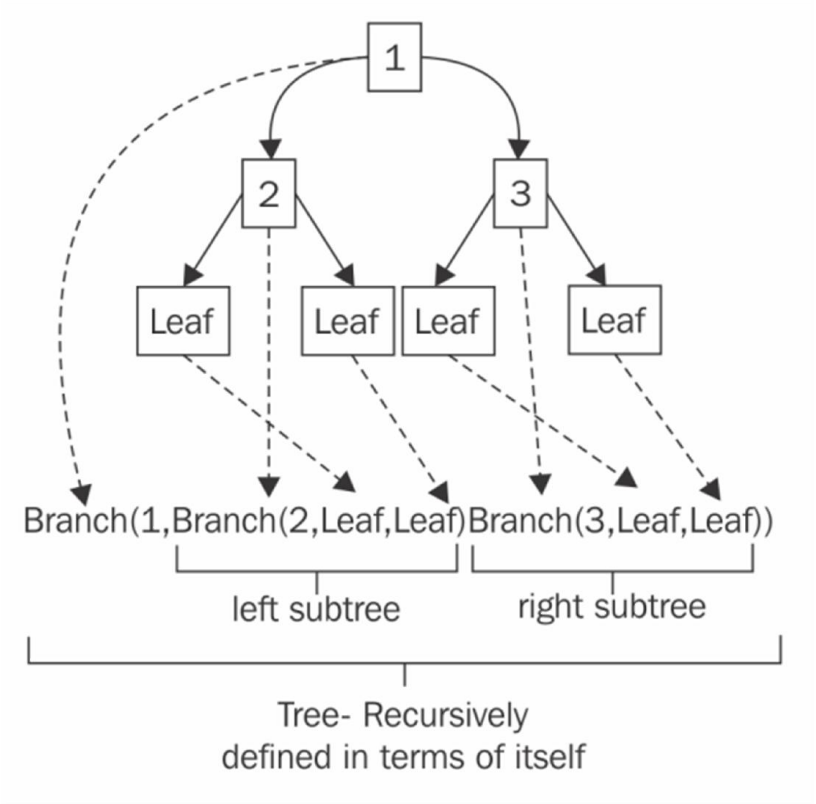

(1, List(1,2,3,4))
case (i, x :: xs) => s"i = ${i}, x = ${x}"

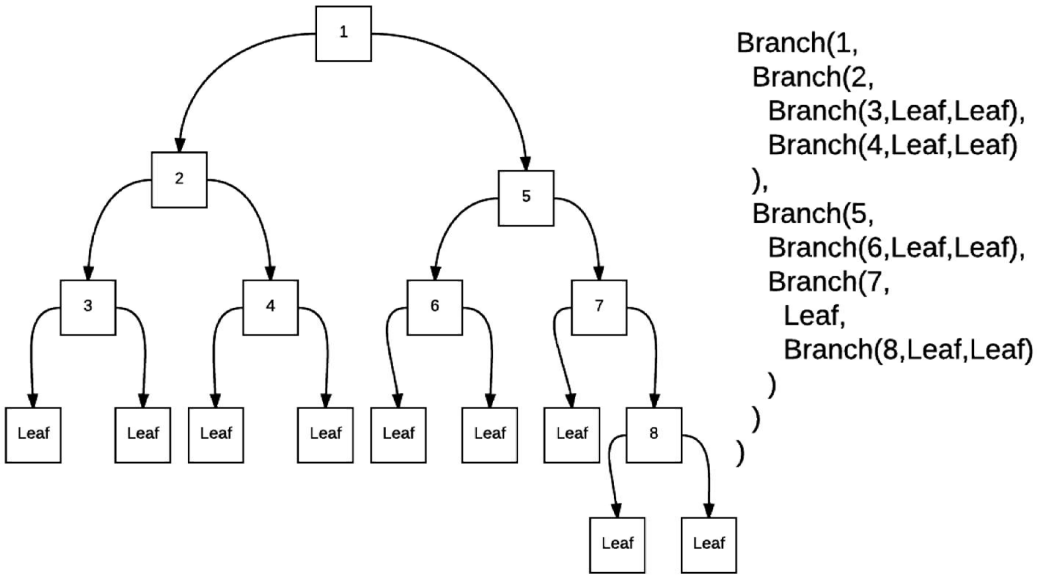

V0
V1
12
99
37
17
199
337
12
99
37
97


V0
V1
12
99
37
17
199
337
12
99
37

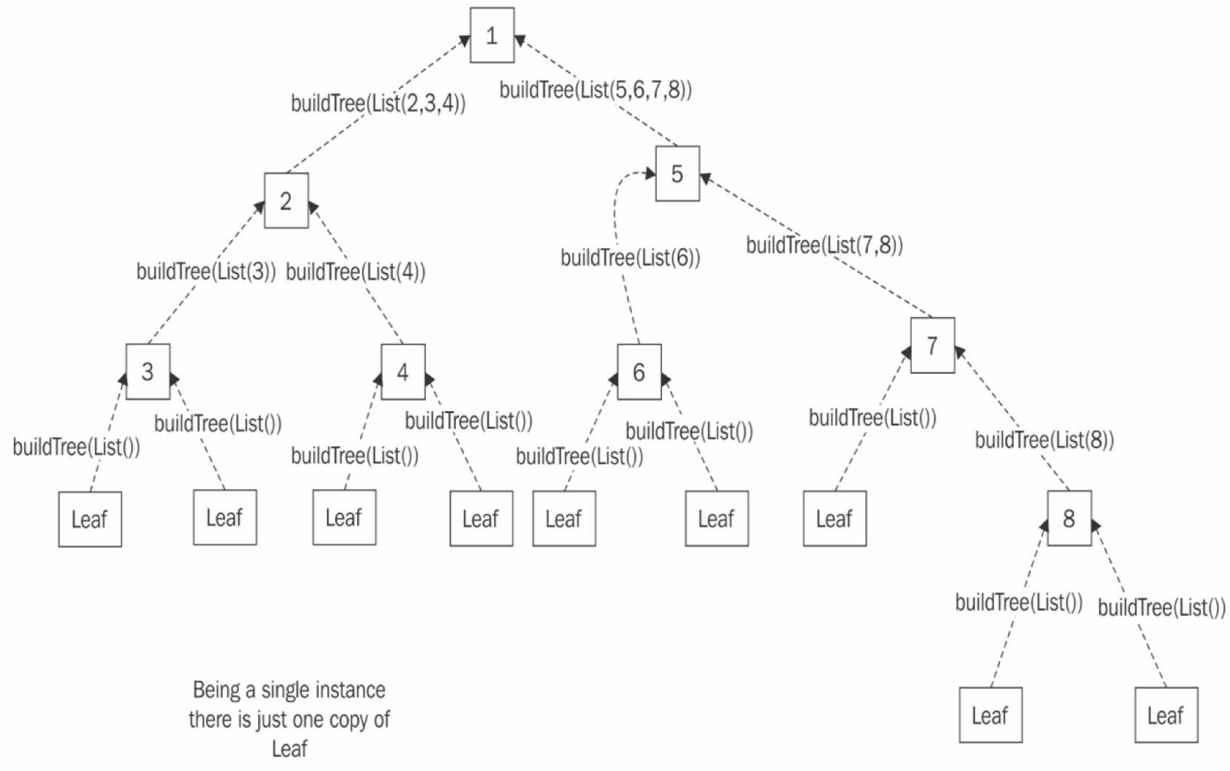
Chapter 4: Binary Trees

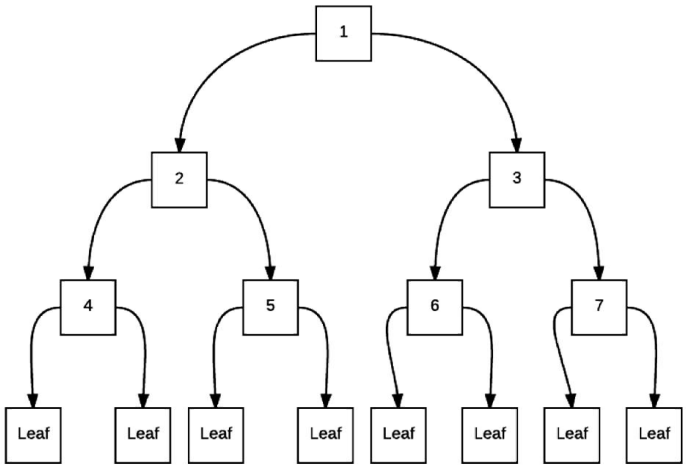




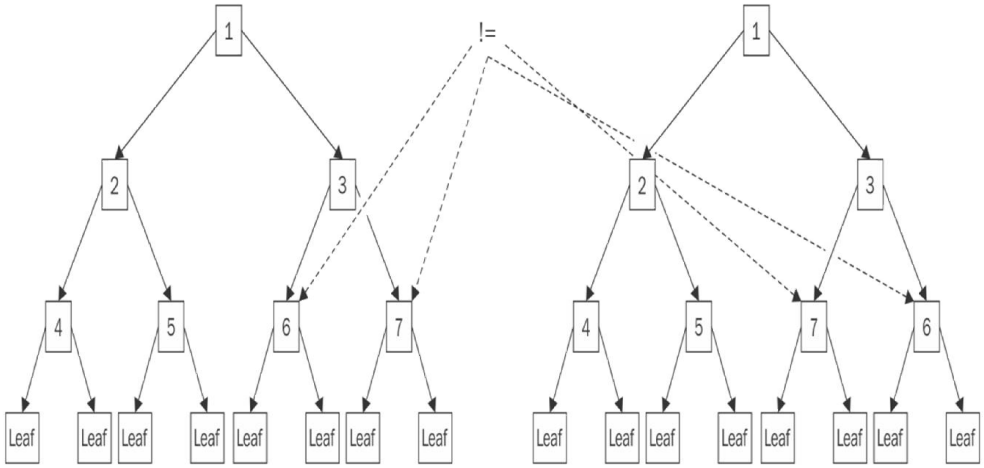


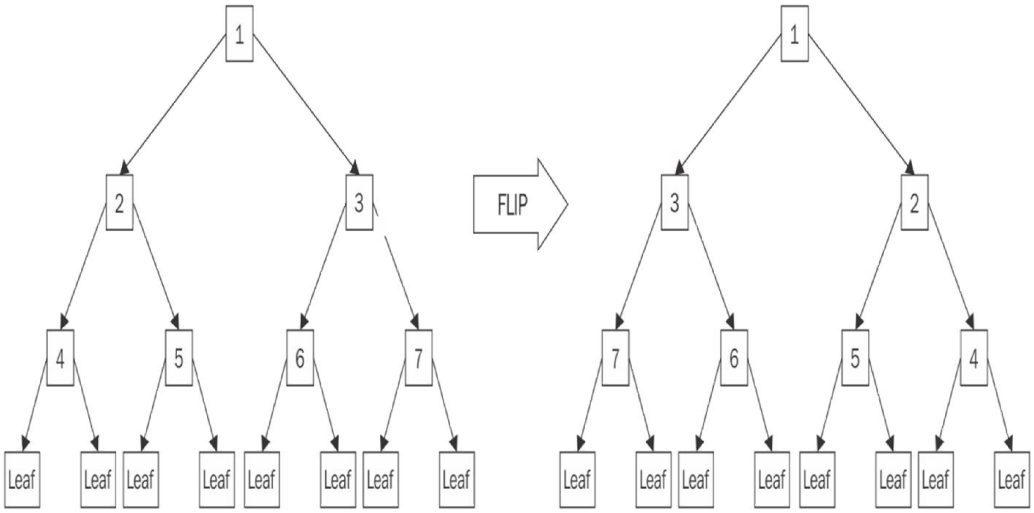

```
Branch(1,
  Branch(2,
    Branch(3,Leaf,Leaf),
    Branch(4,Leaf,Leaf)
  ),
  Branch(5,
    Branch(6,Leaf,Leaf),
    Branch(7,
      Leaf,
      Branch(8,Leaf,Leaf)
    )
  )
)
```

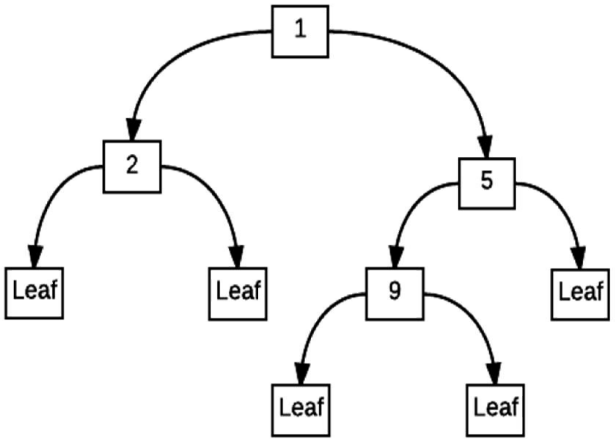





```
Branch(1,
  Branch(2,
    Branch(4,Leaf,Leaf),
    Branch(5,Leaf,Leaf)
  ),
  Branch(3,
    Branch(6,Leaf,Leaf),
    Branch(7,Leaf,Leaf)
  )
)
```







pre
order
Root, Left,Right

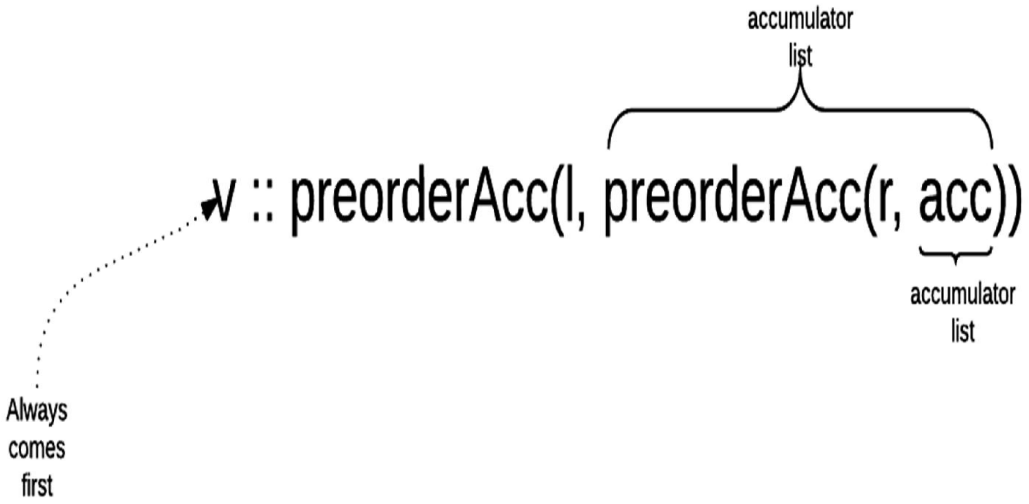
Visit action: print value
1,2,5,9

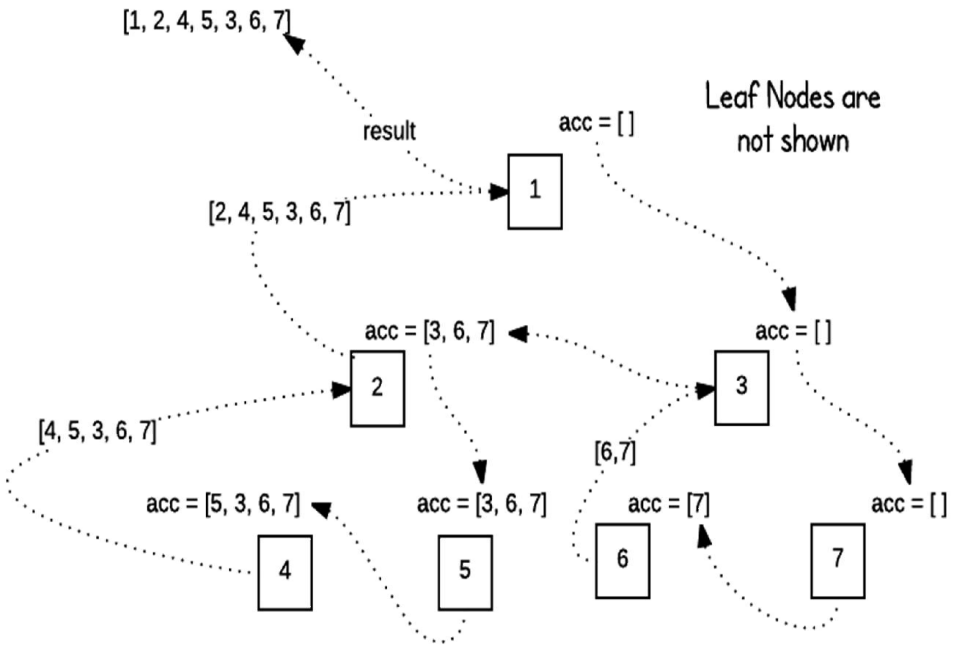
in
order
Left, Root, *Right*

Visit action: print value
2,1,9,5

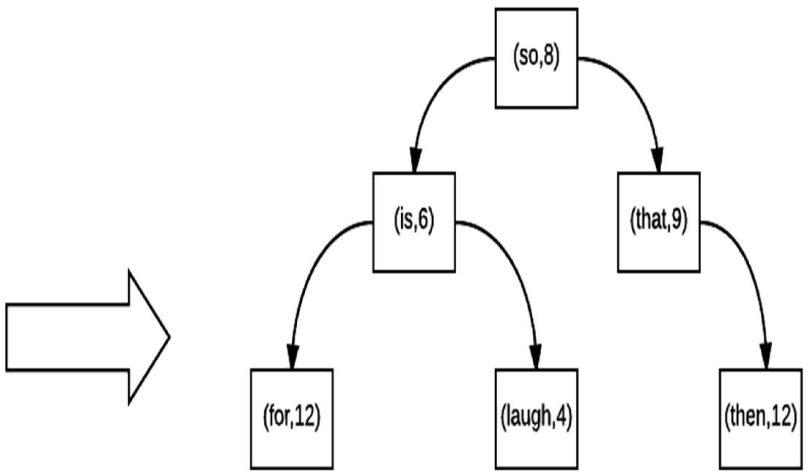
post
order
Left, Right, *Root*

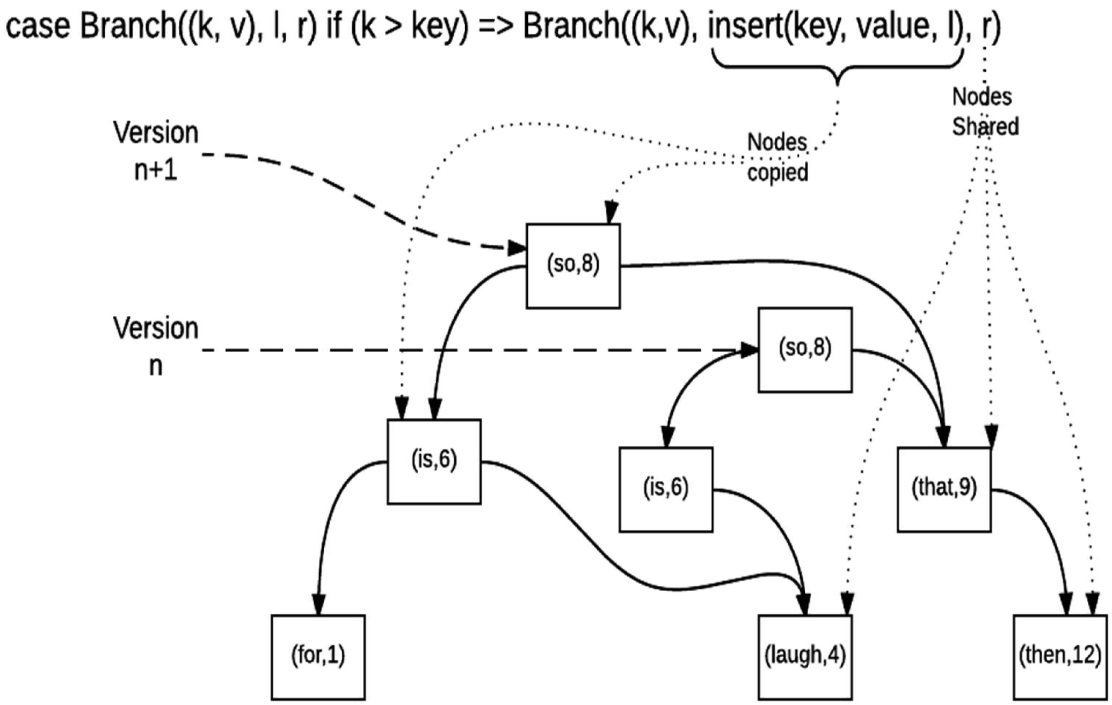
Visit action: print value
2,9,5,1



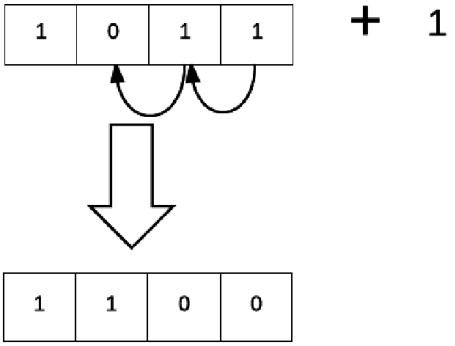


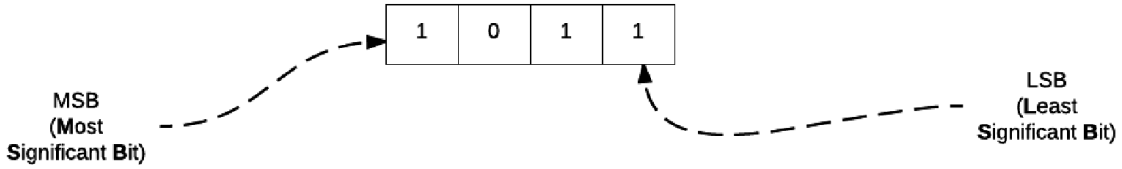
so	8
that	9
is	6
then	12
laugh	4
for	12

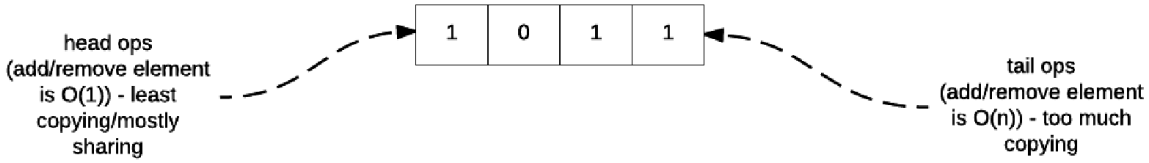


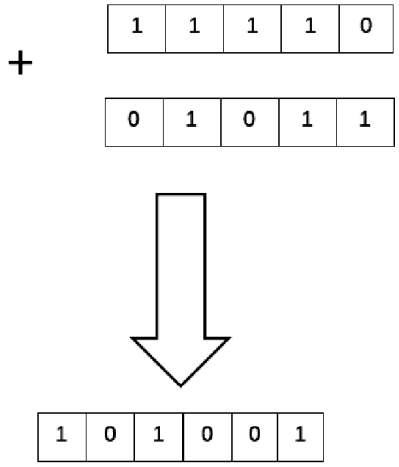

case Branch((k, v), l, r) if (k > key) => Branch((k,v), insert(key, value, l), r)
Nodes
Shared
Nodes
copied
Version
n+1
Version
n
(so,8)
(so,8)
(is,6)
(is,6)
(that,9)
(for,1)
(laugh,4)
(then,12)

