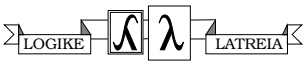
THOMAS VANDRUNEN

ALGORITHMIC COMMONPLACES FIGURES-ONLY VERSION

ALGORITHMIC COMMONPLACES

Figures-Only Version

June 2, 2023



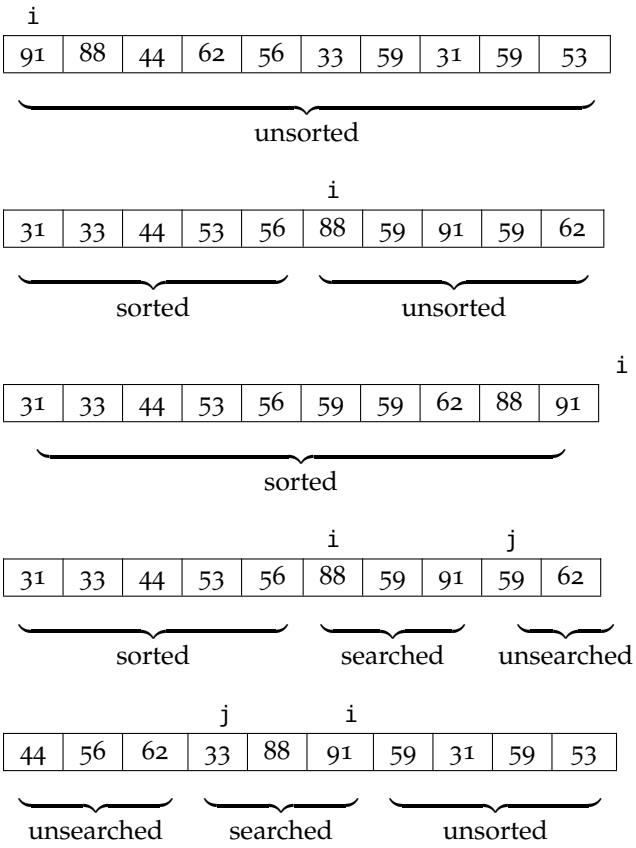
Typeset with LaTeX using the TufteBook document class.

Contents

1
Algorithms

1.1 Pretest

1.2 Algorithms and correctness



1.3 *Algorithms and efficiency*

```
def bounded_linear_search(sequence, P):
 a0  found = False
     i = 0
     while not found and i < len(sequence):   a1(n+1)
     a2n  found = P(sequence[i])
          i += 1
     if found:   a3
      a4  return i - 1
     else :
      a5  return -1
```

```
def binary_search(sequence, TO, item):
 c0  low = 0
     high = len(sequence)
     while high - low > 1:   c1(lg n+1)
   c2 lg n  mid = (low + high) / 2
          compar = TO(item, sequence[mid])
          if compar < 0:    # item comes before mid
               high = mid
          elif compar > 0:  # item comes after mid
               low = mid + 1
          else :            # item is at mid
               assert compar == 0
               low = mid
               high = mid + 1
     if low < high and TO(item, sequence[low]) == 0:   c3
      c4  return low
     else :
      c5  return -1
```

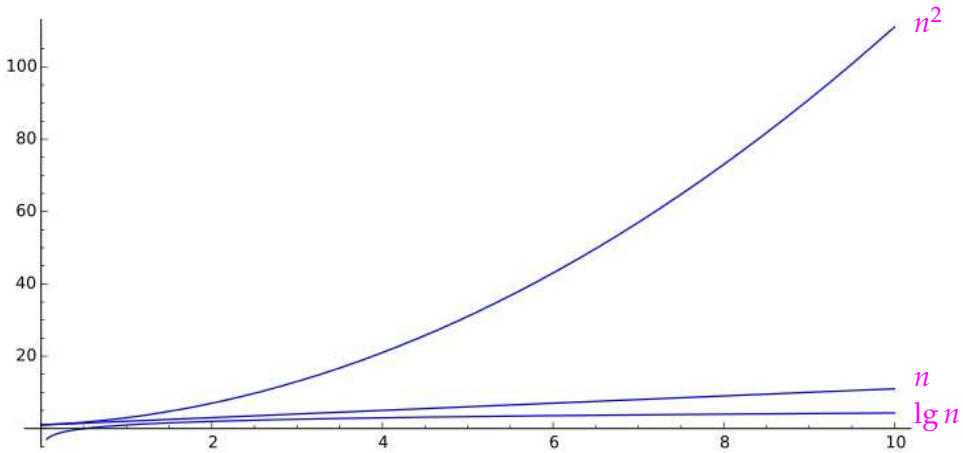
```
def selection_sort(sequence, TO):
    for i in range(len(sequence)):
        min_pos = i
        min = sequence[i]
        for j in range(i + 1, len(sequence)):
            if TO(sequence[j], min) < 0 :
                min = sequence[j]
                min_pos = j
        sequence[min_pos] = sequence[i]
        sequence[i] = min
```

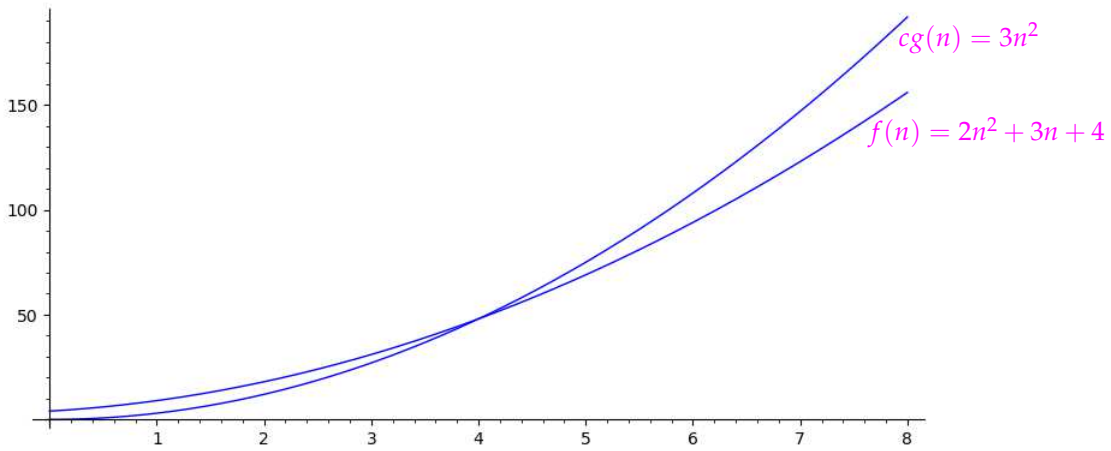
$e_0 + e_1 n$

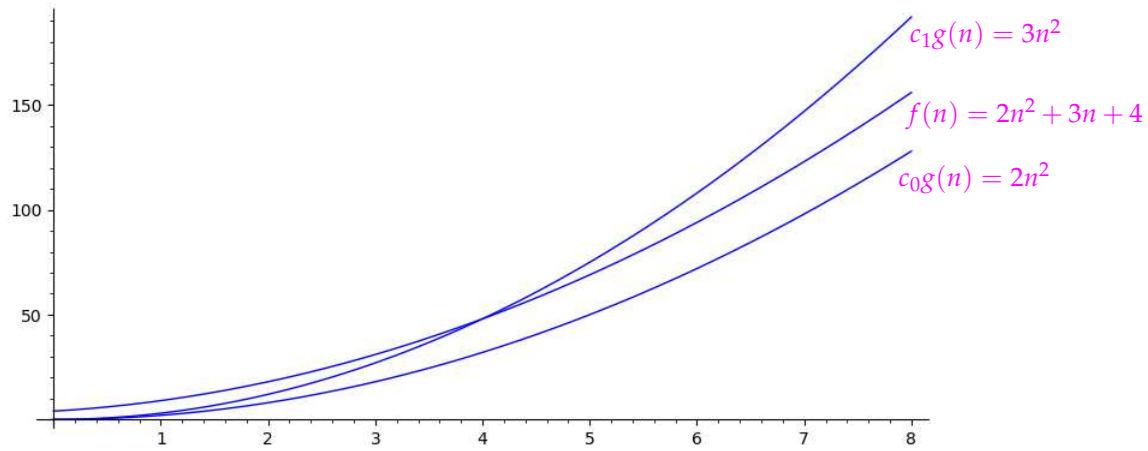
$e_2 n$

$e_3 n + e_4 \sum_{i=0}^{n-1} (n - i - 1)$

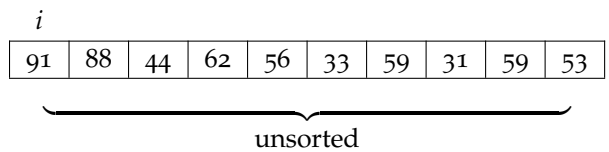
$e_5 \sum_{i=0}^{n-1} (n - i - 1)$

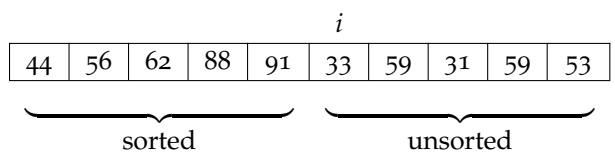


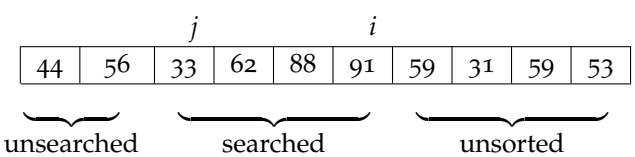


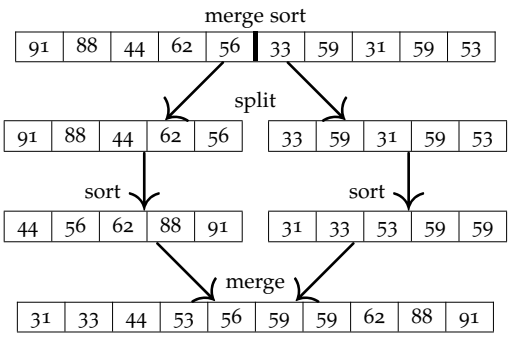


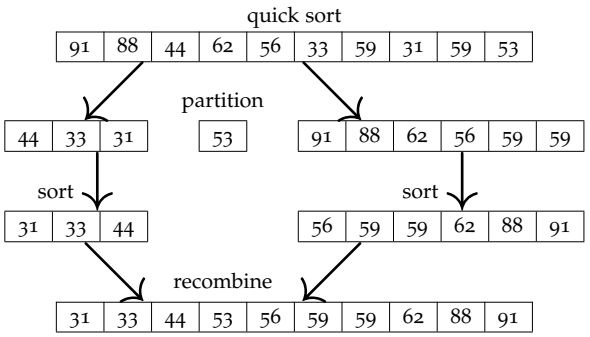

```
def binary_search_recursive(sequence, TO, item, start, stop):
    if start == stop:                                          c0
     c1  return -1
    else :
        mid = (stop + start) / 2
        compar = TO(item, sequence[mid])                       c2
        if compar < 0 :
            return binary_search_recursive(sequence, TO, item, start, mid)
        elif compar > 0 :
            return binary_search_recursive(sequence, TO, item, mid + 1, stop)
        else :
            return mid
```

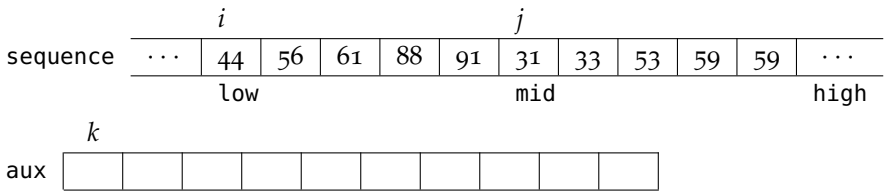




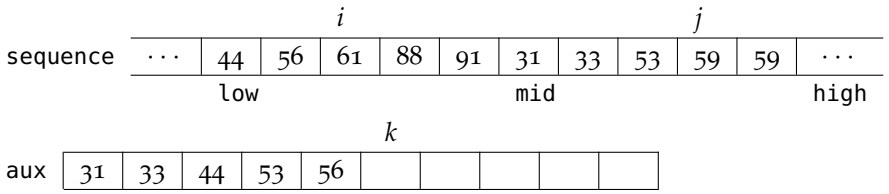




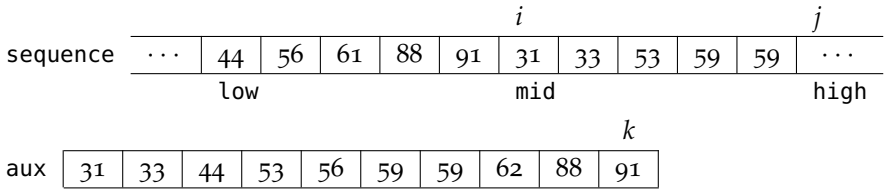


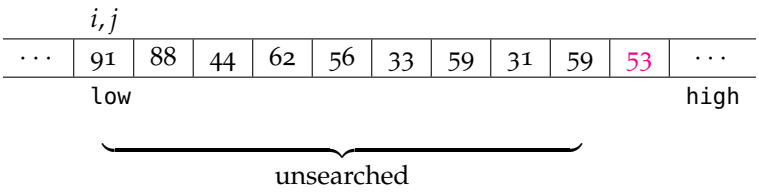


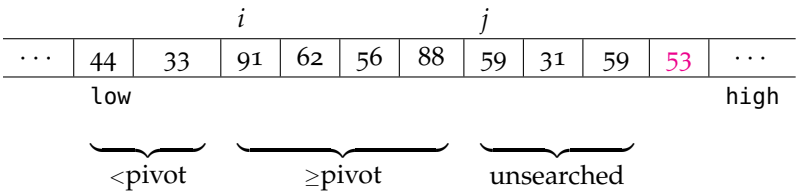
halfway through:

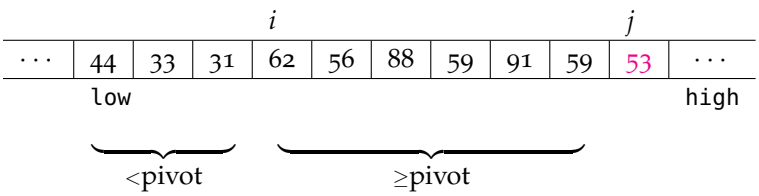


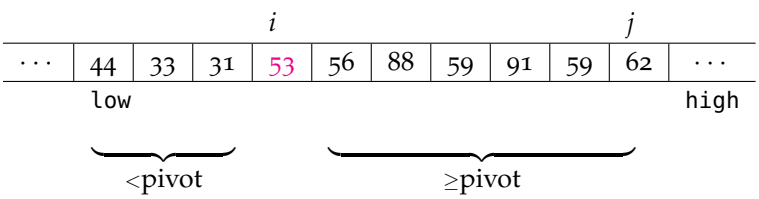
and when finished:

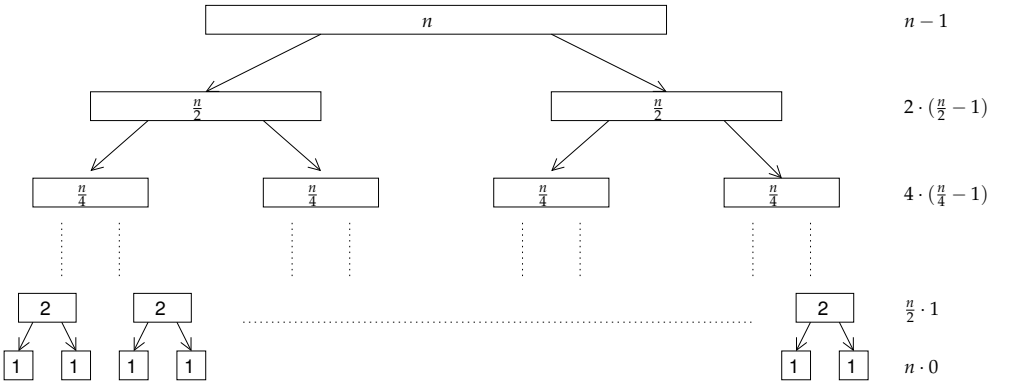


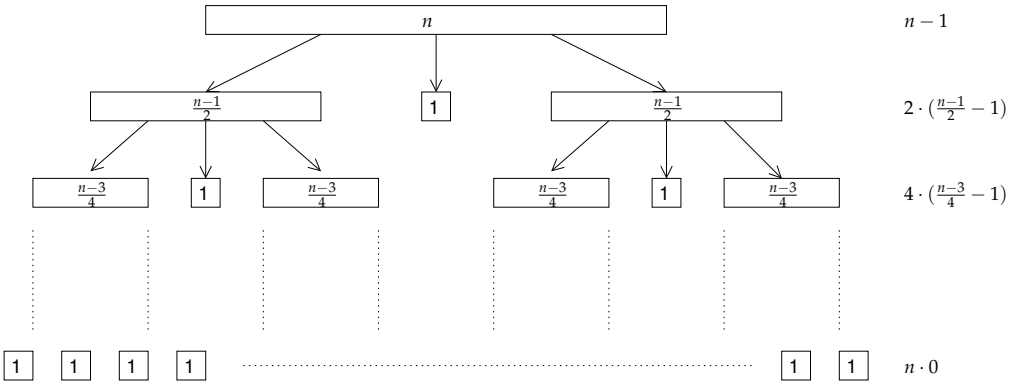

i, j
··· 91 88 44 62 56 33 59 31 59 53 ···
low
high
unsearched

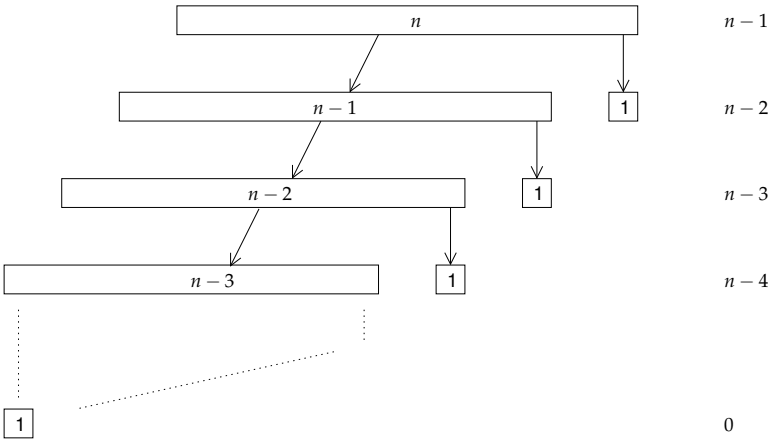

i
j
··· 44 33 91 62 56 88 59 31 59 53 ···
low
high
<pivot
≥pivot
unsearched


i
j
··· 44 33 31 62 56 88 59 91 59 53 ···
low
high
<pivot
≥pivot


i
j
··· 44 33 31 53 56 88 59 91 59 62 ···
low
high
<pivot
≥pivot


n
$\frac{n}{2}$
$\frac{n}{2}$
$\frac{n}{4}$
$\frac{n}{4}$
$\frac{n}{4}$
$\frac{n}{4}$
2
2
2
1
1
1
1
1
1
$n-1$
$2\cdot(\frac{n}{2}-1)$
$4\cdot(\frac{n}{4}-1)$
$\frac{n}{2}\cdot 1$
$n\cdot 0$


n
$\frac{n-1}{2}$
1
$\frac{n-1}{2}$
$\frac{n-3}{4}$
1
$\frac{n-3}{4}$
$\frac{n-3}{4}$
1
$\frac{n-3}{4}$
1
1
1
1
1
1
$n-1$
$2\cdot(\frac{n-1}{2}-1)$
$4\cdot(\frac{n-3}{4}-1)$
$n\cdot 0$


n
$n-1$
1
$n-2$
1
$n-3$
1
1
$n-1$
$n-2$
$n-3$
$n-4$
0

49	7	83	22	8	45	72	91	22	80	53	88	43	29	14	35	55	24	37	84

49	7	83	22	8	45	72	91	22	80	53	88	43	29	14	35	55	24	37	84

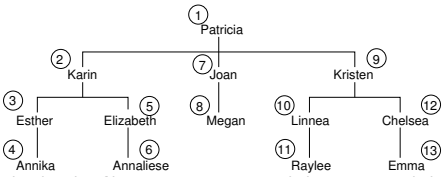
14	7	83	22	8	45	72	49	22	80	53	88	43	29	91	35	55	24	37	84

14	7	83	22	8	45	72	49	22	80	53	88	43	29	91	35	55	24	37	84

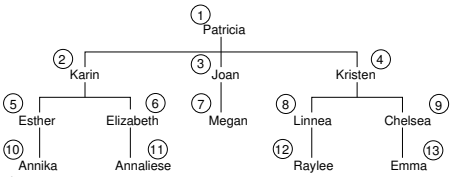
14	7	83	22	8	45	72	49	22	80	53	88	43	29	91	35	55	24	37	84

14	7	55	22	8	45	72	49	22	80	53	88	43	29	91	35	83	24	37	84

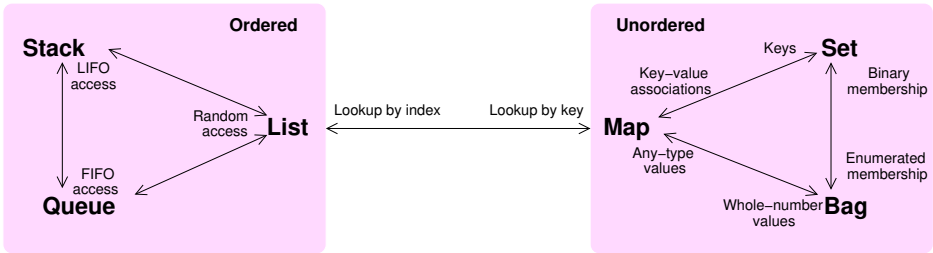
2
Data structures

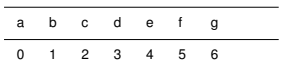



```
Who is the first ancestor?  Patricia Does Patricia have any children?  (y/n) y
Who is the next child?  Karin Does Karin have any children?  (y/n) y
Who is the next child?  Esther Does Esther have any children?  (y/n) y
Who is the next child?  Annika Does Annika have any children?  (y/n) n
Does Esther have any more children?  (y/n) n
Does Karin have any more children?  (y/n) y Who is the next child?  Elizabeth
Does Elizabeth have any children?  (y/n) y Who is the next child?  Annie
Does Annie have any children?  (y/n) n
Does Elizabeth have any more children?  (y/n) n
Does Karin have any more children?  (y/n) n
Does Patricia have any more children?  (y/n) y Who is the next child?  Joan
Does Joan have any children?  (y/n) y Who is the next child?  Megan
Does Megan have any children?  (y/n) n
Does Joan have any more children?  (y/n) n
Does Patricia have any more children?  (y/n) y Who is the next child?  Kristen
Does Kristen have any children?  (y/n) y Who is the next child?  Linnea
Does Linnea have any children?  (y/n) y Who is the next child?  Raylee
Does Raylee have any children?  (y/n) n
Does Linnea have any more children?  (y/n) n
Does Kristen have any more children?  (y/n) y Who is the next child?  Chelsea
Does Chelsea have any children?  (y/n) y Who is the next child?  Emma
Does Emma have any children?  (y/n) n
Does Chelsea have any more children?  (y/n) n
Does Kristen have any more children?  (y/n) n
Does Patricia have any more children?  (y/n) n
```

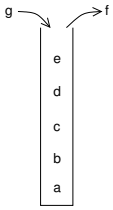



```
Who is the first ancestor?  Patricia
How many children does Patricia have?  3
Who is Patricia's next child?  Karin
Who is Patricia's next child?  Joan
Who is Patricia's next child?  Kristen
How many children does Karin have?  2
Who is Karin's next child?  Esther
Who is Karin's next child?  Elizabeth
How many children does Joan have?  1
Who is Joan's next child?  Megan
How many children does Kristen have?  2
Who is Kristen's next child?  Linnea
Who is Kristen's next child?  Chelsea
How many children does Esther have?  1
Who is Esther's next child?  Annika
How many children does Elizabeth have?  1
Who is Elizabeth's next child?  Annie
How many children does Megan have?  0
How many children does Linnea have?  1
Who is Linnea's next child?  Raylee
How many children does Chelsea have?  1
Who is Chelsea's next child?  Emma
How many children does Annika have?  0
How many children does Annie have?  0
How many children does Raylee have?  0
How many children does Emma have?  0
```

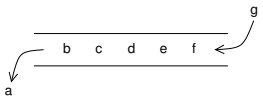




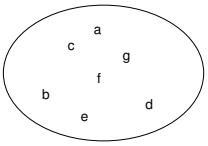
List



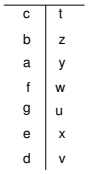
Stack



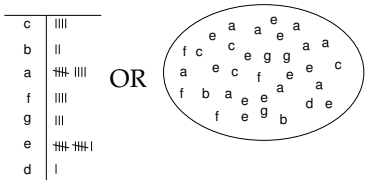
Queue



Set

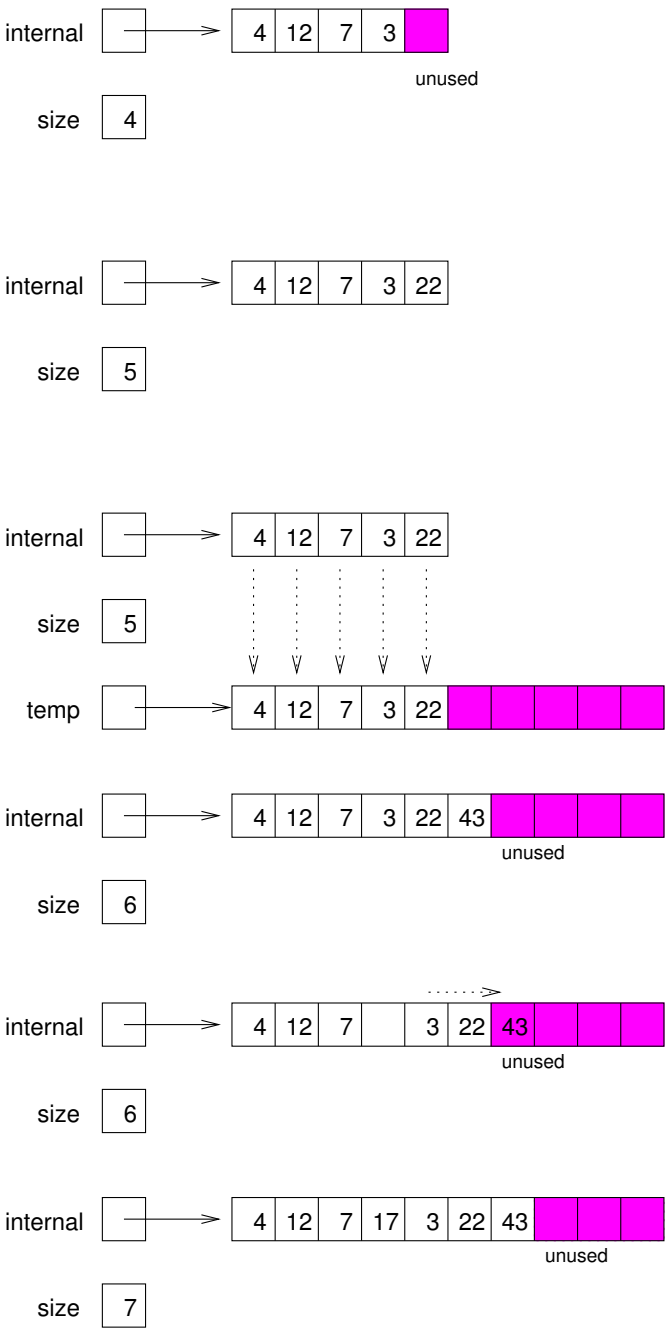


Map

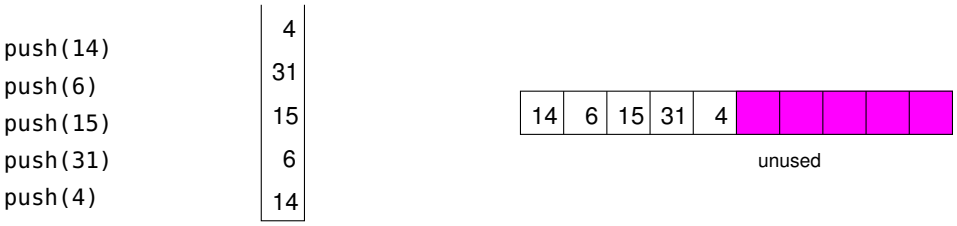


Bag

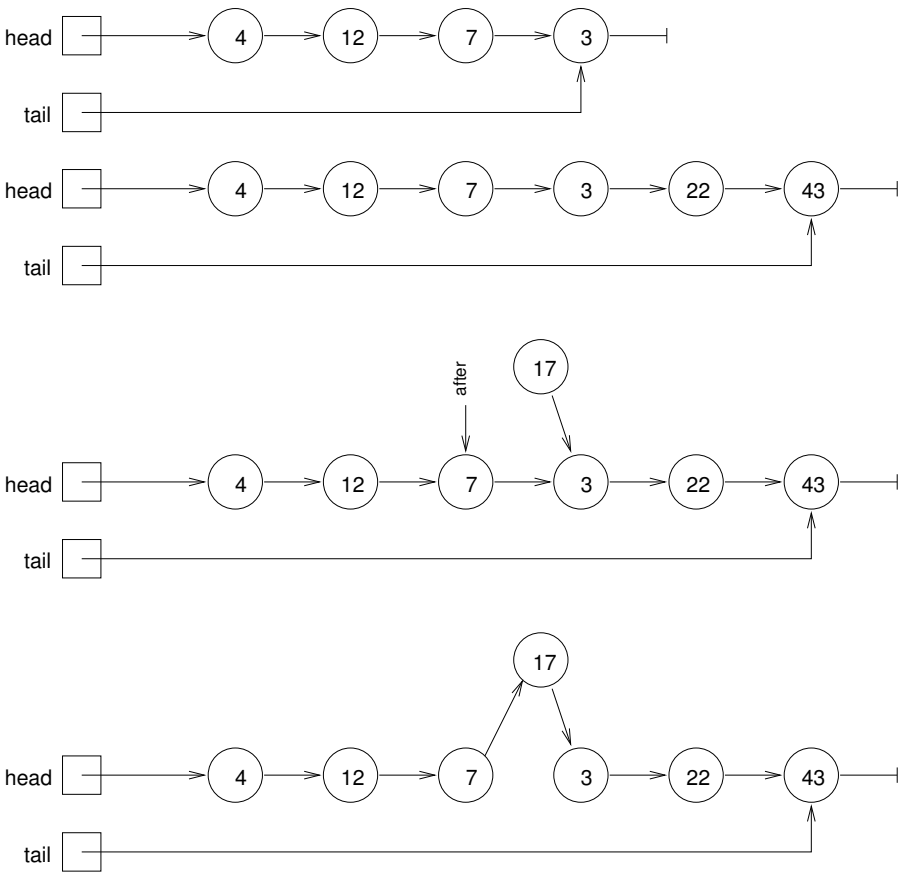
2.1 *Array-based data structures*



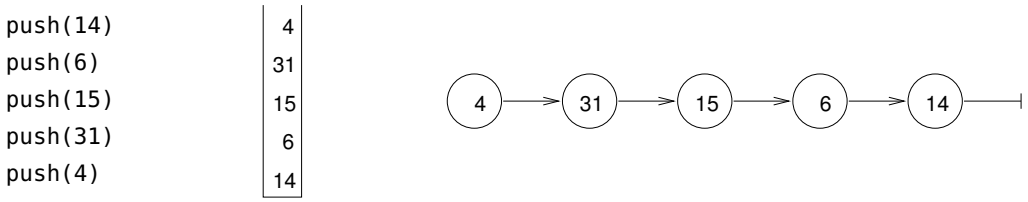
Call to `add()`	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Array length	1	2	4	4	8	8	8	8	16	16	16	16	16	16	16	16	32
Writes performed	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17
Running total	1	3	6	7	12	13	14	15	24	25	26	27	28	29	30	31	48