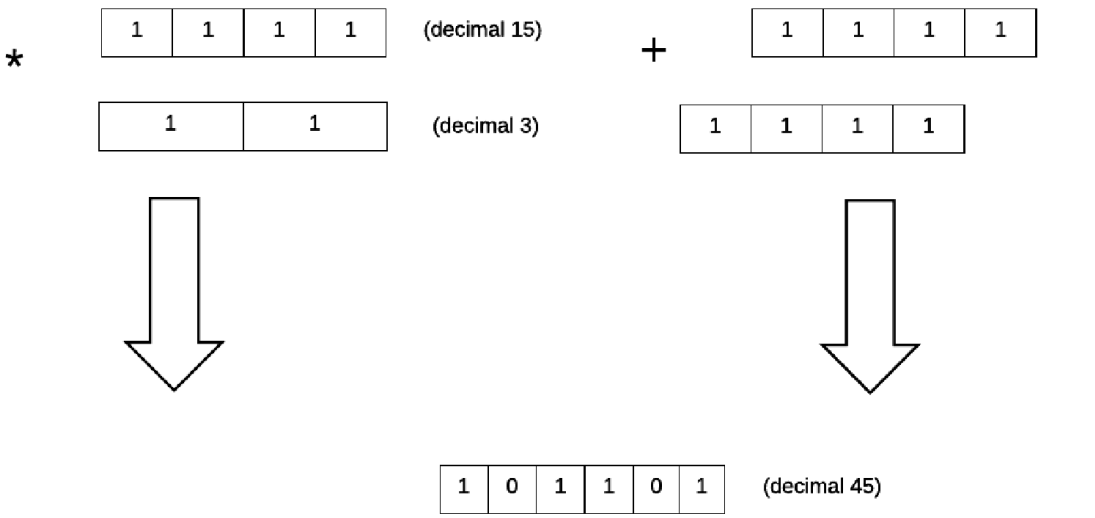
Chapter 5: More List Algorithms

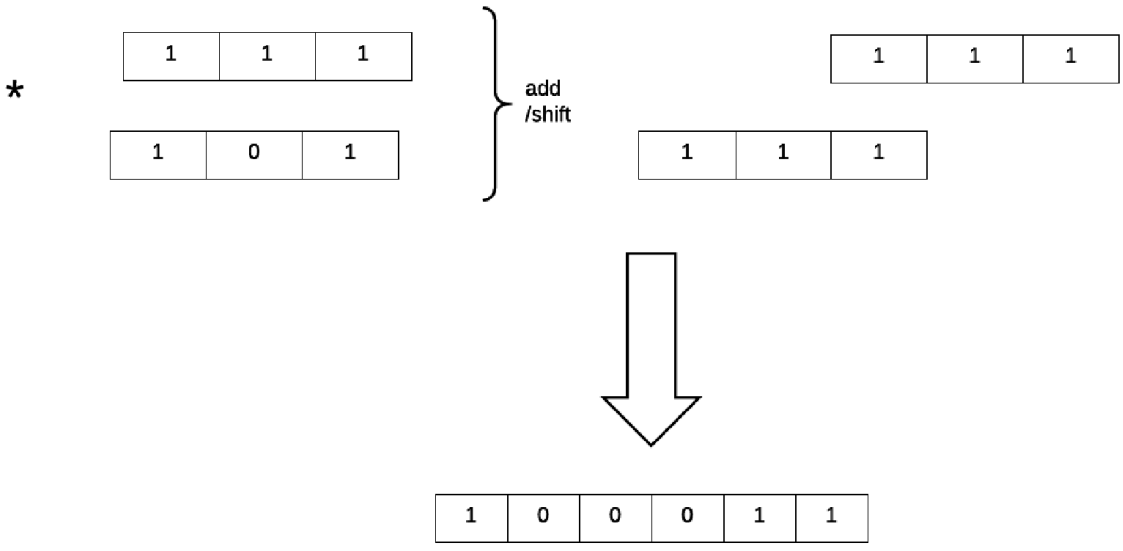


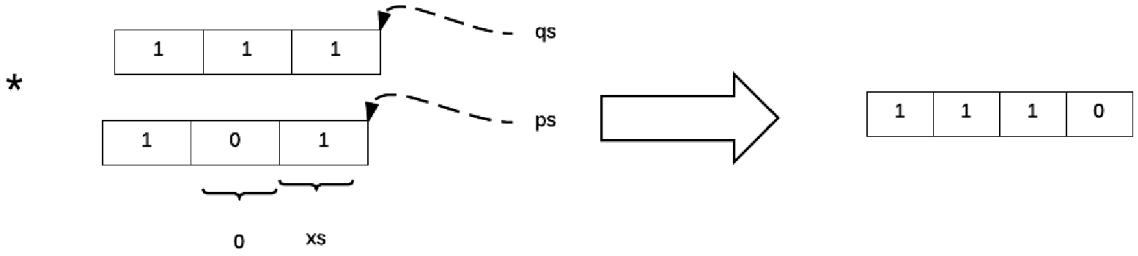


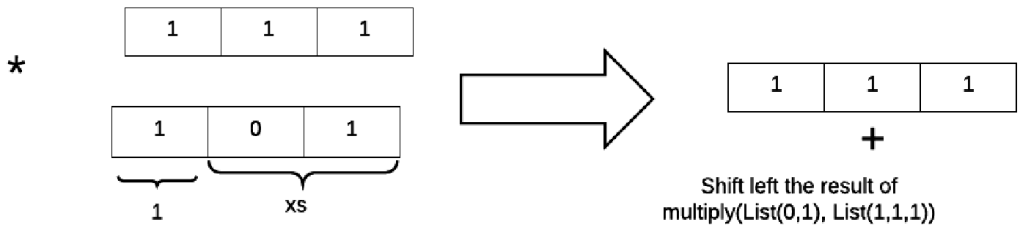


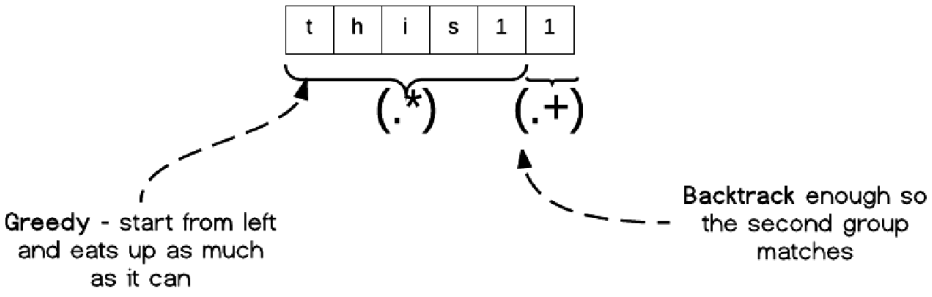


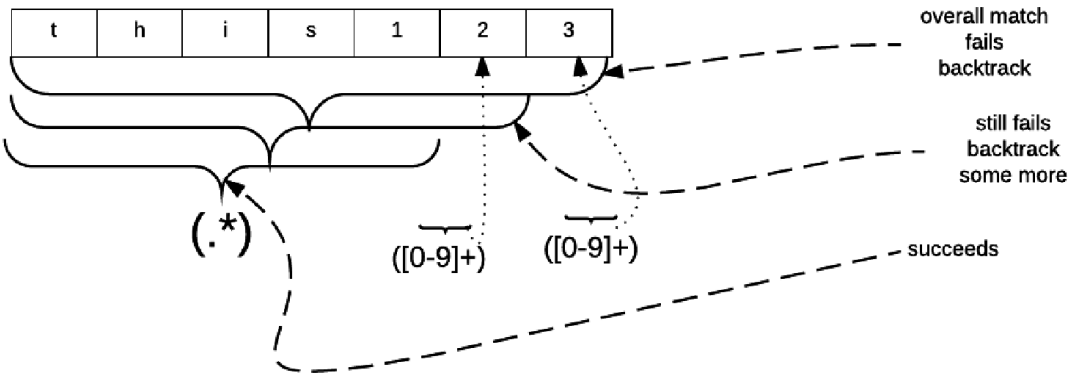

*
1 1 1 1
(decimal 15)
+
1 1 1 1
1 1
(decimal 3)
1 1 1 1
1 0 1 1 0 1
(decimal 45)

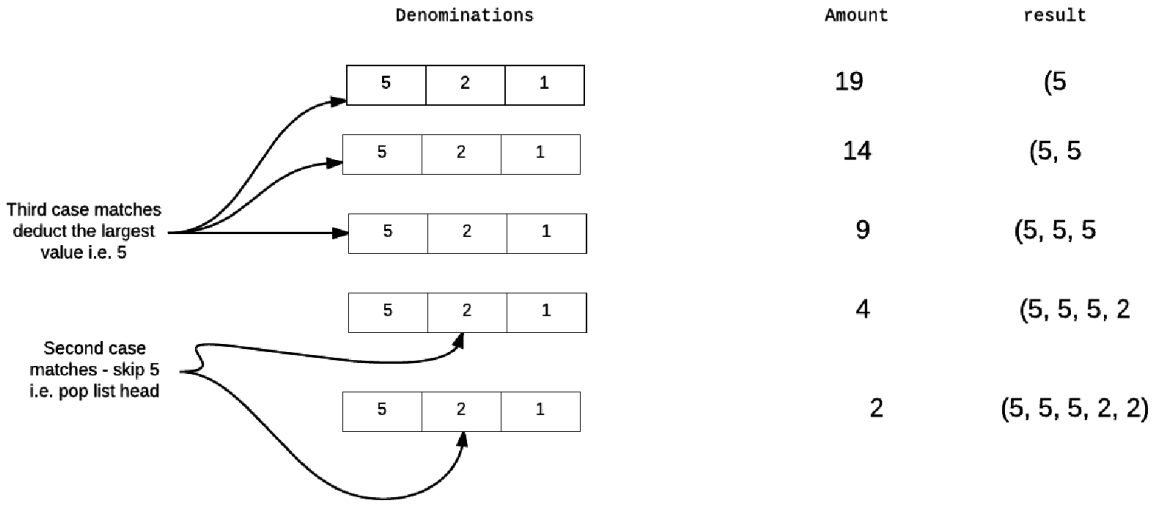

*
1 1 1
1 0 1
add
/shift
1 1 1
1 1 1
1 0 0 0 1 1

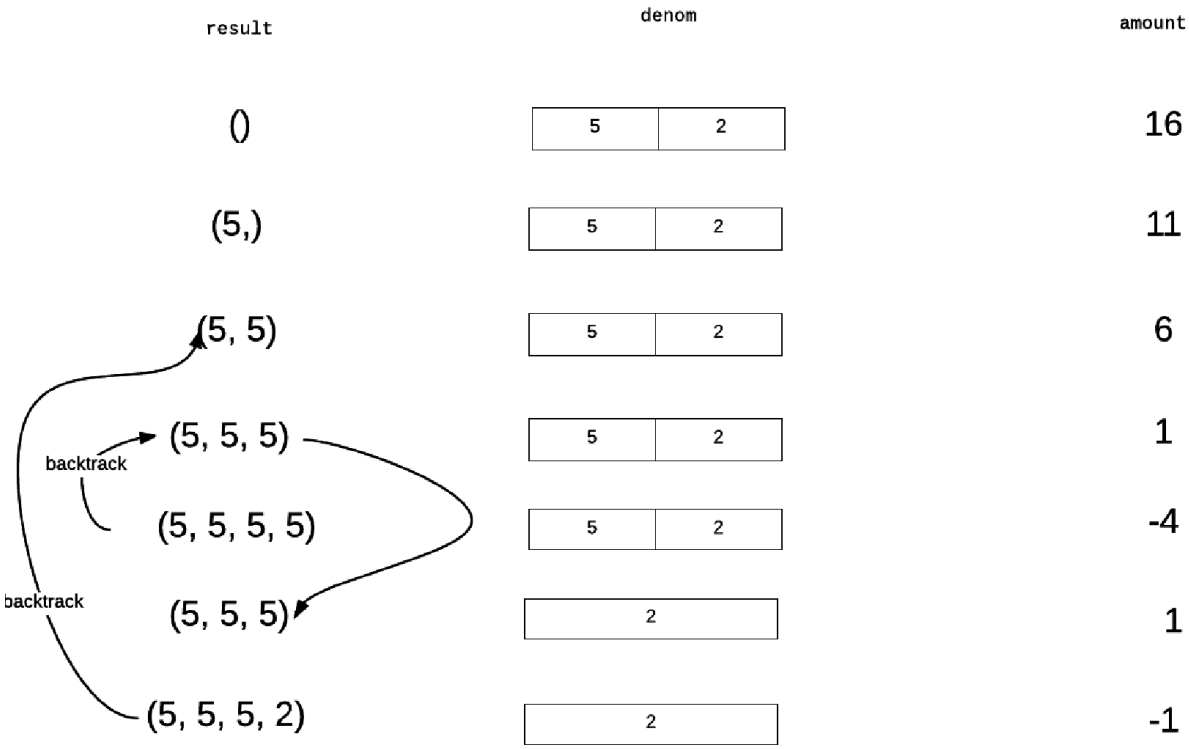

*
1 1 1
qs
1 0 1
ps
0
xs
1 1 1 0


*
1
1
1
1
0
1
1
xs
1
1
1
+
Shift left the result of
multiply(List(0,1), List(1,1,1))

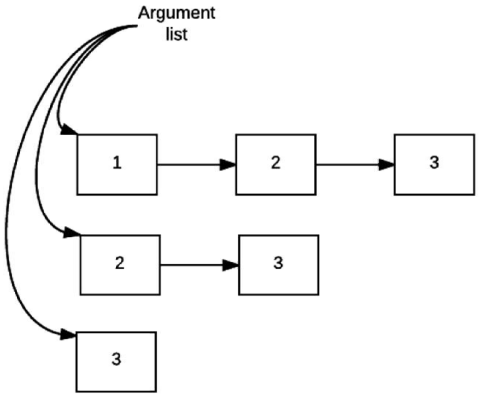

t
h
i
s
1
1
(.*)
(.+)
Greedy - start from left
and eats up as much
as it can
Backtrack enough so
the second group
matches

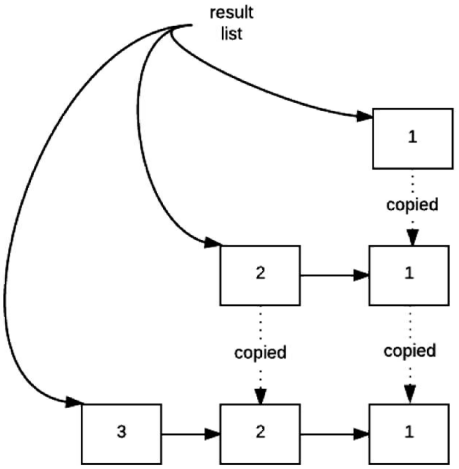

t
h
i
s
1
2
3
overall match
fails
backtrack
still fails
backtrack
some more
succeeds
(.*)
([0-9]+)
([0-9]+)

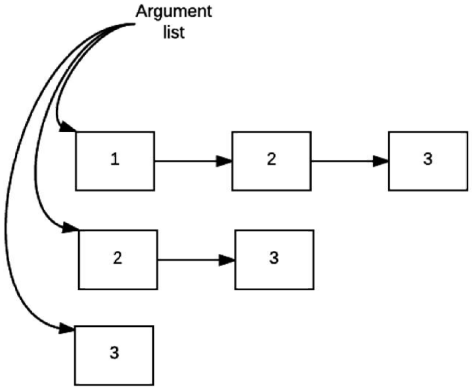

Denominations
Amount
result
5 2 1
19
(5
5 2 1
14
(5, 5
Third case matches deduct the largest value i.e. 5
5 2 1
9
(5, 5, 5
5 2 1
4
(5, 5, 5, 2
Second case matches - skip 5 i.e. pop list head
5 2 1
2
(5, 5, 5, 2, 2)

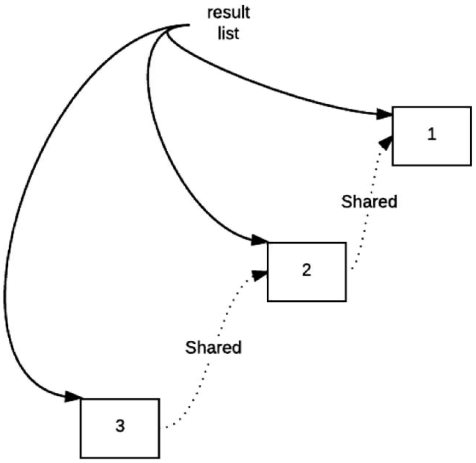

result
denom
amount
()
5 2
16
(5,)
5 2
11
(5, 5)
5 2
6
(5, 5, 5)
5 2
1
backtrack
(5, 5, 5, 5)
5 2
-4
backtrack
(5, 5, 5)
2
1
(5, 5, 5, 2)
2
-1

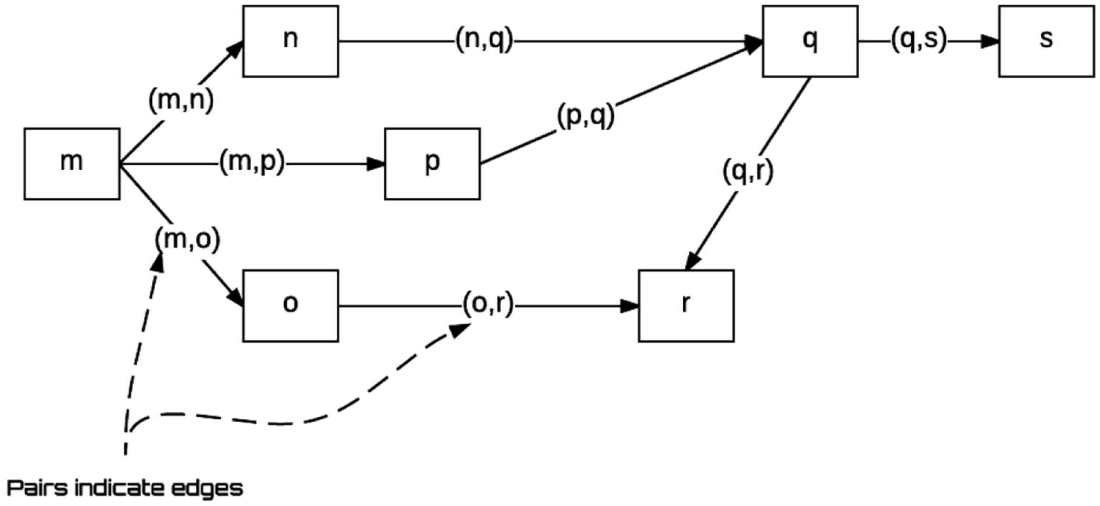
Chapter 6: Graph Algorithms

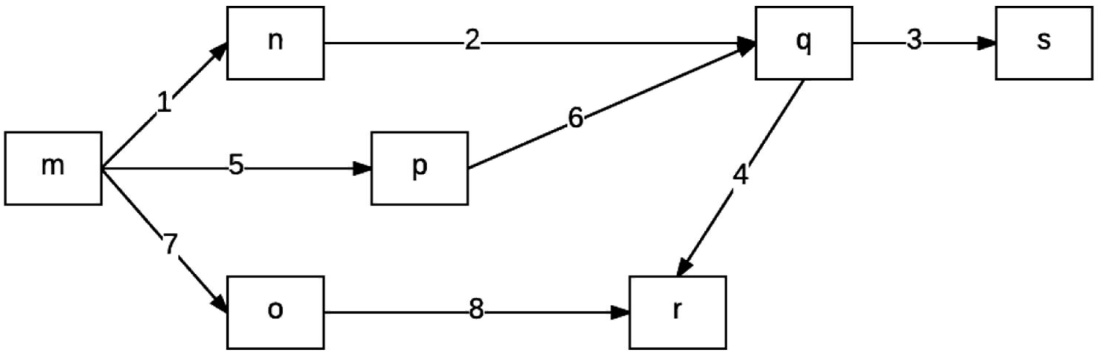


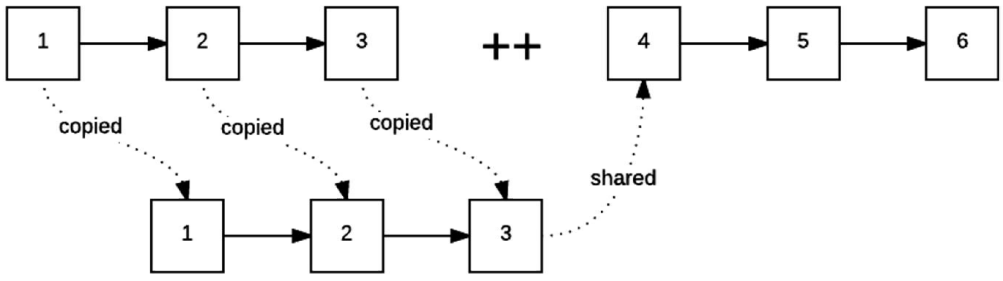


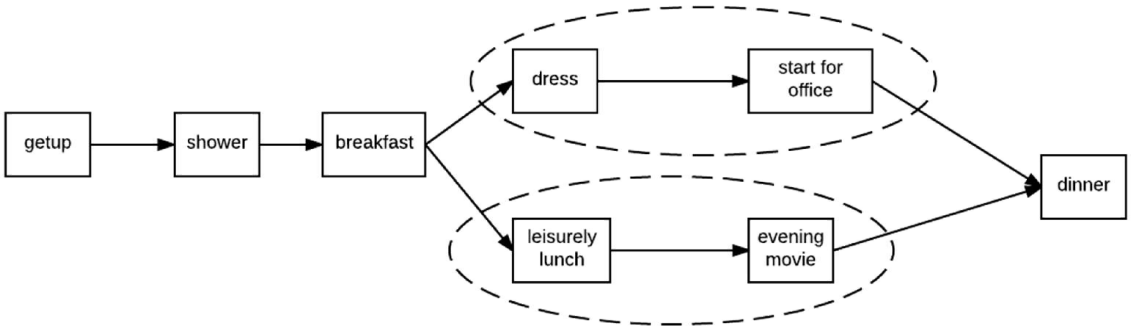


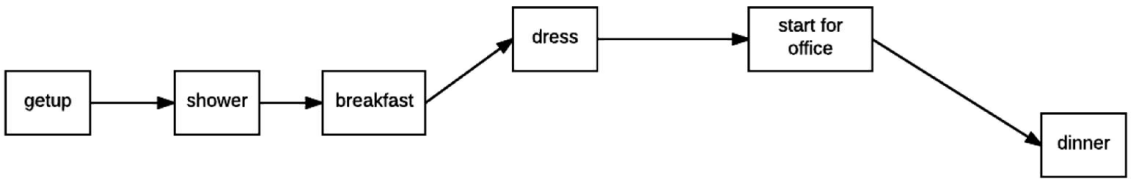


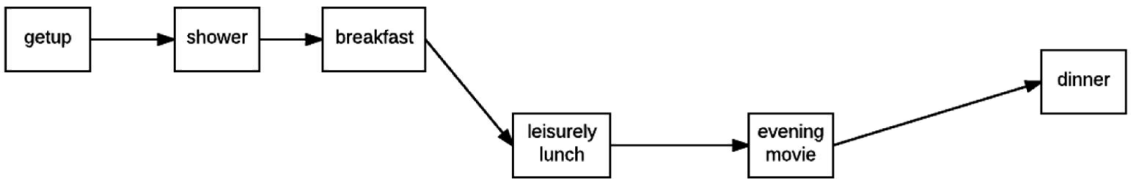

n
(n,q)
q
(q,s)
s
(m,n)
m
(m,p)
p
(p,q)
(q,r)
(m,o)
o
(o,r)
r
Pairs indicate edges