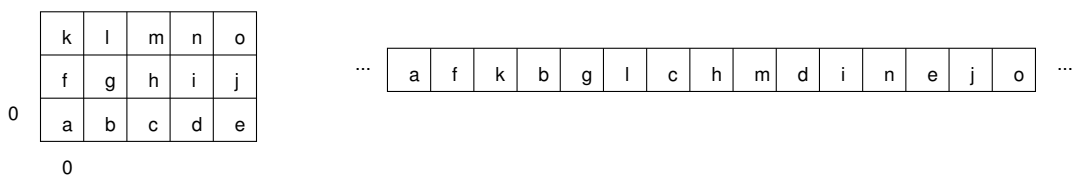
2.2 *Linked data structures*

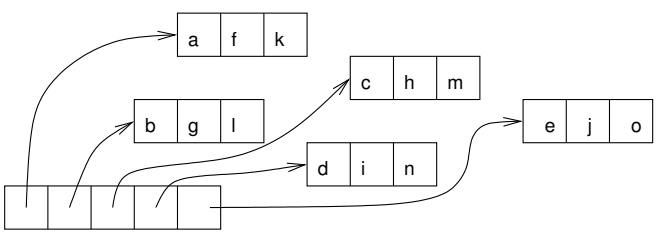


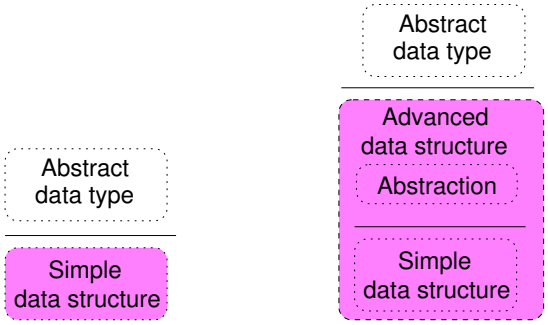
	`ArrayList`	`LinkedList`
`add()`	$\Theta(1)$ (amortized)	$\Theta(1)$
`set()`	$\Theta(1)$	$\Theta(n)$ (worst and expected)
`get()`	$\Theta(1)$	$\Theta(n)$ (worst and expected)
`remove()`	$\Theta(n)$ (worst and expected)	$\Theta(n)$ (worst and expected)
`insert()`	$\Theta(n)$ (worst and expected)	$\Theta(n)$ (worst and expected)



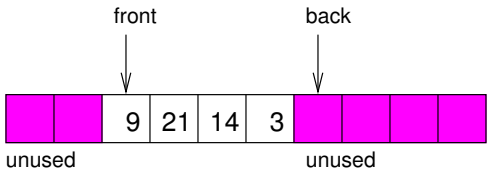
2.3 *Data structures built from abstractions*



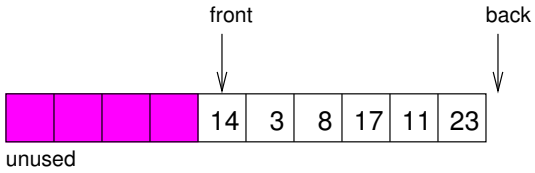




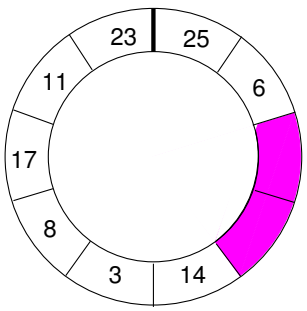

```
enqueue(5)      remove()
enqueue(12)     enqueue(14)
enqueue(9)      enqueue(3)
enqueue(21)     remove()
```

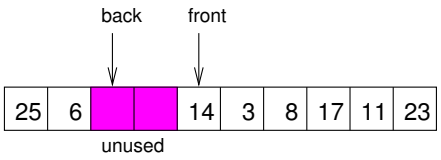


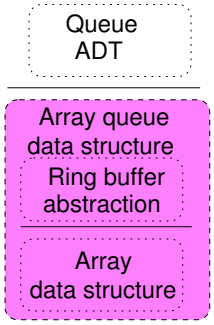

```
enqueue(8)      enqueue(11)
remove()        remove()
enqueue(17)     enqueue(23)
```

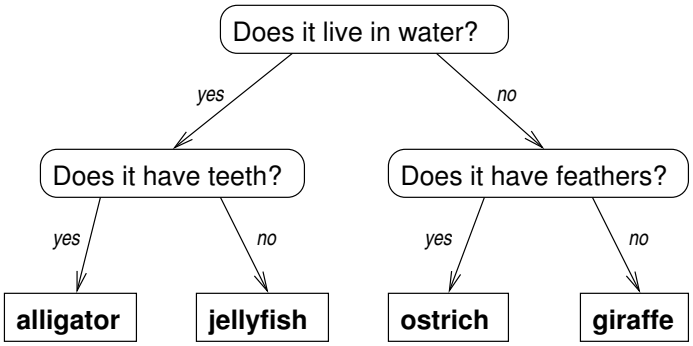



```
enqueue(25)
enqueue(6)
```





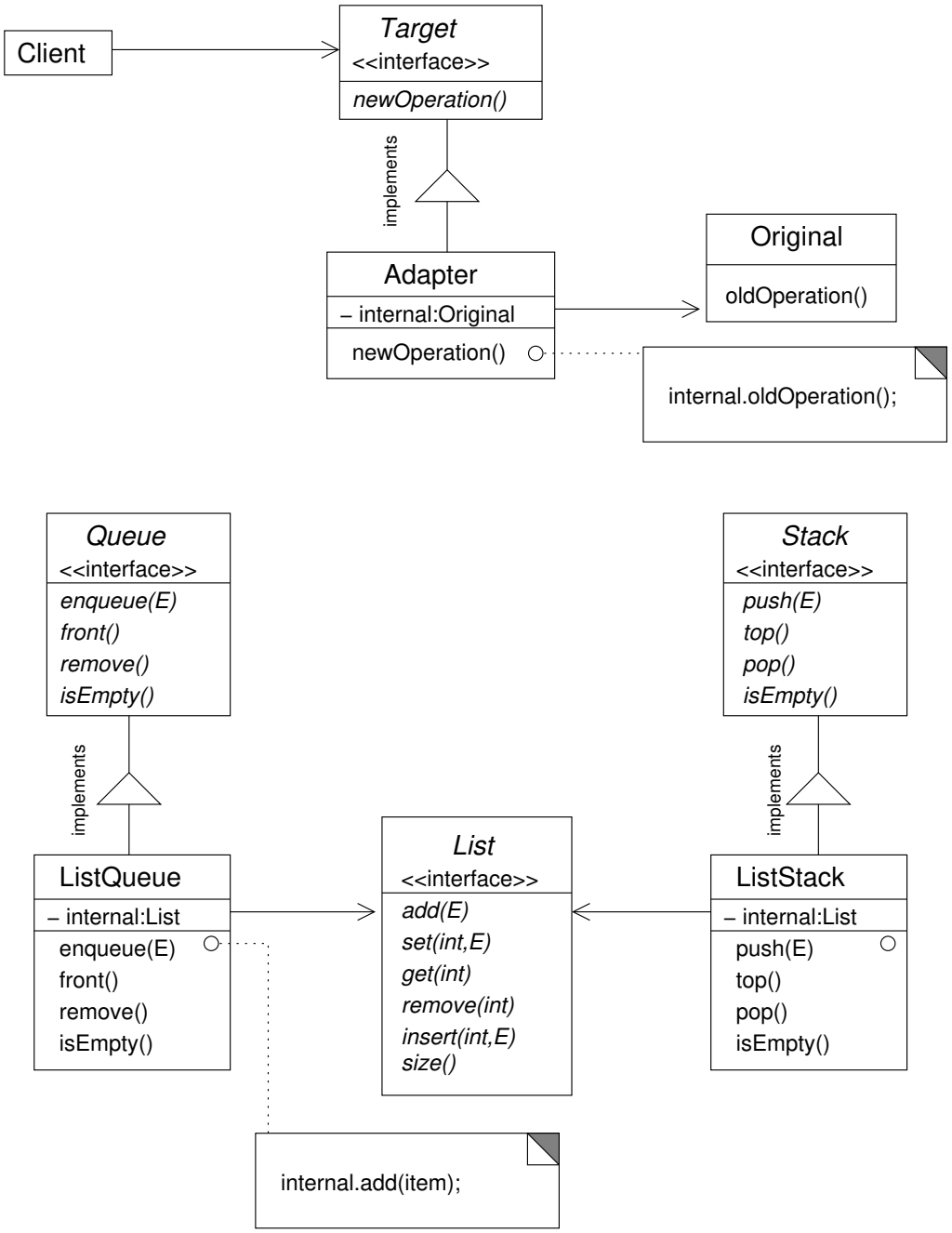


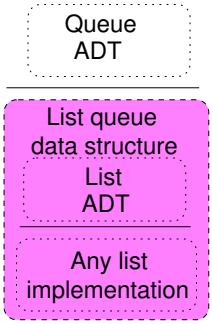


$$\begin{pmatrix} 0.0 & 0.0 & 5.0 & 0.0 & 9.0 \\ 1.0 & 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 6.0 & 7.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 8.0 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

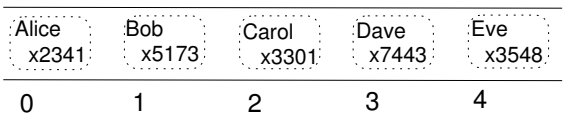
`entries`	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0
`rows`	1	4	1	2	0	2	2	3	0
`colStarts`	0	2	4	7	8				

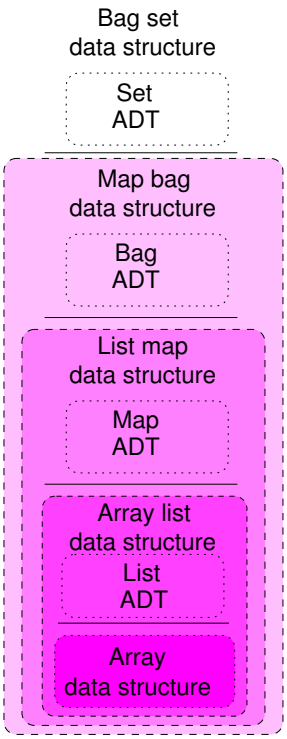
2.4 *Data structures adapted from ADTs*



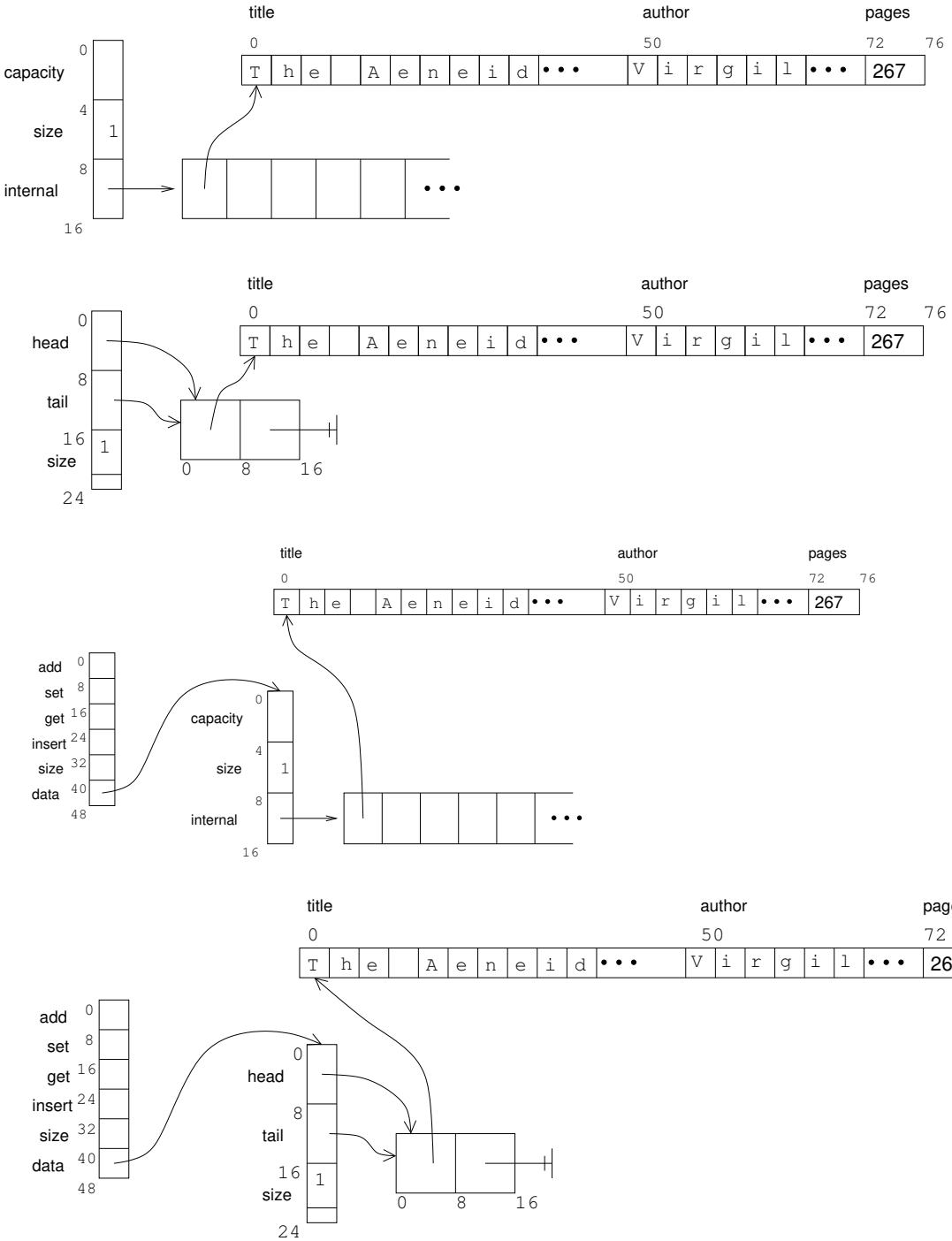


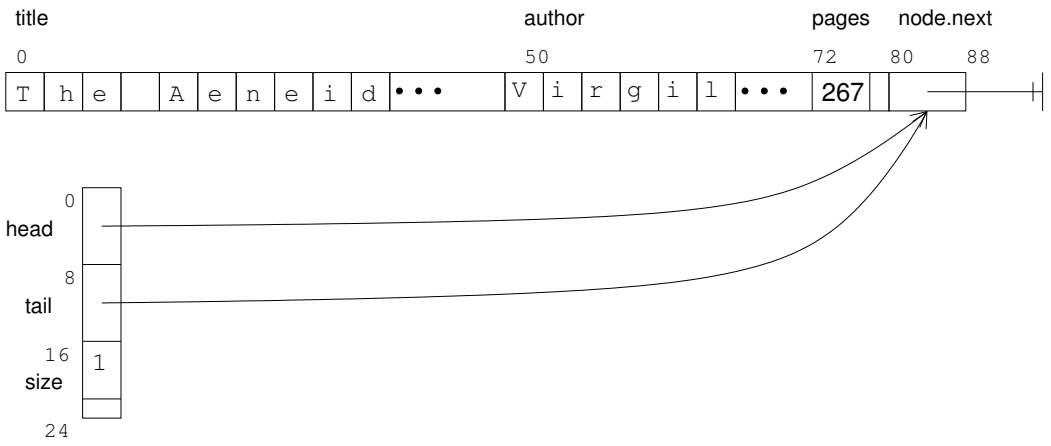
Alice	x2341
Bob	x5173
Carol	x3301
Dave	x7443
Eve	x3548

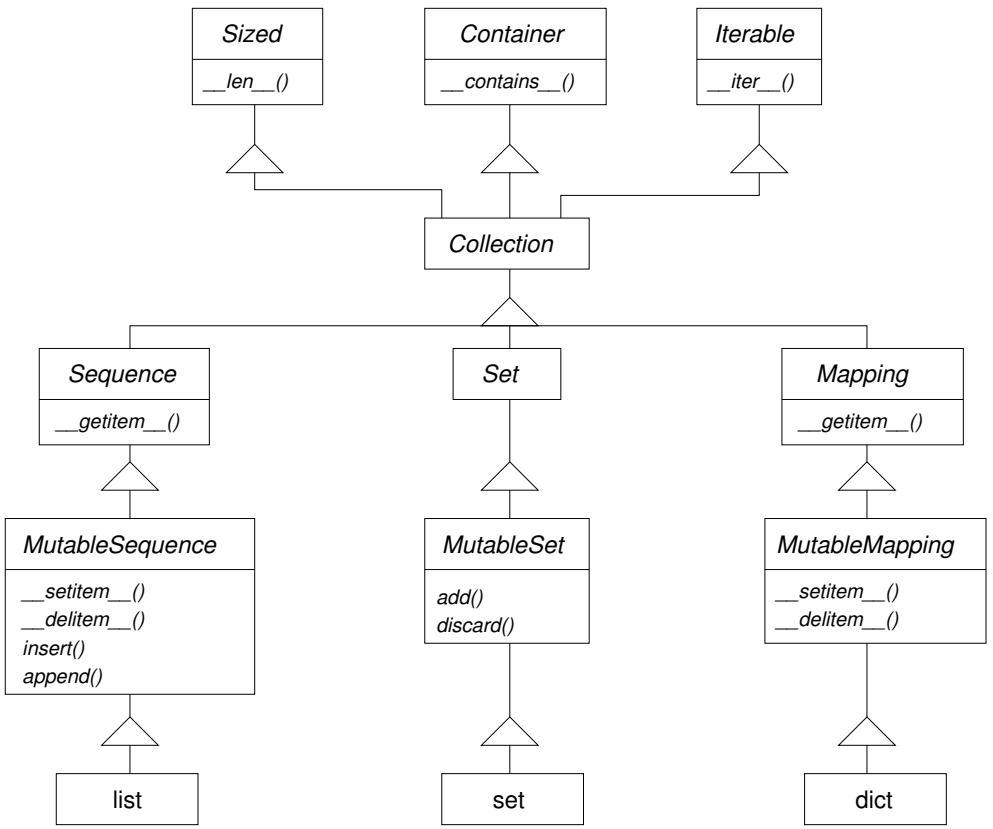




2.5 *ADTs and data structures in other languages*

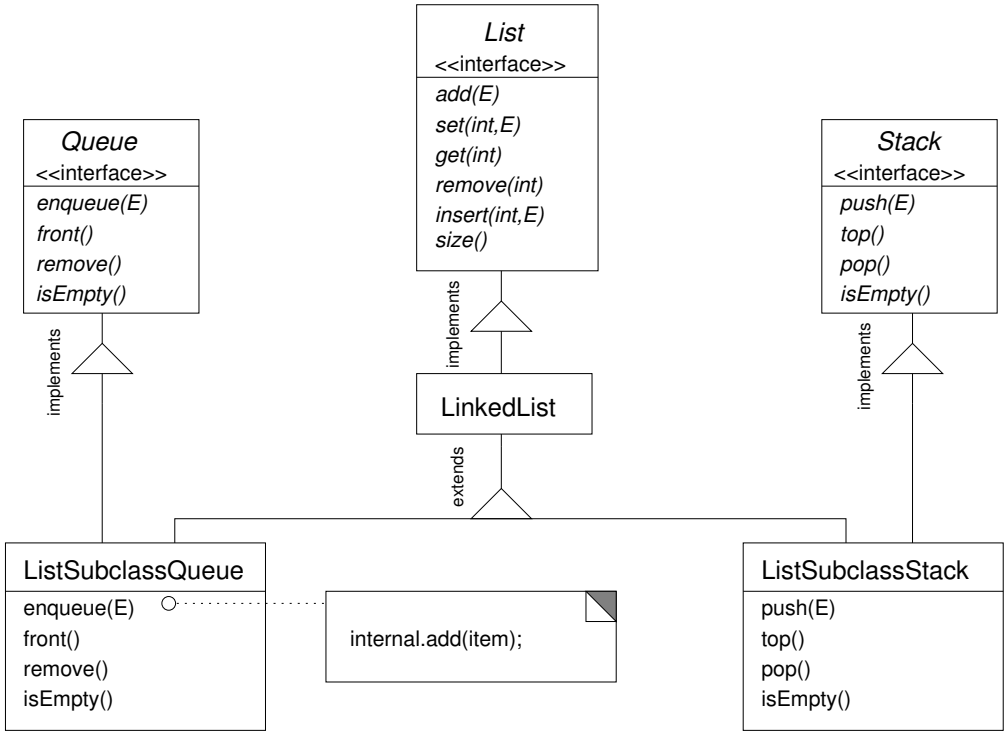



title
author
pages
node.next
0
50
72
80
88
T h e A e n e i d • • •
V i r g i l • • •
267
0
head
8
tail
16
size
1
24


Sized
__len__()
Container
__contains__()
Iterable
__iter__()
Collection
Sequence
__getitem__()
Set
Mapping
__getitem__()
MutableSequence
__setitem__()
__delitem__()
insert()
append()
MutableSet
add()
discard()
MutableMapping
__setitem__()
__delitem__()
list
set
dict

2.6 Programming practices

	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16	17
2	18	19	20	21	22	23	24	25	26
3	27	28	29	30	31	32	33	34	35
4	36	37	38	39	40	41	42	43	44
5	45	46	47	48	49	50	51	52	53
6	54	55	56	57	58	59	60	61	62
7	63	64	65	66	67	68	69	70	71
8	72	73	74	75	76	77	78	79	80



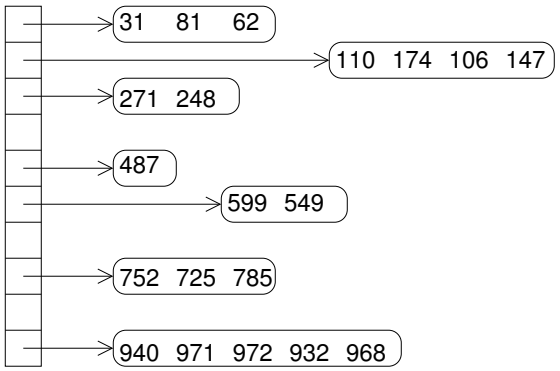
2.7 The road ahead

2.8 Chapter summary

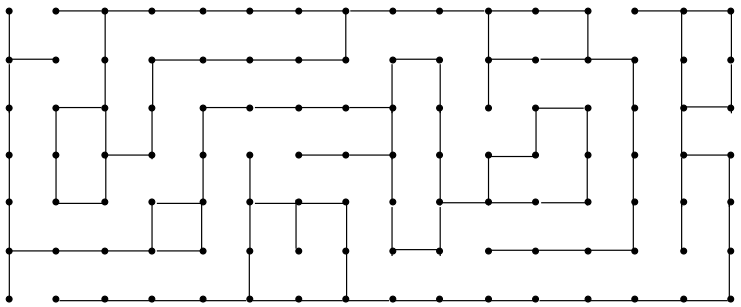
3
Case studies

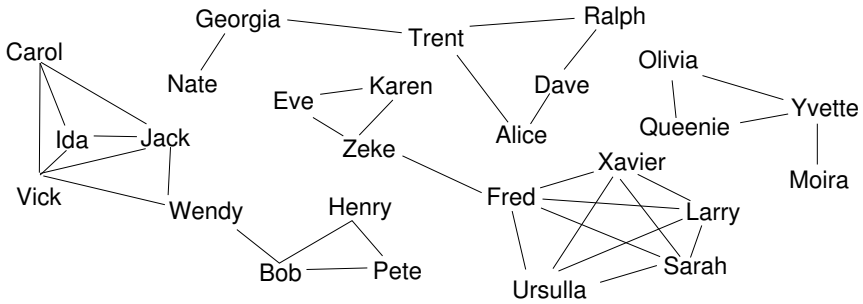
3.1 Linear-time sorting algorithms

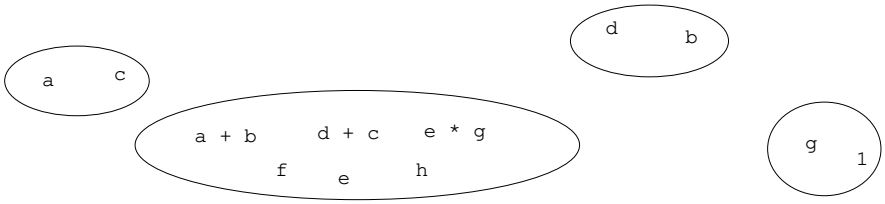
Alice	Bob	Carol	Dave	Eve	Fred	Georgia	Henry	Ida	Jack	Karen	...
2	4	2	1	1	3	4	0	2	3	0	



3.2 Disjoint sets and array forests







Initial: 0 1 2 3 4 5 6 7

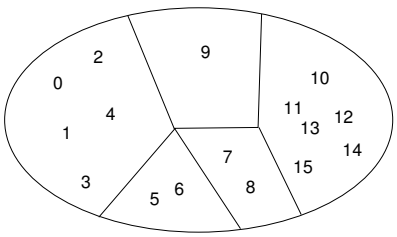
`union(3, 5)` 0 1 2 3 5 4 6 7

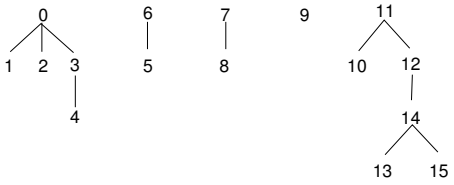
`union(1, 7)` 0 1 7 2 3 5 4 6

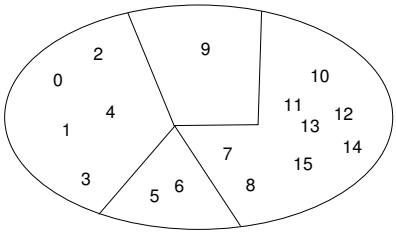
`union(2, 3)` 0 1 7 2 3 5 4 6

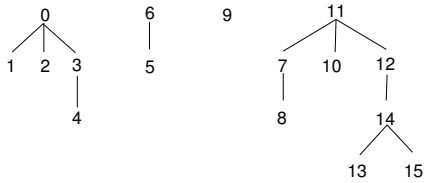
`union(0, 4)` 0 4 1 7 2 3 5 6

`union(0, 3)` 0 4 2 3 5 1 7 6

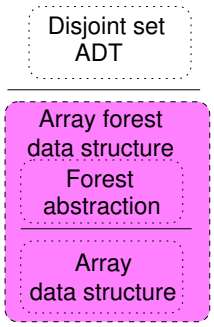


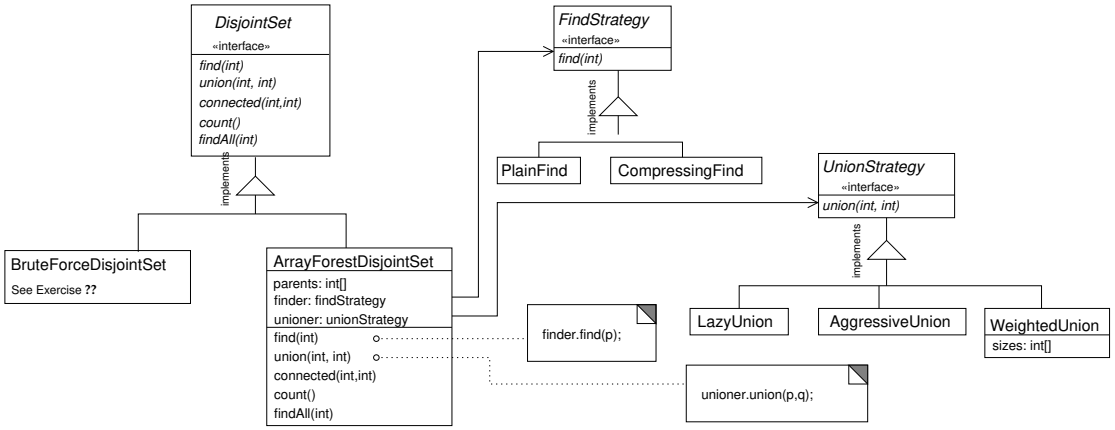






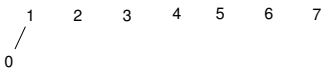
0	0	0	0	3	6	6	11	7	9	11	11	11	14	12	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



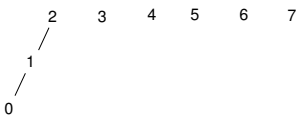


Initial state

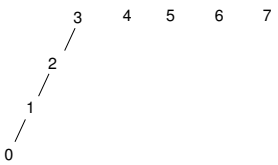
After `union(0, 1)`



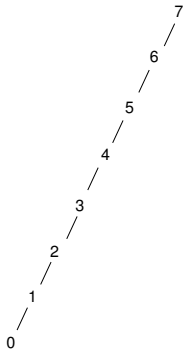
After `union(1, 2)`



After `union(2, 3)`



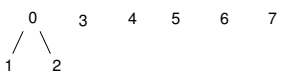
Final state



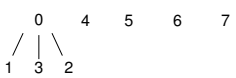
Initial state

After `union(0, 1)`

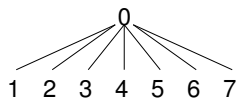
After `union(1, 2)`



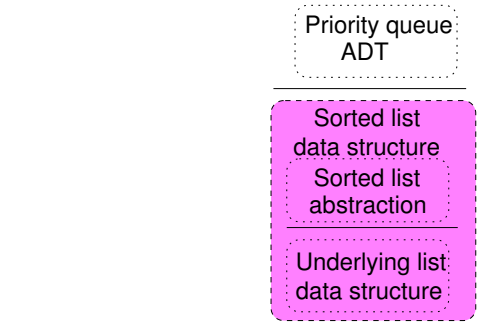
After `union(2, 3)`

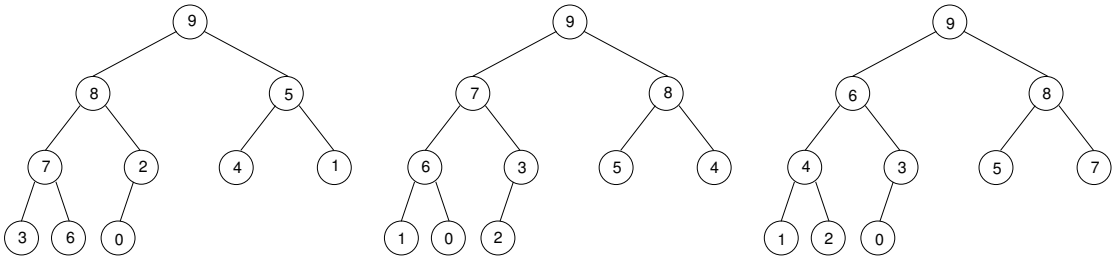


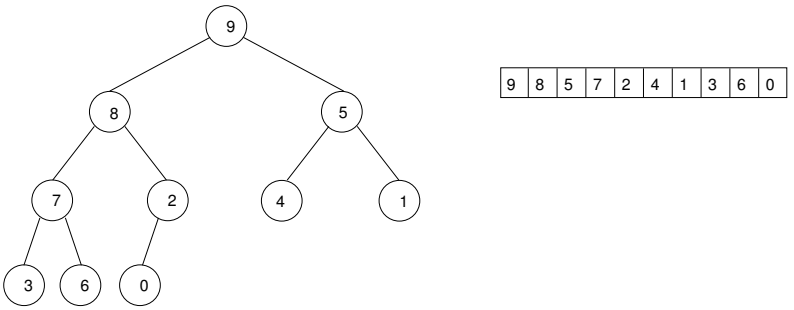
Final state

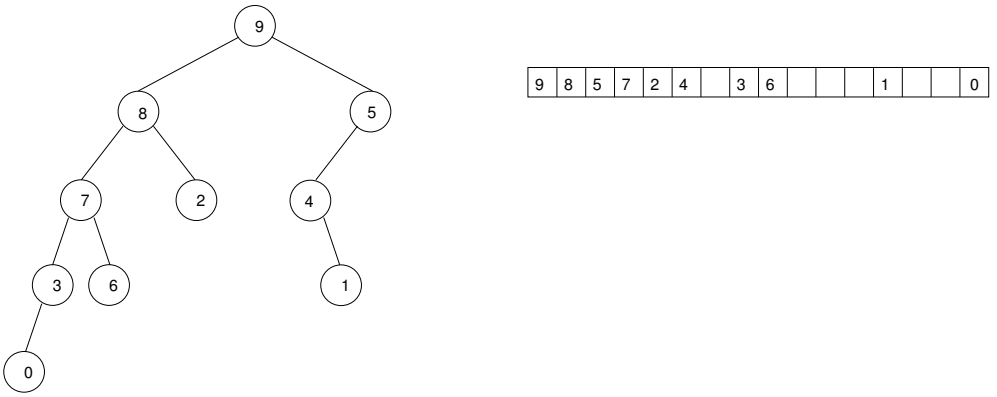


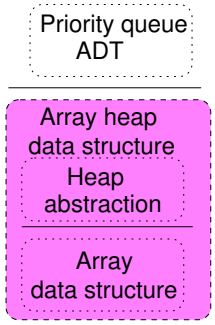
3.3 *Priority queues and heaps*

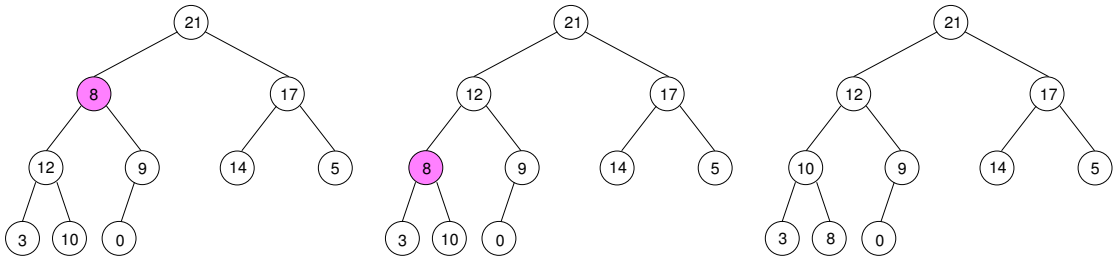


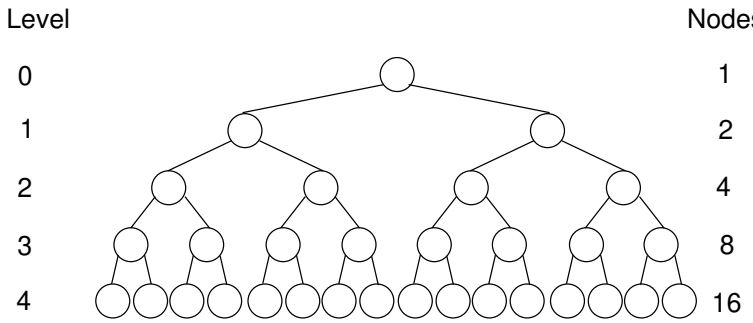


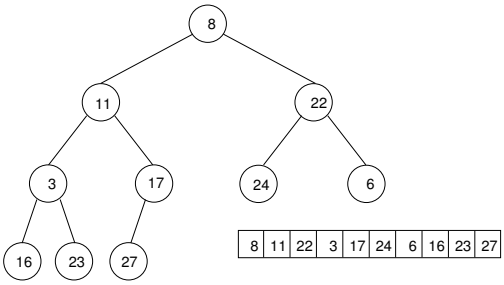


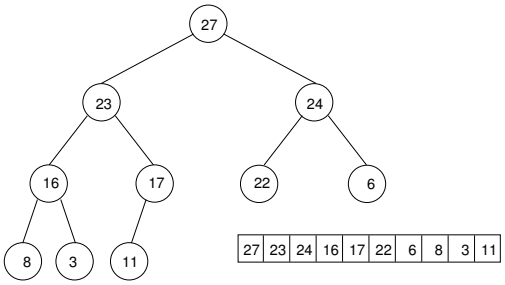


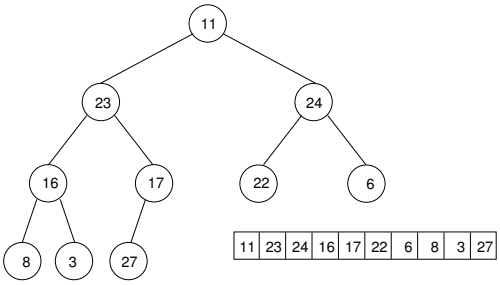

Priority queue ADT
Array heap data structure
Heap abstraction
Array data structure

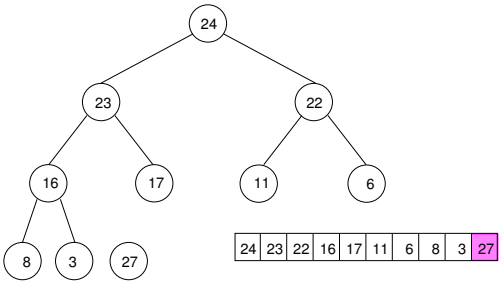

21
8
17
12
9
14
5
3
10
0
21
12
17
8
9
14
5
3
10
0
21
12
17
10
9
14
5
3
8
0

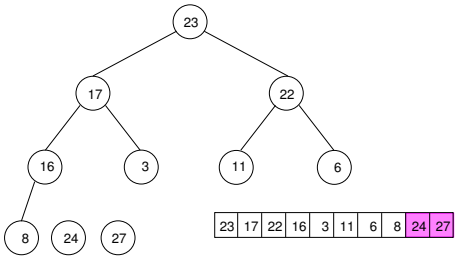

Level
Nodes
0
1
2
3
4
1
2
4
8
16

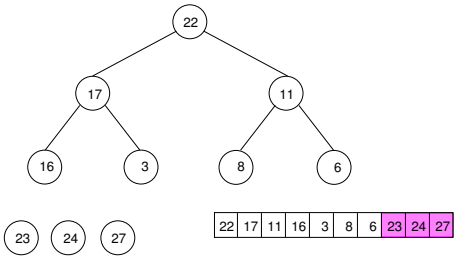

8
11
22
3
17
24
6
16
23
27
8 11 22 3 17 24 6 16 23 27

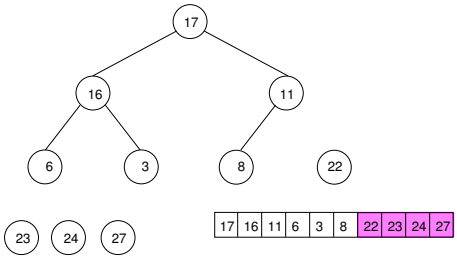

27
23
24
16
17
22
6
8
3
11
27 23 24 16 17 22 6 8 3 11

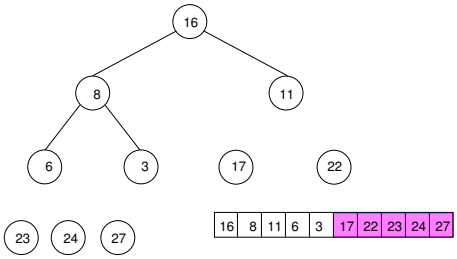

11
23
24
16
17
22
6
8
3
27
11 23 24 16 17 22 6 8 3 27

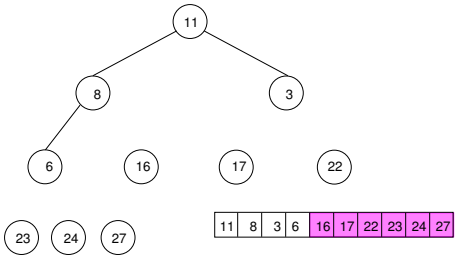

24
23
22
16
17
11
6
8
3
27
24 23 22 16 17 11 6 8 3 27

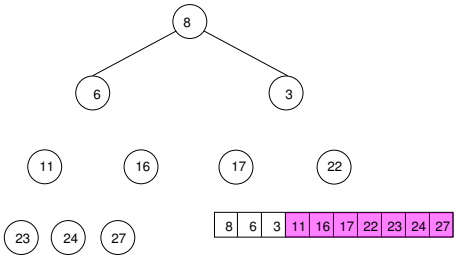

23
17
22
16
3
11
6
8
24
27
23 17 22 16 3 11 6 8 24 27

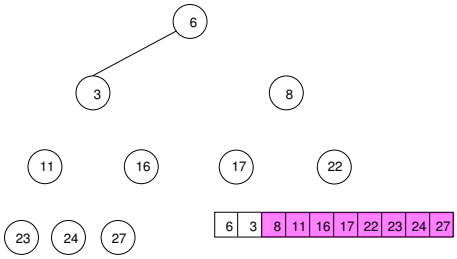

22
17
11
16
3
8
6
23
24
27
22 17 11 16 3 8 6 23 24 27

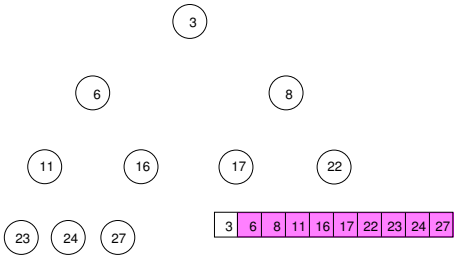

17
16
11
6
3
8
22
23
24
27
17 16 11 6 3 8 22 23 24 27

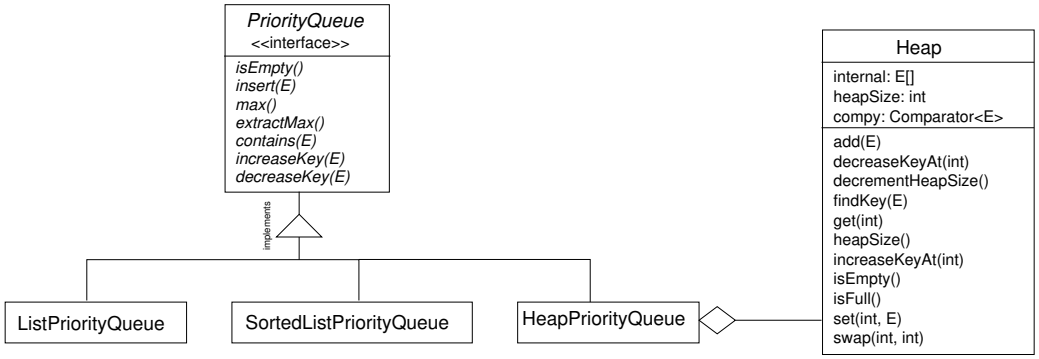

16
8
11
6
3
17
22
23
24
27
16 8 11 6 3 17 22 23 24 27

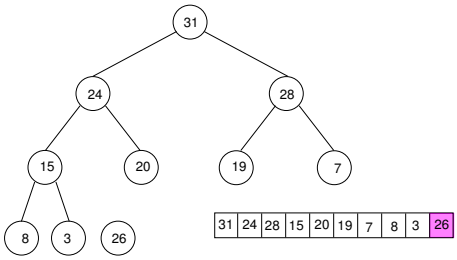

11
8
3
6
16
17
22
23
24
27
11 8 3 6 16 17 22 23 24 27

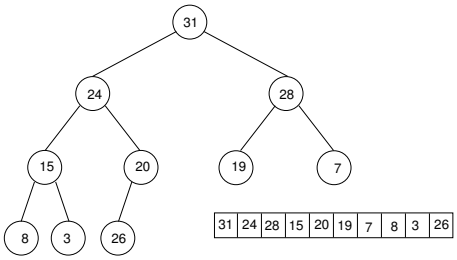

8
6
3
11
16
17
22
23
24
27
8 6 3 11 16 17 22 23 24 27

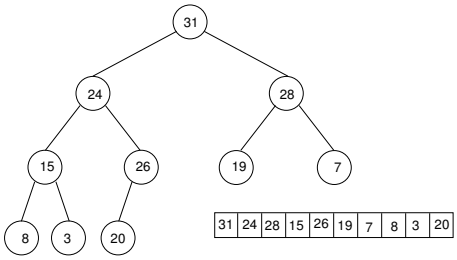

6
3
8
11
16
17
22
23
24
27
6 3 8 11 16 17 22 23 24 27

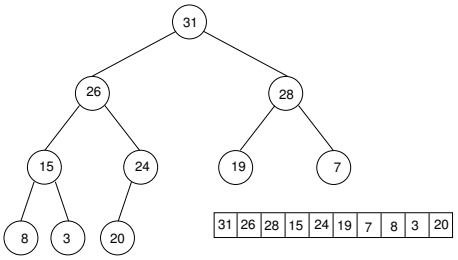

3
6
8
11
16
17
22
23
24
27
3 6 8 11 16 17 22 23 24 27

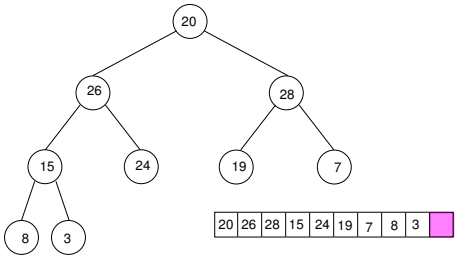

PriorityQueue
<<interface>>
isEmpty()
insert(E)
max()
extractMax()
contains(E)
increaseKey(E)
decreaseKey(E)
implements
Heap
internal: E[]
heapSize: int
compy: Comparator<E>
add(E)
decreaseKeyAt(int)
decrementHeapSize()
findKey(E)
get(int)
heapSize()
increaseKeyAt(int)
isEmpty()
isFull()
set(int, E)
swap(int, int)
ListPriorityQueue
SortedListPriorityQueue
HeapPriorityQueue

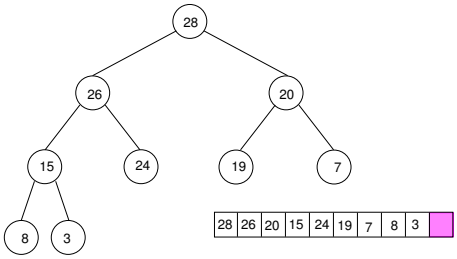

31
24
28
15
20
19
7
8
3
26
31 24 28 15 20 19 7 8 3 26


31
24
28
15
20
19
7
8
3
26
31 24 28 15 20 19 7 8 3 26


31
24
28
15
26
19
7
8
3
20
31 24 28 15 26 19 7 8 3 20

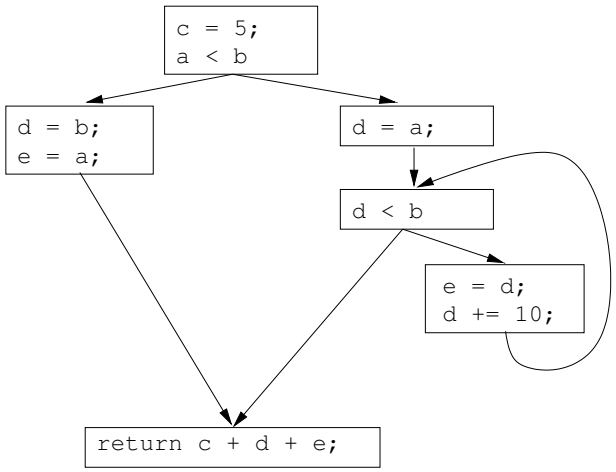

31
26
28
15
24
19
7
8
3
20
31 26 28 15 24 19 7 8 3 20

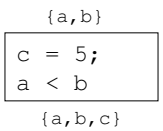

20
26
28
15
24
19
7
8
3
20 26 28 15 24 19 7 8 3

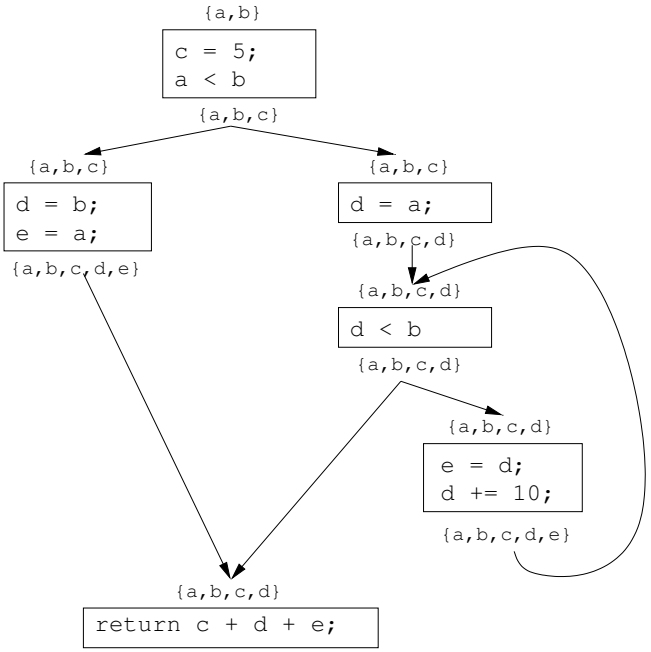

28
26
20
15
24
19
7
8
3
28 26 20 15 24 19 7 8 3

3.4 *N-Sets and bit vectors*

```
static int m(int a, int b) {
    int c, d, e;
    c = 5;
    if (a < b) {
        d = b;
        e = a;
    }
    else {
        d = a;
        while (d < b) {
            e = d;
            d += 10;
        }
    }
    return c + d + e;
}
```

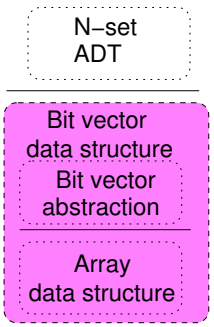




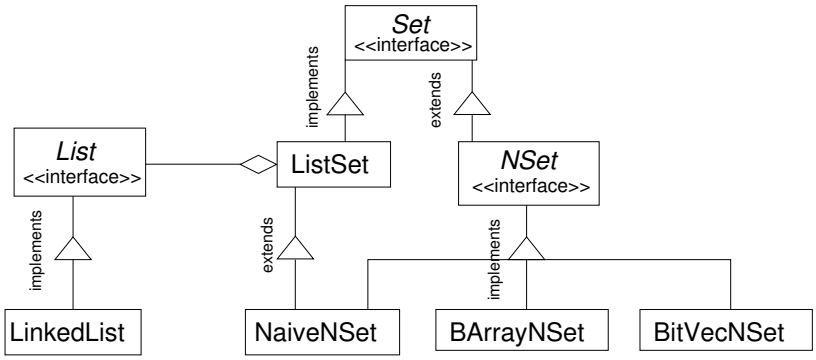


T	*T*	*F*	*T*	*F*	*T*	*T*	*T*

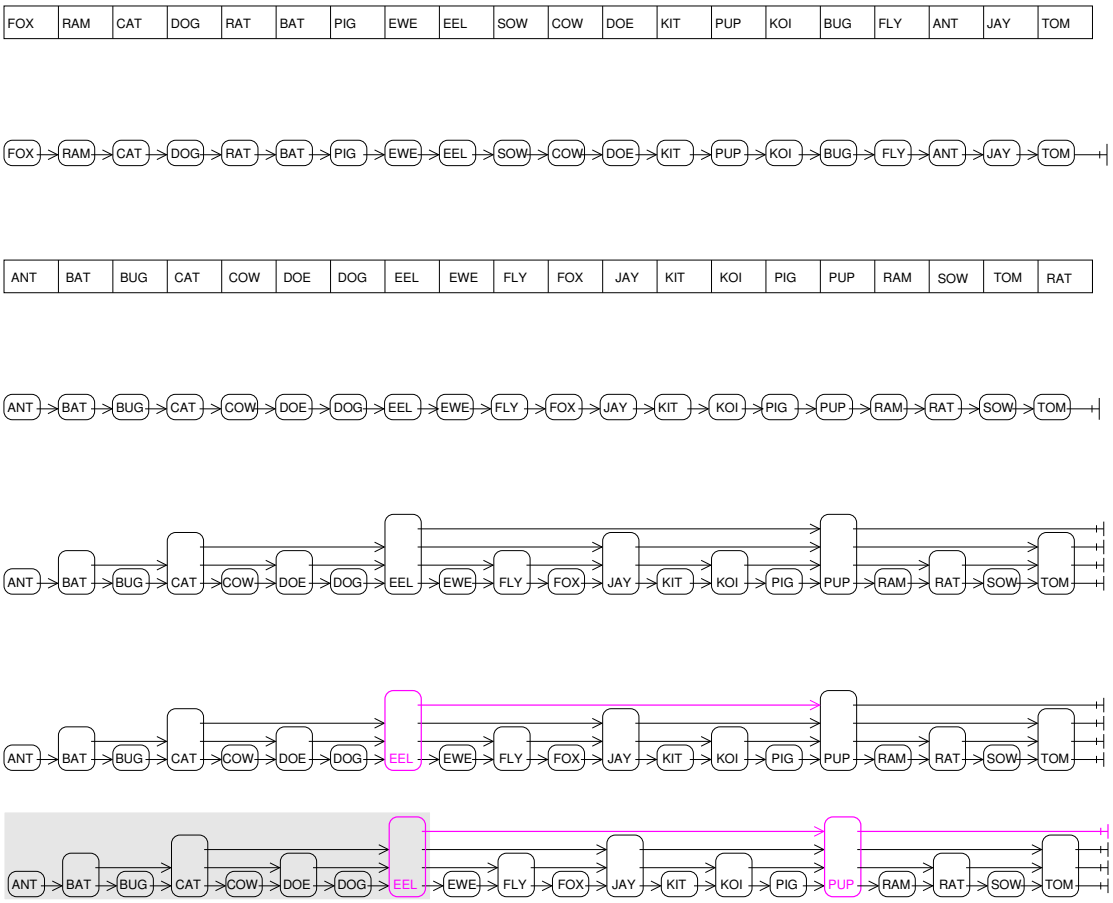
.

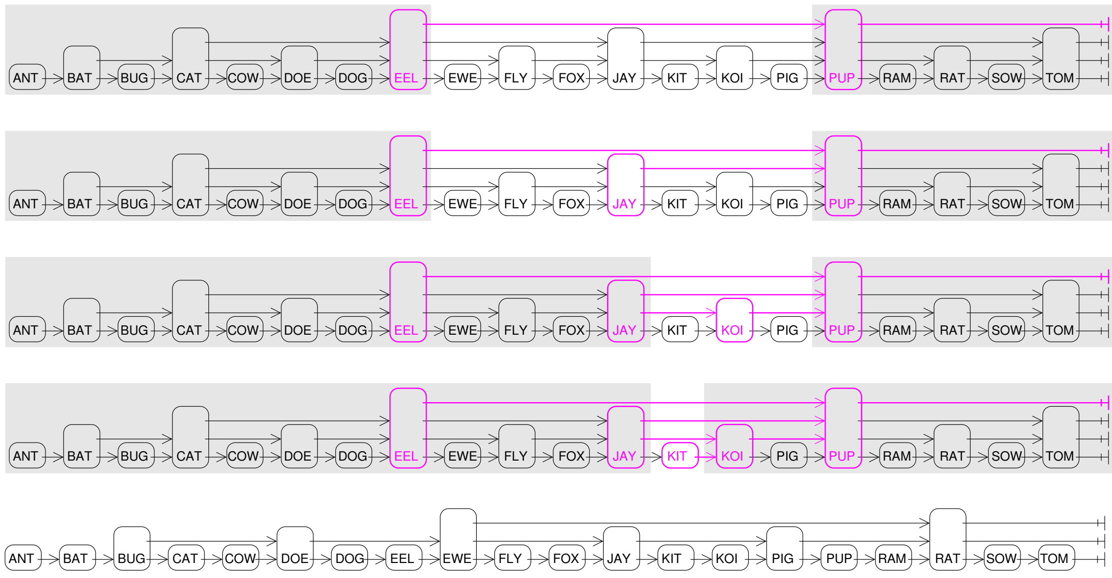


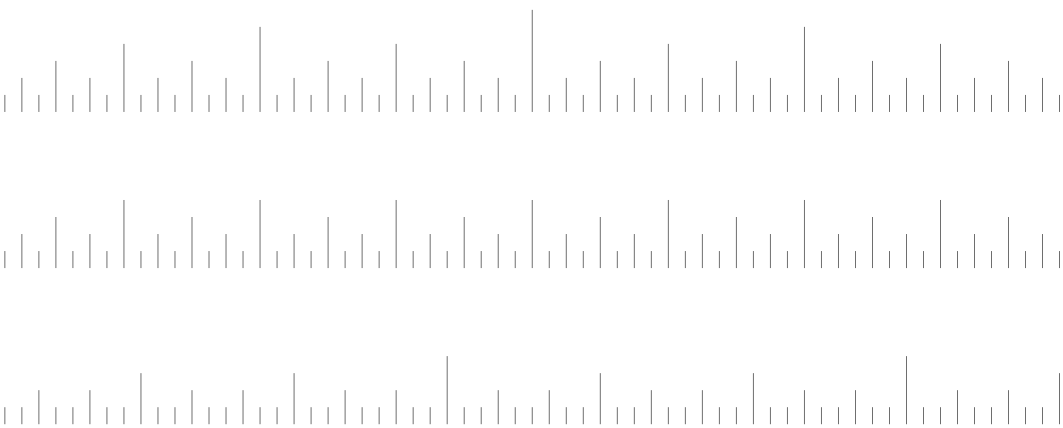
sparse	1	3		4			0	5					2		
dense	6	0	12	1	3	7									

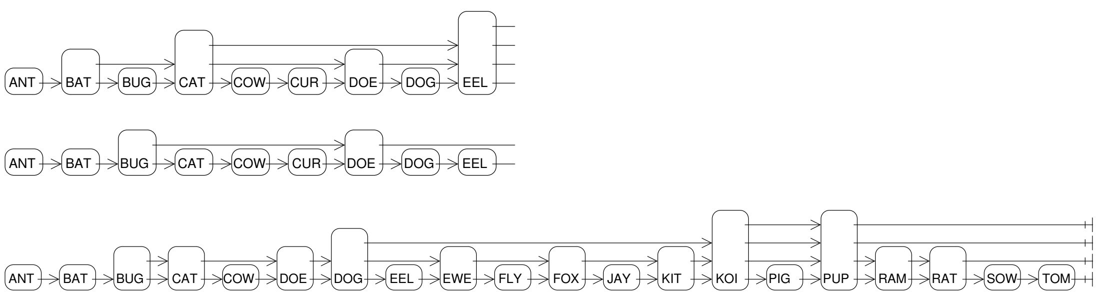


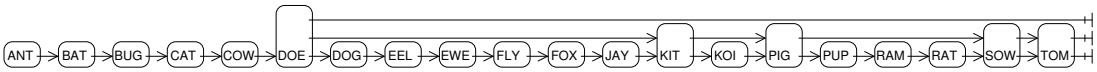
3.5 *Skip lists*

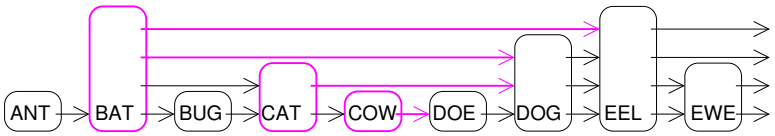


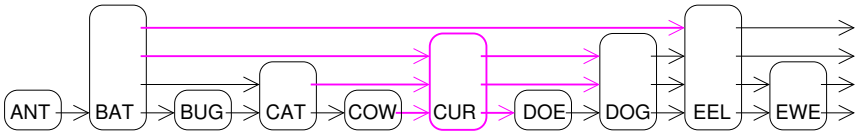

ANT BAT BUG CAT COW DOE DOG EEL EWE FLY FOX JAY KIT KOI PIG PUP RAM RAT SOW TOM
ANT BAT BUG CAT COW DOE DOG EEL EWE FLY FOX JAY KIT KOI PIG PUP RAM RAT SOW TOM
ANT BAT BUG CAT COW DOE DOG EEL EWE FLY FOX JAY KIT KOI PIG PUP RAM RAT SOW TOM
ANT BAT BUG CAT COW DOE DOG EEL EWE FLY FOX JAY KIT KOI PIG PUP RAM RAT SOW TOM
ANT BAT BUG CAT COW DOE DOG EEL EWE FLY FOX JAY KIT KOI PIG PUP RAM RAT SOW TOM

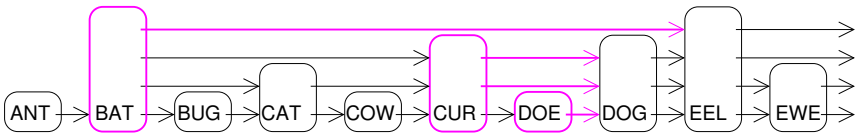


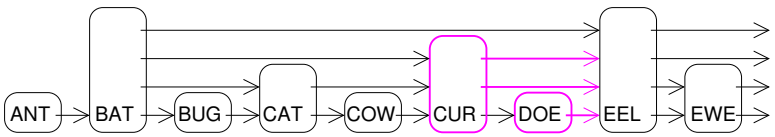

ANT BAT BUG CAT COW CUR DOE DOG EEL
ANT BAT BUG CAT COW CUR DOE DOG EEL
ANT BAT BUG CAT COW DOE DOG EEL EWE FLY FOX JAY KIT KOI PIG PUP RAM RAT SOW TOM









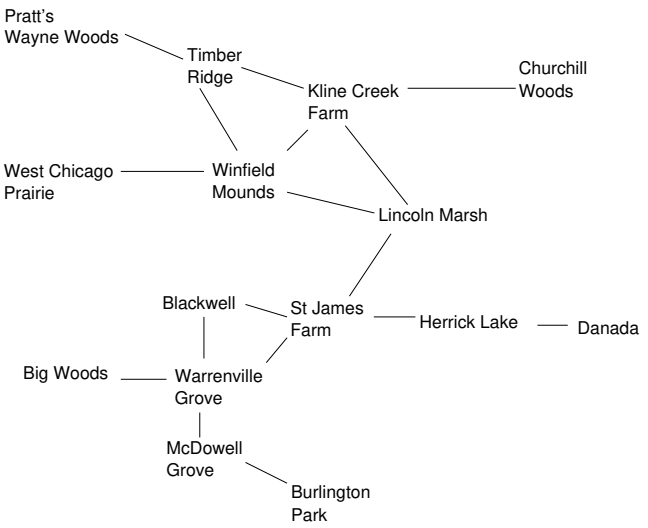


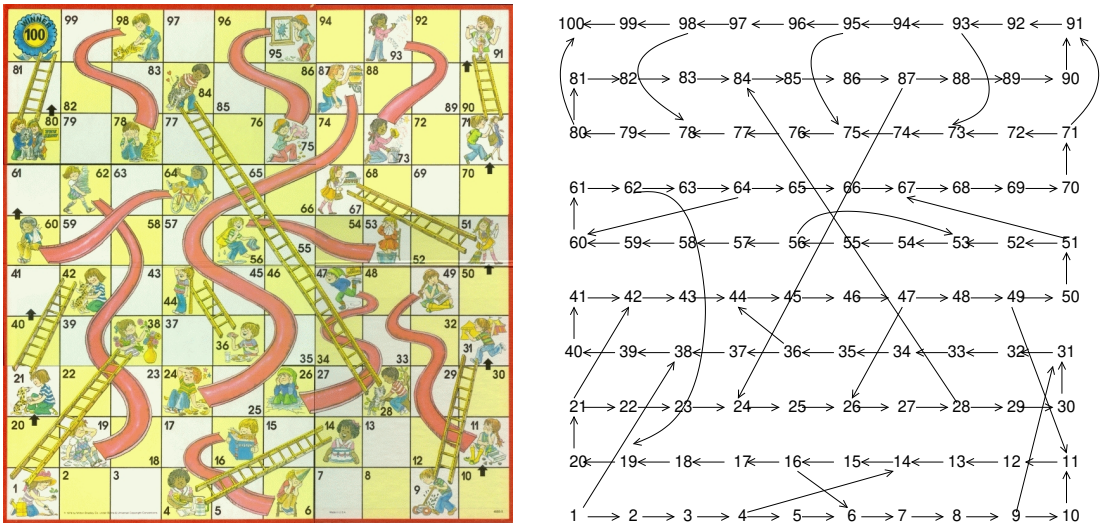
3.6 *Chapter summary*

4
Graphs

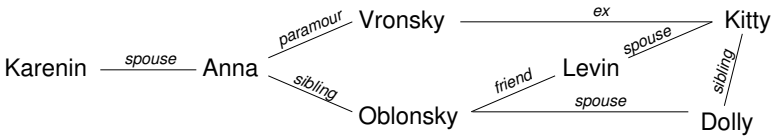
4.1 Concepts

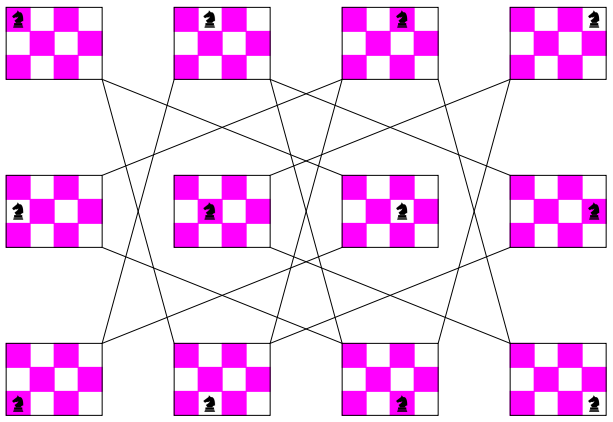
`https://www.dupageforest.org/places-to-go/forest-preserves`



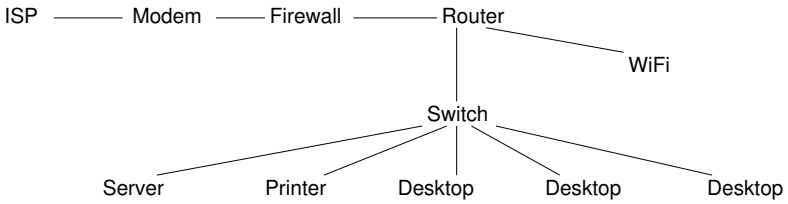


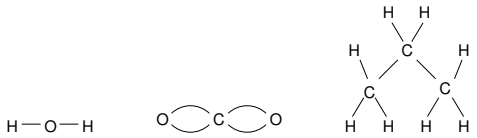
Milton Bradley. *Chutes and Ladders.* 1974 The modern published version is based on the ancient Indian game Moksha Patam.