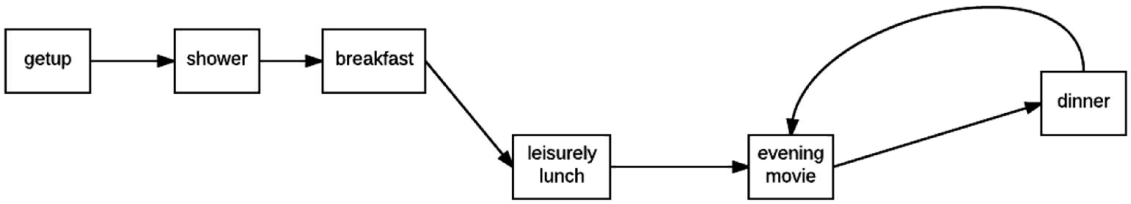

n
2
q
3
s
1
m
6
5
p
4
7
o
8
r


1
2
3
++
4
5
6
copied
copied
copied
shared
1
2
3

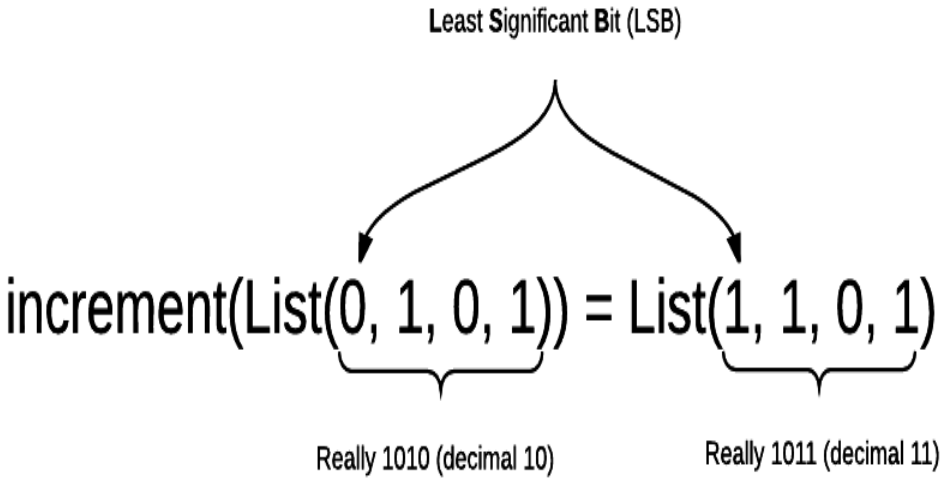

getup
shower
breakfast
dress
start for office
leisurely lunch
evening movie
dinner

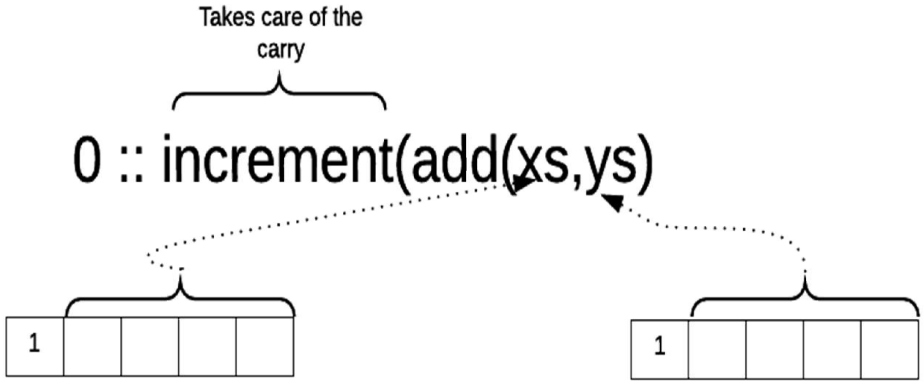

getup
shower
breakfast
dress
start for office
dinner

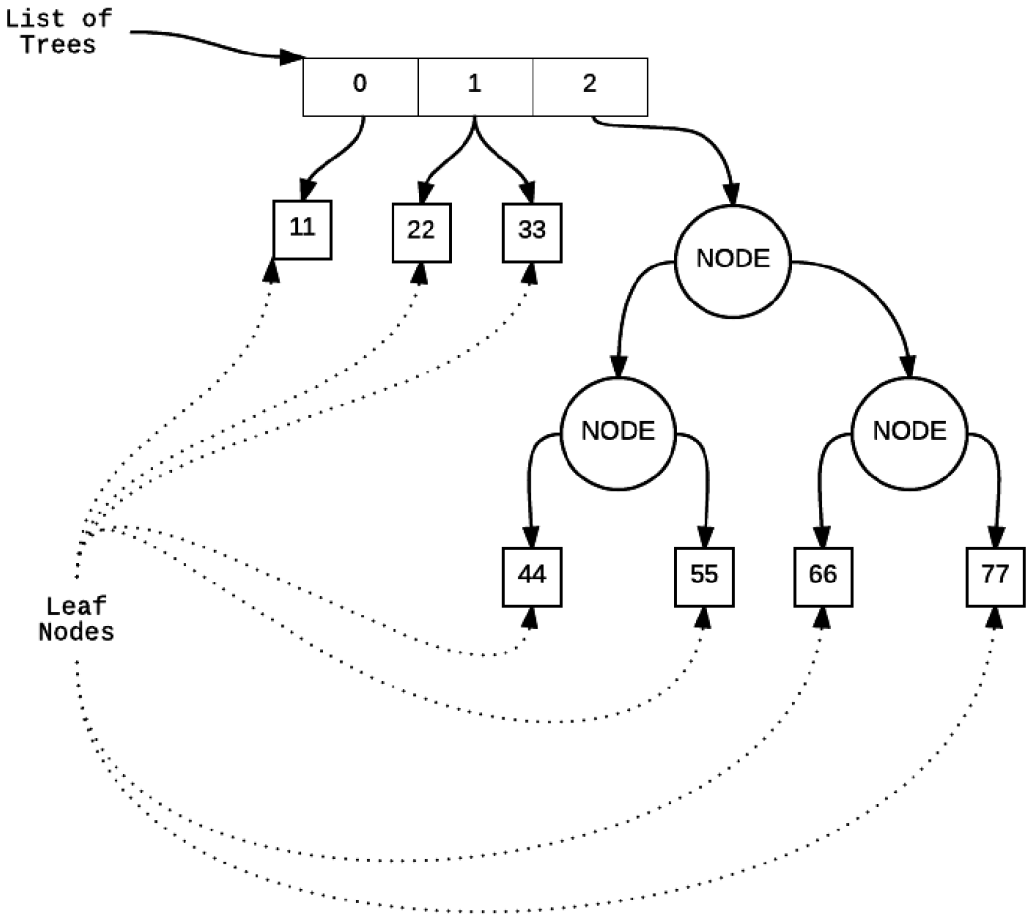

getup
shower
breakfast
leisurely lunch
evening movie
dinner

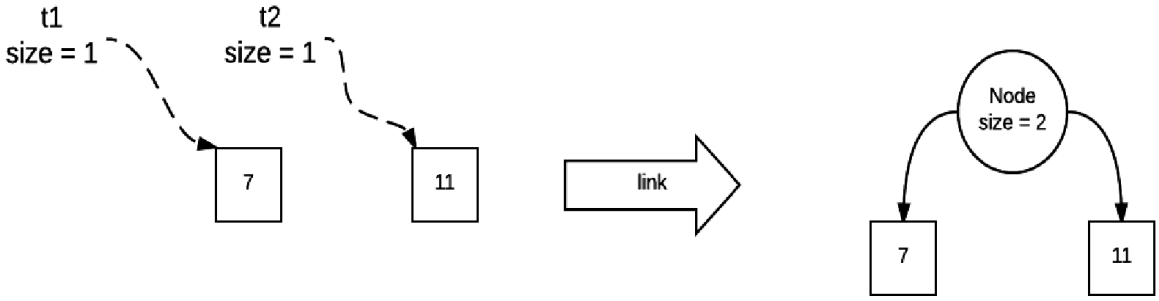

getup
shower
breakfast
leisurely lunch
evening movie
dinner

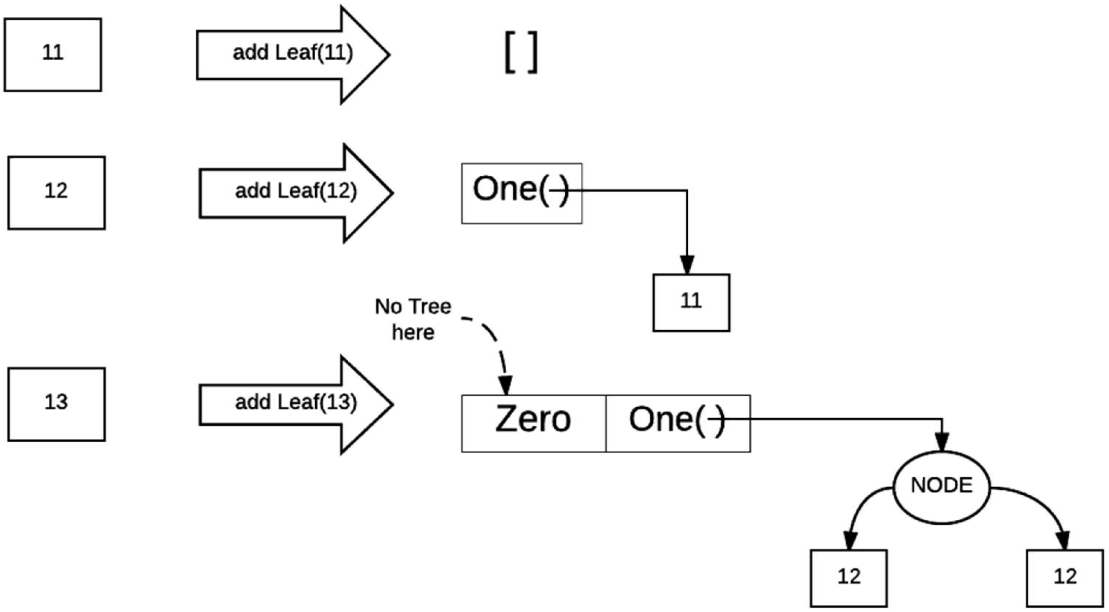
Chapter 7: Random Access Lists

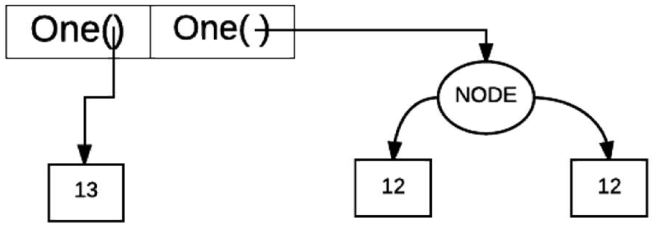


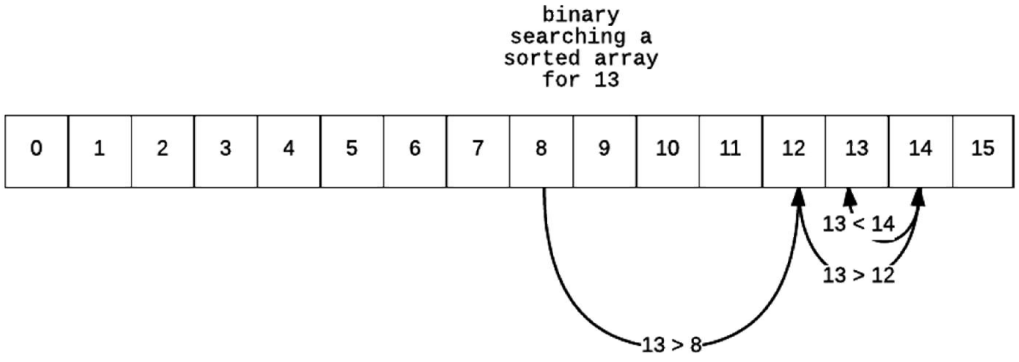


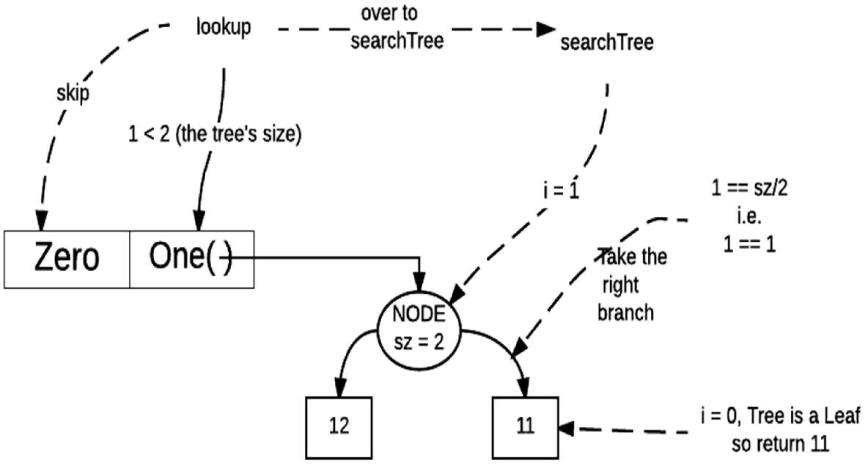

List of
Trees
0
1
2
11
22
33
NODE
NODE
NODE
44
55
66
77
Leaf
Nodes


t1
size = 1
t2
size = 1
7
11
link
Node
size = 2
7
11

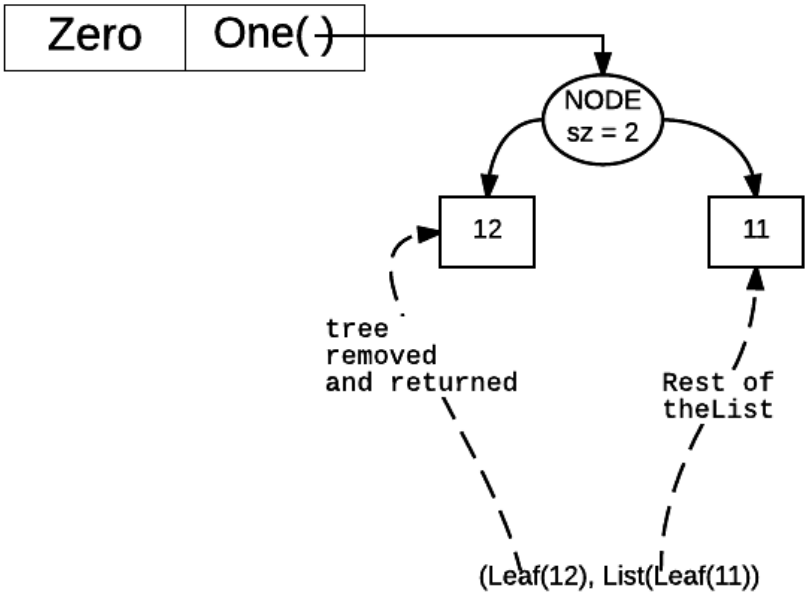

11
add Leaf(11)
[ ]
12
add Leaf(12)
One( )
11
No Tree
here
13
add Leaf(13)
Zero
One( )
NODE
12
12


One( )
One( )
NODE
13
12
12

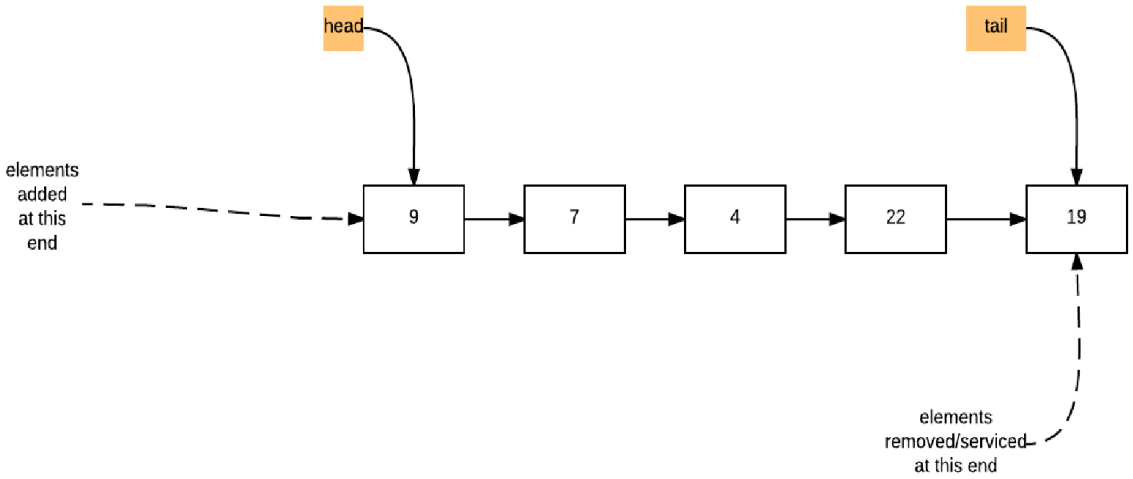

binary
searching a
sorted array
for 13
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
13 < 14
13 > 12
13 > 8