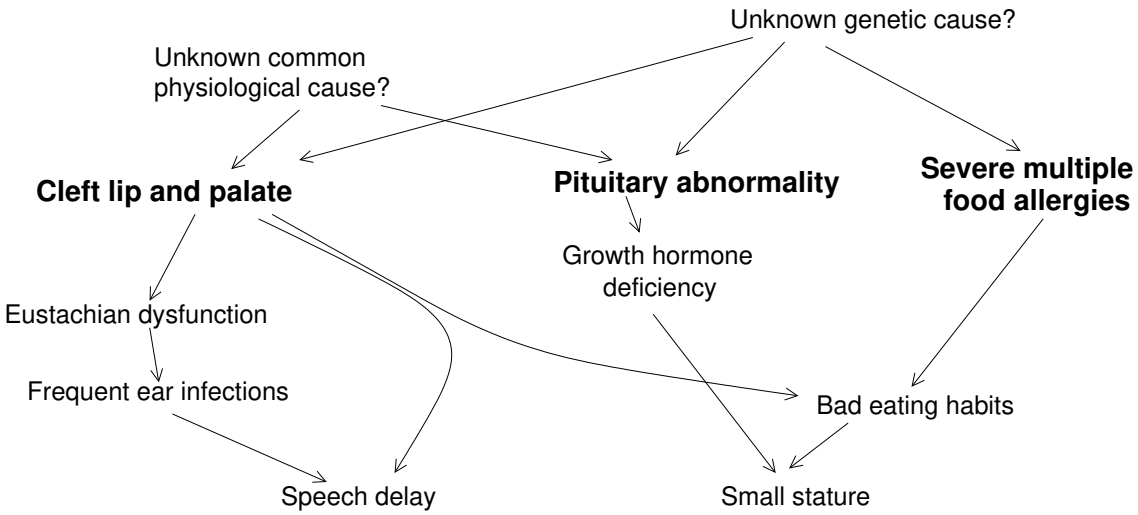


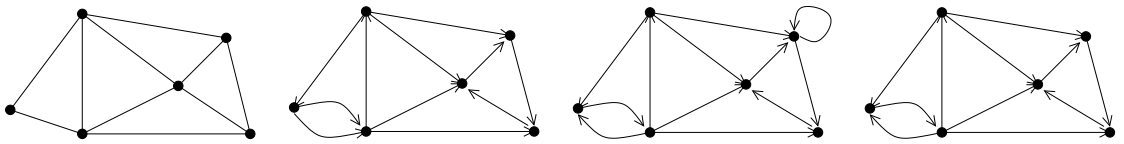
Chess knight from the Free Icons Library, `http://chittagongit.com/icon/knight-chess-icon-16.html`





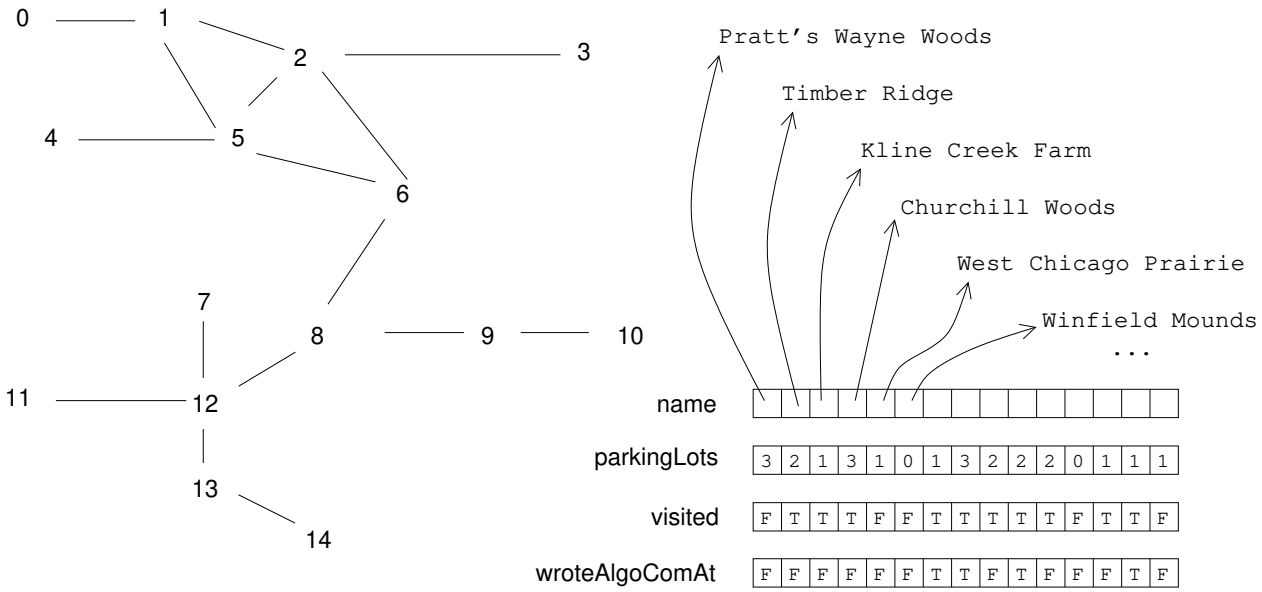
Water Carbon dioxide Propane

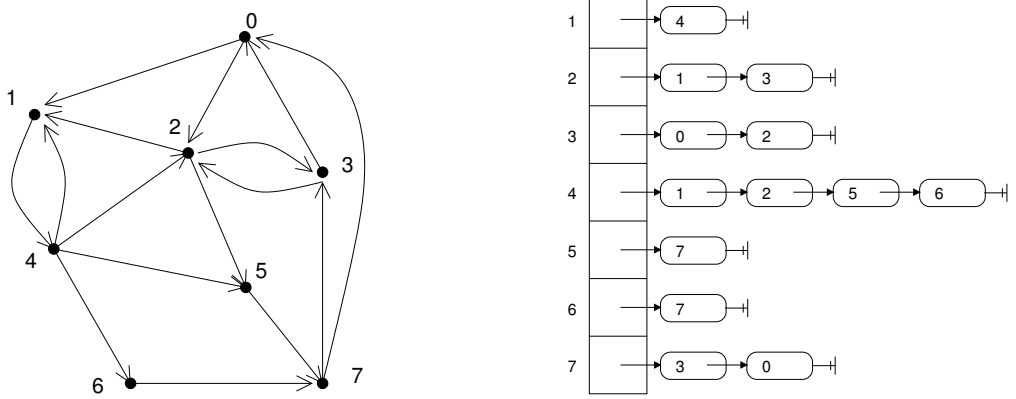


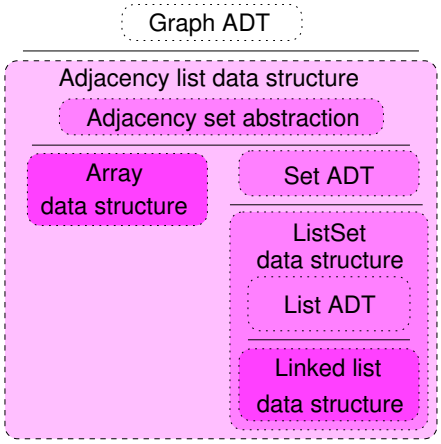


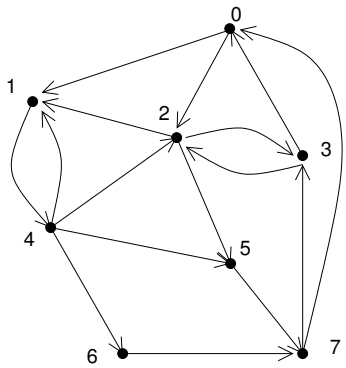
An undirected graph | A directed graph with parallel edges | A directed graph with a self-loop | A simple directed graph

4.2 *Implementation*

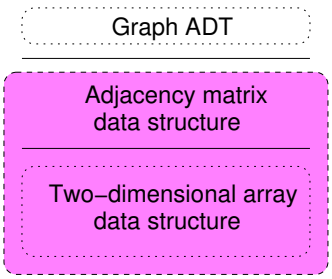


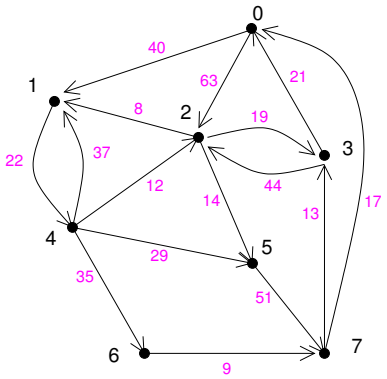






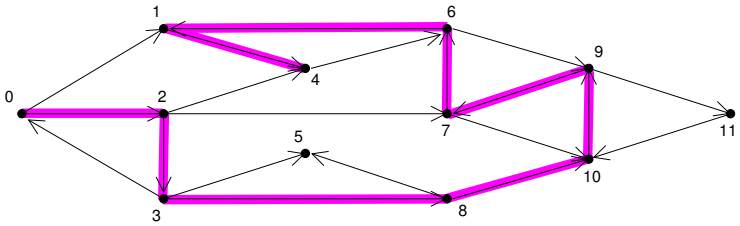
	0	1	2	3	4	5	6	7
0				*T*				*T*
1	*T*		*T*		*T*			
2	*T*			*T*	*T*			
3			*T*					*T*
4		*T*						
5			*T*		*T*			
6					*T*			
7						*T*	*T*	

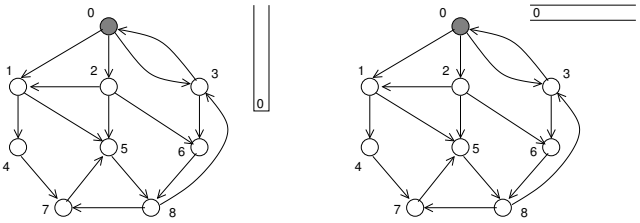




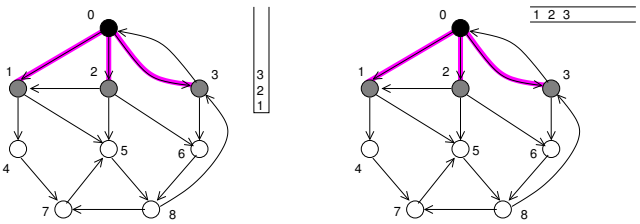
	0	1	2	3	4	5	6	7
0	0	∞	∞	21	∞	∞	∞	17
1	40	0	8	∞	37	∞	∞	∞
2	63	∞	0	19	12	∞	∞	∞
3	∞	∞	19	0	∞	∞	∞	13
4	∞	22	∞	∞	0	∞	∞	∞
5	∞	∞	14	∞	29	0	∞	∞
6	∞	∞	∞	∞	35	∞	0	∞
7	∞	∞	∞	∞	∞	51	9	0

4.3 *Traversal*

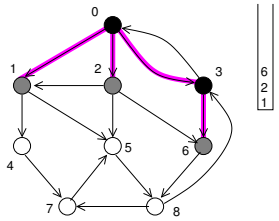


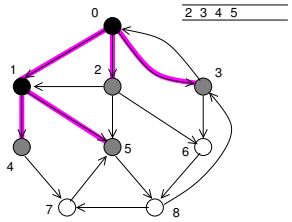


Initially the starting point, 0, is discovered and in the worklist—the top of the DFT stack and the front of the BFT queue.

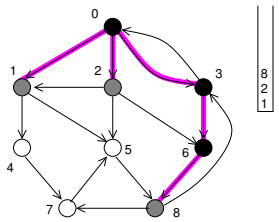


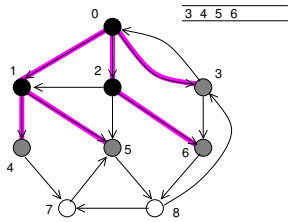
In either algorithm, 0 is removed from the worklist and its adjacents—all of them newly discovered—are added to the worklist. For DFT, 3 is at the top of the stack, but for BFT, 1 is at the front of the queue.



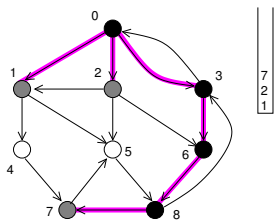


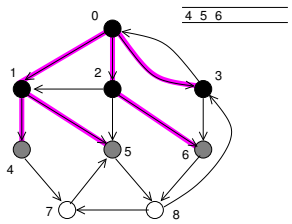
DFT pops and visits 3, whose adjacents are 0 and 6. DFT ignores 0 because it's already discovered, but pushes 6. BFT removes 1, visits it, and enqueues its adjacents 4 and 5.



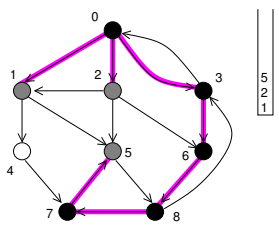


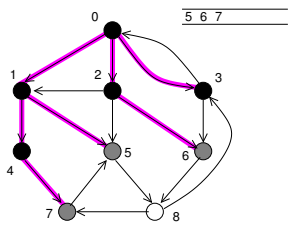
DFT pops 6. There is one adjacent vertex, 8, and since it's undiscovered, it is pushed. BFT removes 2. The adjacents 1 and 5 are already discovered, but newly discovered adjacent 6 is enqueued.



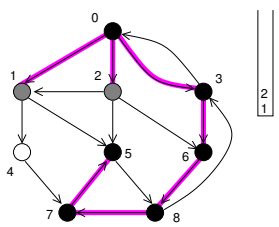


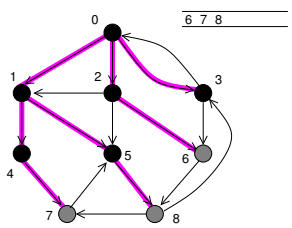
DFT pops 8, which is adjacent to 3 and 7. Of them only 7 is newly discovered, and it is pushed. BFT removes 3, but its adjacents 0 and 6 are already discovered. Nothing is enqueued.



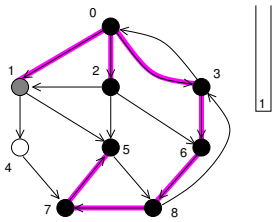


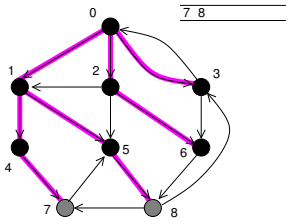
DFT pops 7 and discovers the adjacent 5, which is pushed. BDF removes 4. The only adjacent, 7, is newly discovered and enqueued.



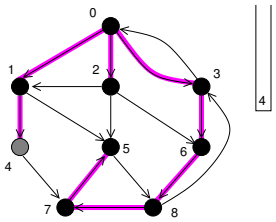


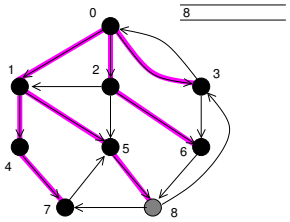
DFT pops 5, but its only adjacent, 8, is already discovered. This terminates a path from 0 to 5 explored over the previous five steps. BFT also removes 5, but here the adjacent 8 is newly discovered and enqueued. Now BFT has discovered all vertices; its remaining steps empty the queue.



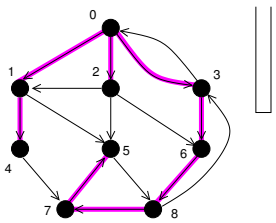


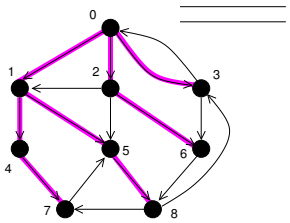
DFT effectively backtracks back to vertex 2, which is popped. The adjacent vertices 1, 5, and 6 are already discovered, so nothing is pushed. BFT removes 6, but its only adjacent 8 is already discovered, and nothing is enqueued.



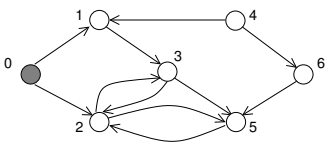


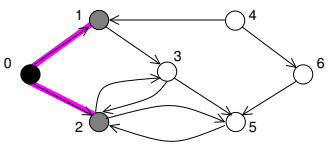
DFT pops 1, which has adjacents 4 and 5, and pushes the newly discovered 4. DFT has now discovered all vertices. BFT removes 7, but its only adjacent, 5, is already discovered.

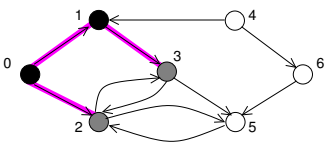


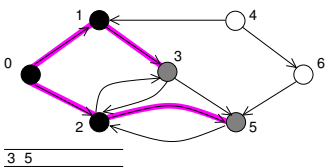


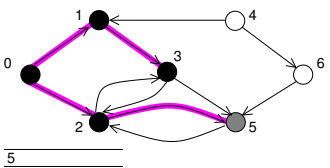
DFT pops 4, and BFT removes 8. Both worklists now emptied, the algorithms terminate.

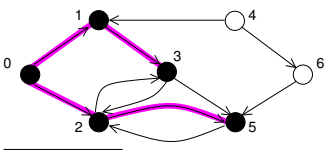


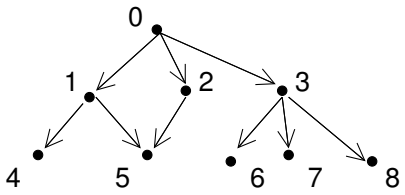


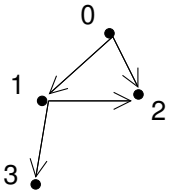


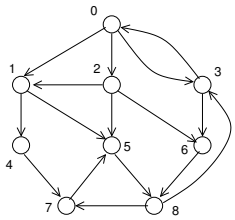


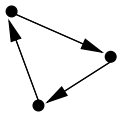


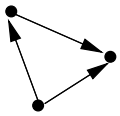




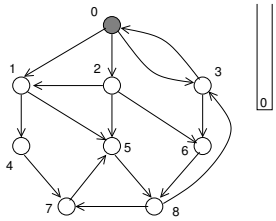




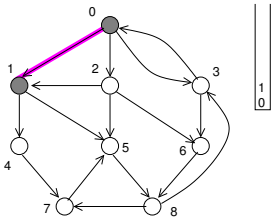




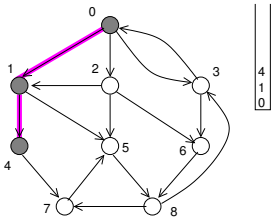
```
def traverse(g, s, worklist, f):
    parents = [None for v in range(g.num_vertices())]                    c0 V
c1  worklist.add(s)
    parents[s] = s
    while not worklist.is_empty():                                       c2 (V + 1)
c3 V    v = worklist.remove()
        f(v)
        for u in g.adjacents(v):                                         c4 (V + E)
            if parents[u] == None:                                       c5 E
c6 (V − 1)      parents[u] = v
                worklist.add(u)
    return parents                                                       c7
```



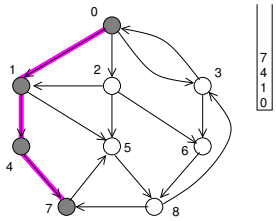
The call to visit the starting point is on the stack, a root of subsequent calls.



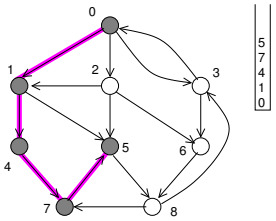
The loop over 0's adjacents discovers 1 which it visits, recursively calling the function.



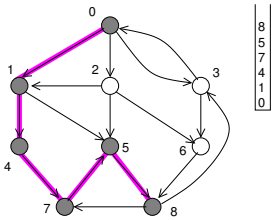
The consequent loop over 1's adjacents results in a call to visit 4.



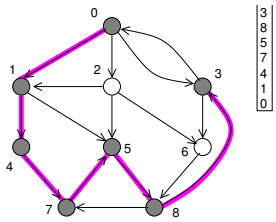
From 4 we discover 7...



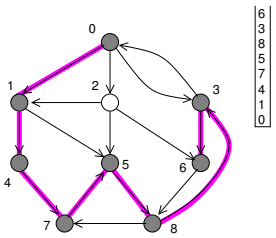
...from 7 we discover 5 ...



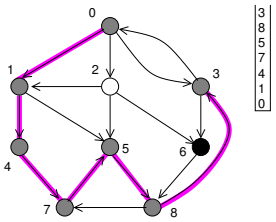
...and from 5 we discover 8.



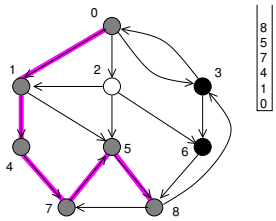
From 8 we discover 3. By this point we have seven calls to `depth_first_r()` that are active. The loops over adjacent vertices in all the calls below the top are paused.



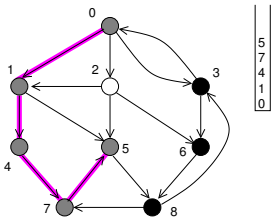
In the loop over 3's adjacents, 0 is already discovered and so we skip it, and call `depth_first_r()` on the newly discovered 6.



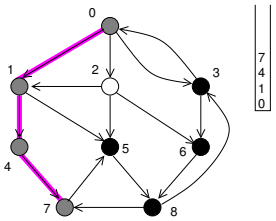
Now 6's only adjacent, 8, is already discovered. The call to visit 6 returns, and 6 is finished.



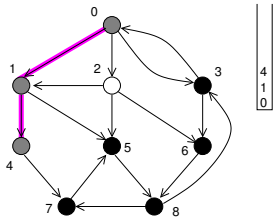
The loop over 3's adjacents resumes but is immediately complete. The call to visit 3 returns, and 3 is finished.



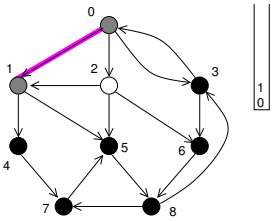
The loop over 8's adjacents resumes, but since 7 is already discovered, the call to visit 8 returns, and 8 is finished.



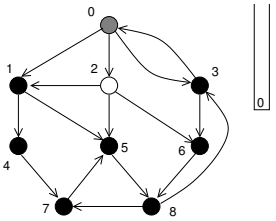
Similarly the call to visit 5 returns, since its loop terminates immediately after resuming. 5 is now finished.



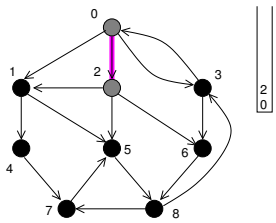
The call to visit 7 also returns immediately after resuming. 7 is now finished.



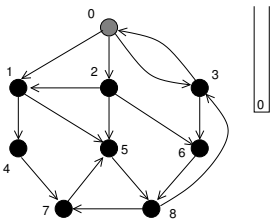
The recursion winds its way back to the call to visit 1.



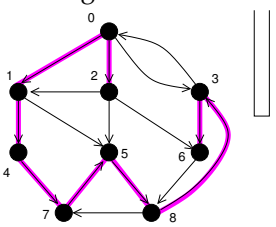
The call returns, and 1 is finished. Now we're back to the call visiting 0.



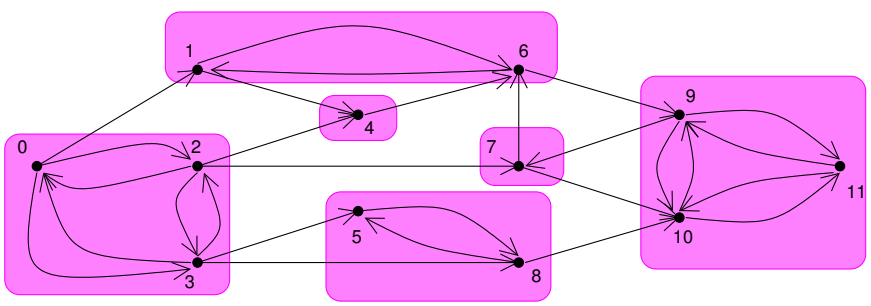
Next `depth_first_r()` is called on 0's adjacent 2. All of 2's adjacents are already discovered, so visiting 2 results in no more recursive calls.



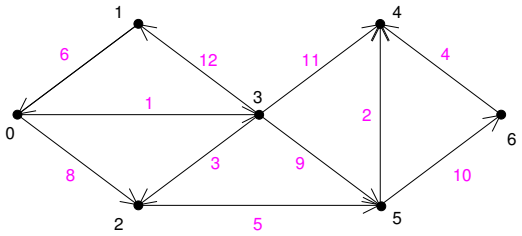
Back at 0 again, its remaining adjacent 3 is already discovered, so again no more recursive calls are made.

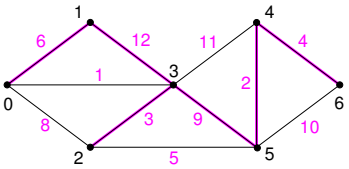


All recursive calls have returned. We re-highlight all the edges that were followed so we can see the complete traversal tree as captured by the `parents` list

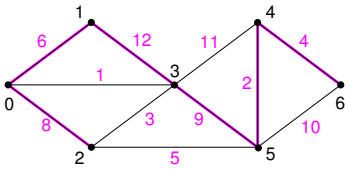


4.4 *Minimum spanning trees*

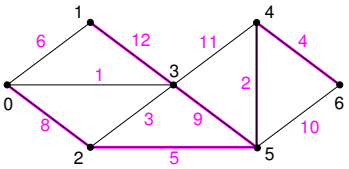




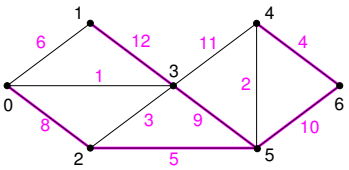
Total weight 47



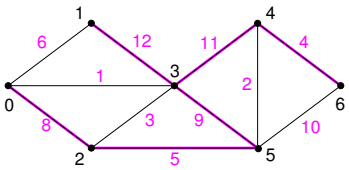
Total weight 41



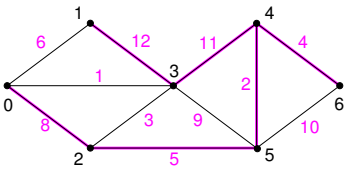
Total weight 40



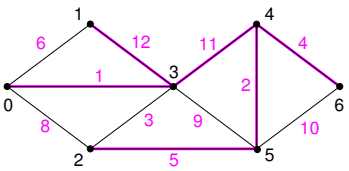
Total weight 48



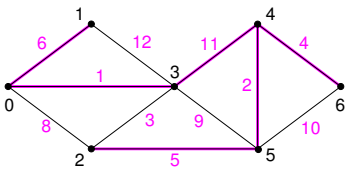
Total weight 49



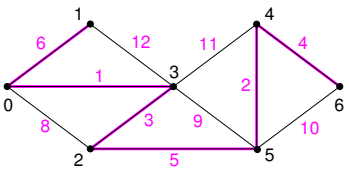
Total weight 42



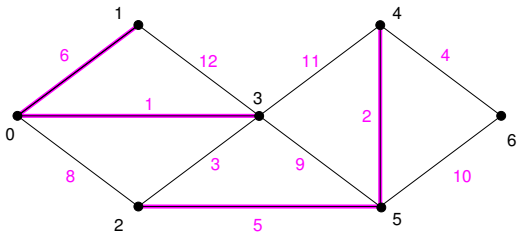
Total weight 35

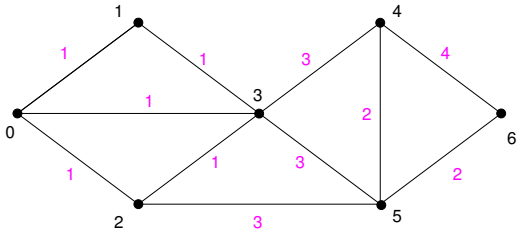


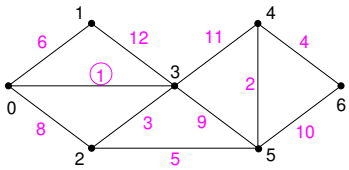
Total weight 29

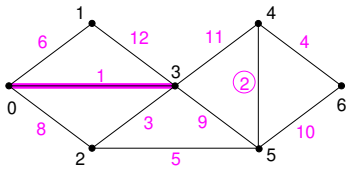


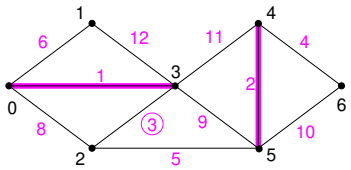
Total weight 21

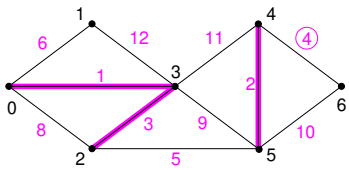


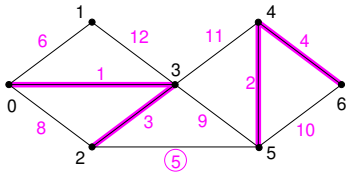

0
1
2
3
4
5
6
1
1
1
1
3
3
4
2
1
3
2

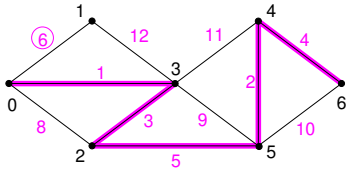

0
1
2
3
4
5
6
6
12
11
4
8
3
9
10
5
2
①

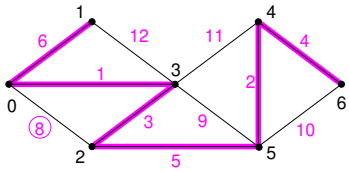

0
1
2
3
4
5
6
6
12
11
4
1
8
3
9
10
5
2
②

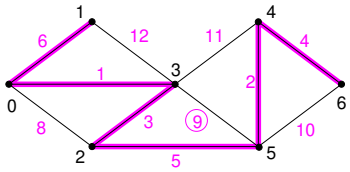

0
1
2
3
4
5
6
6
12
11
4
1
8
③
9
10
5
2

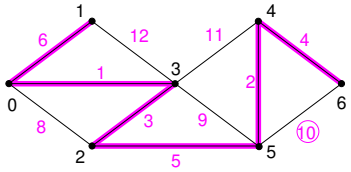

0
1
2
3
4
5
6
6
12
11
④
1
8
3
9
10
5
2

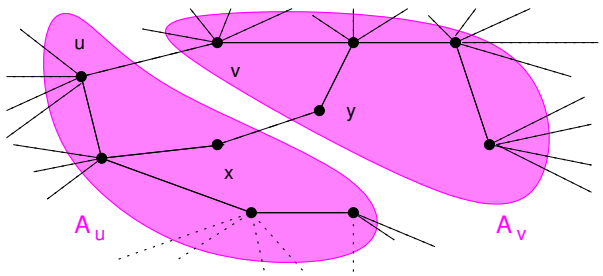

0
1
2
3
4
5
6
6
12
11
4
1
8
3
9
10
⑤
2


0
1
2
3
4
5
6
⑥
12
11
4
1
8
3
9
10
5
2

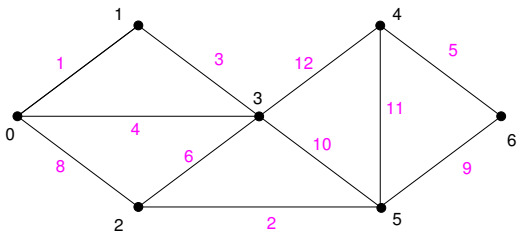

0
1
2
3
4
5
6
6
12
11
4
1
⑧
3
9
10
5
2

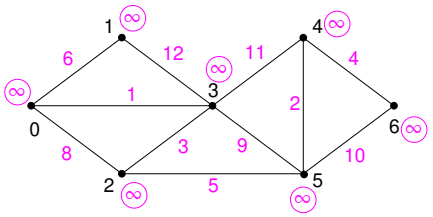

0
1
2
3
4
5
6
6
12
11
4
1
8
3
⑨
10
5
2


0
1
2
3
4
5
6
6
12
11
4
1
8
3
9
⑩
5
2

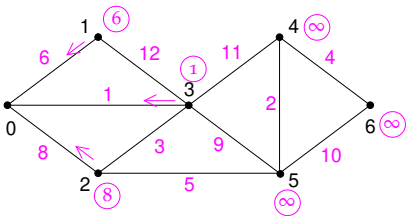

u
v
y
x
A u
A v

```
def mst_kruskal(g):
    components = DisjointSet(g.num_vertices())          Θ(V)
Θ(E) edges = undirected_weighted_edges(g)
    counting_sort(edges, lambda e: e[2])                Θ(E)
Θ(1) A = set()
    for (v,u,w) in edges:                               Ω(E)
        if not components.connected(v, u):              O(E lg V)
O(V lg V)   components.union(v, u)
            A.add((v, u))                               Θ(V)
    return A
```

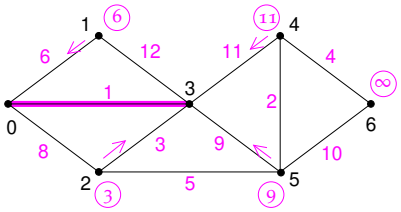




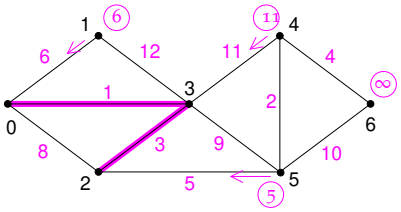
Initially all vertices have no prospective parent and infinite distance bound.