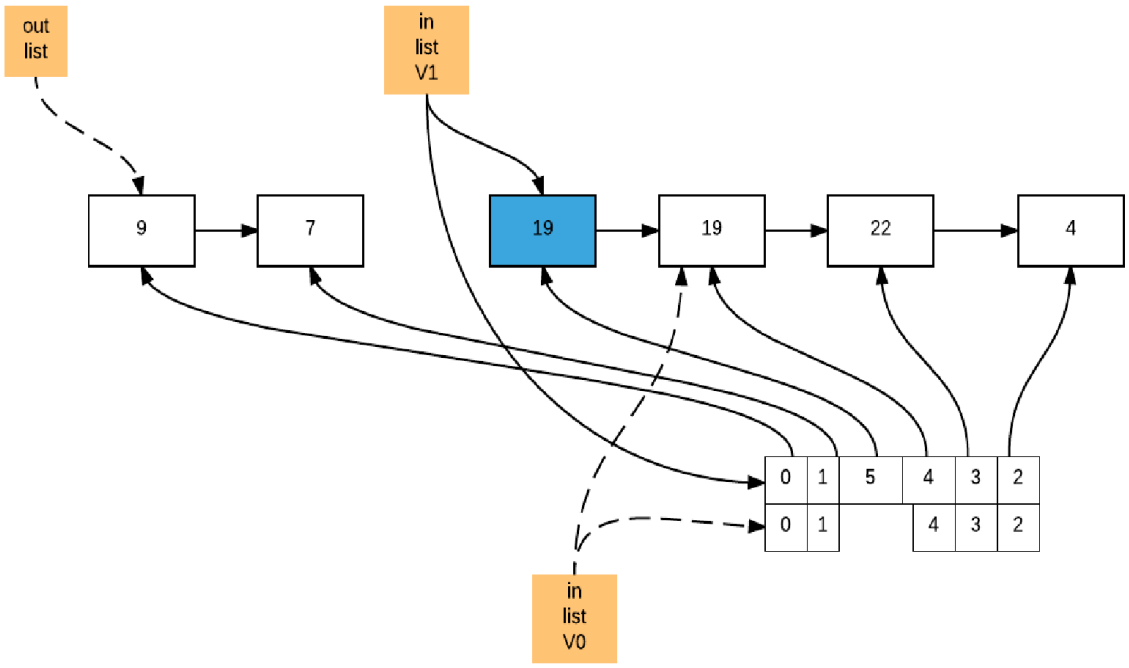


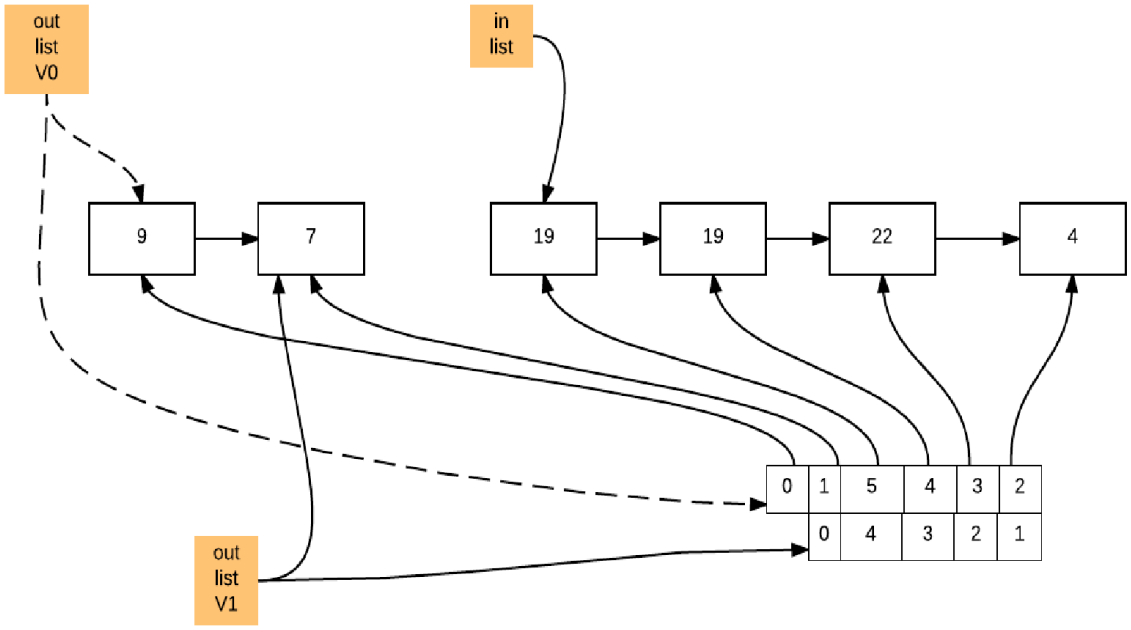
Searching
for
index 1

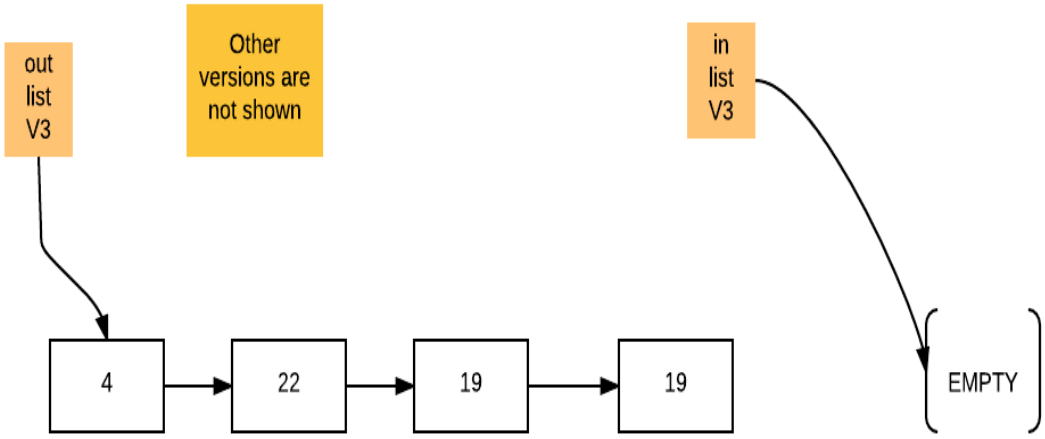


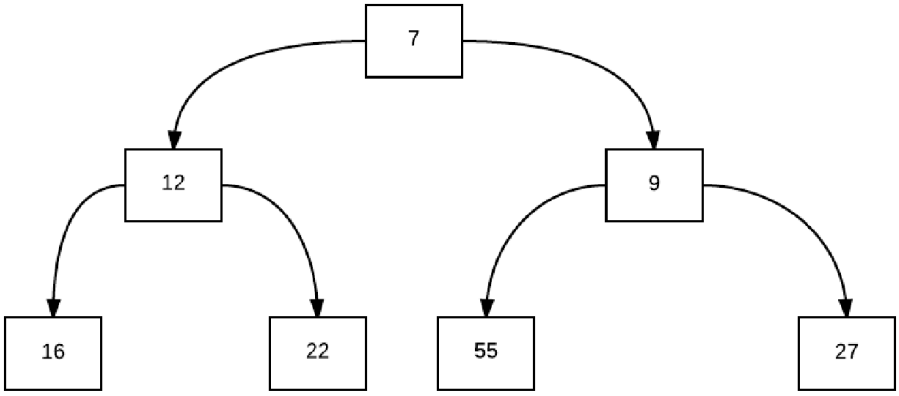
Chapter 8: Queues

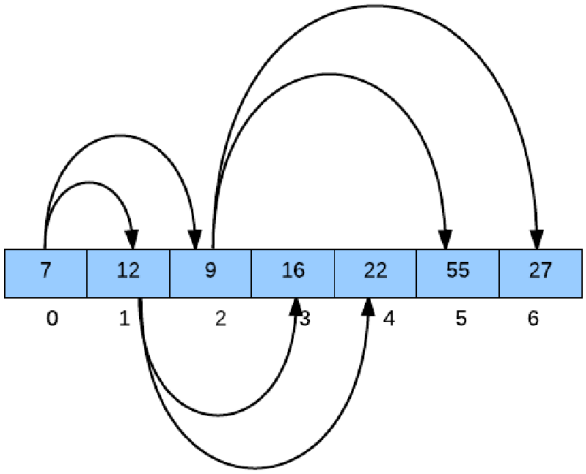


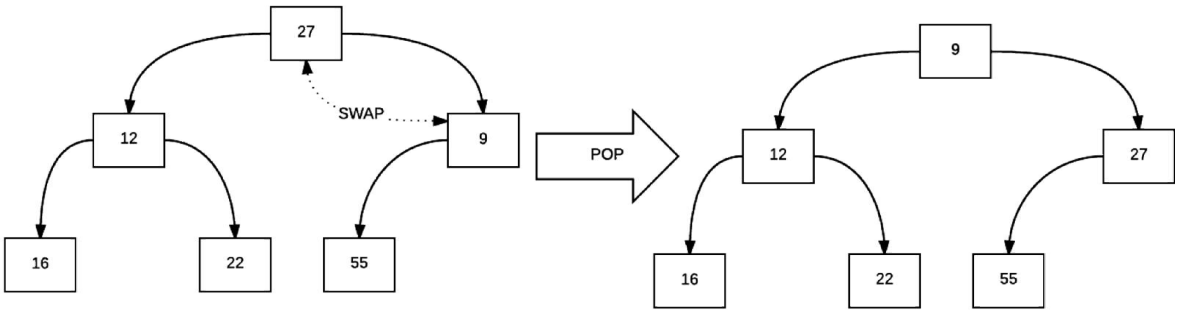


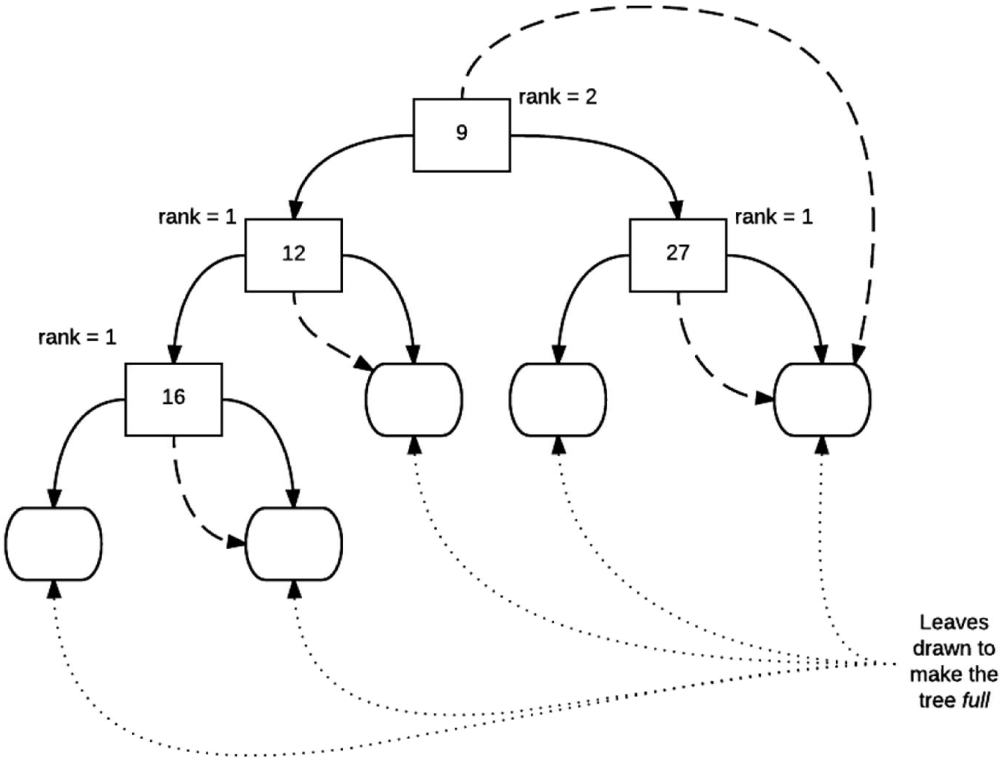

out
list
V0
in
list
9
7
19
19
22
4
0
1
5
4
3
2
0
4
3
2
1
out
list
V1

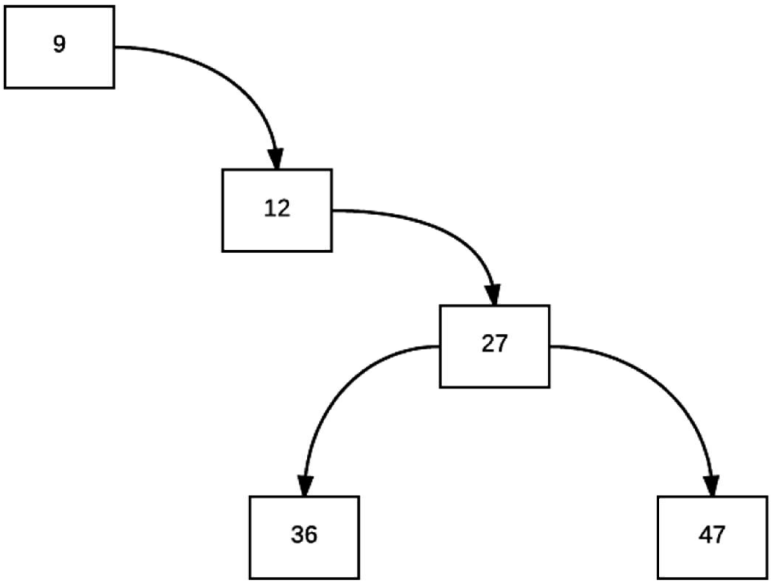

out
list
V3
Other
versions are
not shown
in
list
V3
4
22
19
19
EMPTY

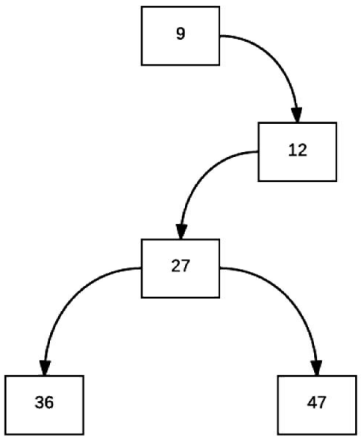

7
12
9
16
22
55
27

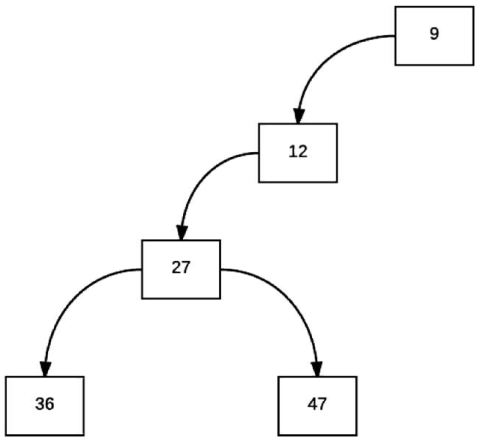

7
12
9
16
22
55
27
0
1
2
3
4
5
6

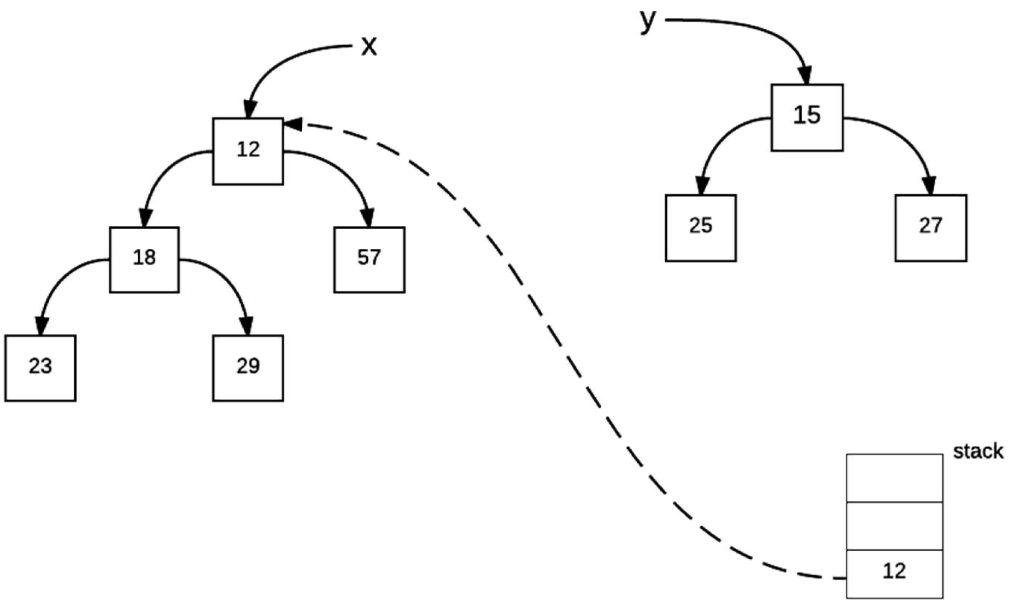

27
12
SWAP
9
16
22
55
POP
9
12
27
16
22
55

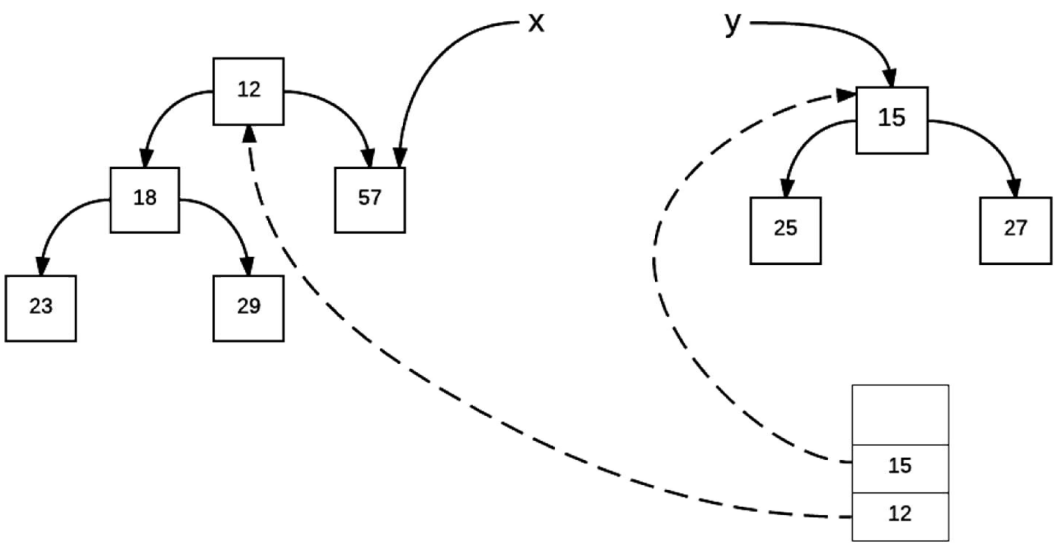

rank = 2
9
rank = 1
12
rank = 1
27
rank = 1
16
Leaves
drawn to
make the
tree *full*

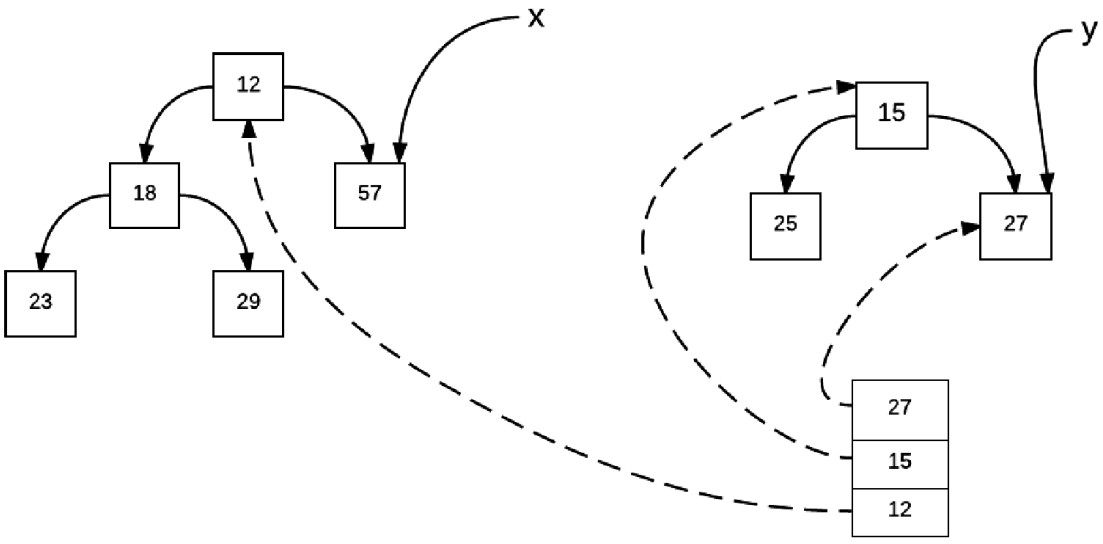

9
12
27
36
47

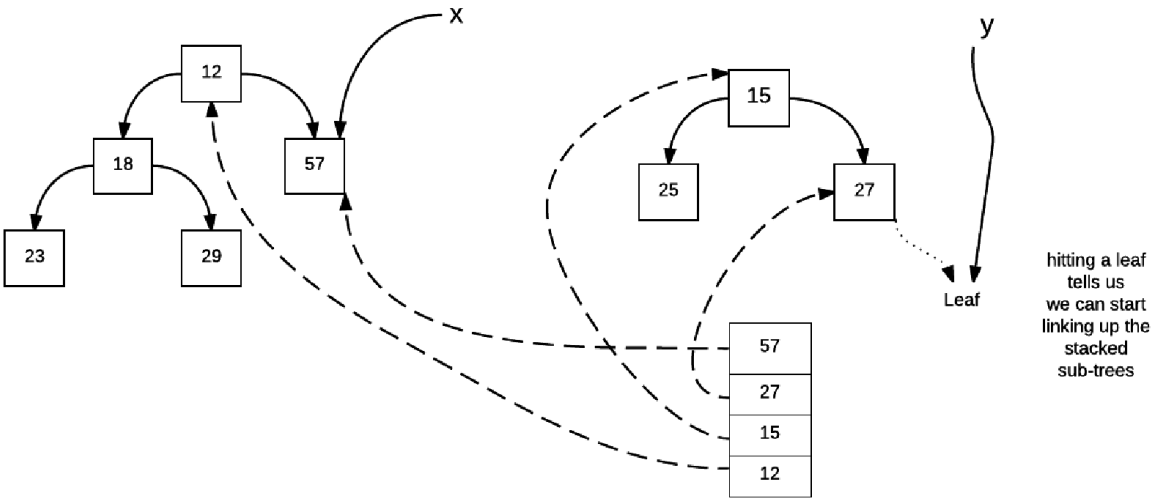

9
12
27
36
47

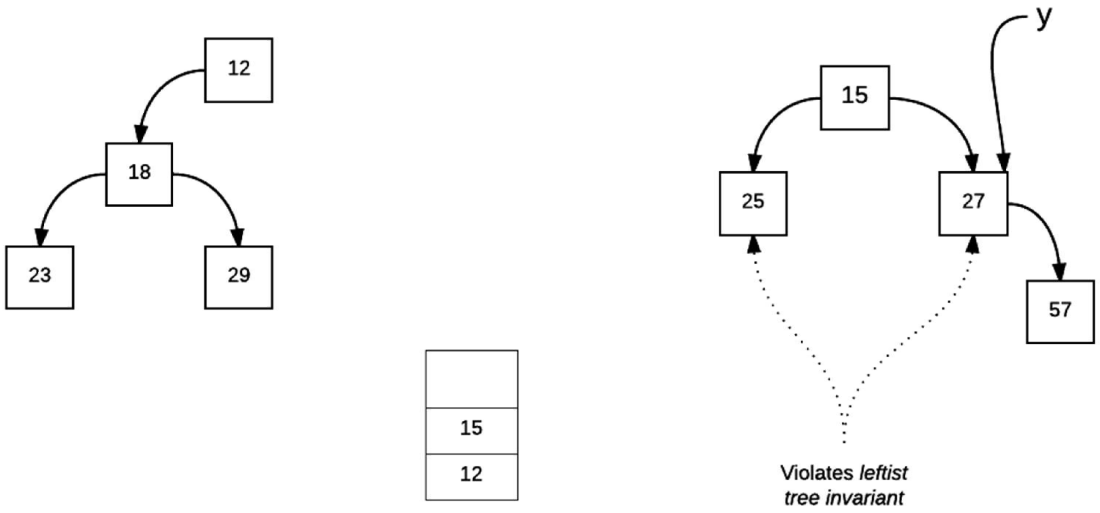

9
12
27
36
47

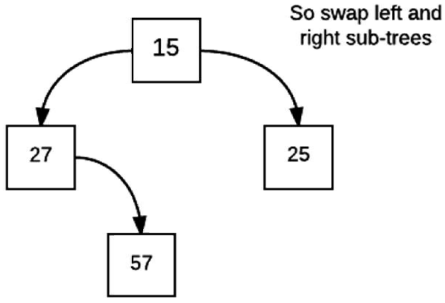

x
12
18
57
23
29
y
15
25
27
stack
12

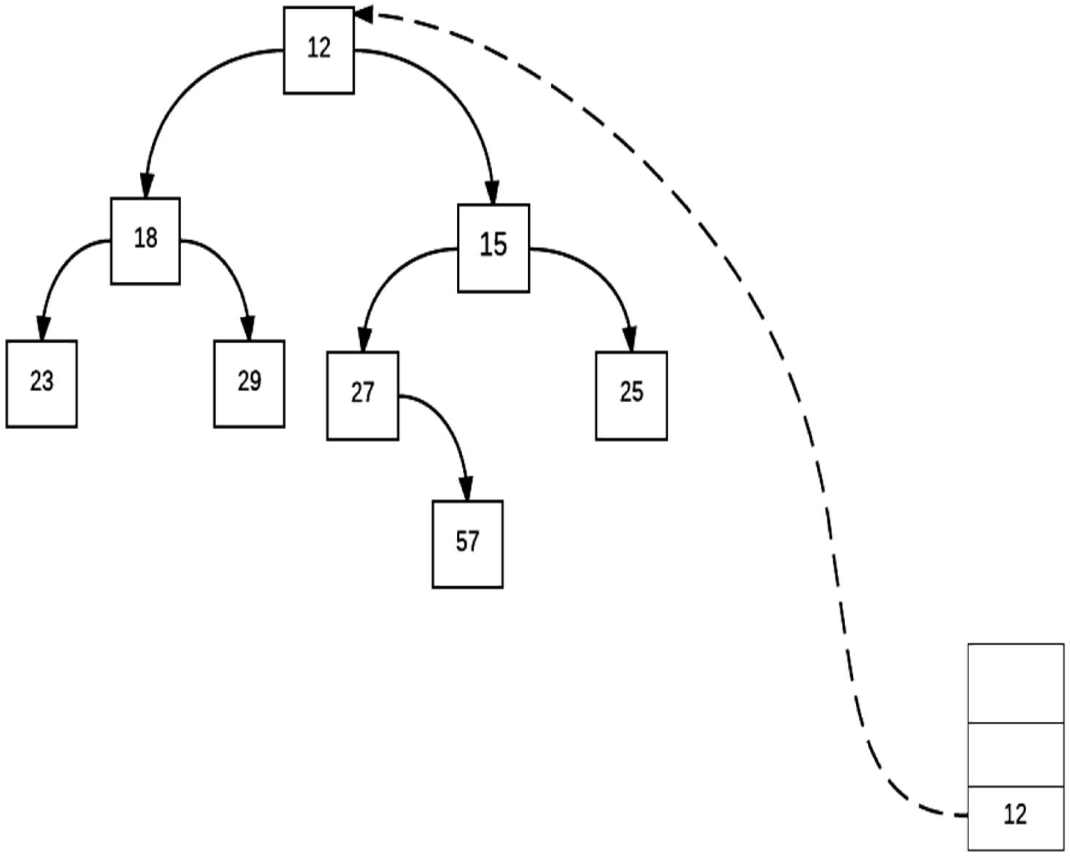

x
y
12
18
57
23
29
15
25
27
15
12


x
y
12
18
57
23
29
15
25
27
27
15
12

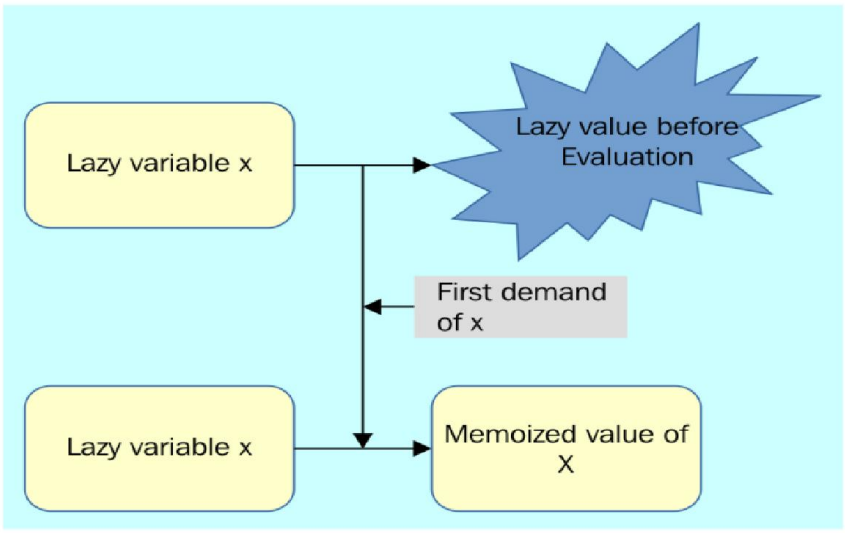

x
12
18
57
23
29
y
15
25
27
Leaf
hitting a leaf
tells us
we can start
linking up the
stacked
sub-trees
57
27
15
12