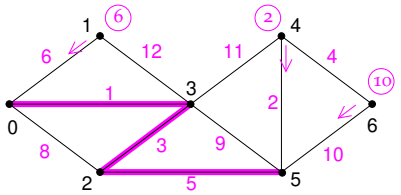
Suppose 0 is removed from the worklist. Relaxing its edges gives vertices 1, 2, and 3 smaller distance bounds and 0 as prospective parent.



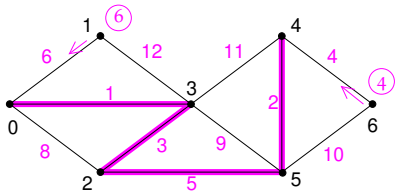
Having least distance bound, 3 is removed from the worklist and edge $(0,3)$ is added to the tree. Relaxing 3's edges results in new distance bounds for 2, 4, and 5, and 3 is their new prospective parent, but no change for 1.



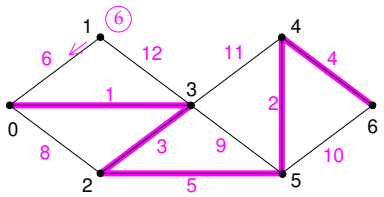
With distance bound 3, vertex 2 is next out of the worklist. We relax the edge $(2,5)$, with 5's prospective parent changing to 2. The other vertices adjacent to 2 are already connected.



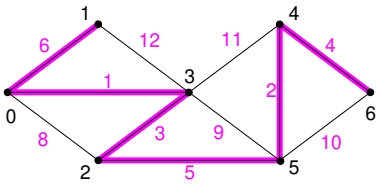
Vertex 5 has least distance bound, and its prospective parent 2 becomes its definite parent in the tree. Relaxing edges $(5,4)$ and $(5,6)$ makes 5 the prospective parent of 4 and 6.



With distance bound 2, vertex 4 is next. The edge $(4,6)$ is a shorter edge to connect 6 to the tree than $(5,6)$, and so 4 becomes 6's prospective parent.



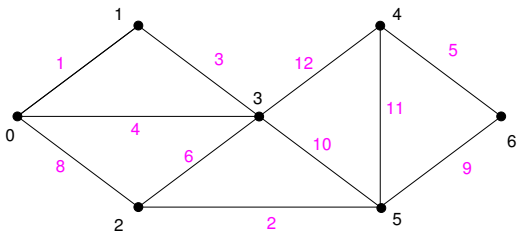
Vertex 6 is removed from the worklist and edge $(4,6)$ is added to the tree. All of 6's adjacents are already connected—no further changes.



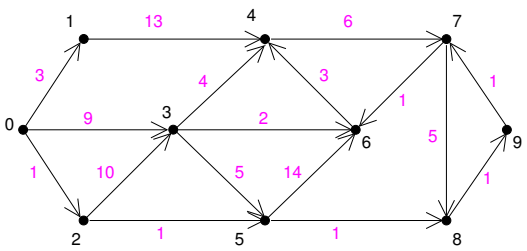
The last vertex in the worklist is removed, and edge $(0,1)$ is added to the tree. The algorithm is finished.

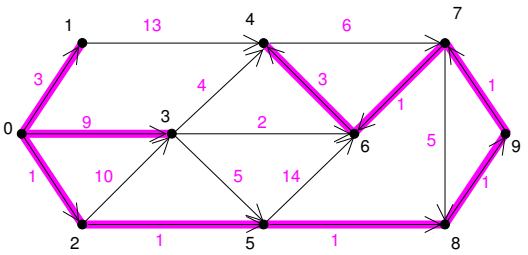
```
def mst_prim(g):
    parents = [None for v in range(g.num_vertices())]
    distances = [sys.maxint for v in range(g.num_vertices())]
    pq = PriorityQueue(g.num_vertices(), distances)
    A = set()
    while not pq.is_empty():
        v = pq.extract_max()
        if parents[v] == None :
            assert len(A) == 0
        else :
            A.add((parents[v], v))
        for u in g.adjacents(v):
            if (pq.contains(u) and
                (parents[u] == None or g.weight(v, u) < distances[u])) :
                parents[u] = v
                distances[u] = g.weight(v, u)
                pq.increase_key(u)
    return A
```

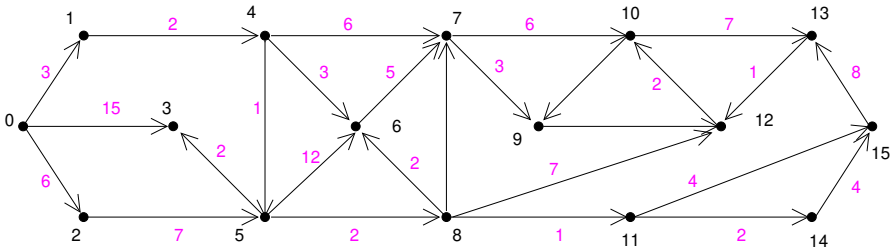
$\Theta(V)$ $\Theta(1)$ $\Omega(V)$ $O(V \lg V)$ $\Theta(V)$ $\Omega(E)$ $O(E)$ $O(E)$ $O(E \lg V)$

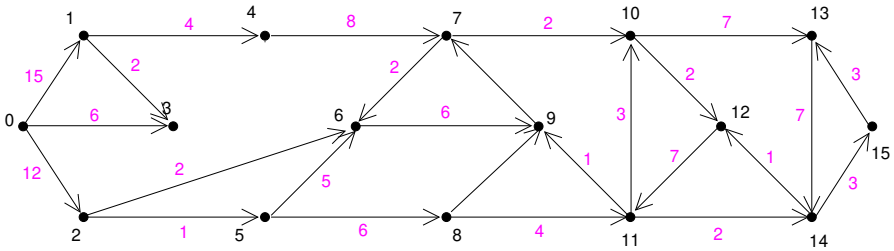


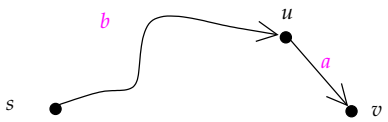
4.5 *Shortest paths*

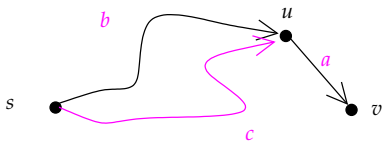


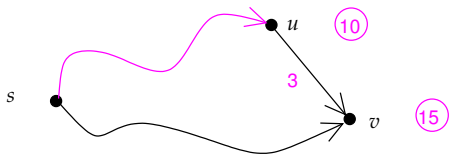


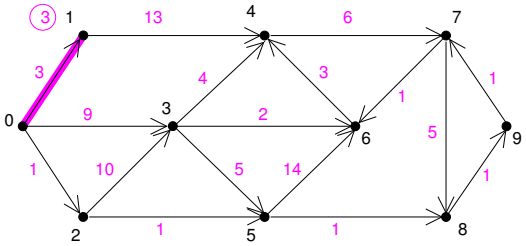

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

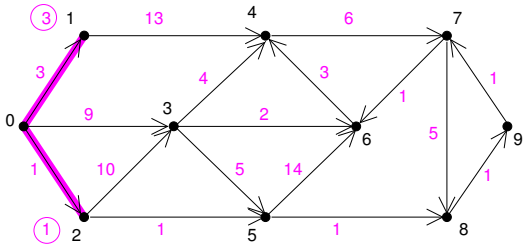

b
u
a
s
v


b
u
a
s
v
c

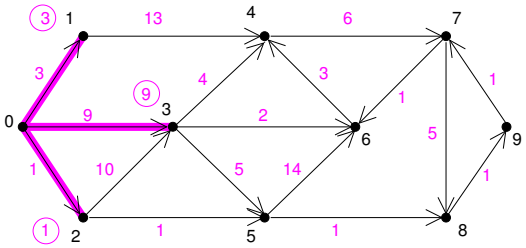

u
10
3
s
v
15



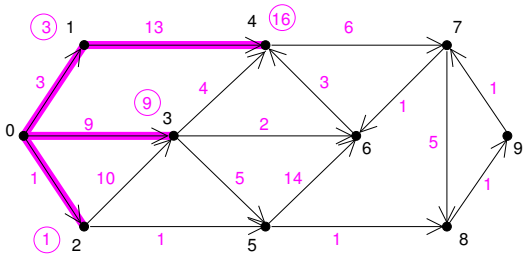
Relaxing edge $(0,1)$ discovers a distance bound for vertex 1 and sets 1's prospective parent to 0.



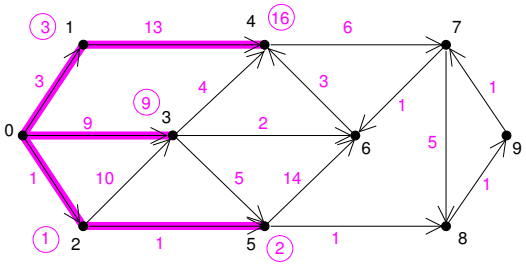
We relax edge $(0,2)$.



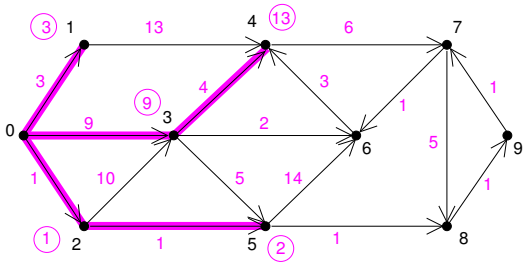
We relax edge $(0,3)$.



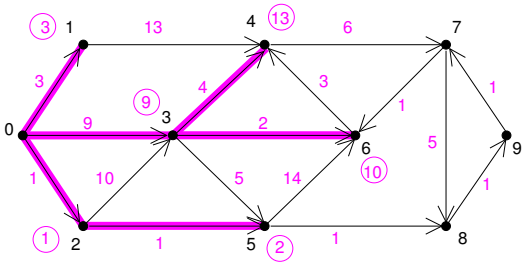
We relax edge $(1,4)$.



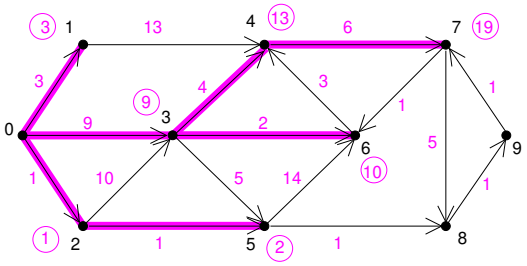
Relaxing edge $(2,3)$ does not find an improvement for vertex 3, but relaxing edge $(2,5)$ sets 5's distance bound to 2.



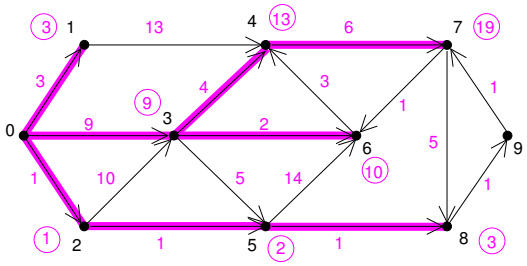
Relaxing edge $(3,4)$ discovers a shorter path to 4. We update distance bound and prospective parent.



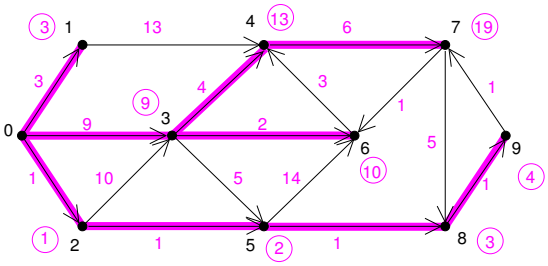
We relax edge $(3,5)$ to no improvement. We also relax edge $(3,6)$.



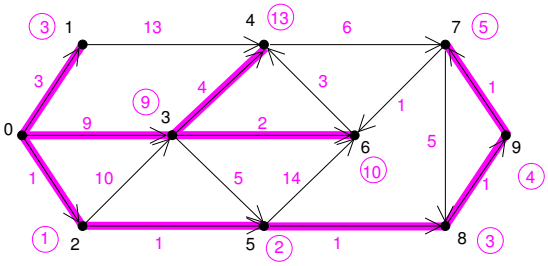
We relax edge $(4,7)$ to find distance bound 19 for vertex 7.



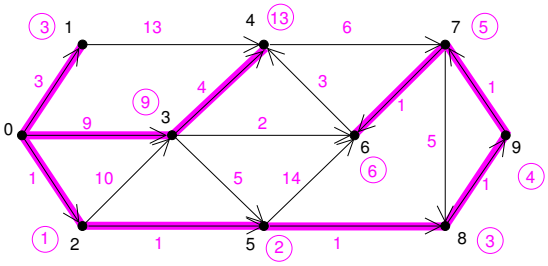
Relaxing edge $(5,6)$ provides no improvement for vertex 6, but relaxing edge $(5,8)$ finds a distance bound of 3 for vertex 8.



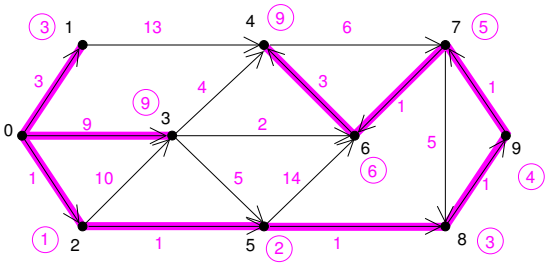
Relaxing the outgoing edges from vertices 6 and 7 finds no improvement. We relax edge $(8,9)$ to set 9's distance bound to 4.



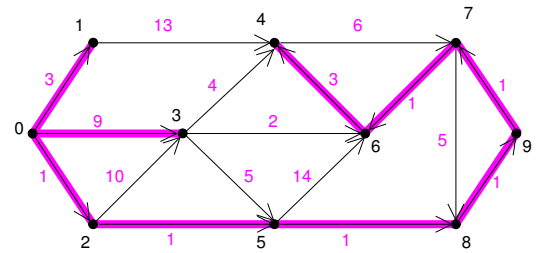
Relaxing edge $(9,7)$ finds an improvement for vertex 7. This completes the first round of relaxations.



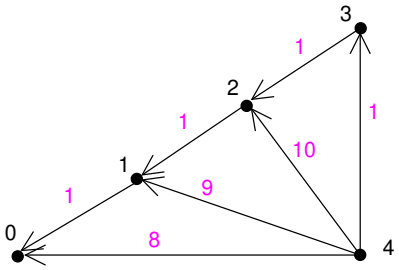
The second round of relaxations repeats the entire process, but the only improvement is found by relaxing edge $(7,6)$. Vertex 7 is 6's new prospective parent.

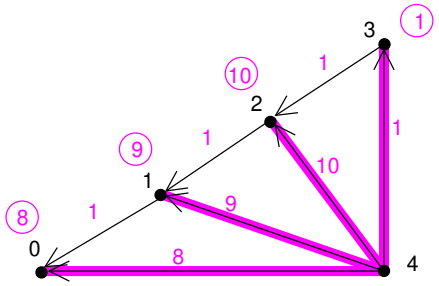


A third round of relaxations discovers a shorter path for vertex 4 when edge $(6,4)$ is relaxed.

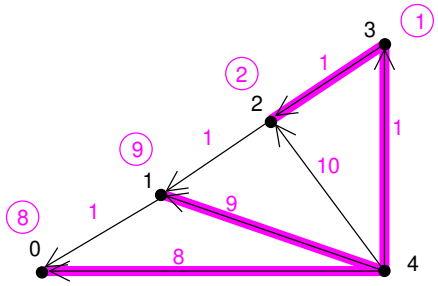


A fourth round of relaxations discovers no improvements. We have converged on the shortest path tree.

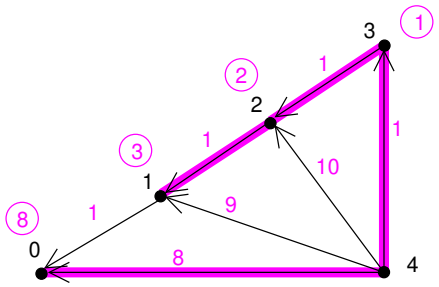




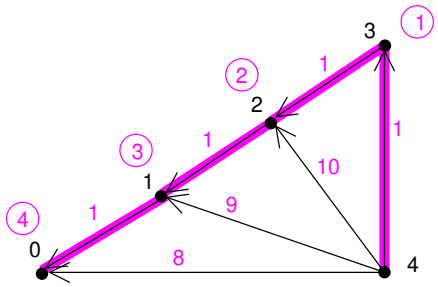
Vertices 1, 2, and 3 each have only one outgoing edge. Their distance bound is ∞ when those edges are relaxed on the first round, which doesn't change the initial distance bounds and parents of vertices 0, 1, and 2. When the outgoing edges of 4 are relaxed, 4 becomes the prospective parent of each of the other vertices. That results in a correct distance bound for vertex 3, but no others.



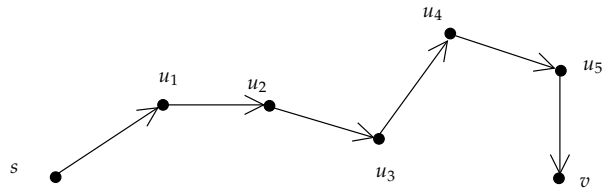
Relaxing edge $(1, 0)$ on the second round reveals a path of total weight $9 + 1 = 10$ to vertex 0. This is worse than the distance 8 already known, so there is no change to 0. Similarly, the relaxation of $(2, 1)$ effects no change. But relaxing edge $(3, 2)$ uncovers the shortest path to vertex 2.

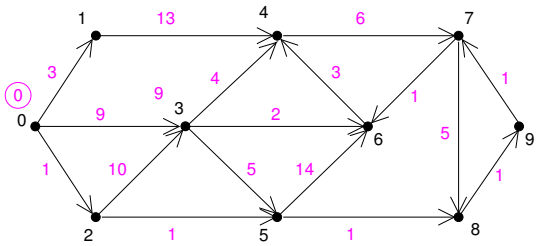


With 2's correct distance having been discovered on the second round, relaxing $(2, 1)$ reveals 1's correct distance on the third round. Unfortunately, the third-round relaxation of edge $(1, 0)$ has already happened, and so ...

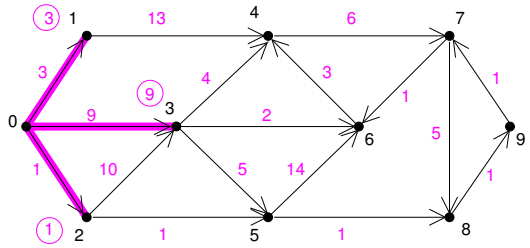


... it is not until the fourth round that 1's correct distance bound allows us to find the correct distance for 0 when $(1, 0)$ is relaxed.

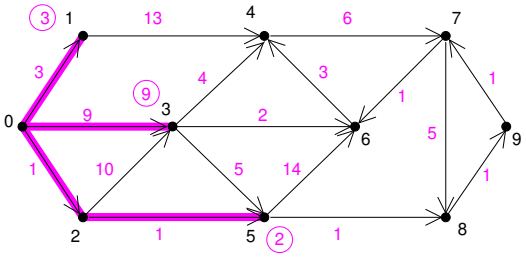




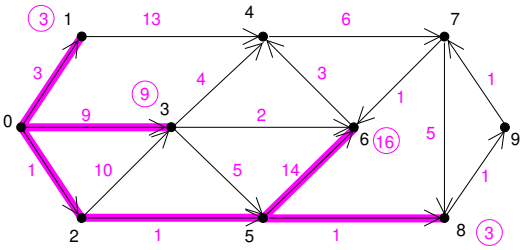
Initially all vertices have infinite distance except the source, 0.



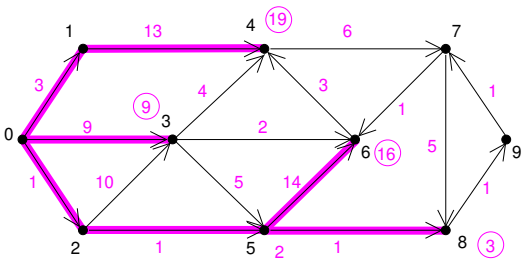
After relaxing 0's edges, 0 is the prospective parent of vertices 1, 2, and 3, which now have finite distance bounds.



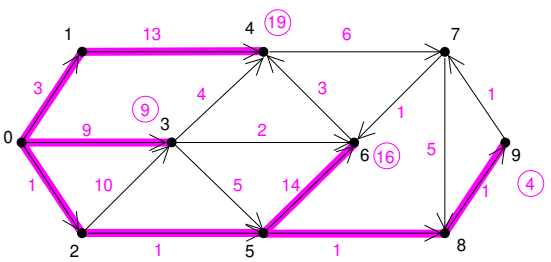
Vertex 2 has lowest distance bound. Relaxing $(2,3)$ doesn't change 3, but relaxing $(2,5)$ sets 2 as 5's prospective parent.



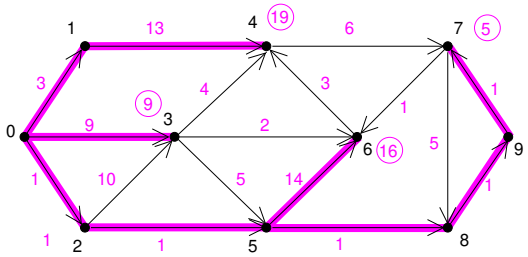
Vertex 5 has lowest distance bound. We relax its outgoing edges, and now 6 and 8 have finite distance bounds.



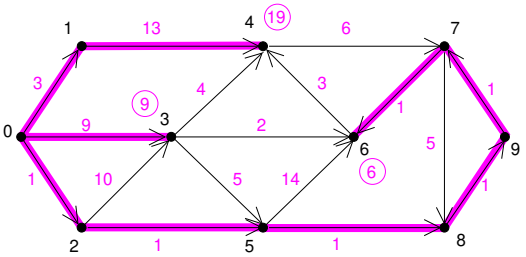
Vertices 1 and 8 are tied for least bound. Let's break the tie by relaxing 1. Vertex 4 now has distance bound 19.



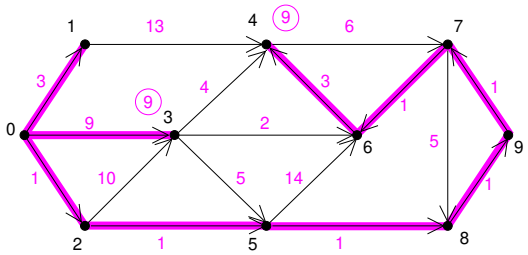
Vertex 8 has lowest distance bound. Relaxing $(8,9)$ sets 8 as 9's prospective parent. Vertex 9 has distance bound 4.



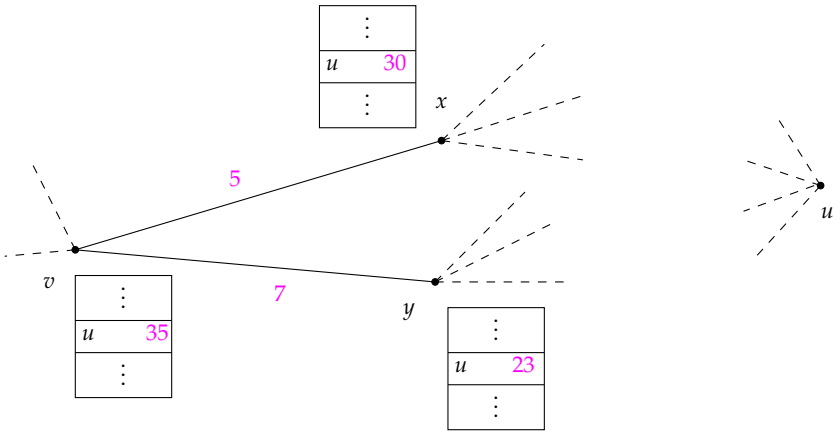
Vertex 9 is next, and we relax its outgoing edge $(9,7)$. Vertex 7 has distance bound 5.



Vertex 7 is next. Relaxing the edge $(7,6)$ reveals a shorter path for vertex 6. We update distance bound and prospective parent.



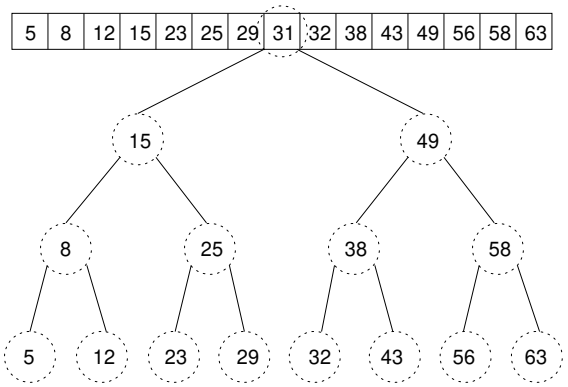
Next we relax the outgoing edges from 6. This finds a shorter path to 4. We still need to relax the edges from vertices 3 and 4, but they effect no change. With the priority queue empty, the algorithm terminates.