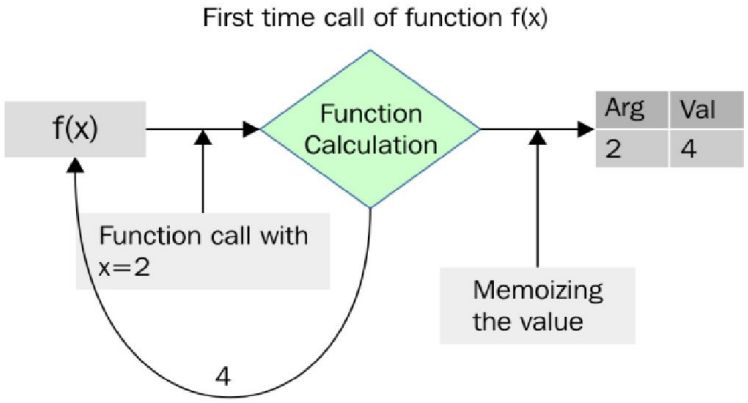

12
18
23
29
15
12
y
15
25
27
57
Violates *leftist*
tree invariant

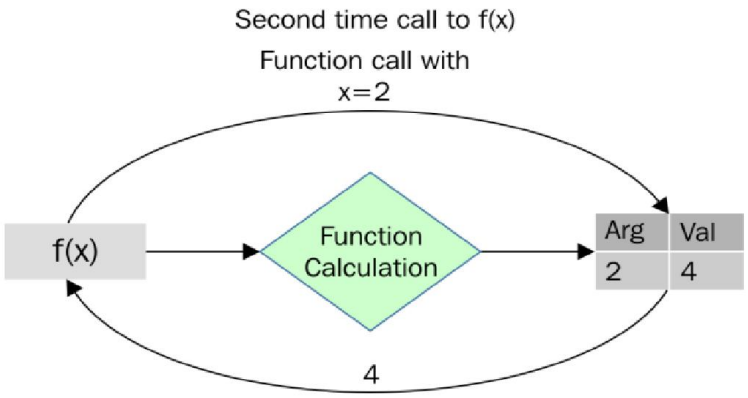

So swap left and
right sub-trees
15
27
25
57


12
18
15
23
29
27
25
57
12

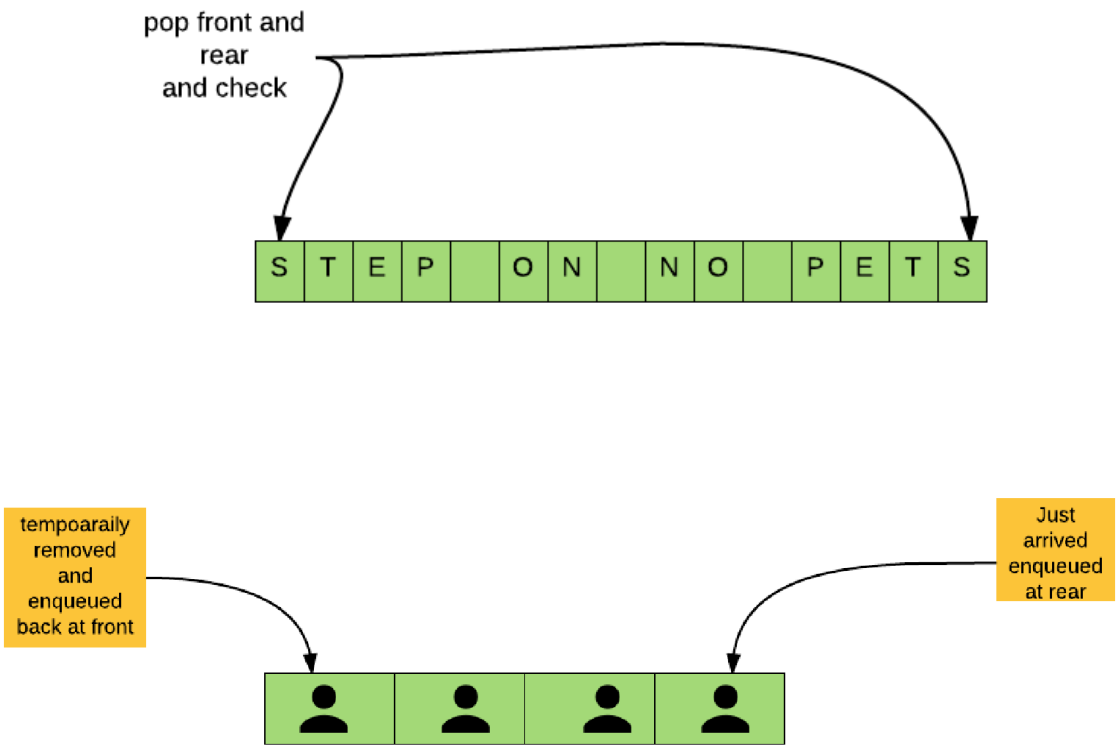
Chapter 9: Streams, Laziness, and Algorithms

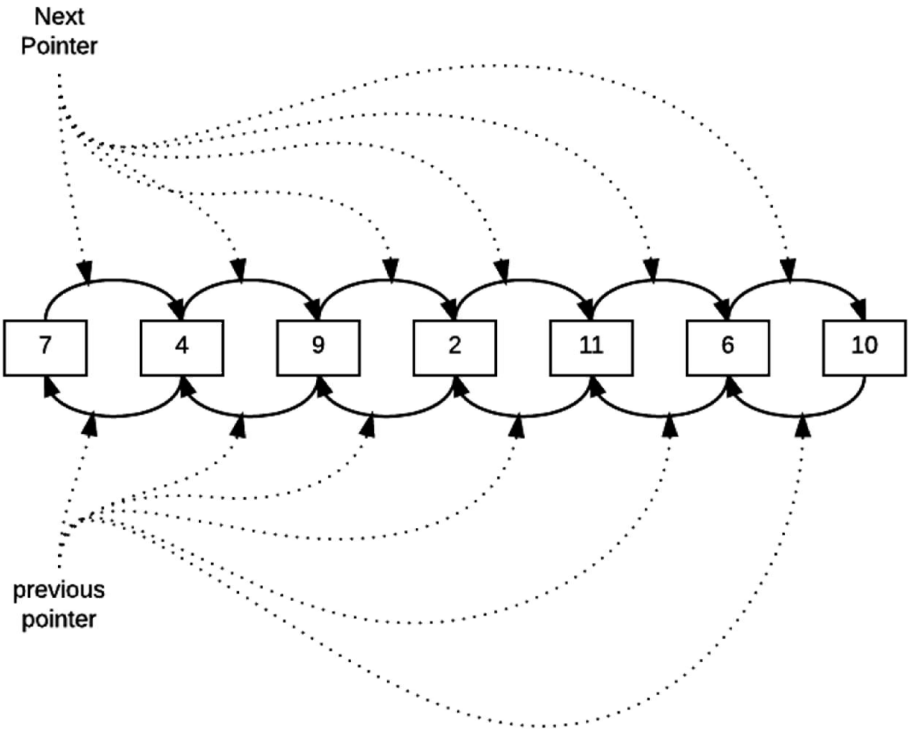


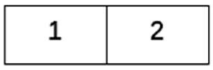




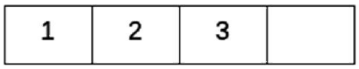
Chapter 10: Being Lazy - Queues and Deques



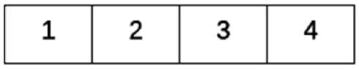




add 3 - no space

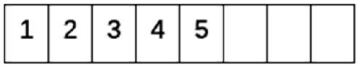


allocate new array *double* the size (4)
copy elements
(We can now absorb 3)

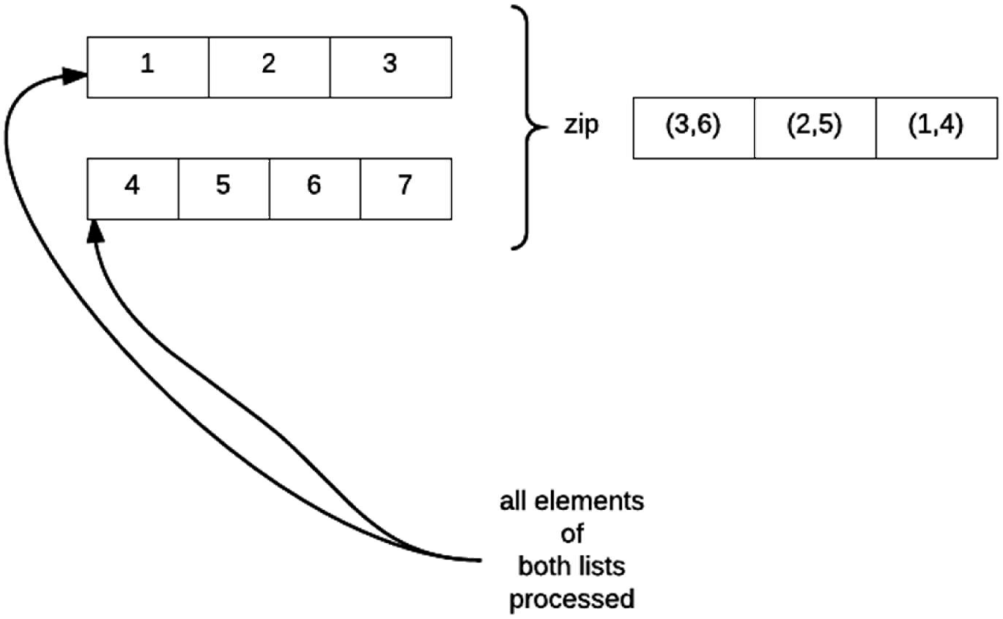


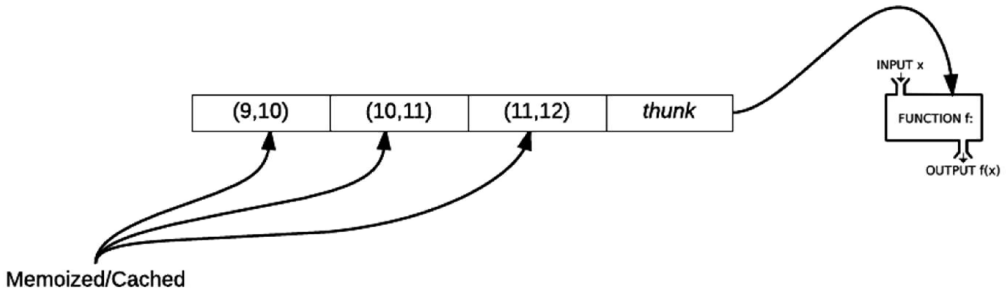
add 4

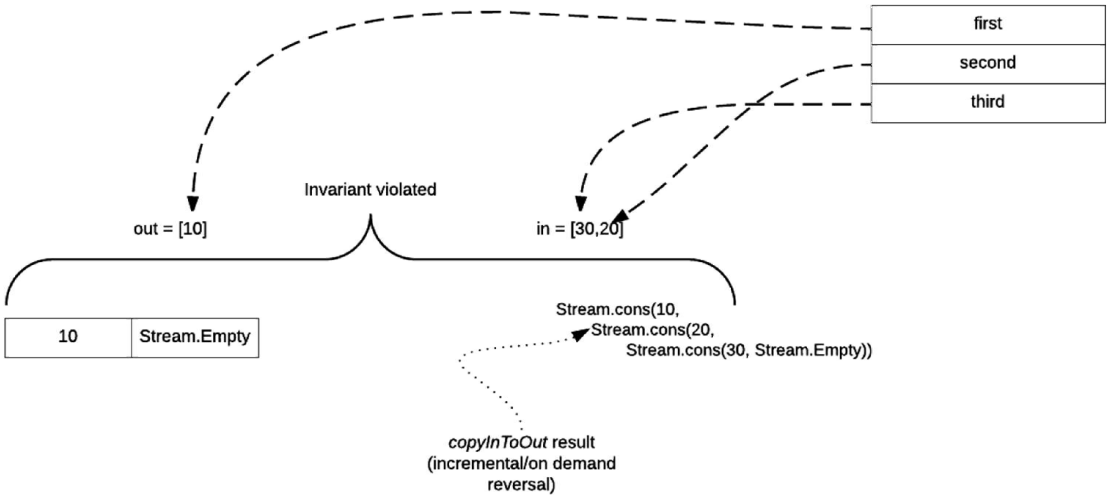
add 5 - no space

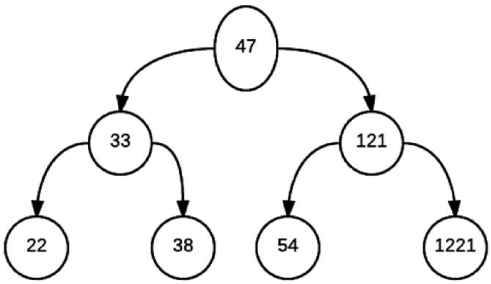


allocate new array *double* the size (8)
copy elements
(We can now absorb 5)

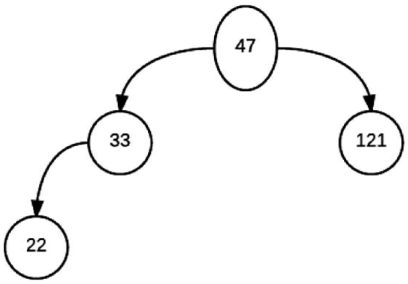

1
2
3
4
5
6
7
zip
(3,6)
(2,5)
(1,4)
all elements
of
both lists
processed


(9,10)
(10,11)
(11,12)
thunk
INPUT x
FUNCTION f:
OUTPUT f(x)
Memoized/Cached

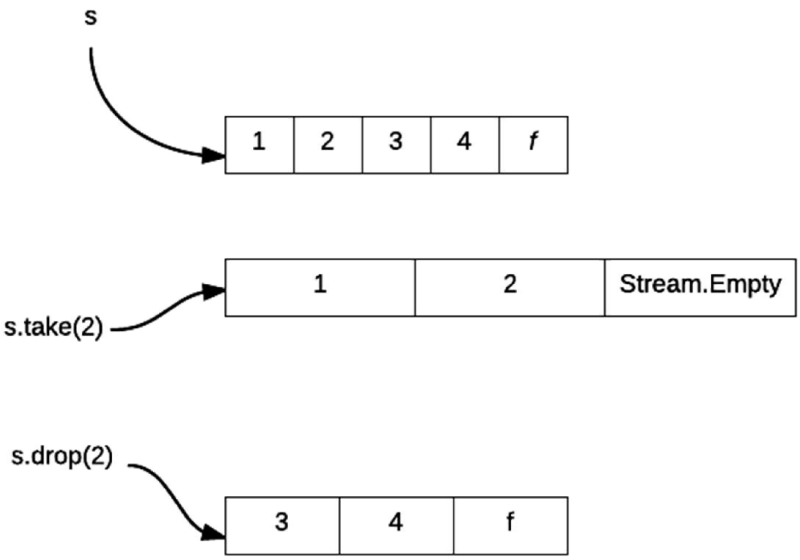

first
second
third
Invariant violated
out = [10]
in = [30,20]
10
Stream.Empty
Stream.cons(10,
Stream.cons(20,
Stream.cons(30, Stream.Empty))
copyInToOut result
(incremental/on demand
reversal)