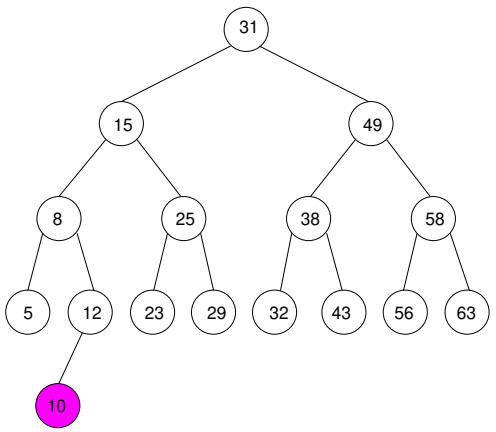


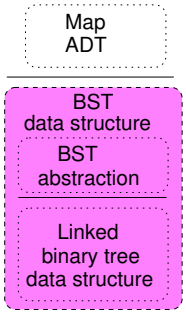
4.6 *Chapter summary*

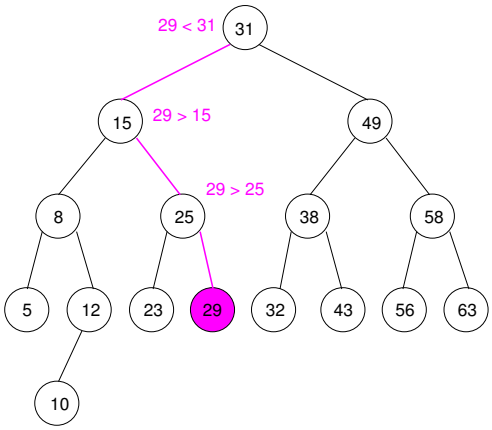
5
Search trees

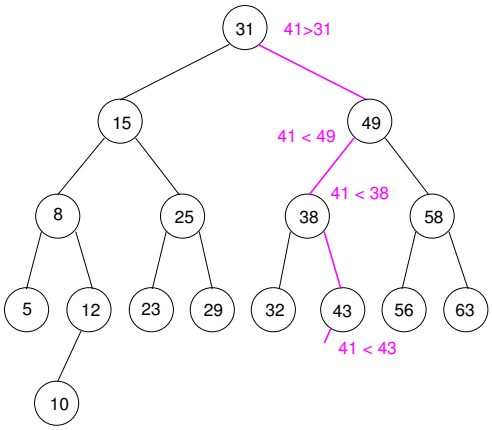
5.1 Binary search trees

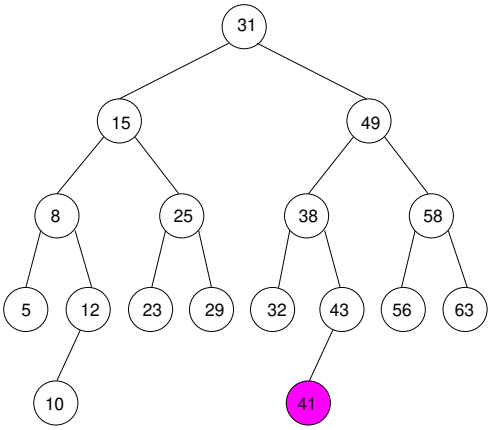


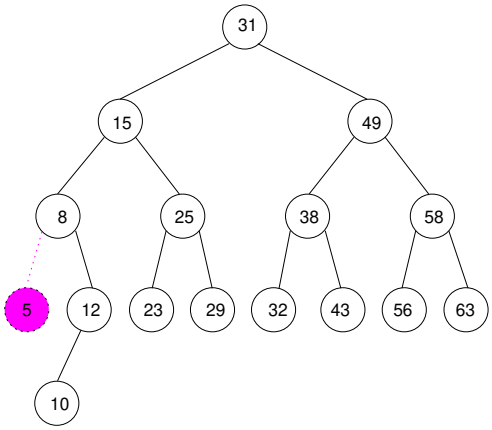


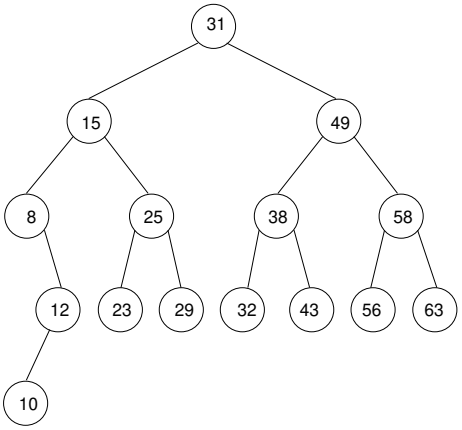

Map
ADT
BST
data structure
BST
abstraction
Linked
binary tree
data structure

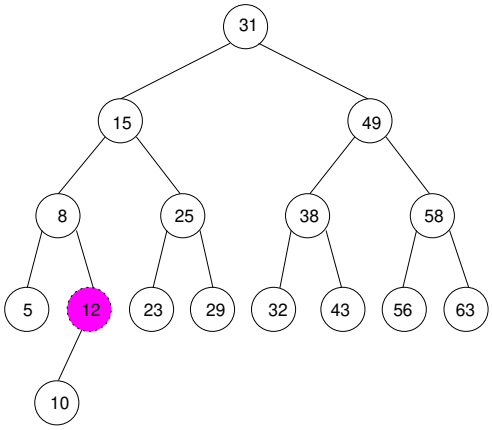

29 < 31
29 > 15
29 > 25
31
15
49
8
25
38
58
5
12
23
29
32
43
56
63
10

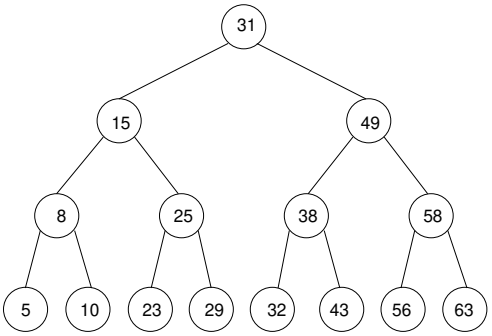

41>31
41 < 49
41 < 38
41 < 43
31
15
49
8
25
38
58
5
12
23
29
32
43
56
63
10

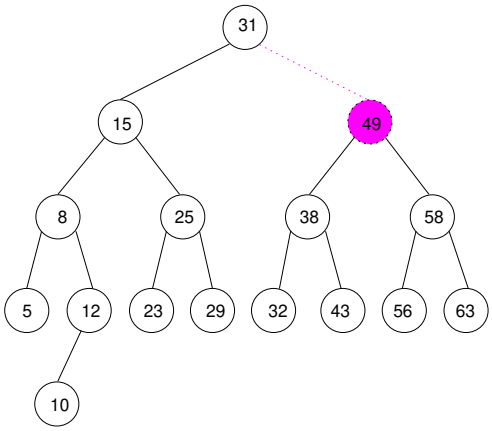

31
15
49
8
25
38
58
5
12
23
29
32
43
56
63
10
41

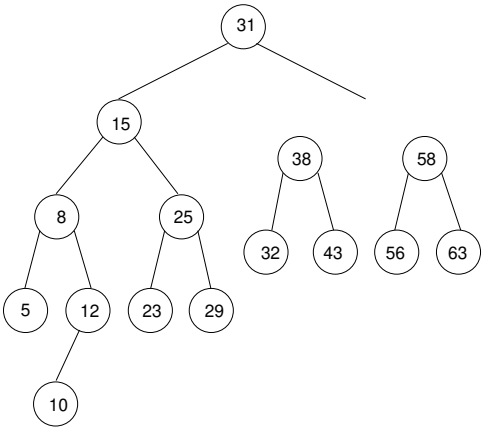

31
15
49
8
25
38
58
5
12
23
29
32
43
56
63
10

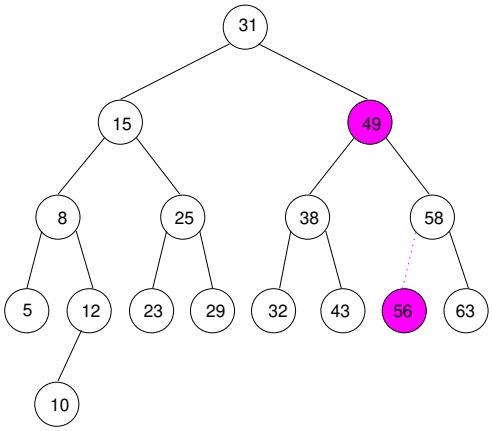

31
15
49
8
25
38
58
12
23
29
32
43
56
63
10

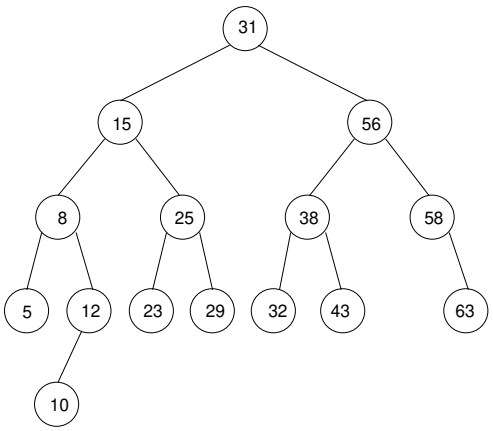

31
15
49
8
25
38
58
5
12
23
29
32
43
56
63
10

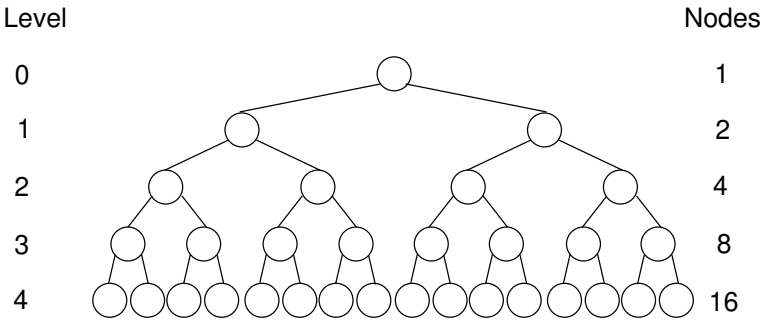

31
15
49
8
25
38
58
5
10
23
29
32
43
56
63


31
15
49
8
25
38
58
5
12
23
29
32
43
56
63
10

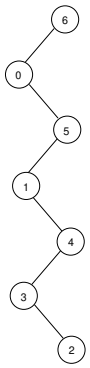

31
15
38
58
8
25
32
43
56
63
5
12
23
29
10


31
15
49
8
25
38
58
5
12
23
29
32
43
56
63
10


31
15
56
8
25
38
58
5
12
23
29
32
43
63
10

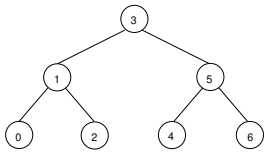

Level
Nodes
0
1
1
2
2
4
3
8
4
16

6, 0, 5, 1, 4, 2, 3



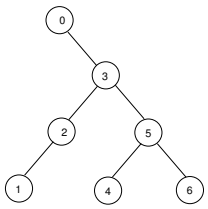
height 7
total depth 21
ANI 4

3, 1, 5, 0, 2, 4, 6



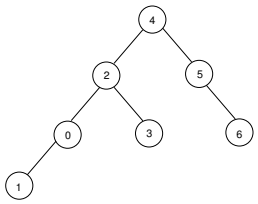
height 3
total depth 10
ANI 2.43

0, 3, 5, 2, 6, 1, 4



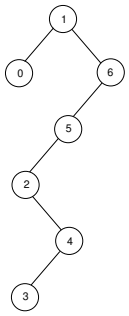
height 4
total depth 14
ANI 3

4, 2, 5, 3, 0, 1, 6



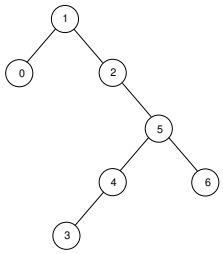
height 4
total depth 11
ANI 2.57

1, 6, 5, 2, 4, 3, 0



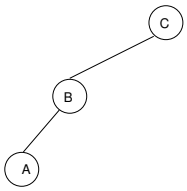
height 6
total depth 16
ANI 3.29

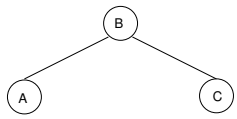
1, 2, 5, 4, 3, 0, 6

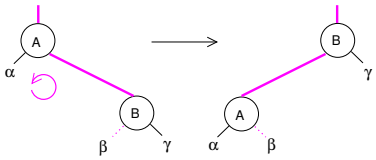


height 5
total depth 14
ANI 3

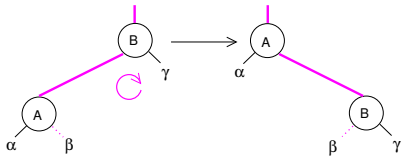
5.2 *The balanced tree problem*



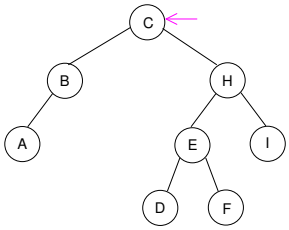




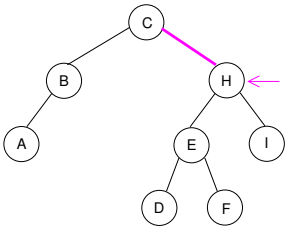
Left rotation



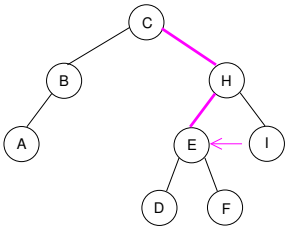
Right rotation



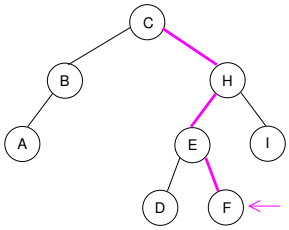
The call to `put()` starts at the root.



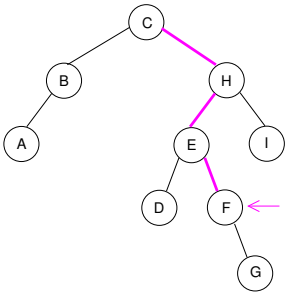
Descending to the right, `put()` is called on `H`.



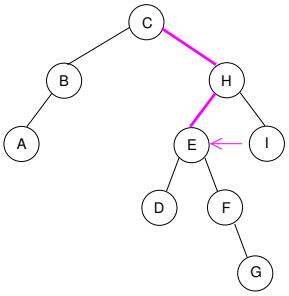
Descending to the left, `put()` is called on `E`.



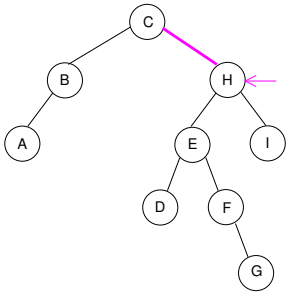
The process descends to the right again with another recursive call to `put()`.



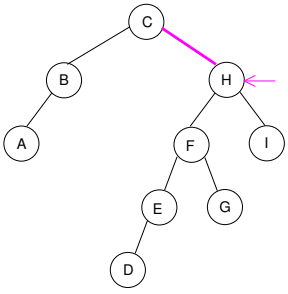
`G` is inserted as `F`'s right child. After checking for imbalance, the call returns.



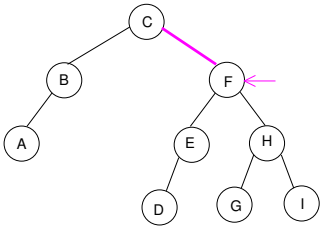
The call to `put()` on `E` resumes, checks for imbalance, and returns.



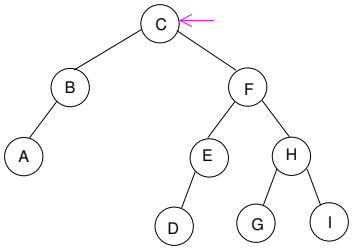
Back at `H`, suppose the imbalance at this subtree exceeds acceptable levels.



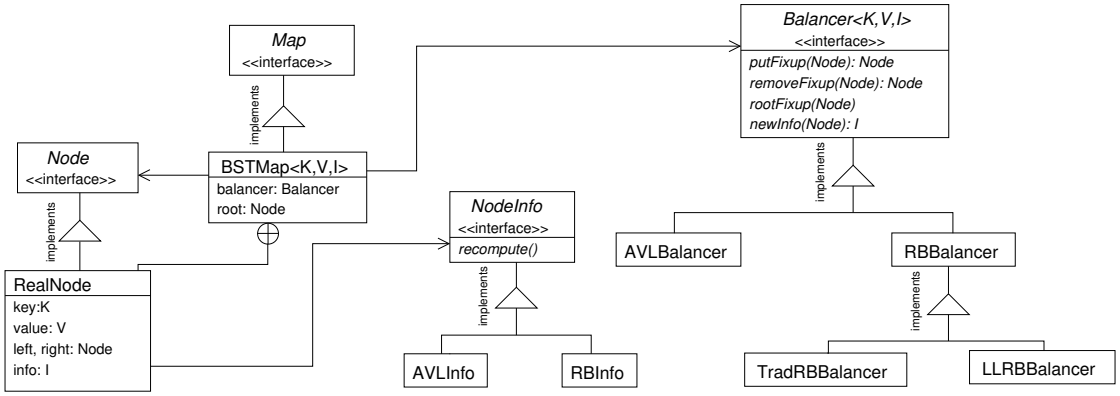
A left rotation is performed about `H`'s left child.



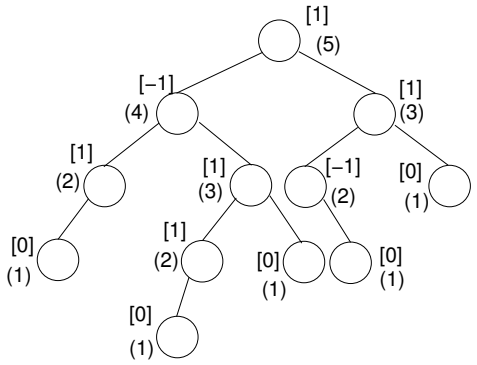
This is followed by a right rotation about `H`. That call to `put()` returns.

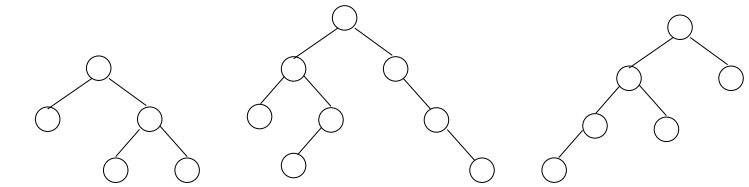


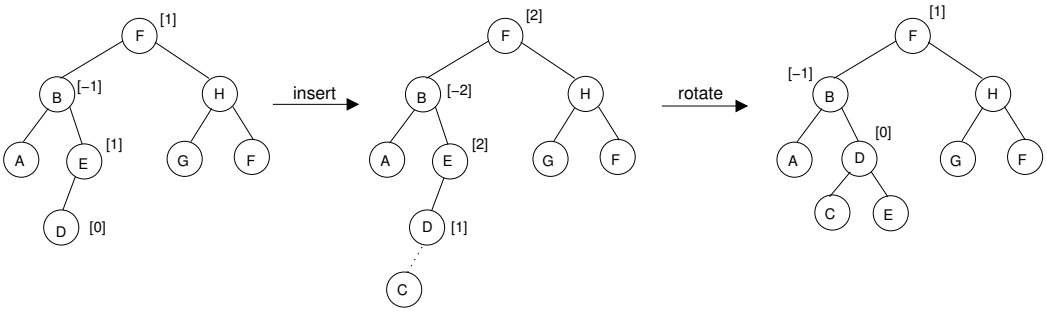
The call to `put()` on the root resumes, checks for imbalance, and returns.

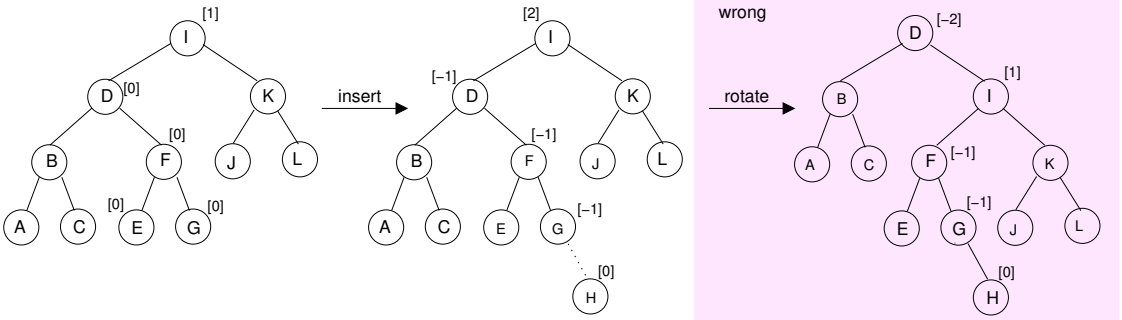


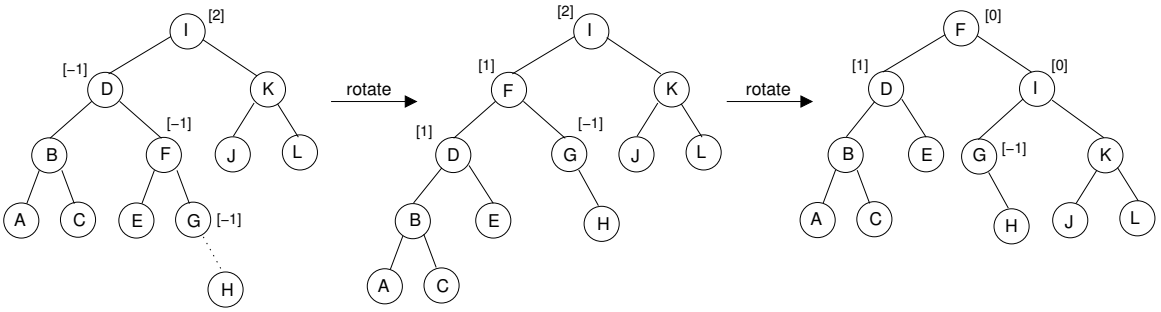
5.3 *AVL trees*

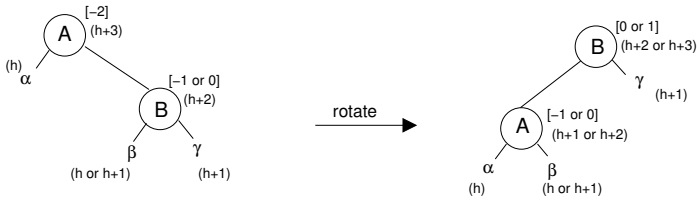


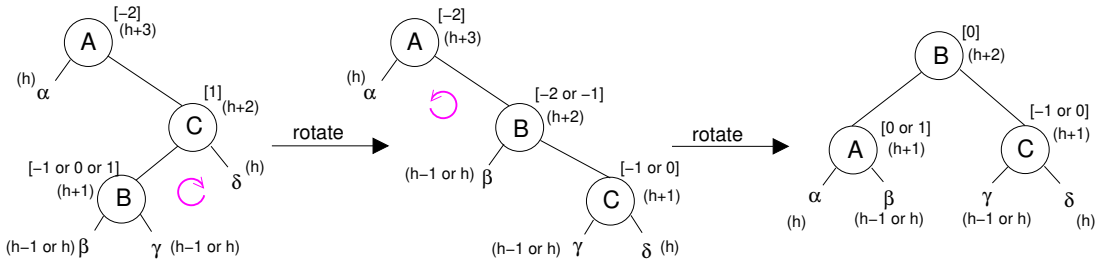



F [1]
B [−1]
H
A
E [1]
G
F
D [0]
insert
F [2]
B [−2]
H
A
E [2]
G
F
D [1]
C
rotate
F [1]
B [−1]
H
A
D [0]
G
F
C
E

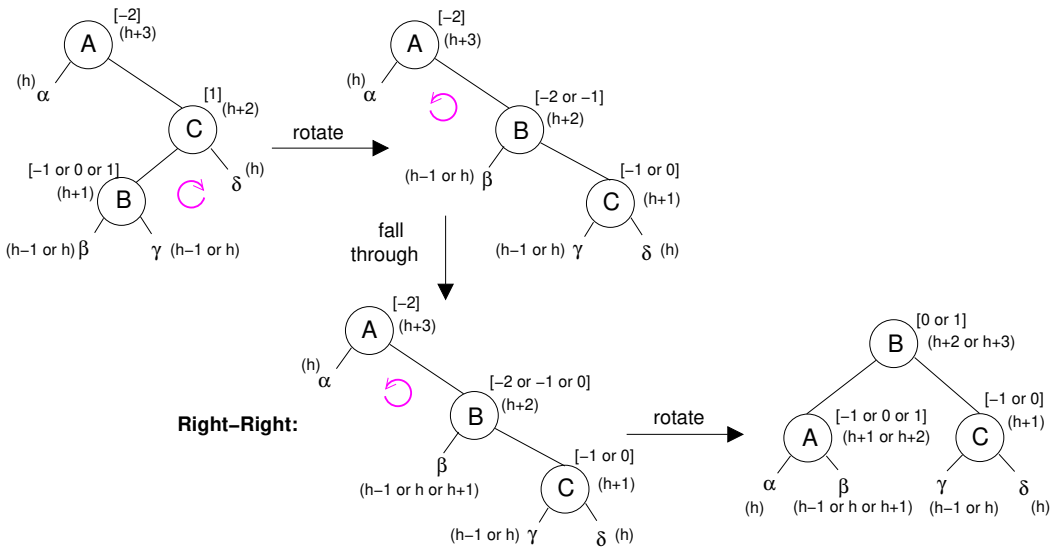

I [1]
D [0]
K
B
F [0]
J
L
A
C
E
G [0] [0]
insert
I [2]
D [−1]
K
B
F [−1]
J
L
A
C
E
G [−1]
H [0]
wrong
rotate
D [−2]
B
I [1]
A
C
F [−1]
K
E
G [−1]
J
L
H [0]

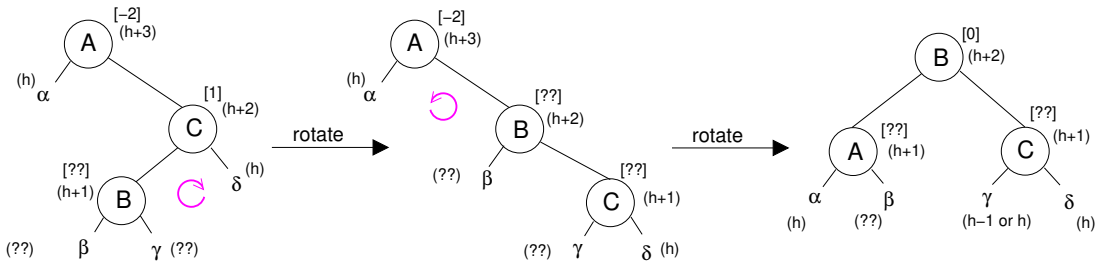

I [2]
D [−1]
K
B
F [−1]
J
L
A
C
E
G [−1]
H
rotate
I [2]
F [1]
K
D [1]
G [−1]
J
L
B
E
H
A
C
rotate
F [0]
D [1]
I [0]
B
E
G [−1]
K
A
C
H
J
L

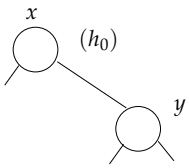

A
[−2]
(h+3)
(h)
α
B
[−1 or 0]
(h+2)
β
γ
(h or h+1)
(h+1)
rotate
B
[0 or 1]
(h+2 or h+3)
γ
(h+1)
A
[−1 or 0]
(h+1 or h+2)
α
β
(h)
(h or h+1)


A
[−2]
(h+3)
(h)
α
C
[1]
(h+2)
B
[−1 or 0 or 1]
(h+1)
δ
(h)
(h−1 or h)
β
γ
(h−1 or h)
rotate
A
[−2]
(h+3)
(h)
α
B
[−2 or −1]
(h+2)
(h−1 or h)
β
C
[−1 or 0]
(h+1)
(h−1 or h)
γ
δ
(h)
rotate
B
[0]
(h+2)
A
[0 or 1]
(h+1)
C
[−1 or 0]
(h+1)
α
β
γ
δ
(h)
(h−1 or h)
(h−1 or h)
(h)

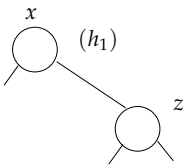
Right–Left:


A
[−2]
(h+3)
(h)
α
C
[1]
(h+2)
B
[−1 or 0 or 1]
(h+1)
δ
(h)
(h−1 or h)
β
γ
(h−1 or h)
rotate
A
[−2]
(h+3)
(h)
α
B
[−2 or −1]
(h+2)
(h−1 or h)
β
C
[−1 or 0]
(h+1)
(h−1 or h)
γ
δ
(h)
fall
through
Right–Right:
A
[−2]
(h+3)
(h)
α
B
[−2 or −1 or 0]
(h+2)
β
(h−1 or h or h+1)
C
[−1 or 0]
(h+1)
(h−1 or h)
γ
δ
(h)
rotate
B
[0 or 1]
(h+2 or h+3)
A
[−1 or 0 or 1]
(h+1 or h+2)
C
[−1 or 0]
(h+1)
α
β
γ
δ
(h)
(h−1 or h or h+1)
(h−1 or h)
(h)

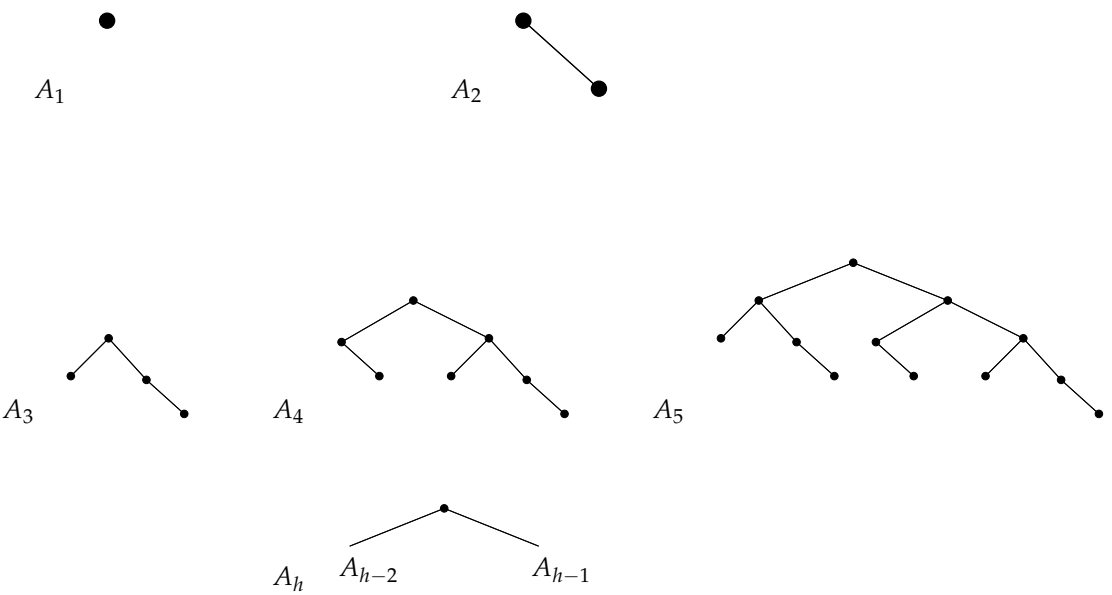




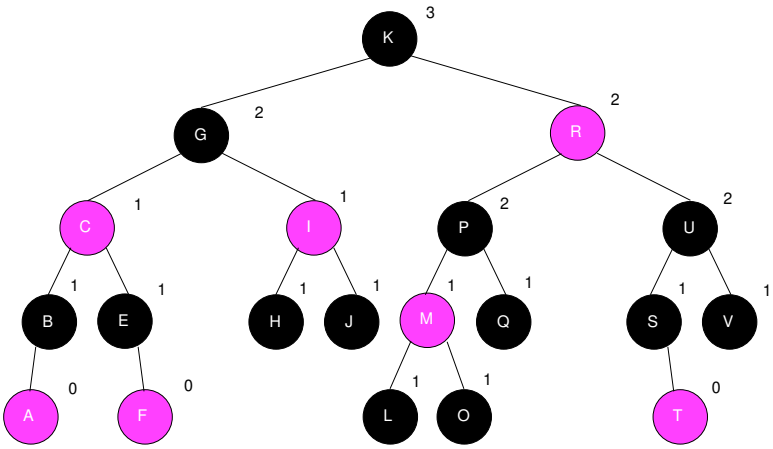
Before recursive call on right child



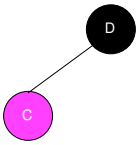
After recursive call on right child



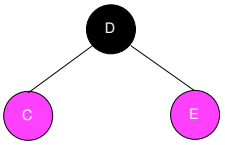
5.4 *Traditional red-black trees*



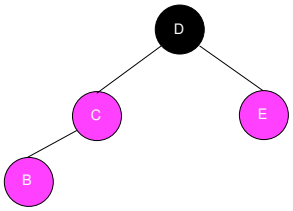
After the initial put we have one node, which we paint black.



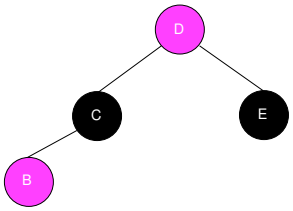
A second put, with a key less than the root, is inserted as a left child and leaf. New leaves are red.



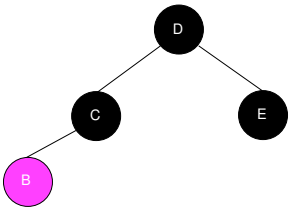
Similarly, a put with a key greater than the root results in another red leaf. The entire tree has black height 1, with a "black level" and a "red level."



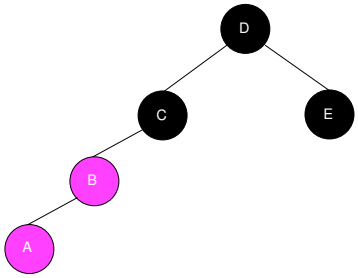
Putting a new minimum key, `B`, entails inserting a red leaf off the `C` node. This incurs a *double-red* violation.



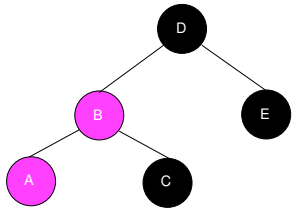
We fix this by pushing the redness of the `C`-`E` level up.



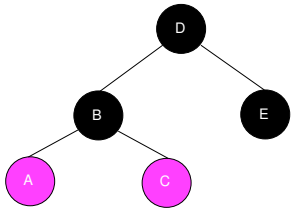
The tree has grown. We recognize this by blackening the root, which increments the entire tree's black height.



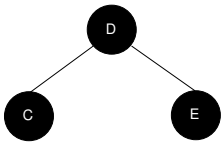
Putting yet another new minimum key, `A`, also incurs a `double-red` violation, but this one cannot be fixed by mere recoloring.

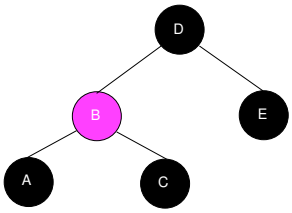


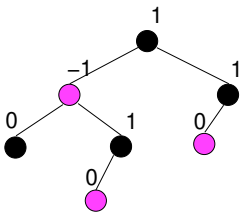
Instead, we rotate about `C`. The result is more balanced, but not a legal red-black tree. . .