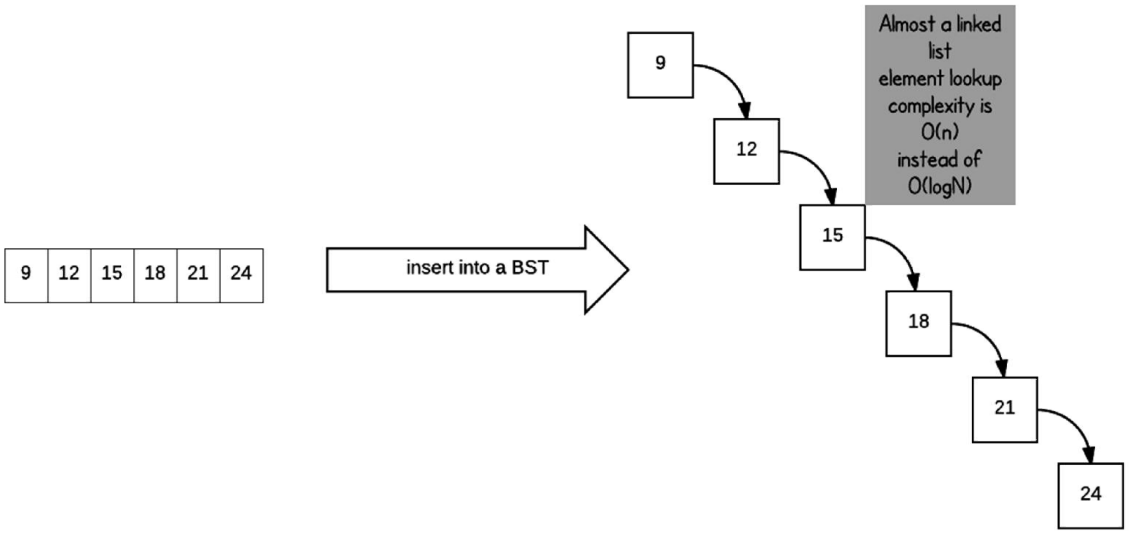
A PERFECTLY BALANCED

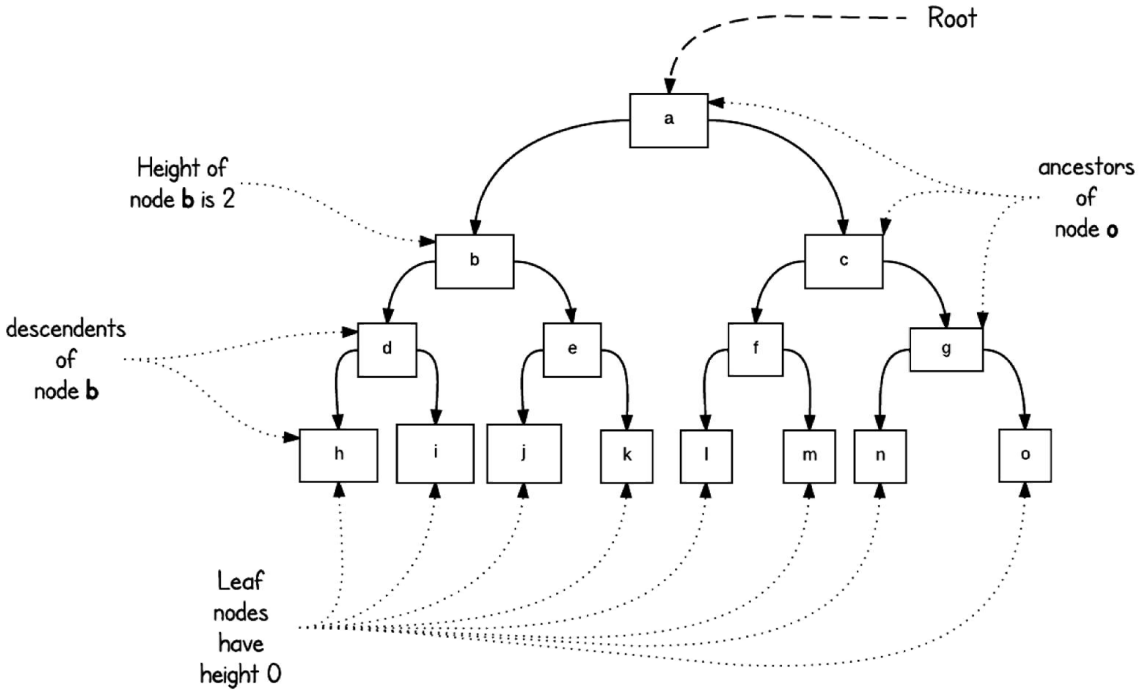


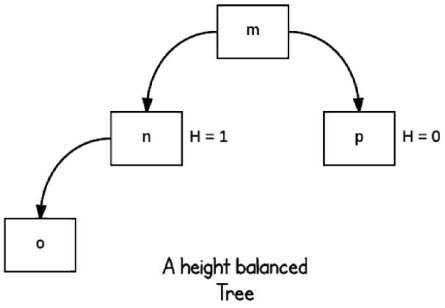
ALMOST PERFECTLY BALANCED



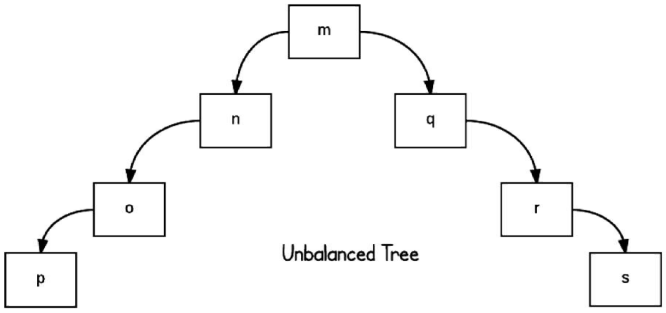
Chapter 11: Red-Black Trees





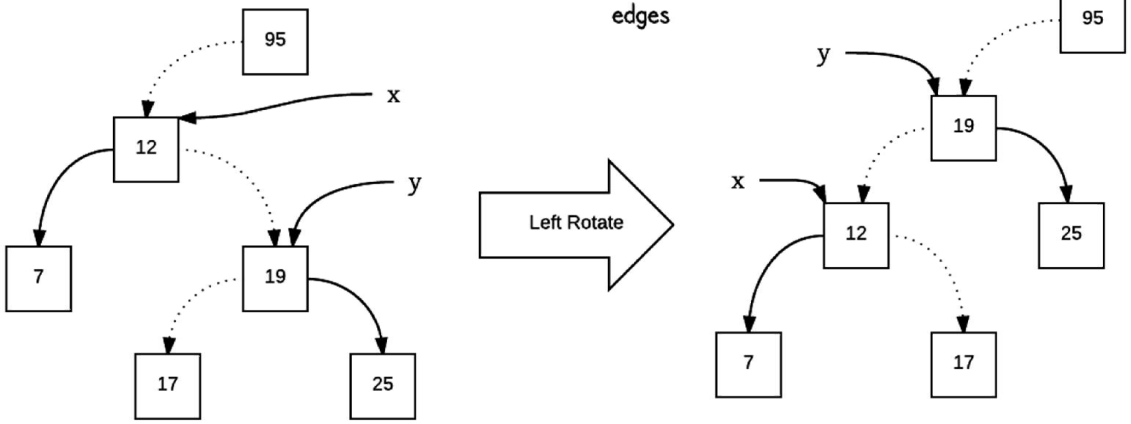


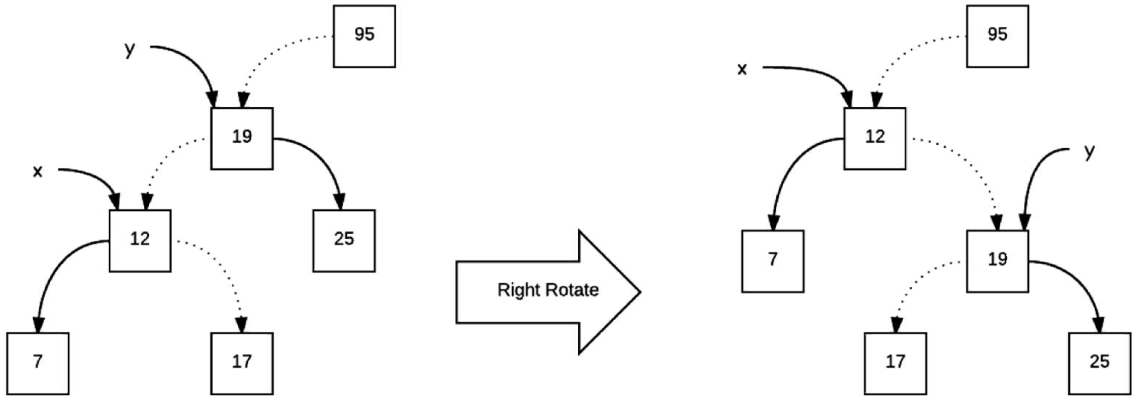
A height balanced Tree

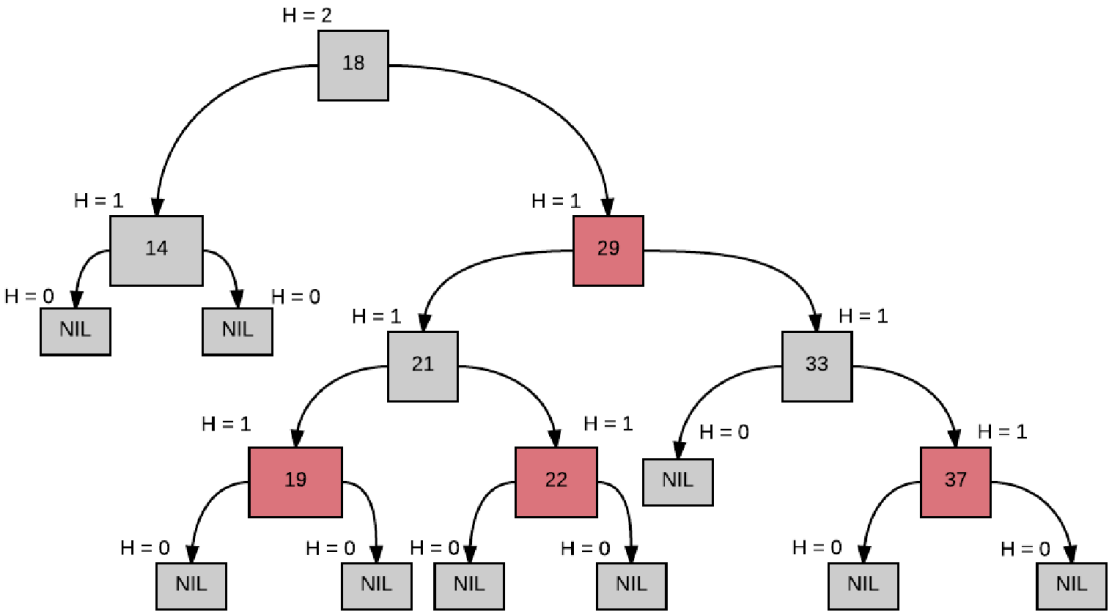


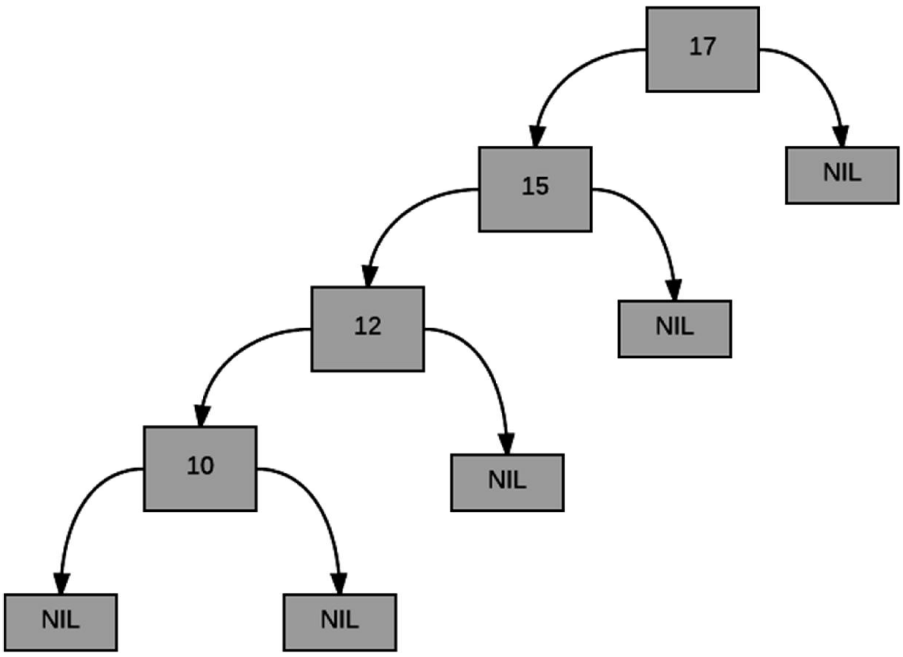
Unbalanced Tree

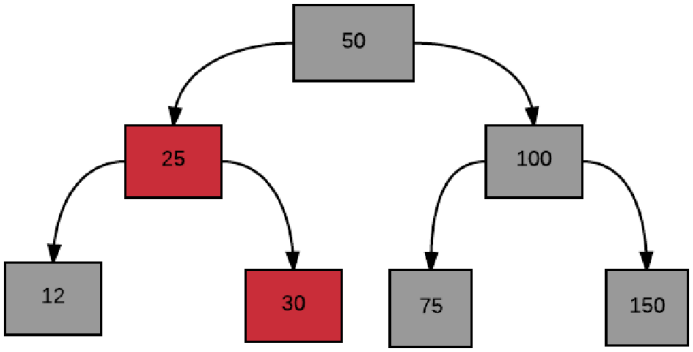
Dotted edges indicate changed edges

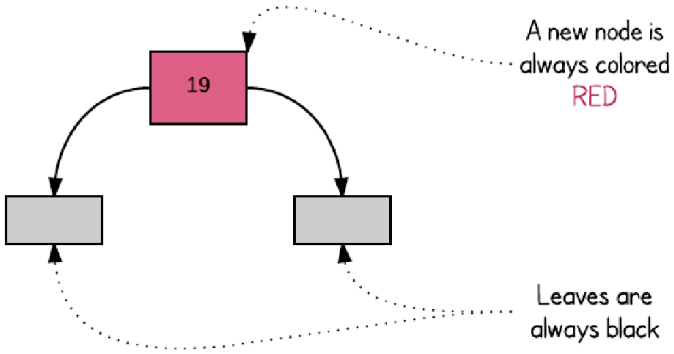


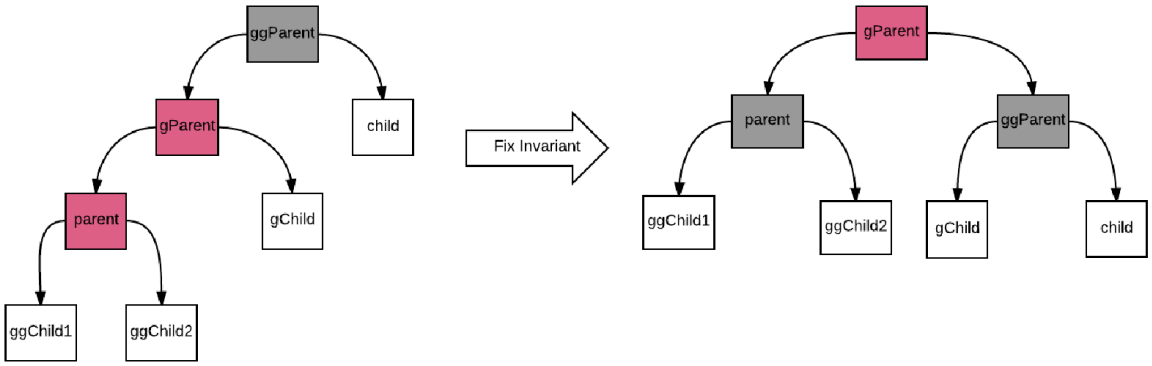


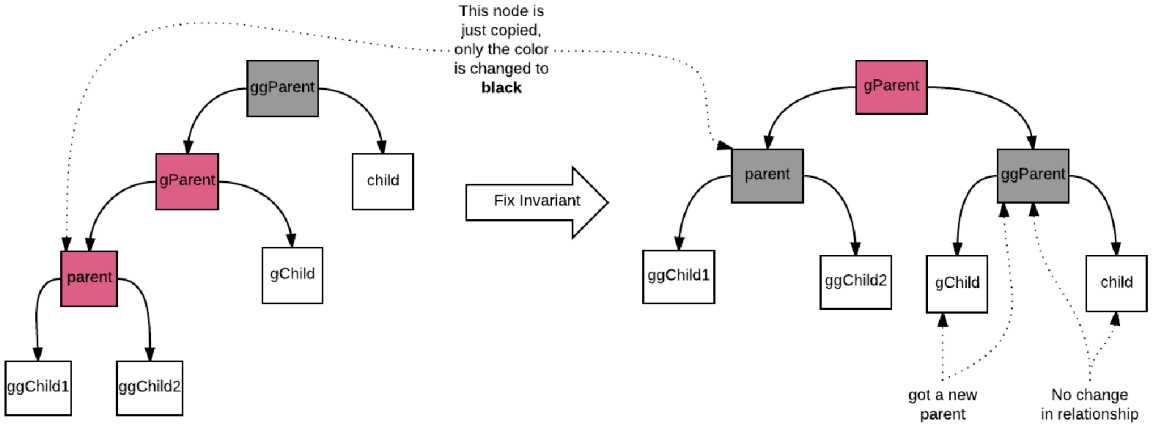

H = 2
18
H = 1
14
H = 1
29
H = 0
NIL
H = 0
NIL
H = 1
21
H = 1
33
H = 1
19
H = 1
22
H = 0
NIL
H = 1
37
H = 0
NIL
H = 0
NIL
H = 0
NIL
H = 0
NIL
H = 0
NIL
H = 0
NIL


17
15
NIL
12
NIL
10
NIL
NIL
NIL

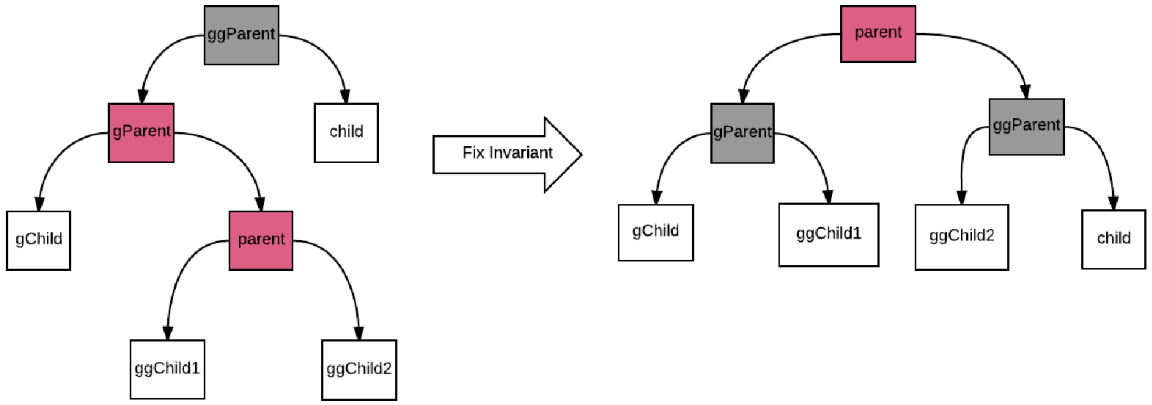

50
25
100
12
30
75
150

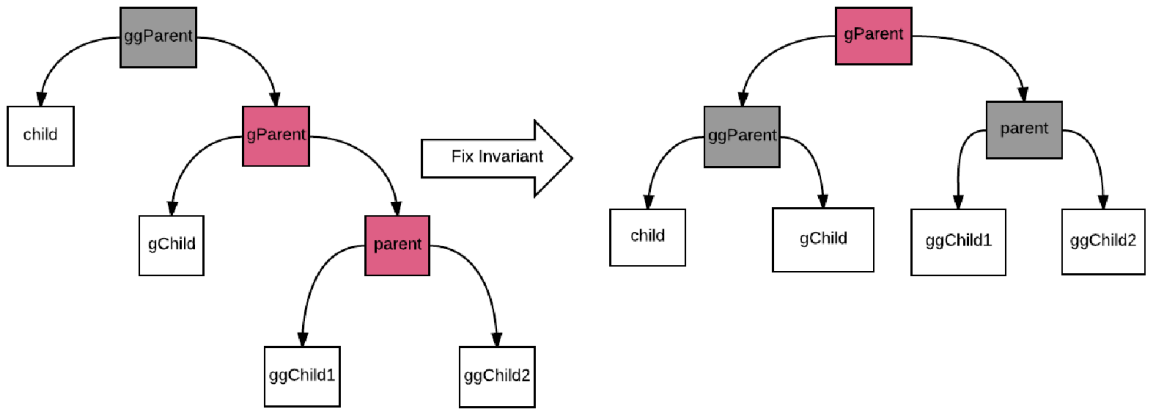

19
A new node is
always colored
RED
Leaves are
always black

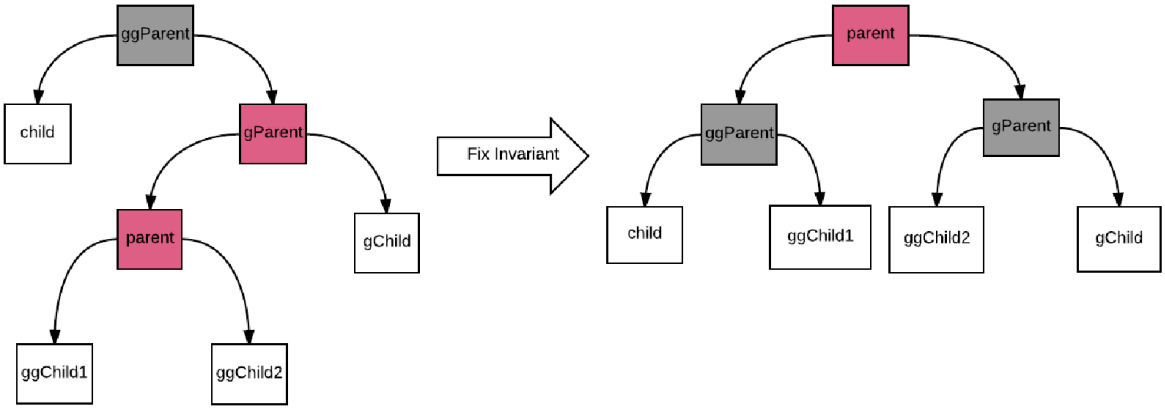

ggParent
gParent
child
parent
gChild
ggChild1
ggChild2
Fix Invariant
gParent
parent
ggParent
ggChild1
ggChild2
gChild
child

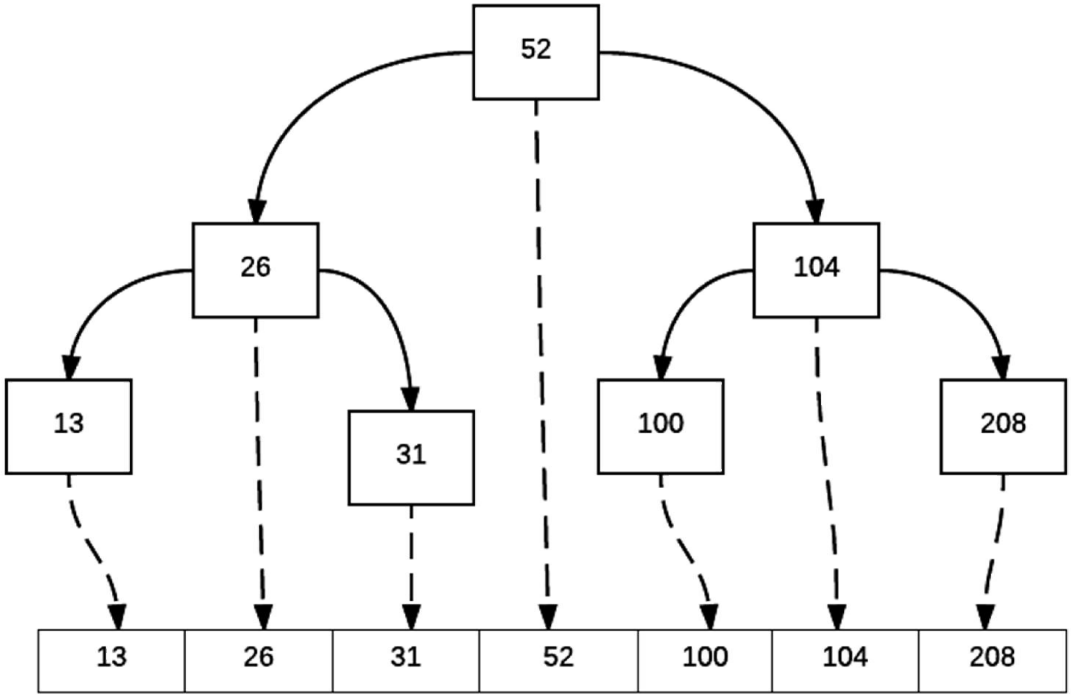


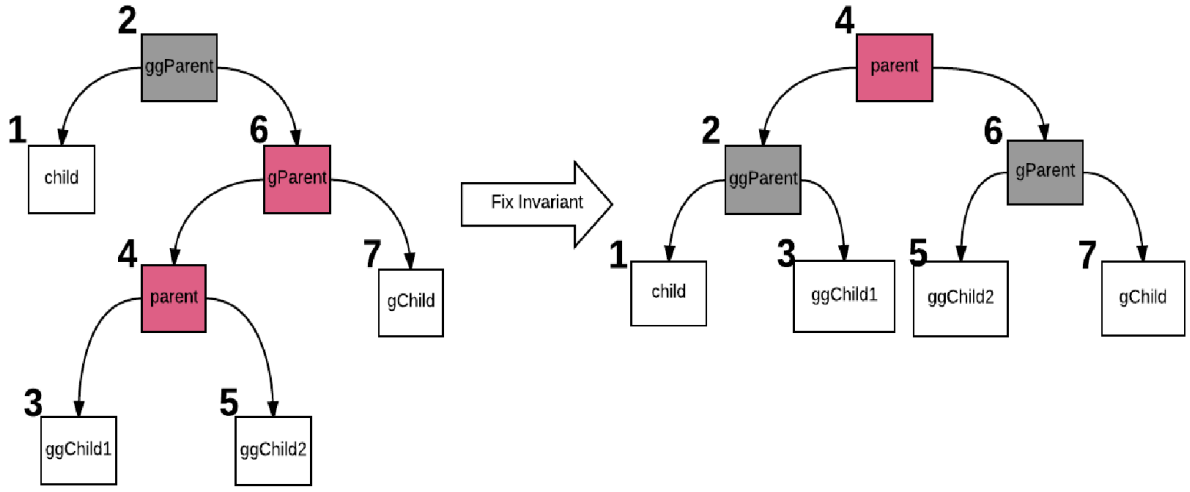

```
gParent.copy(color = Red,
left = parent.copy(color = Black),
      right = ggParent.copy(color = Black, left =
    gChild, right = child))
```



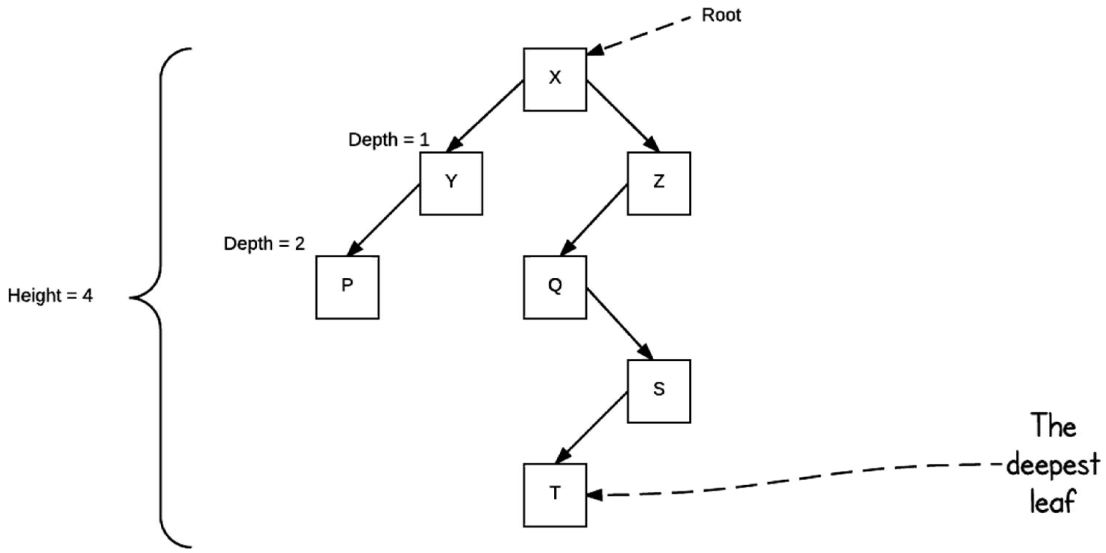


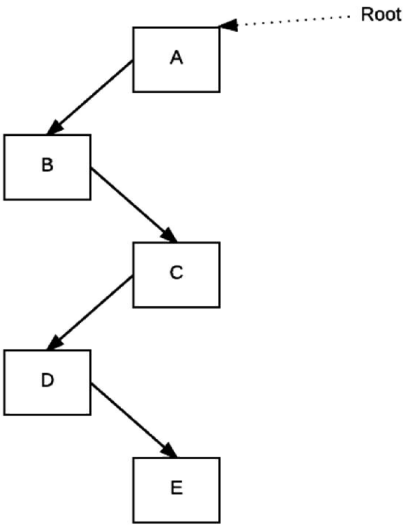

ggParent
child
gParent
parent
gChild
ggChild1
ggChild2
Fix Invariant
parent
ggParent
gParent
child
ggChild1
ggChild2
gChild

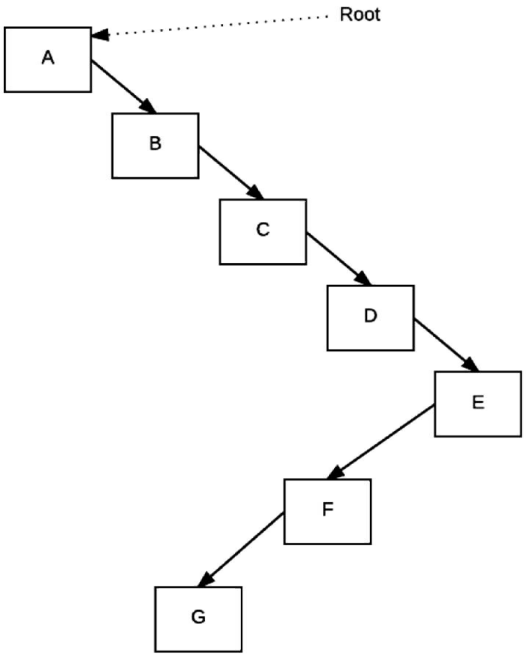

52
26
104
13
31
100
208
13
26
31
52
100
104
208

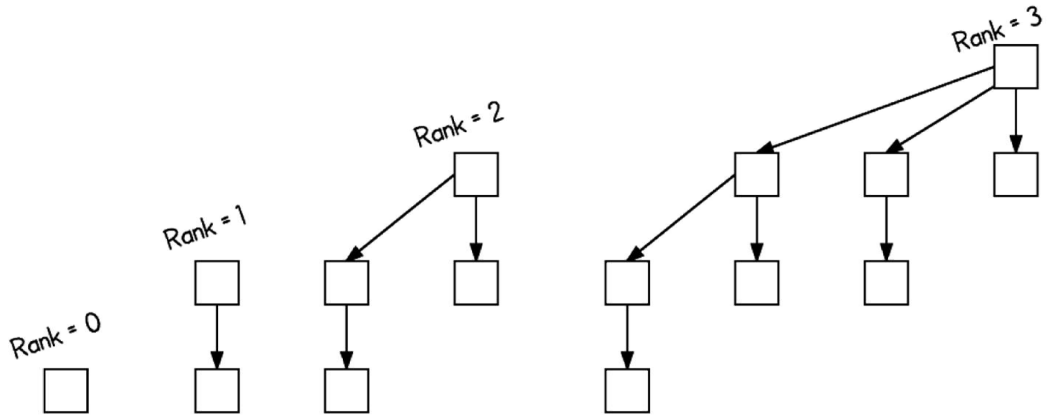

2
ggParent
1
child
6
gParent
4
parent
7
gChild
3
ggChild1
5
ggChild2
Fix Invariant
4
parent
2
ggParent
6
gParent
1
child
3
ggChild1
5
ggChild2
7
gChild