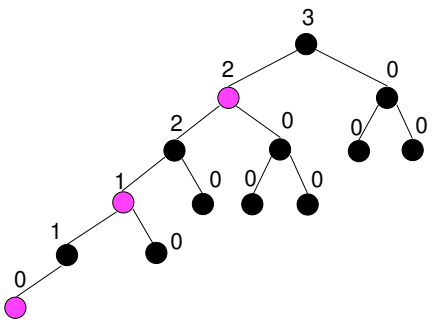


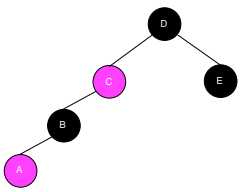
. . . until we recolor. Nodes `A`, `C` make an incomplete red level.

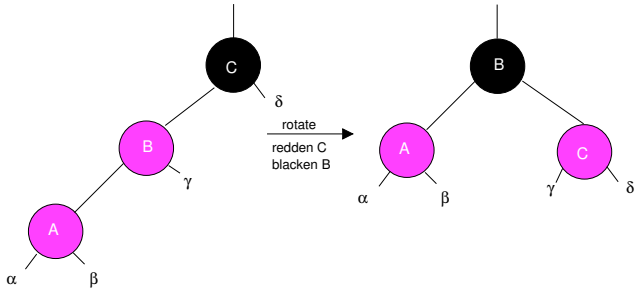




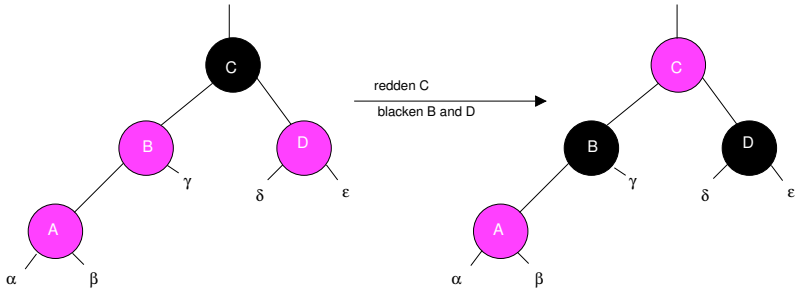




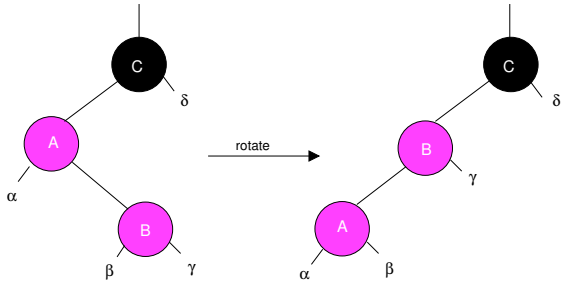


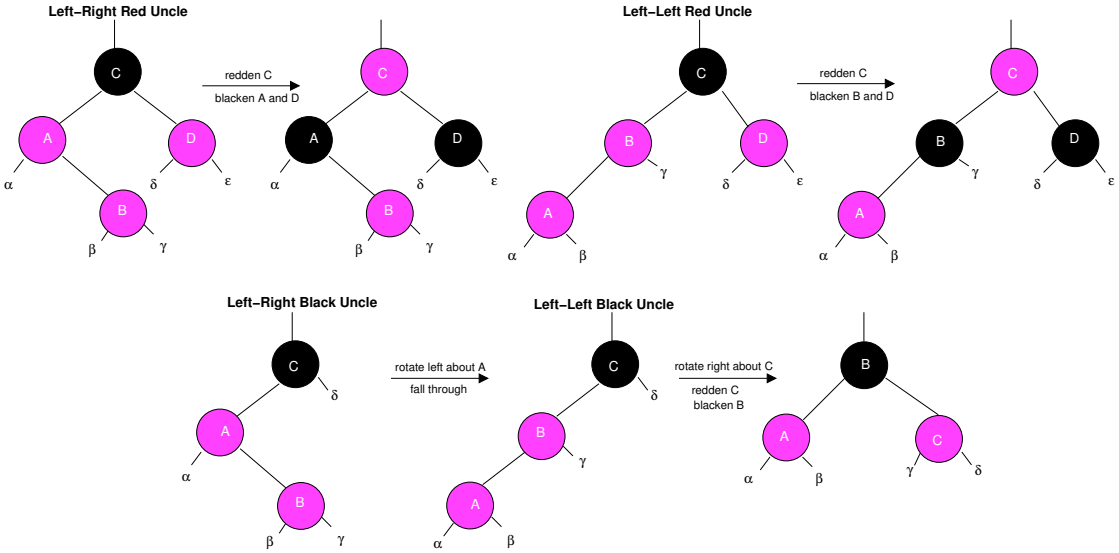


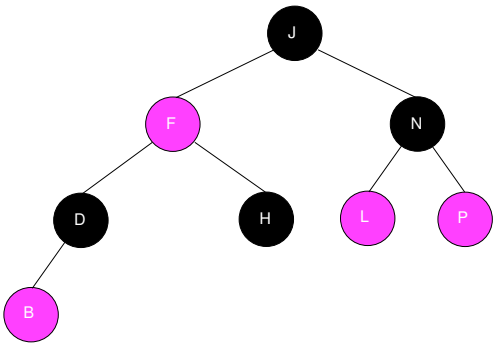
Left-left red uncle

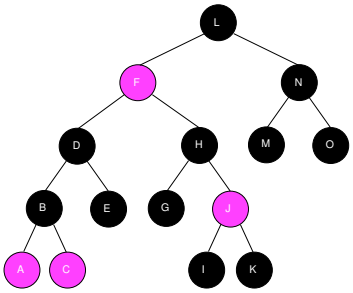


Left-right black uncle

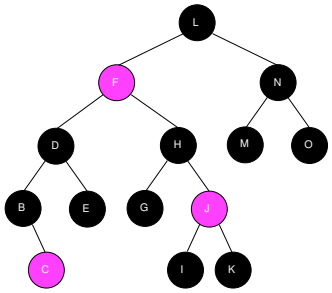



Left–Right Red Uncle
redden C
blacken A and D
Left–Left Red Uncle
redden C
blacken B and D
Left–Right Black Uncle
rotate left about A
fall through
Left–Left Black Uncle
rotate right about C
redden C
blacken B

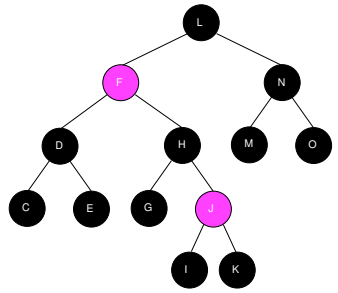

J
F
N
D
H
L
P
B



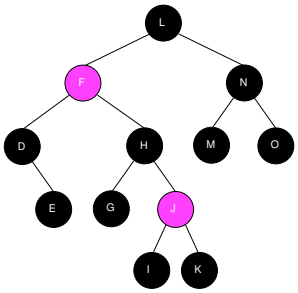
Initially the tree has black height 3. We can delete a red leaf like the A node...



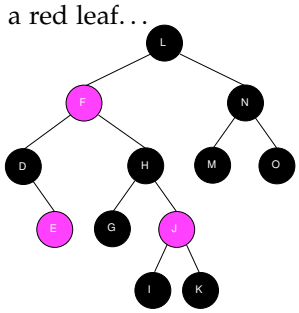
...without any violations to the tree. Similarly, removing the key B means replacing that key with its successor, C, and deleting that successor node. Since that happens to be a red leaf...



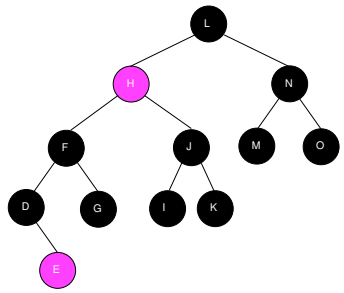
...that deletion also incurs no violation. But suppose now we remove key C, whose node is a leaf but is black.



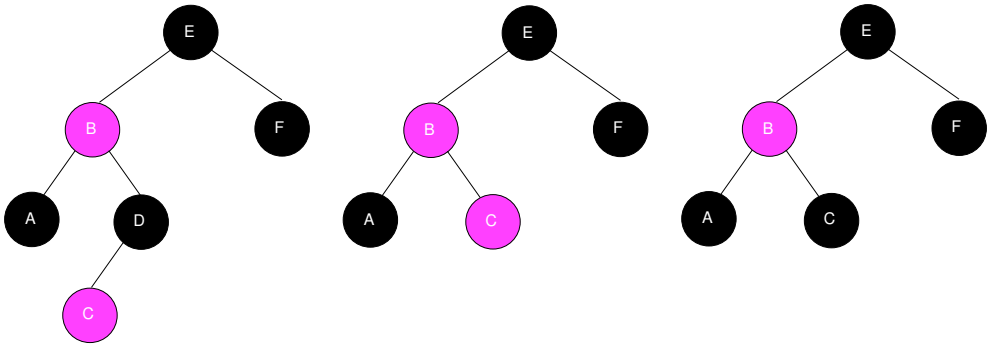
Now the node D has an inconsistent black height—0 on the left and 1 on the right. This needs to be fixed.



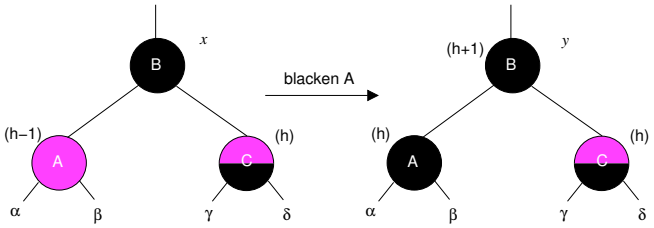
Since D's child with greater black height is itself black and has no red children, we can reduce its black height by reddening it. D's black height is consistent, but it has decreased. Its parent F now has inconsistent black height—1 on the left and 2 on the right.



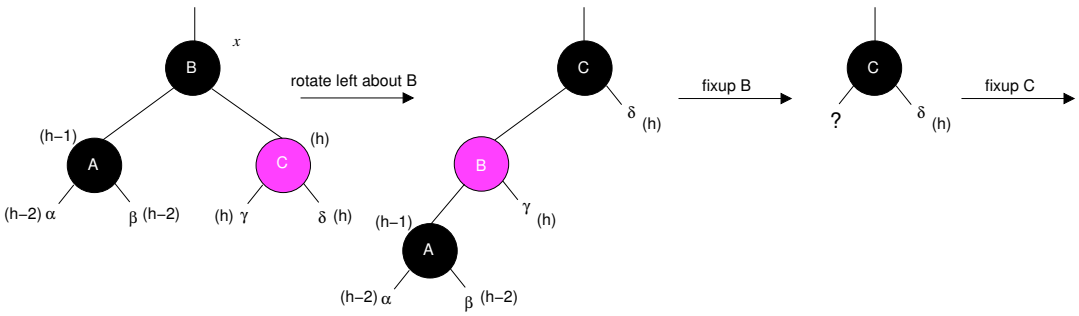
A left-rotation about F and appropriate recoloring produces a replacement subtree rooted at H that not only has a consistent black height but also a black height equal to that of the subtree it replaces.



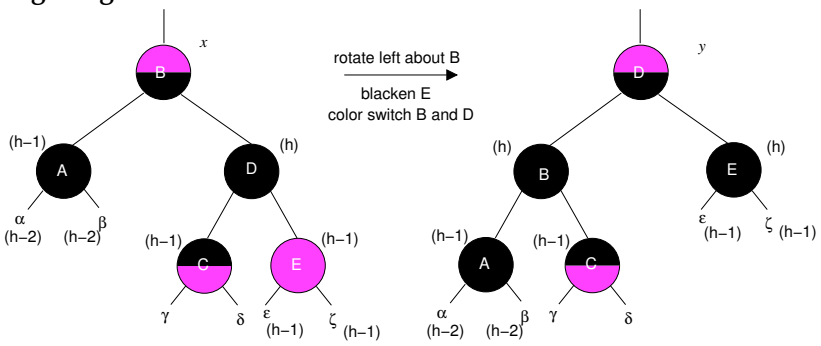
Left red.



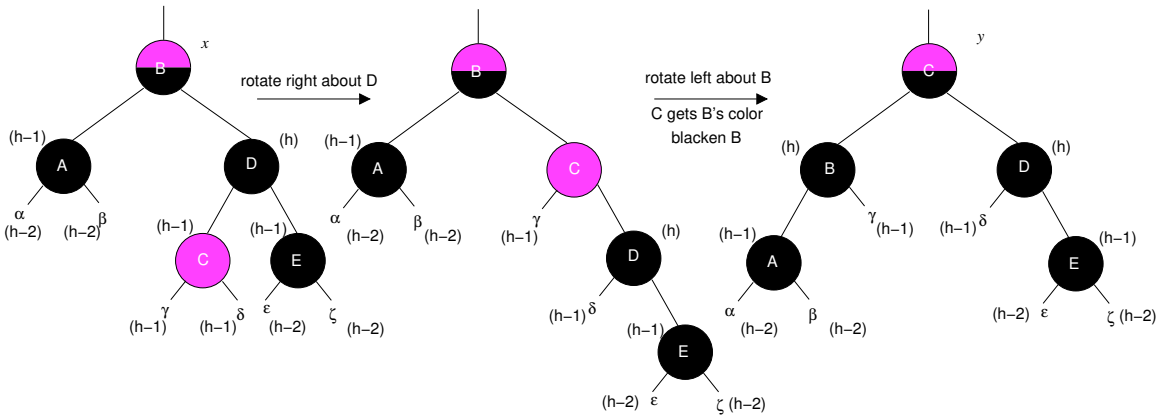
Right red.



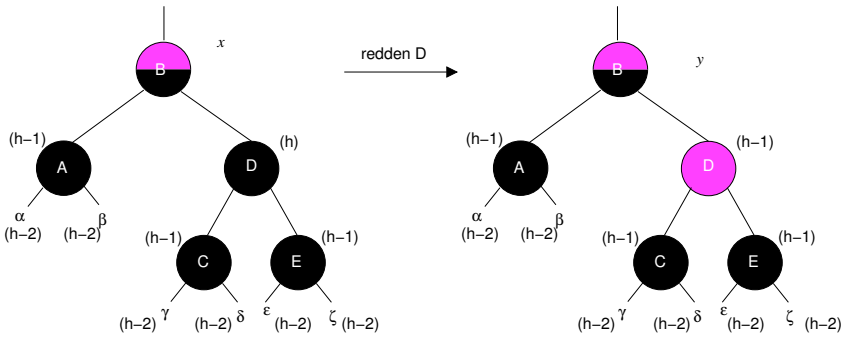
Right-right red.

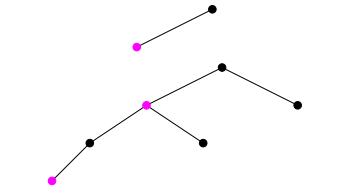
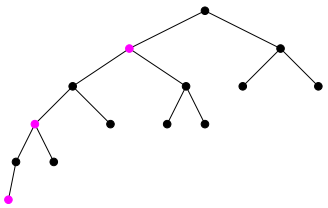
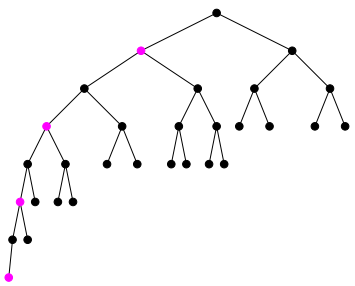


Right-left red.

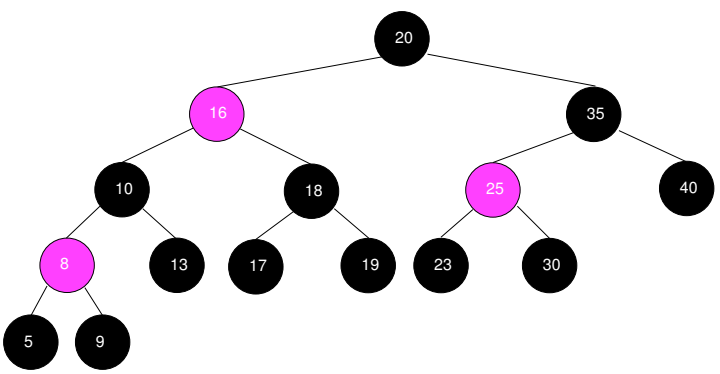


All black.

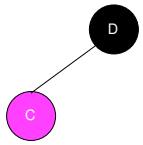


	Black height	Height	Nodes
	1	2	2
	2	4	6
	3	6	14
	4	8	30

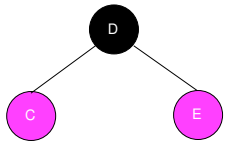
5.5 *Left-leaning red-black trees*



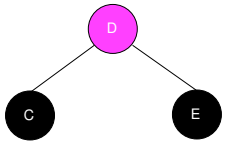
After the initial put we have one node, which we paint black.



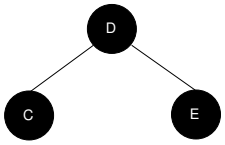
A second put, with a key less than the root, is inserted as a left child and leaf. New leaves are red.



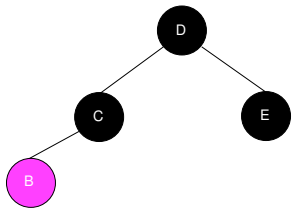
A put with a key greater than the root results in red leaf to the right. This is a *right red* violation.



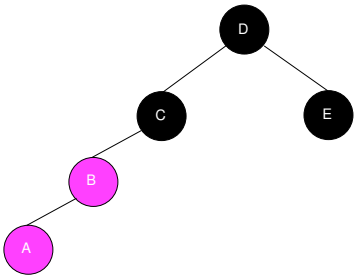
Since the red right child has a red sibling, we fix the violation by pushing the redness up to the parent.



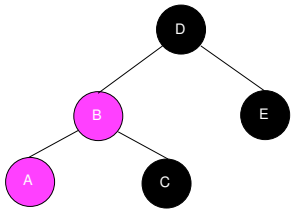
The tree has grown. We recognize this by blackening the root, which increments the entire tree's black height.



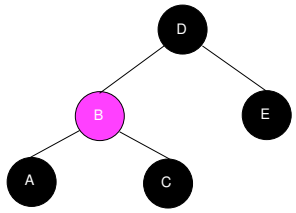
Putting a new minimum key, `B`, means inserting a red leaf off the `C` node. There is no violation.



Putting yet another minimum key, `A`, incurs a *double-red* violation, which cannot be fixed by recoloring.

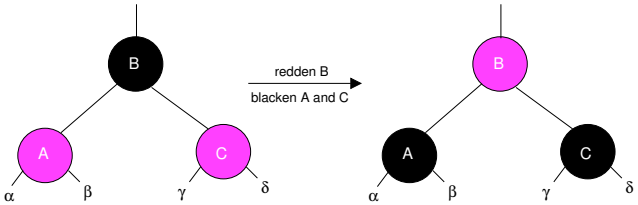


Instead, we rotate about `C` . . .

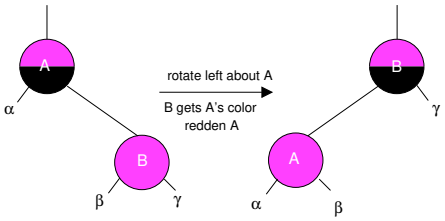


. . . and recolor, though differently from a traditional red-black tree.

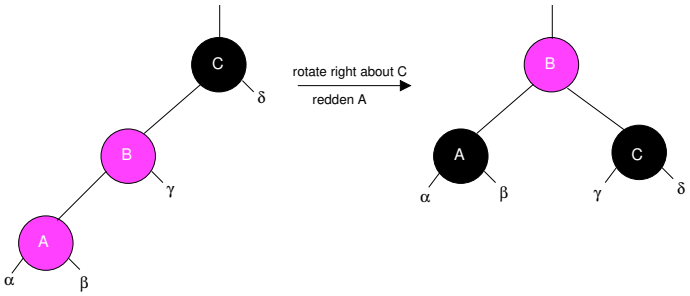
Both red



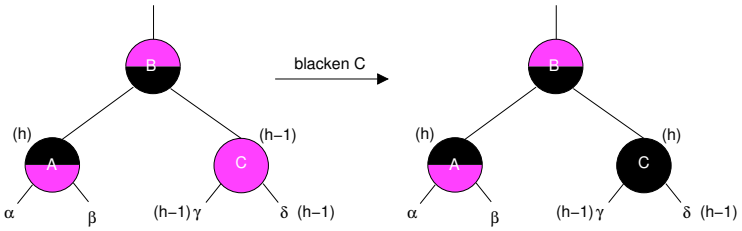
Right red



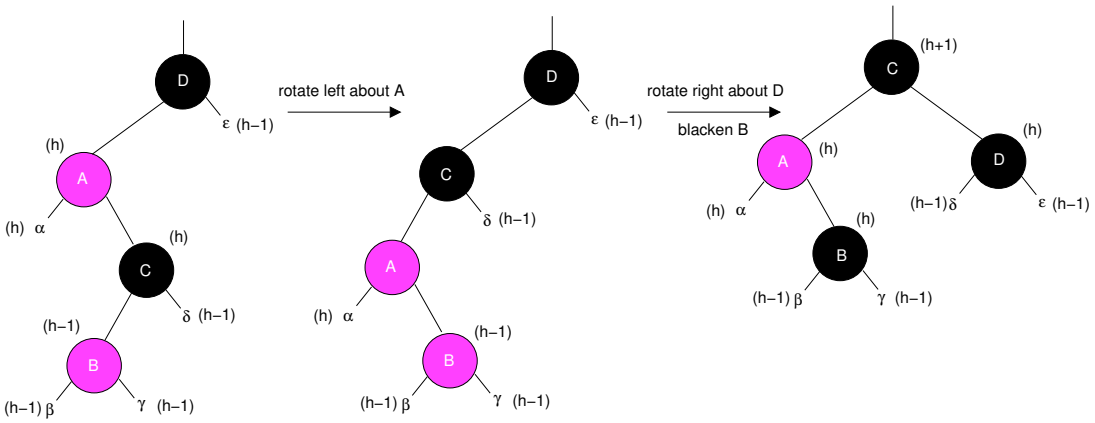
Left-left double-red



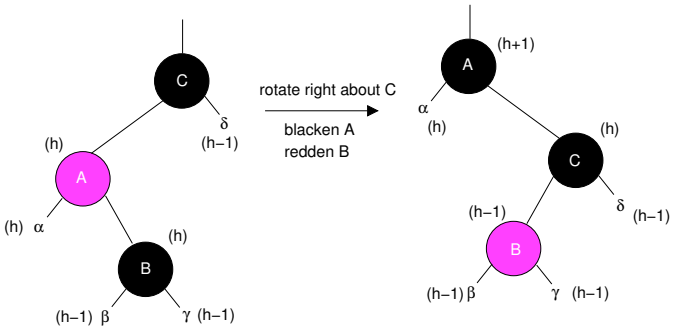
Right deficient, right red.



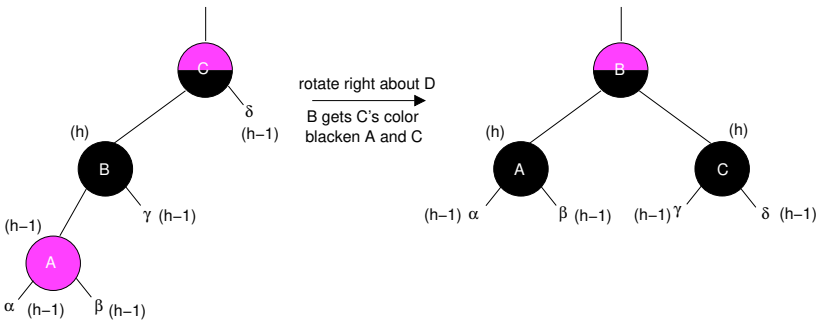
Right deficient, left red, left-right-left red.



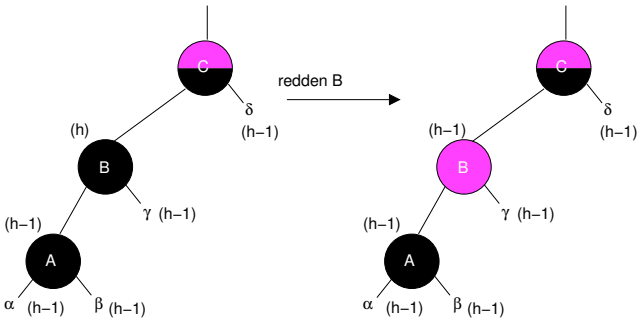
Right deficient, left red, left-right-left black.



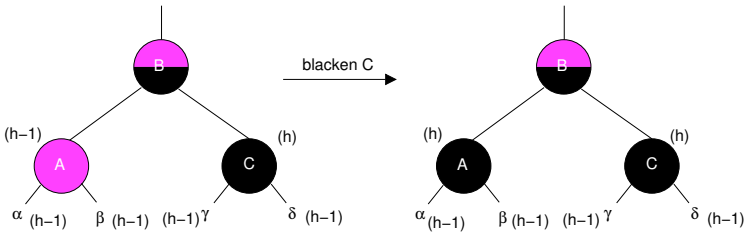
Right deficient, left-left red.



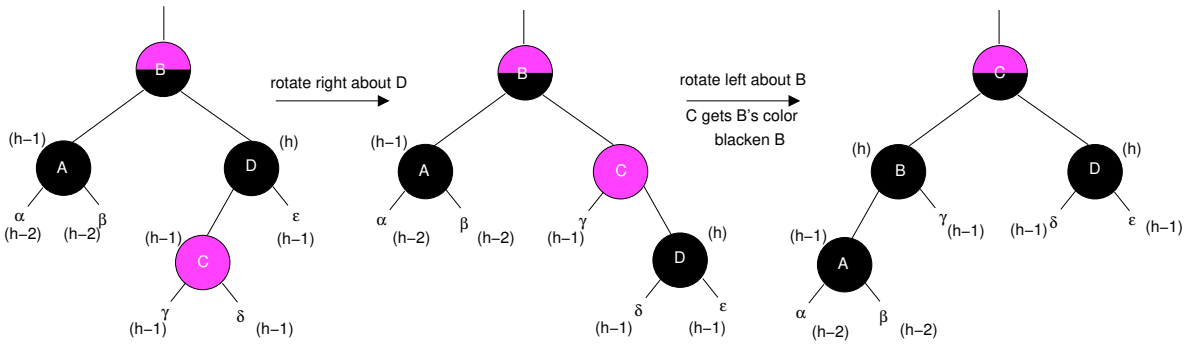
Right deficient, all black.



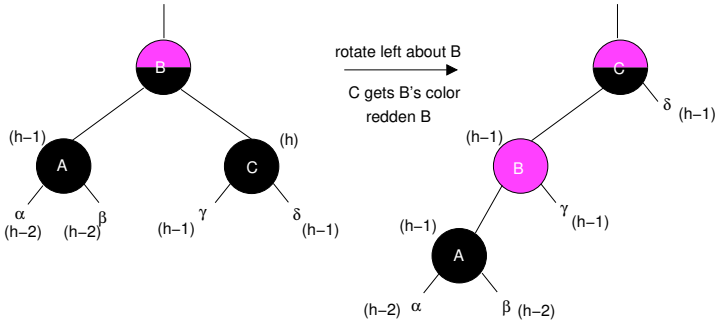
Left deficient, left red.



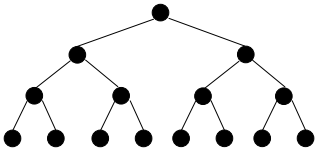
Left deficient, right-left red.



Left deficient, all black.

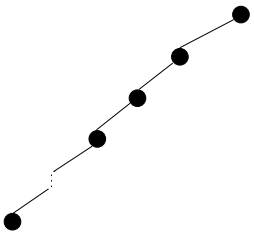


Perfect



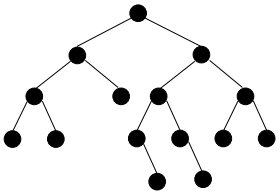
Height: 4
Leaf percentage: 53.5%
Total depth: 49

Worst



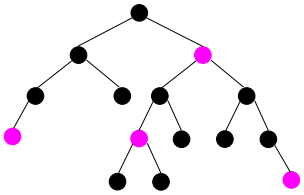
Height: 15
Leaf percentage: 6.7%
Total depth: 120

AVL



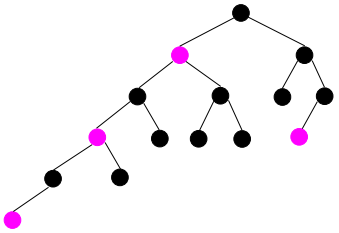
Height: 5
Leaf percentage: 46.7%
Total depth: 51

Red-black



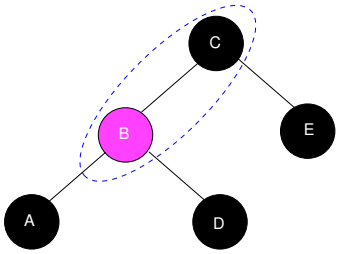
Height: 5
Leaf percentage: 46.7%
Total depth: 52

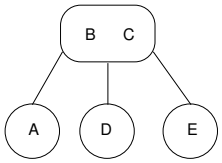
Left-leaning red-black

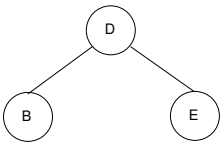


Height: 6
Leaf percentage: 46.7%
Total depth: 53

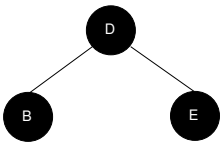
5.6 *B-trees*



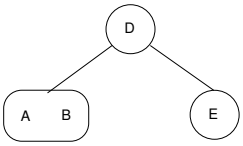




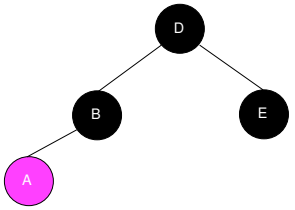
Consider this two-three tree, which happens to have all two-nodes . . .



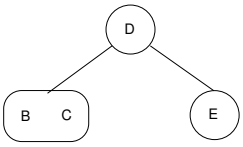
. . . and the equivalent red-black tree, with all black nodes.



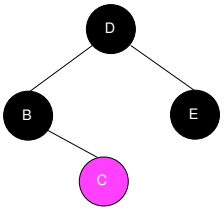
If we insert the key `A`, it is absorbed by the node containing `B`. That node becomes a three-node.



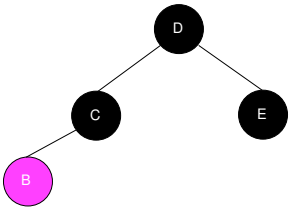
Inserting the same key into the equivalent red-black tree results in a new red leaf.



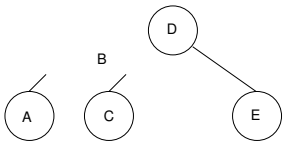
If, instead, we insert the key `C` into the original two-three tree, that new key is absorbed the same way.



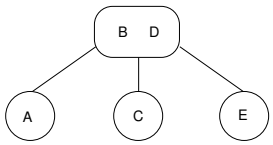
In a red-black tree, the new key is again inserted as a red leaf . . .



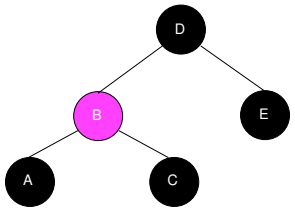
. . . but a left-leaning red-black tree needs to be fixed up by a rotation and recoloring.