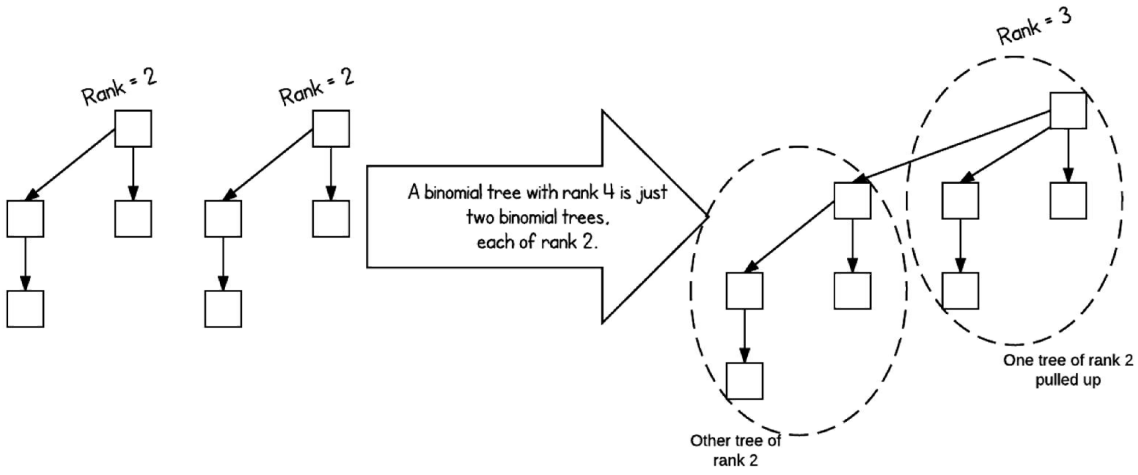
Chapter 12: Binomial Heaps

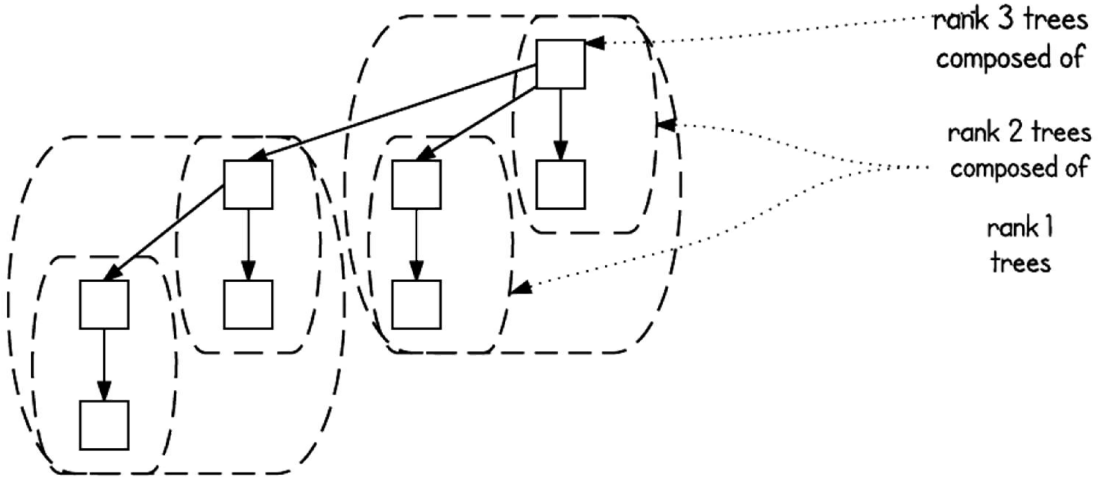


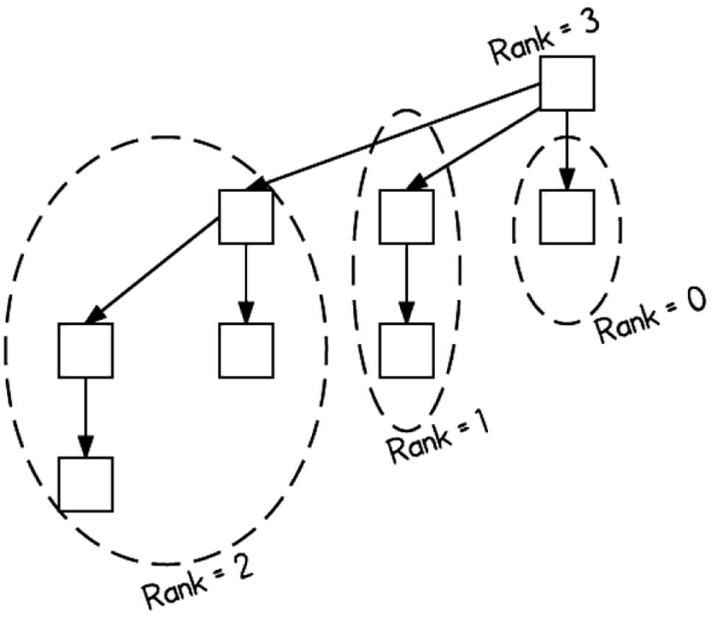


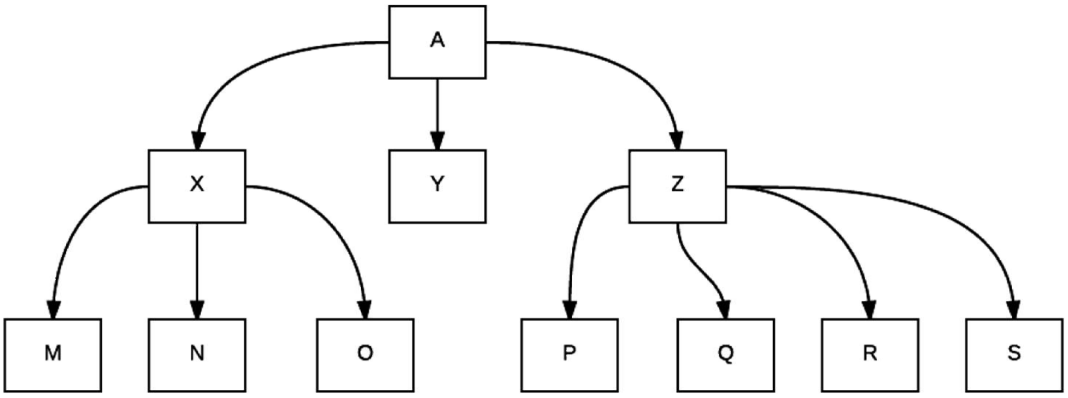


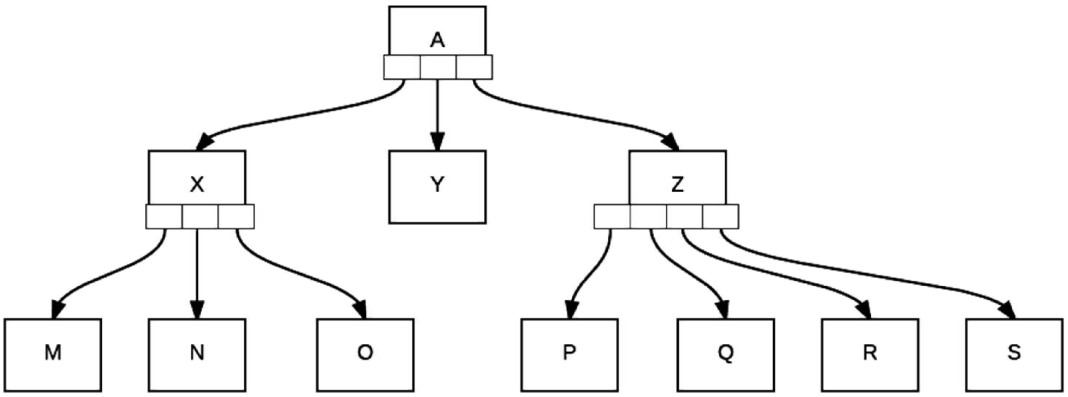

Rank = 0
Rank = 1
Rank = 2
Rank = 3

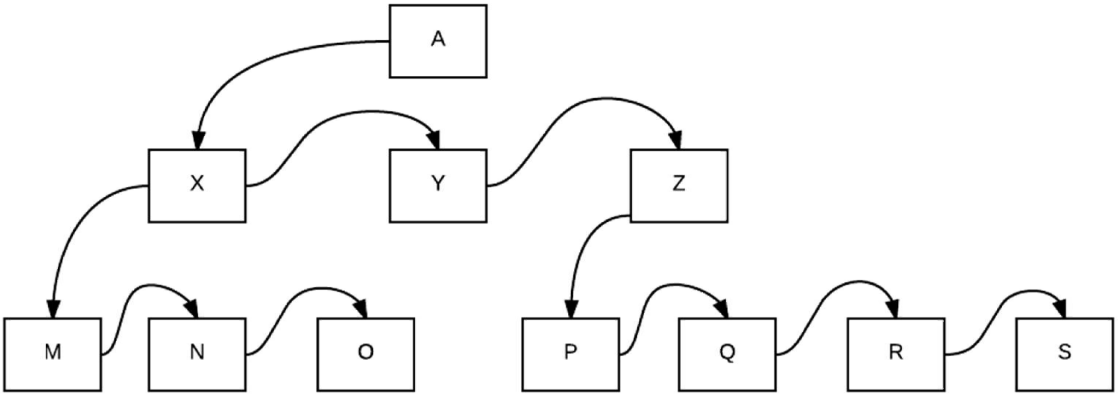

Rank = 2
Rank = 2
A binomial tree with rank 4 is just two binomial trees, each of rank 2.
Rank = 3
One tree of rank 2 pulled up
Other tree of rank 2

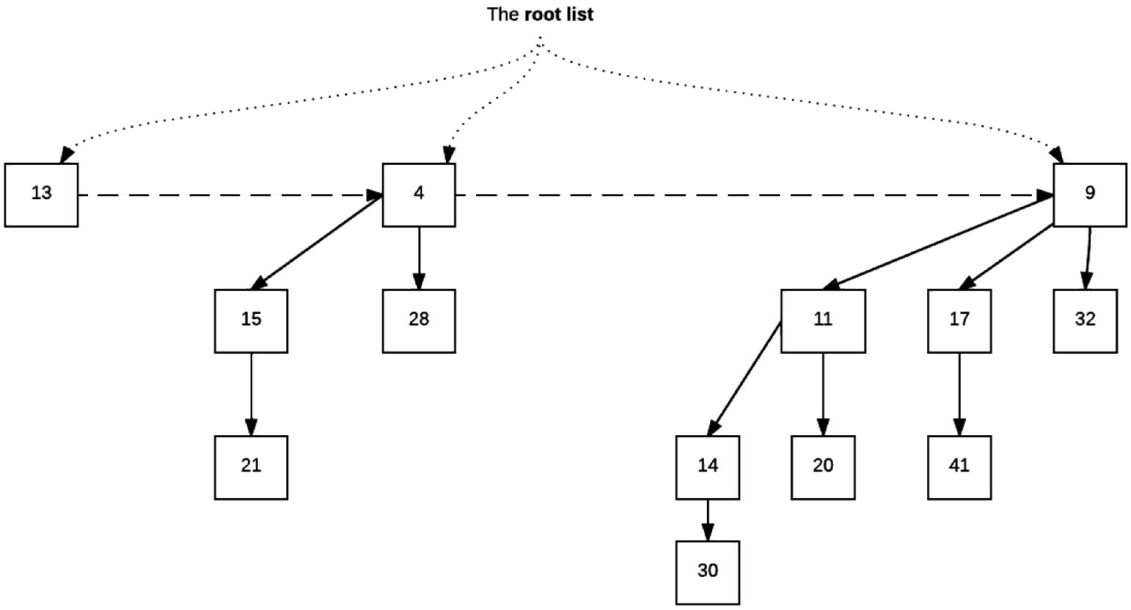

rank 3 trees composed of
rank 2 trees composed of
rank 1 trees

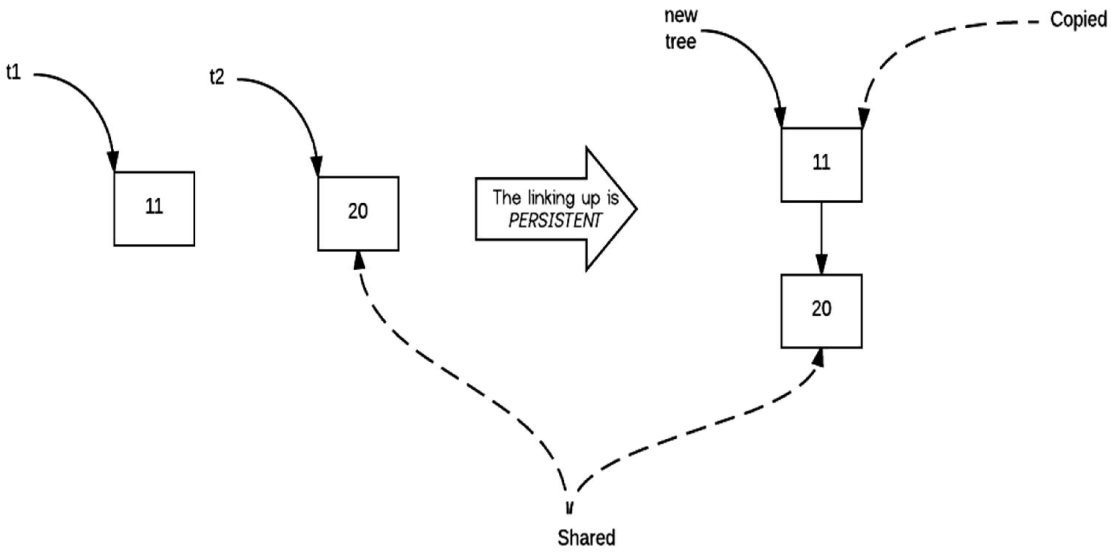

Rank = 3
Rank = 0
Rank = 1
Rank = 2

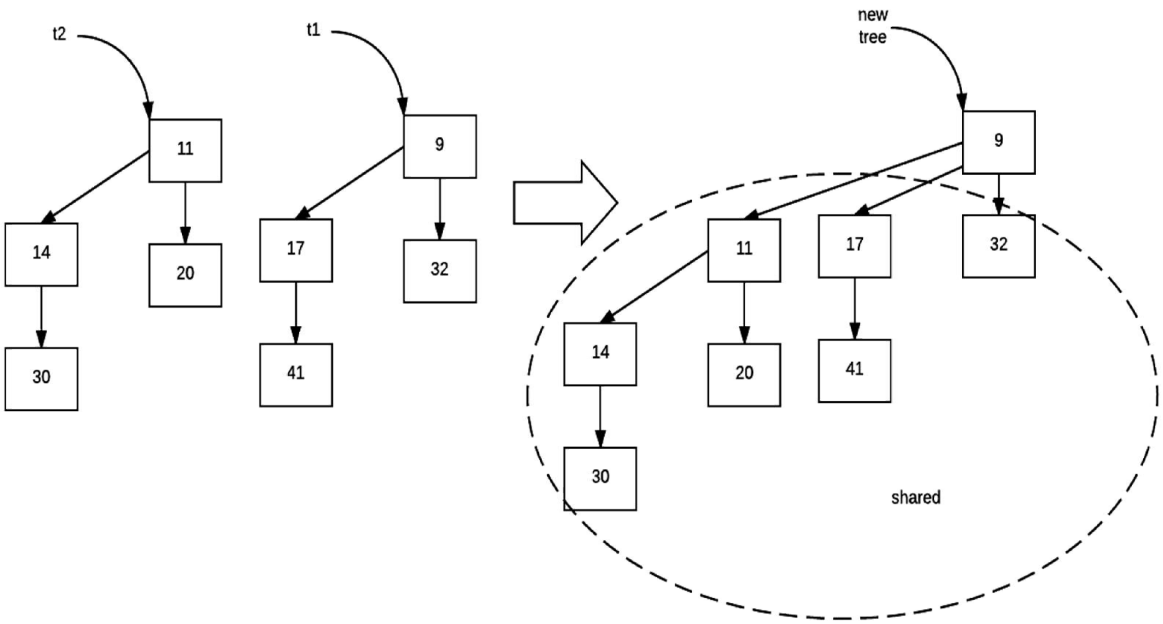

A
X
Y
Z
M
N
O
P
Q
R
S

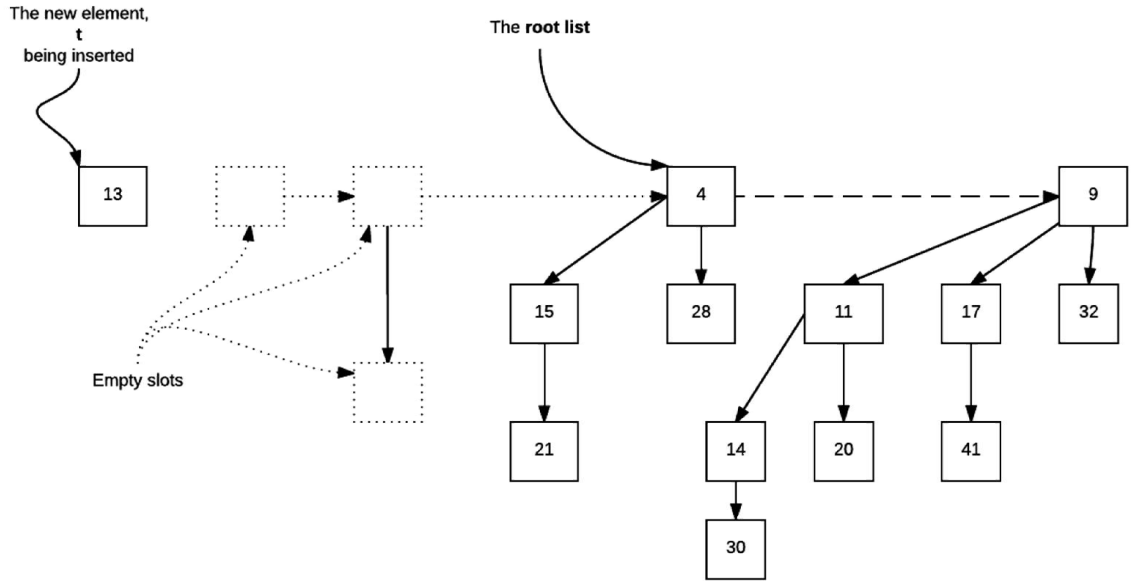

A
X
Y
Z
M
N
O
P
Q
R
S

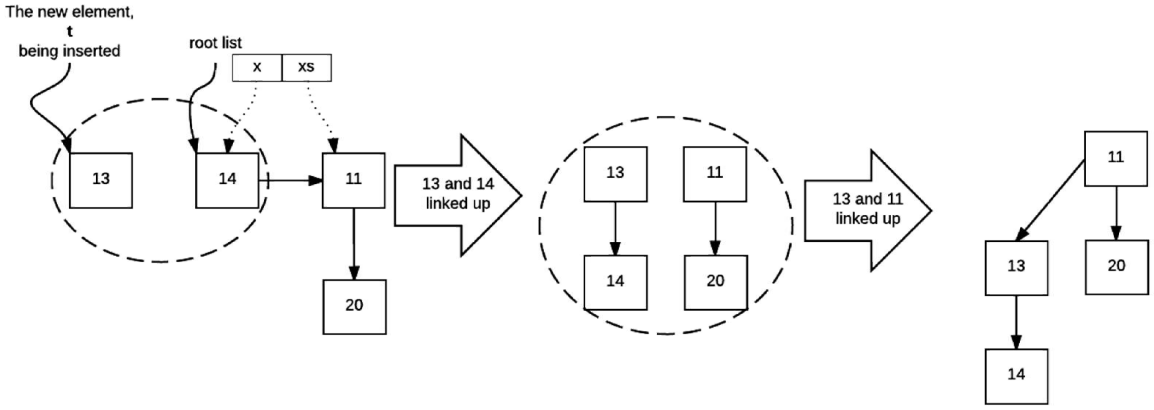

A
X
Y
Z
M
N
O
P
Q
R
S

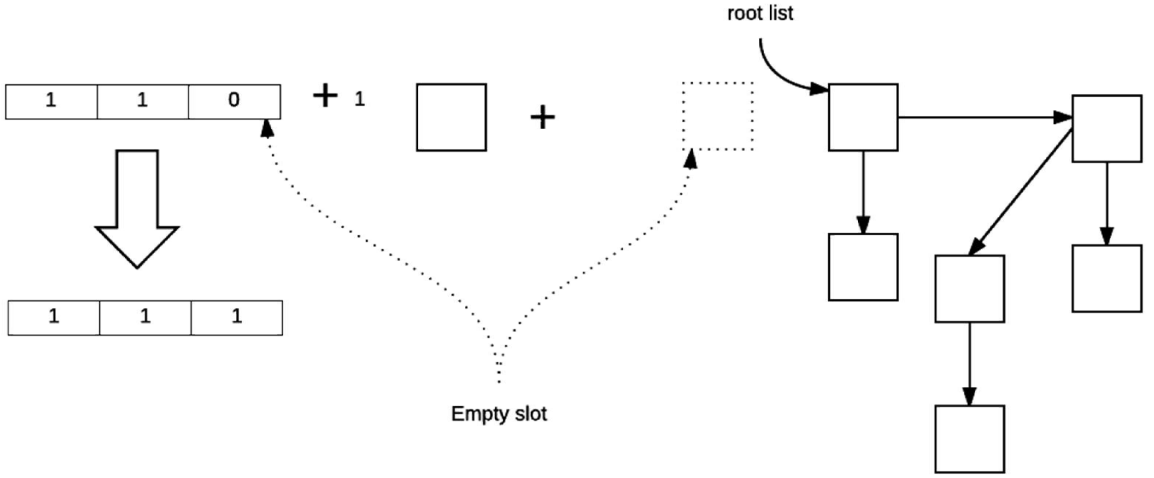

The root list
13
4
9
15
28
11
17
32
21
14
20
41
30

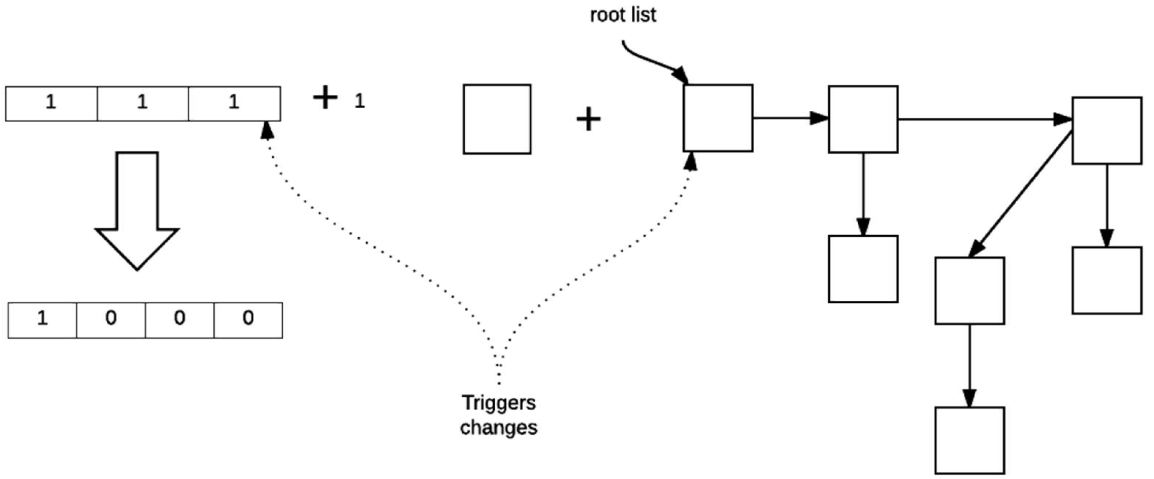

t1
11
t2
20
The linking up is
PERSISTENT
new
tree
Copied
11
20
Shared


t2
11
14
20
30
t1
9
17
32
41
new
tree
9
11
17
32
14
20
41
30
shared

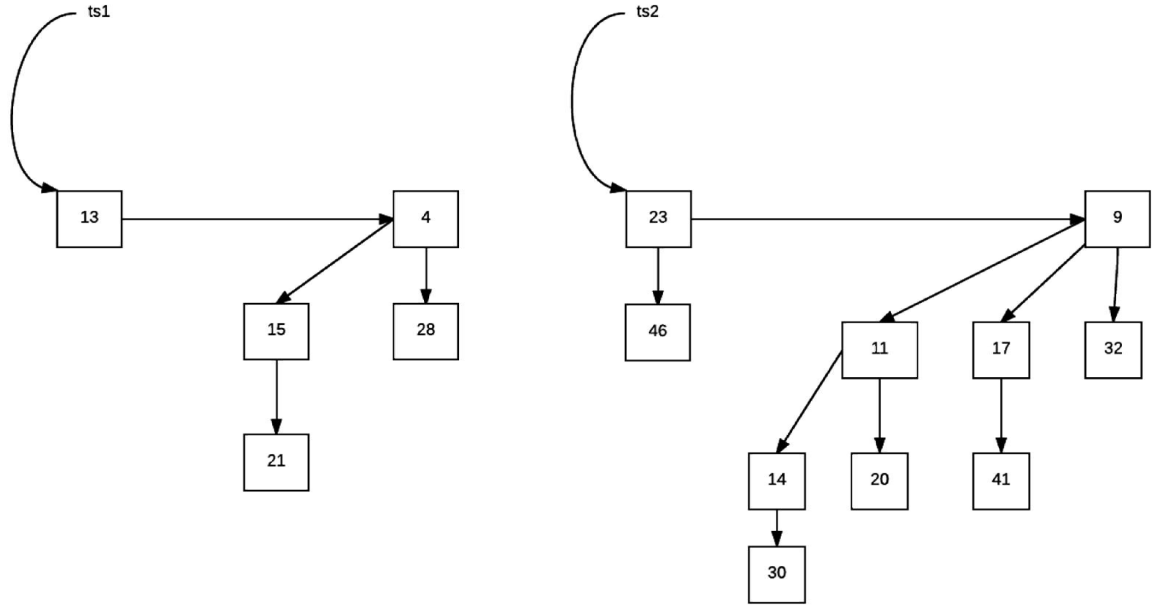

The new element,
t
being inserted
The root list
13
4
9
15
28
11
17
32
Empty slots
21
14
20
41
30

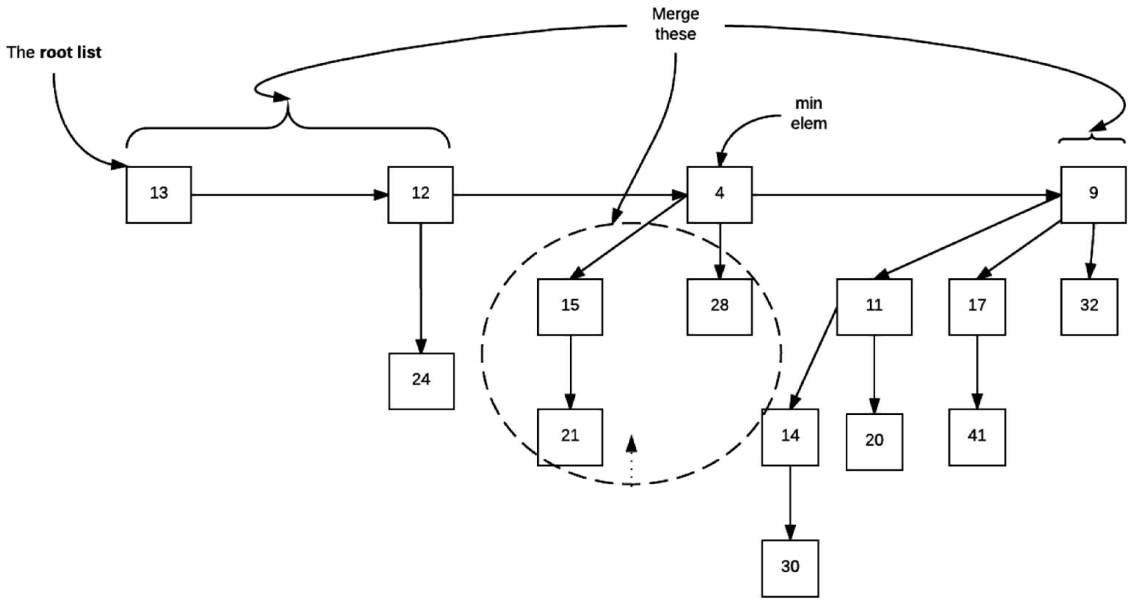

The new element,
t
being inserted
root list
x
xs
13
14
11
20
13 and 14
linked up
13
11
14
20
13 and 11
linked up
11
13
20
14


root list
1
1
0
+ 1
+
1
1
1
Empty slot

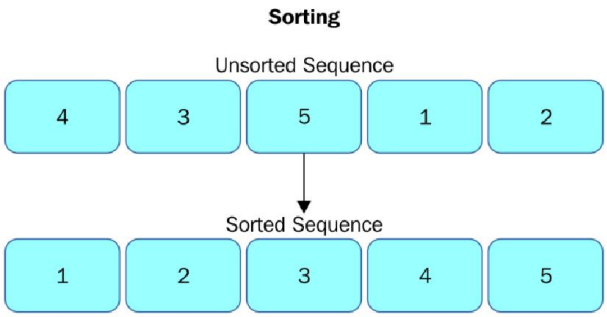

root list
1
1
1
+ 1
+
1
0
0
0
Triggers
changes

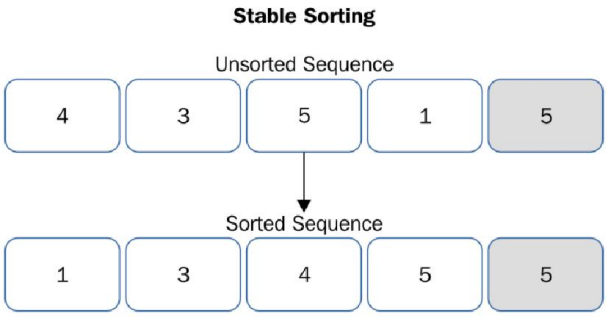
\\\

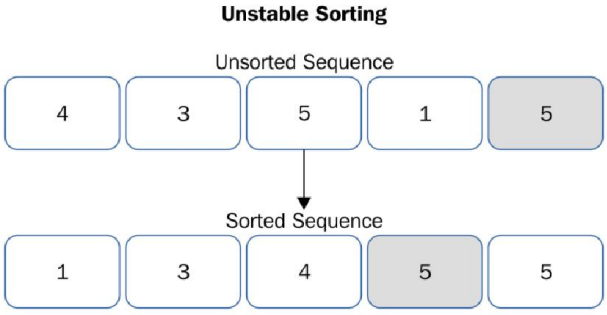

ts1
13
4
15
28
21
ts2
23
9
46
11
17
32
14
20
41
30


Merge
these
The root list
min
elem
13
12
4
9
15
28
11
17
32
24
21
14
20
41
30

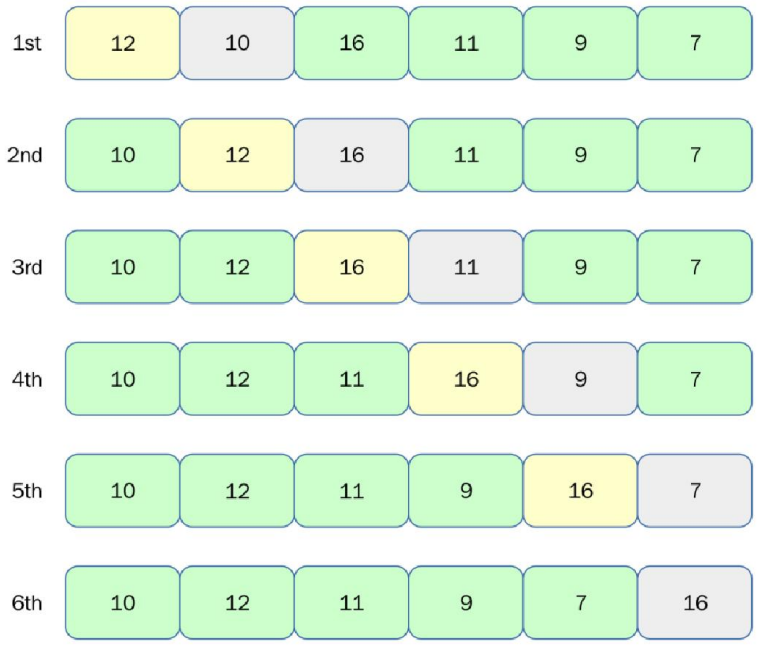
Chapter 13: Sorting



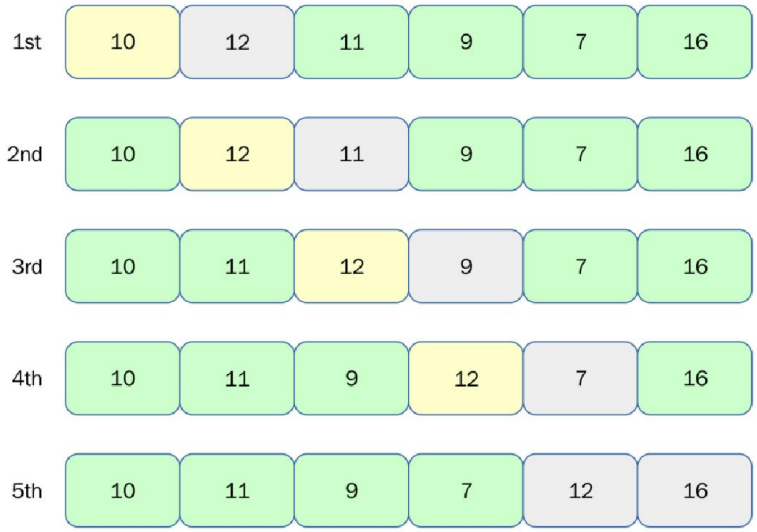




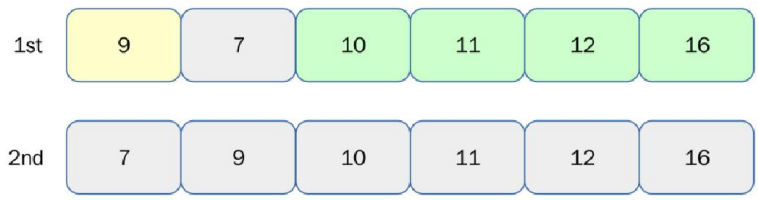
Bubble Sort First Pass



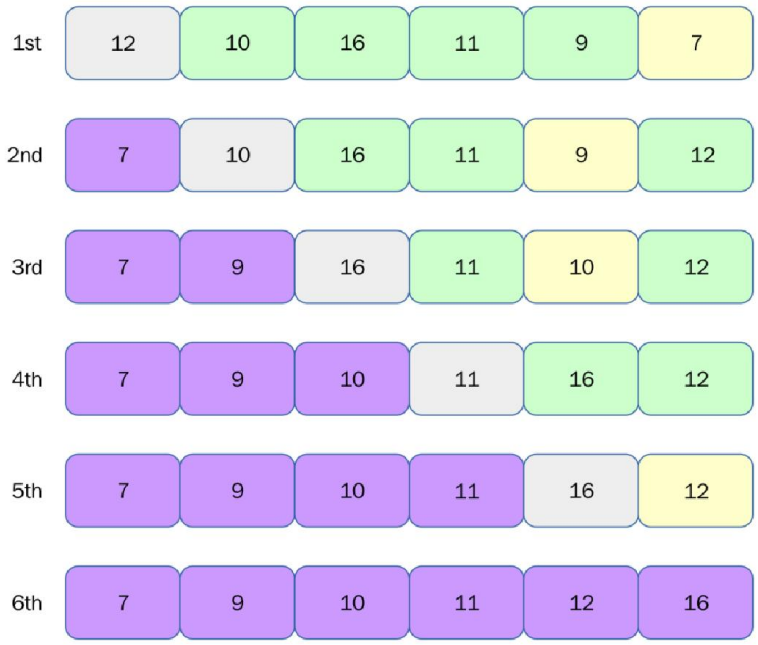
Bubble Sort Second Pass



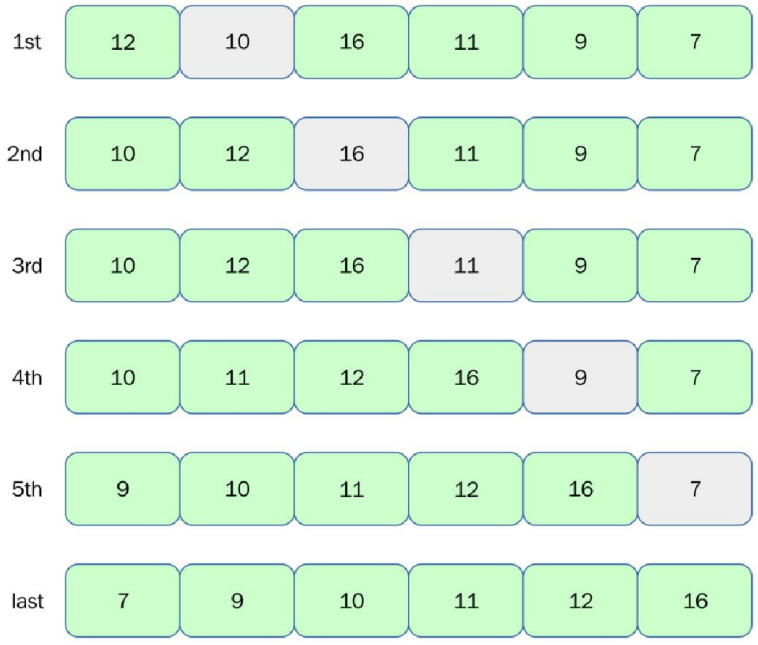
Bubble Sort Last Pass

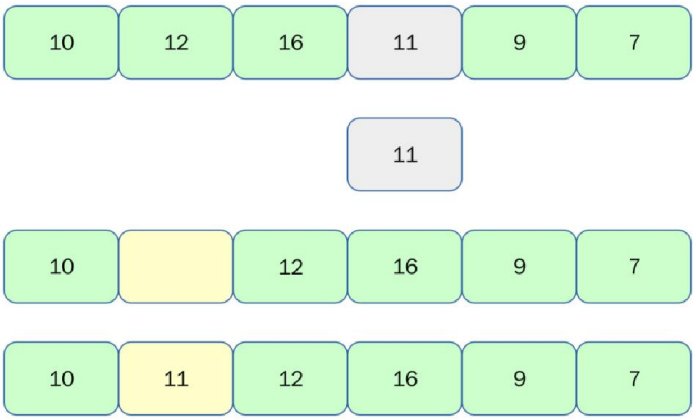


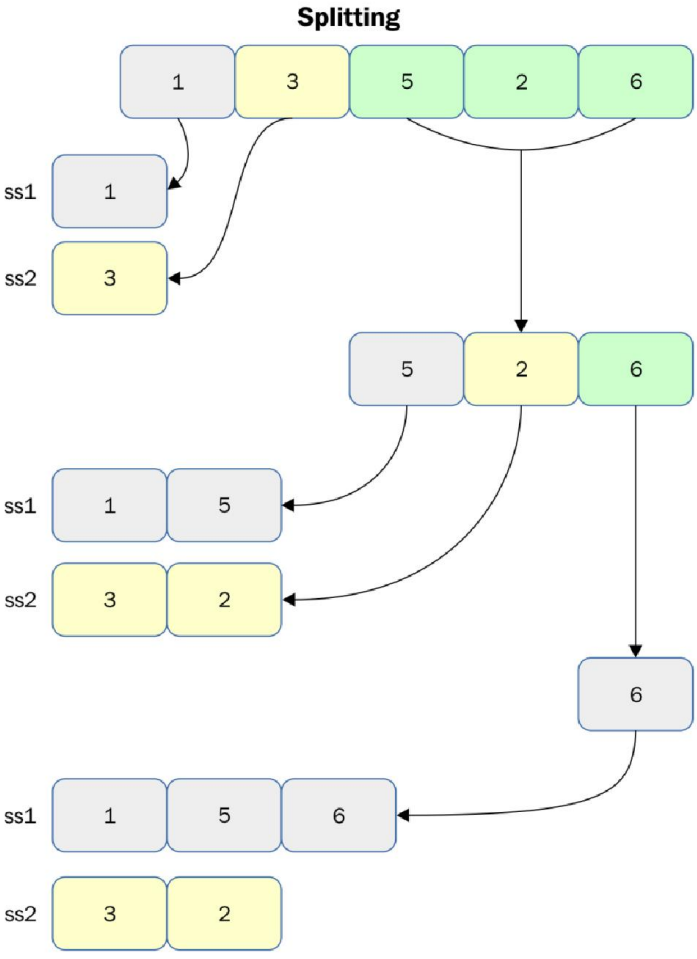
Selection Sort

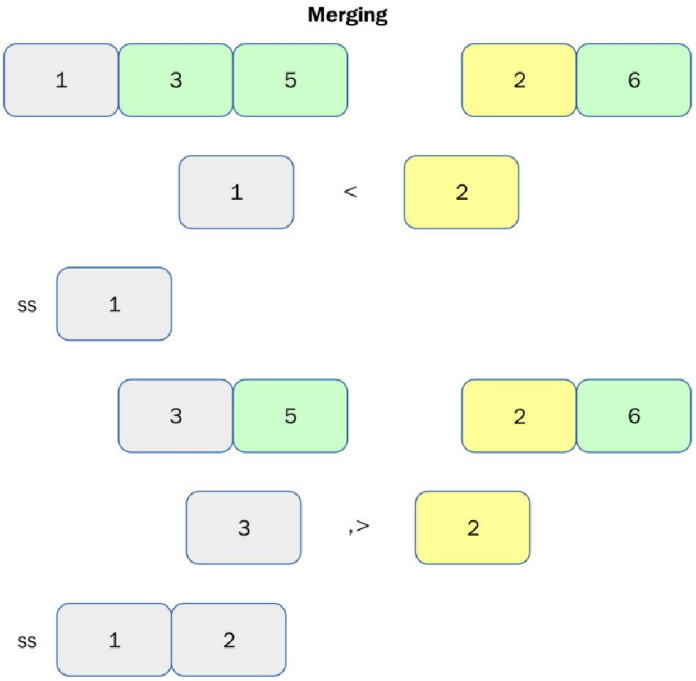


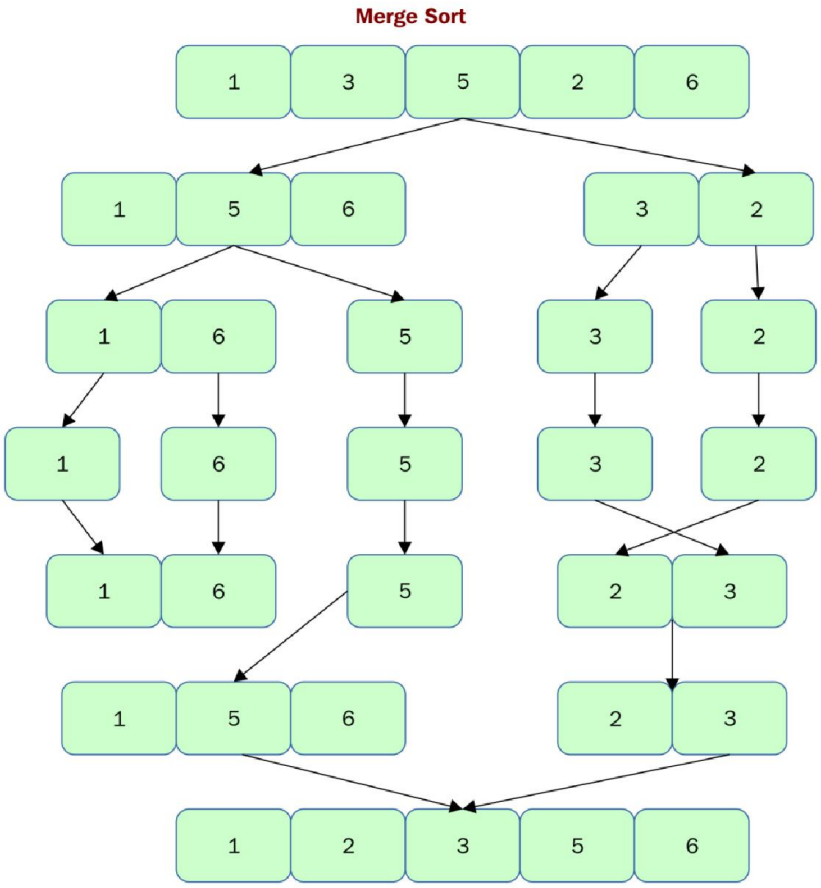
Insertion Sort

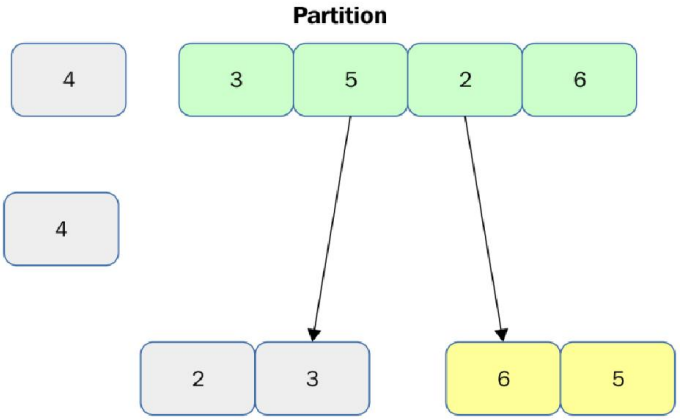


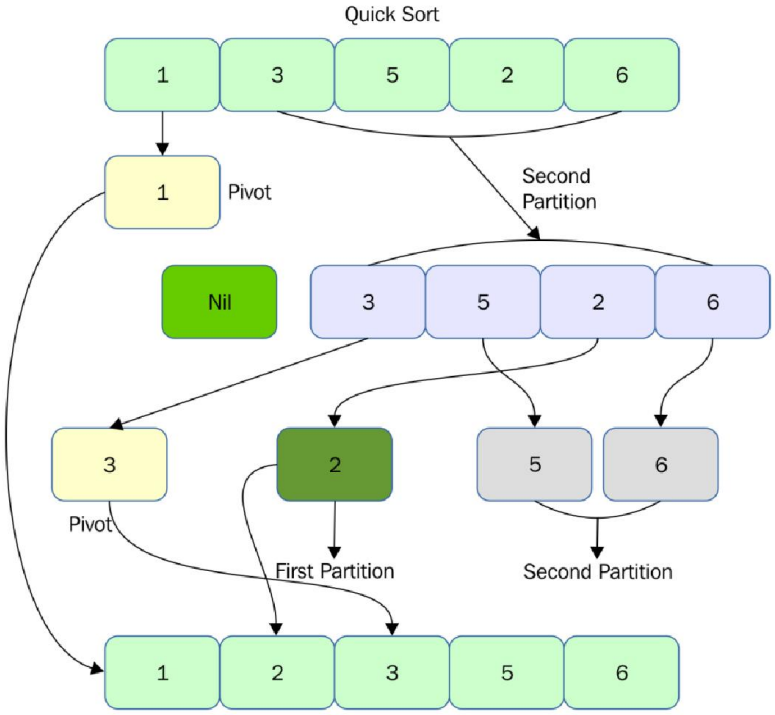

10
12
16
11
9
7
11
10
12
16
9
7
10
11
12
16
9
7


Splitting
1
3
5
2
6
ss1
1
ss2
3
5
2
6
ss1
1
5
ss2
3
2
6
ss1
1
5
6
ss2
3
2


Merging
1
3
5
2
6
1
<
2
ss
1
3
5
2
6
3
,>
2
ss
1
2


Merge Sort
1
3
5
2
6
1
5
6
3
2
1
6
5
3
2
1
6
5
3
2
1
6
5
2
3
1
5
6
2
3
1
2
3
5
6


Partition
4
3
5
2
6
4
2
3
6
5


Quick Sort
1
3
5
2
6
1
Pivot
Second
Partition
Nil
3
5
2
6
3
Pivot
2
First Partition
5
6
Second Partition
1
2
3
5
6