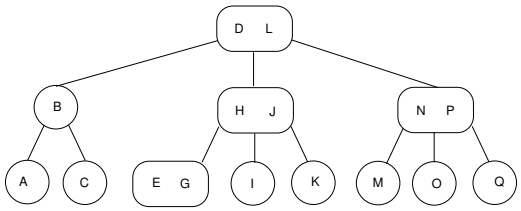
If we take either of the trees from the previous set of examples and add the other key—either `C` after `A` or `A` after `C`—then the three-node splits, resulting in two new two-nodes, plus a key (and value) sent up one level.



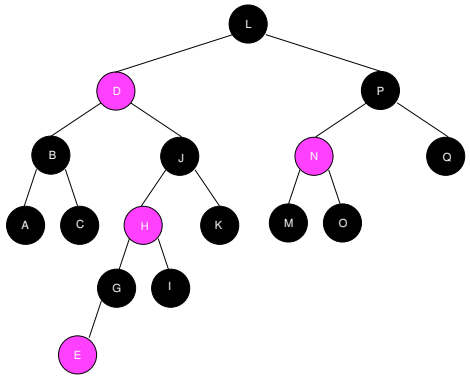
The key `B` is absorbed into the root two-node, which becomes a three-node whose children are the new nodes plus the old right child.



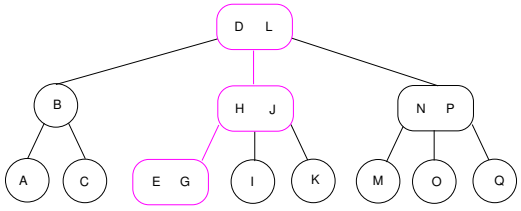
In a left-leaning red-black tree, these situations are handled by "pushing redness up" to the `B` node.



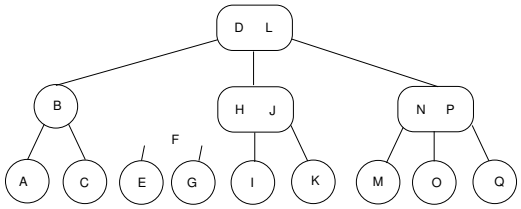
This two-three tree has a path from root to a leaf on which all nodes are three-nodes.



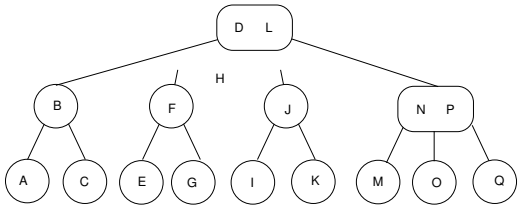
The equivalent path in the equivalent left-leaning red-black tree alternates black and red nodes.



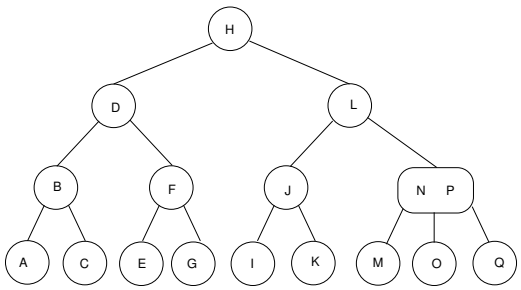
A call to `put()` with the new key `F` follows that path, finding `F`'s would-be place in the three-node with `E` and `G`.



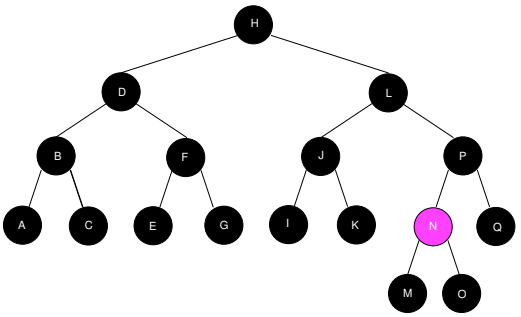
The three-node leaf is split. The "shards"—two new two-nodes and a key (and a value)—are sent up to the parent level.



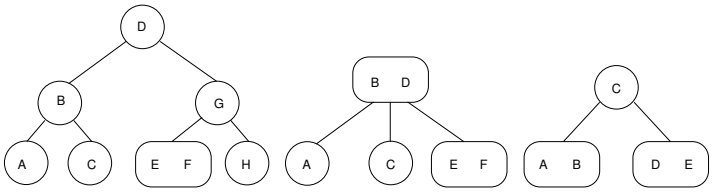
The mid-level three-node that would absorb the shards of the leaf-level split also splits, and sends its shards up to the root.

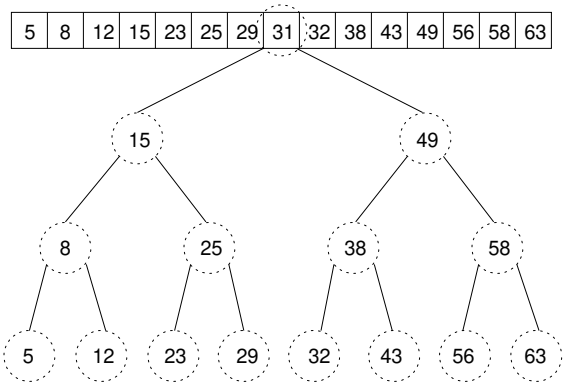


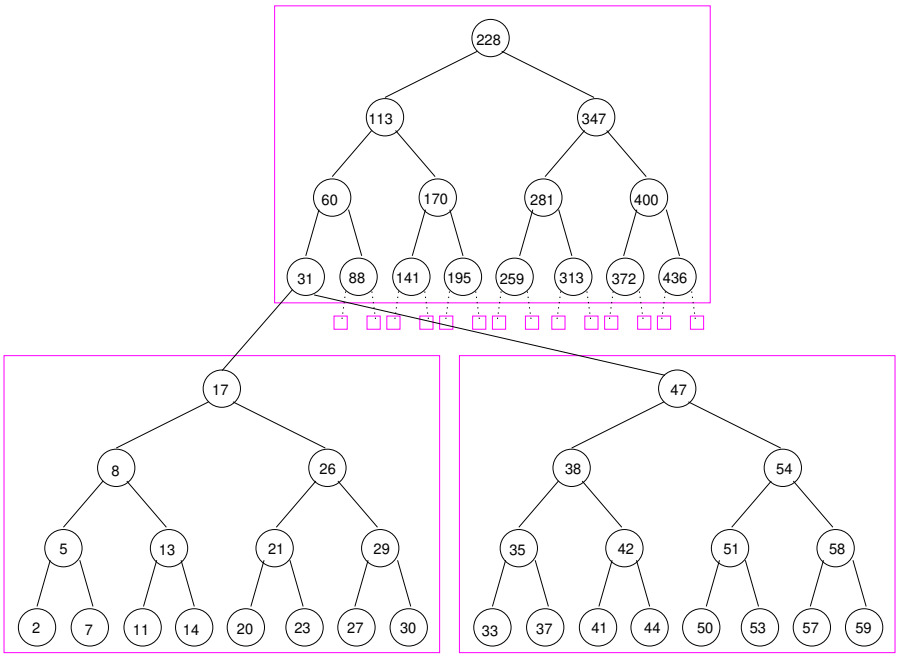
The shards of the root-level split are assembled into a two-node that becomes the new root.

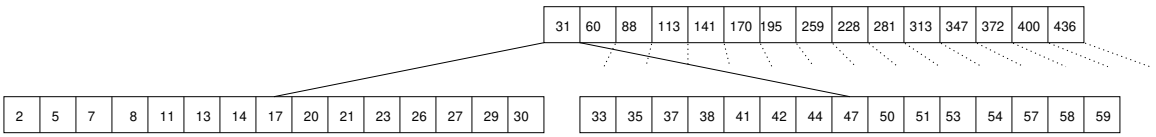


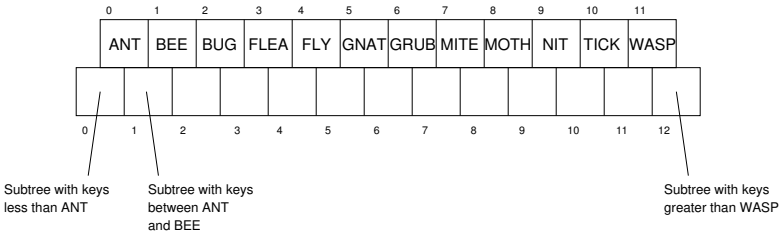
The left-leaning red-black tree that results from the same insertion has black height one higher than before.

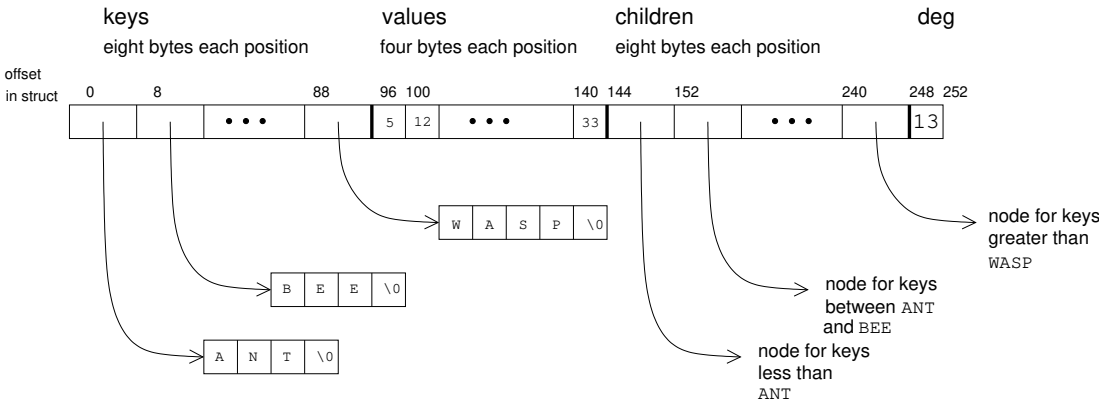


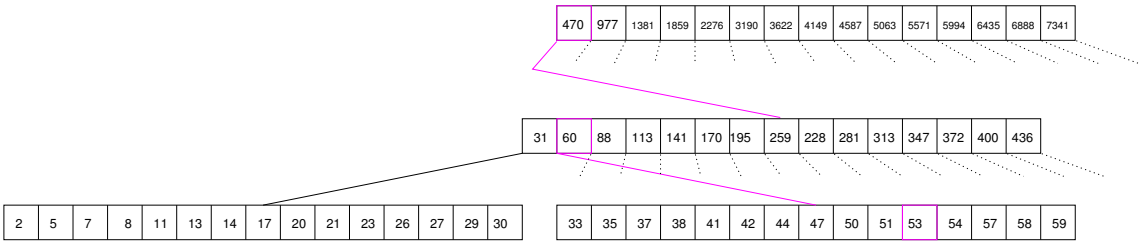

5 8 12 15 23 25 29 31 32 38 43 49 56 58 63
15
49
8
25
38
58
5
12
23
29
32
43
56
63

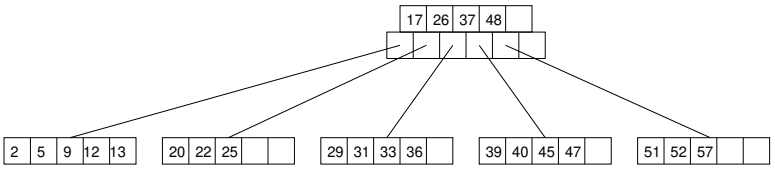

228
113
347
60
170
281
400
31
88
141
195
259
313
372
436
17
47
8
26
38
54
5
13
21
29
35
42
51
58
2
7
11
14
20
23
27
30
33
37
41
44
50
53
57
59

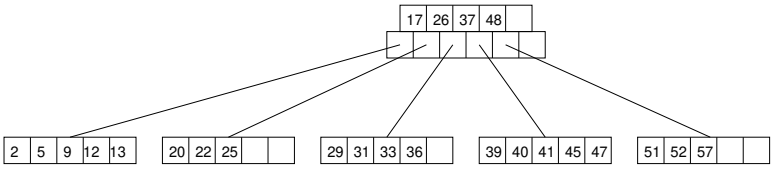

31 60 88 113 141 170 195 259 228 281 313 347 372 400 436
2 5 7 8 11 13 14 17 20 21 23 26 27 29 30
33 35 37 38 41 42 44 47 50 51 53 54 57 58 59

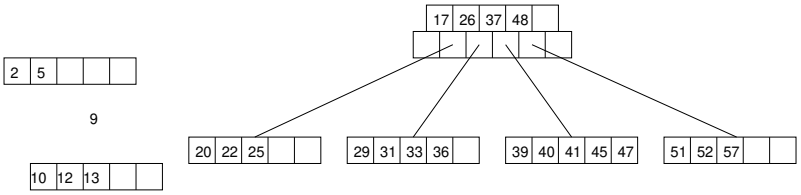

0 1 2 3 4 5 6 7 8 9 10 11
ANT BEE BUG FLEA FLY GNAT GRUB MITE MOTH NIT TICK WASP
0 1 2 3 4 5 6 7 8 9 10 11 12
Subtree with keys less than ANT
Subtree with keys between ANT and BEE
Subtree with keys greater than WASP

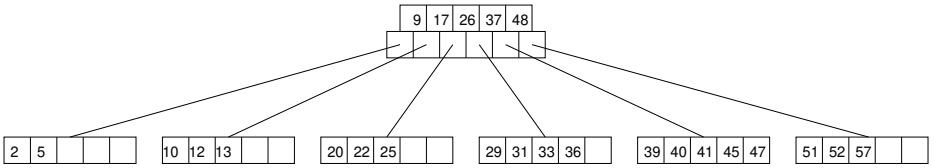

keys
eight bytes each position
values
four bytes each position
children
eight bytes each position
deg
offset in struct
0 8 88 96 100 140 144 152 240 248 252
5 12 33 13
W A S P \0
B E E \0
A N T \0
node for keys greater than WASP
node for keys between ANT and BEE
node for keys less than ANT

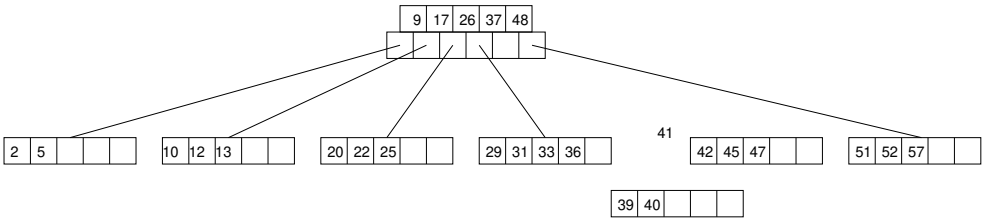

470 977 1381 1859 2276 3190 3622 4149 4587 5063 5571 5994 6435 6888 7341
31 60 88 113 141 170 195 259 228 281 313 347 372 400 436
2 5 7 8 11 13 14 17 20 21 23 26 27 29 30
33 35 37 38 41 42 44 47 50 51 53 54 57 58 59

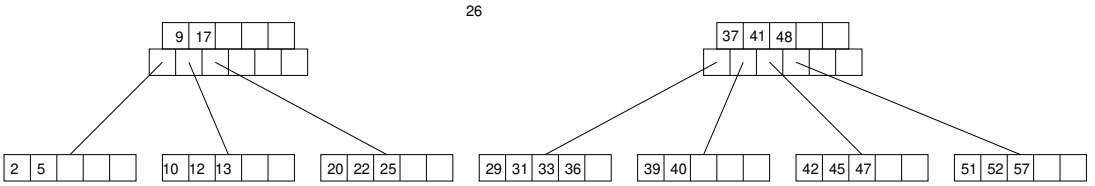

17 26 37 48
2 5 9 12 13
20 22 25
29 31 33 36
39 40 45 47
51 52 57

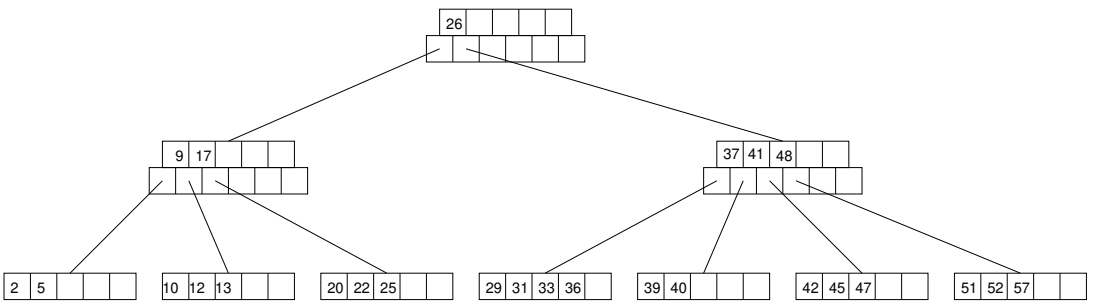

17 26 37 48
2 5 9 12 13
20 22 25
29 31 33 36
39 40 41 45 47
51 52 57

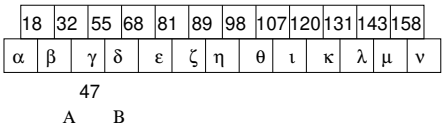

17 26 37 48
2 5
9
10 12 13
20 22 25
29 31 33 36
39 40 41 45 47
51 52 57


9 17 26 37 48
2 5
10 12 13
20 22 25
29 31 33 36
39 40 41 45 47
51 52 57

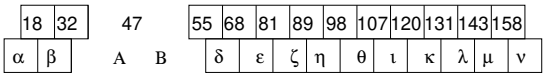

9 17 26 37 48
2 5
10 12 13
20 22 25
29 31 33 36
41
42 45 47
51 52 57
39 40


26
9 17
2 5
10 12 13
20 22 25
37 41 48
29 31 33 36
39 40
42 45 47
51 52 57

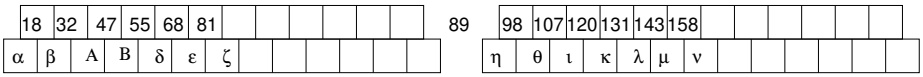

26
9 17
2 5
10 12 13
20 22 25
37 41 48
29 31 33 36
39 40
42 45 47
51 52 57



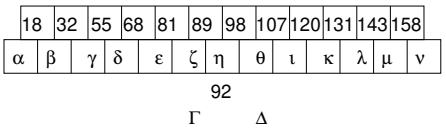
Suppose that the shard key is 47 and that its least upper bound is in postion 2.



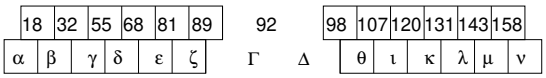
The child it came from, γ, is deleted, and 47 with its shard children are virtually inserted to make an overfull node.



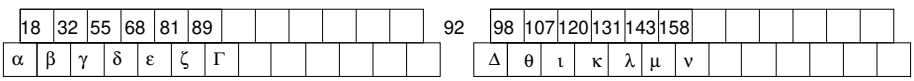
That virtual node is split into two half-filled nodes with key 89 in the middle.



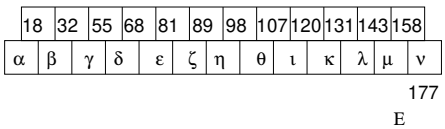
Suppose the shard key is 92 and that its least upper bound position is half the number of keys.



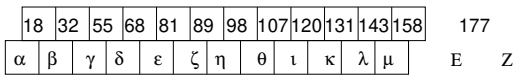
The child it came from, η, is deleted, and 92 with its shard children are virtually inserted to make an overfull node.



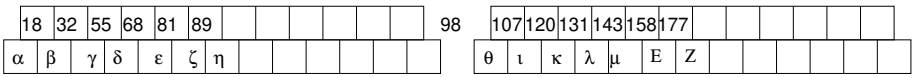
But forget about a virtual node—the real node is split exactly in half, and 92 is the shard key at this level as well.



Suppose the shard key is 177 and that it's the extreme case, greater than all of the keys at this node.

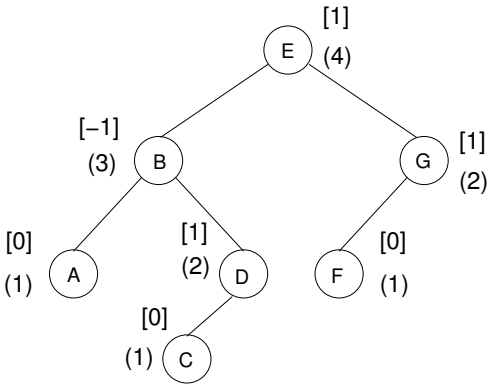


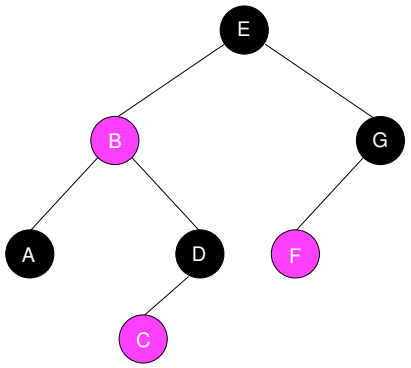
The child it came from, ν, is deleted, and 177 with its shard children are virtually inserted to make an overfull node.

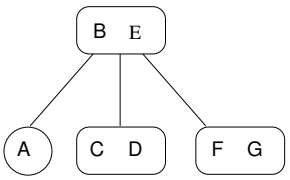


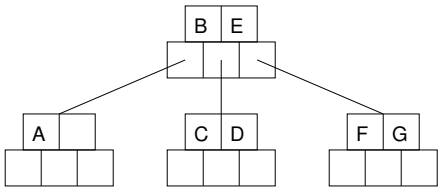
That virtual node is split into two half-filled nodes with key 98 in the middle.

5.7 *Chapter summary*



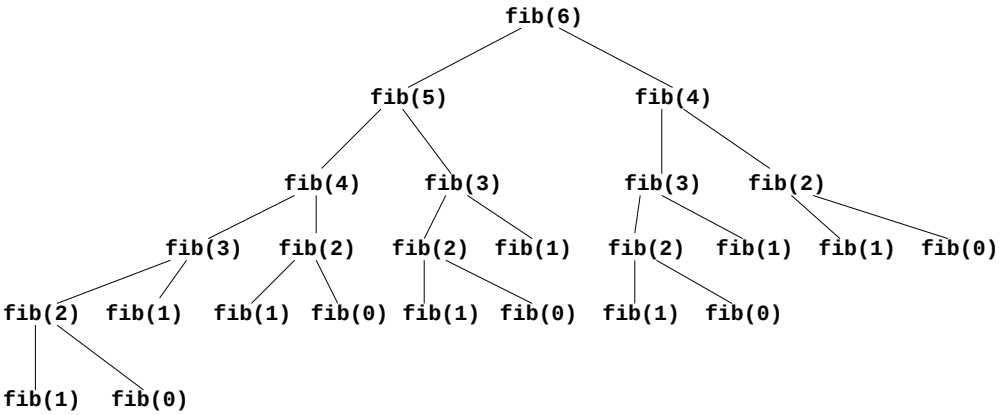


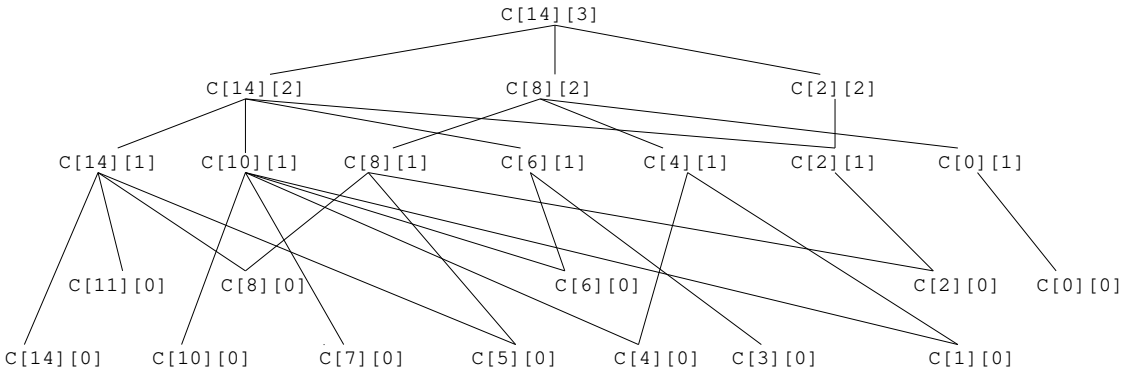




6
Dynamic programming

6.1 Overlapping subproblems





coin values

6	**3**	0	1	2	1	1	2	1	2	2	2	2	3	2	3	3
4	**2**	0	1	2	1	1	2	2	2	2	3	3	3	3	4	4
3	**1**	0	1	2	1	2	3	2	3	4	3	4	5	4	5	6
1	**0**	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**

amounts

```
def make_change(amount, denoms):
    fewest_coins = [[None for j in range(len(denoms))]
                    for i in range(amount + 1)]
    coins_to_take = [[None for j in range(len(denoms))]
                     for i in range(amount + 1)]

    for j in range(len(denoms)) :
        fewest_coins[0][j] = 0
        coins_to_take[0][j] = 0
    for i in range(amount + 1) :
        fewest_coins[i][0] = i
        coins_to_take[i][0] = i

    for i in range(1,amount + 1) :  # For each sub-amount
        for j in range(1,len(denoms)) :  # For each range of denominations

            # Initially assume the best we can do is take 0 coins
            # of the current (jth) denomination
            best_coins = fewest_coins[i][j-1]
            best_take = 0

            k = 1
            while k * denoms[j] <= i :
                coins = k + fewest_coins[i - k * denoms[j]][j-1]
                if coins <= best_coins :
                    best_coins = coins
                    best_take = k
                k += 1
            # Record the smallest number of coins and how many of
            # this denomination to get that number of coins
            fewest_coins[i][j] = best_coins
            coins_to_take[i][j] = best_take
```

3	0/0	1/0	2/0	1/0	1/0	2/0	1/1	2/1	2/0	2/1	2/1	3/1	2/2	3/2	3/1
2	0/0	1/0	2/0	1/0	1/1	2/1	2/0	2/1	2/2	3/2	3/1	3/2	3/3	4/3	4/2
1	0/0	1/0	2/0	1/1	2/1	3/1	2/2	3/2	4/2	3/3	4/3	5/3	4/4	5/4	6/4
0	0/0	1/1	2/2	3/3	4/4	5/5	6/6	7/7	8/8	9/9	10/10	11/11	12/12	13/13	14/14
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**

amounts

6.2 *Three problems*

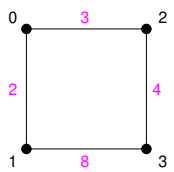
r	e	d	b	l	a	c	k	t	r	e	e
0	1	2	3	4	5	6	7	8	9	10	11

0	1	4	5	8	10
r	e	l	a	t	e
0	1	2	3	4	5

d	a	t	a	s	t	r	u	c	t	u	r	e	s
0	1	2	3	4	5	6	7	8	9	10	11	12	13

a	l	g	o	r	i	t	h	m	s
0	1	2	3	4	5	6	7	8	9

6.3 *Elements of dynamic programming*

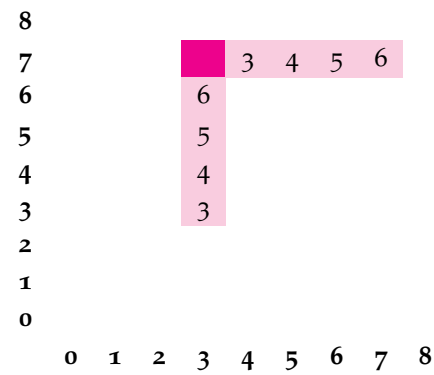


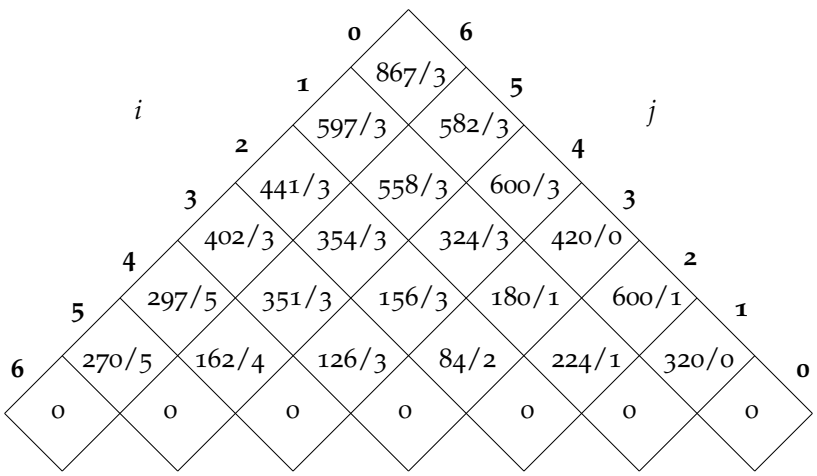
6.4 *Three solutions*

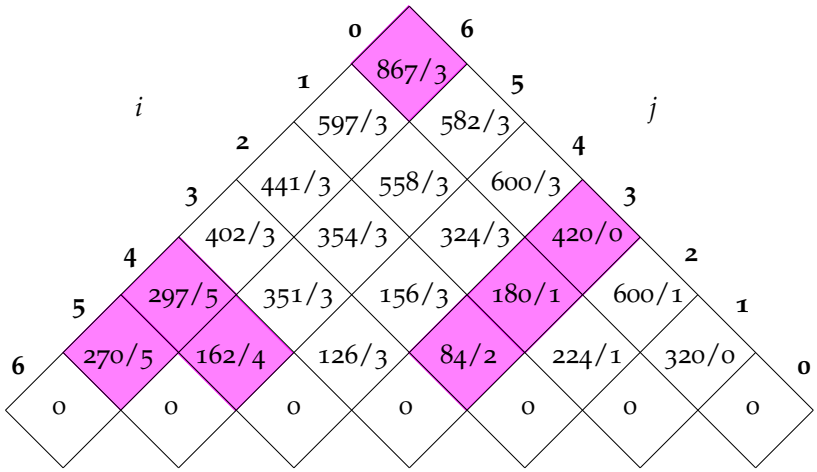
items												
	4	0/S	150/S	150/S	150/S	150/S	190/T	200/S	220/S	220/S	220/S	240/T
	3	0/S	150/S	150/S	150/S	150/S	150/S	200/T	220/S	220/S	220/S	220/S
	2	0/S	150/S	150/S	150/S	150/S	150/S	150/S	220/T	220/T	220/T	220/T
	1	0/S	150/S	150/S	150/S	150/S	150/S	150/S	150/S	150/S	150/S	150/S
	0	0/S	150/T	150/T	150/T	150/T	150/T	150/T	150/T	150/T	150/T	150/T
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

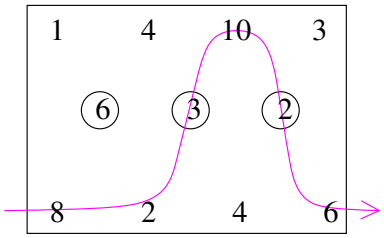
capacities

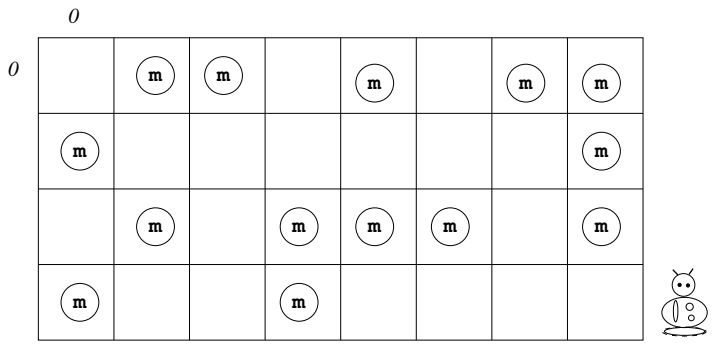
s	**10**	0	0/1	1/1	2/1	2/1	3/0	3/-1	3/-1	3/-1	3/-1	3/1	3/1	3/1	3/1	4/0
m	**9**	0	0/1	1/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	3/1	3/1	3/1	3/1	3/1
h	**8**	0	0/1	1/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	3/1	3/1	3/1	3/1	3/1
t	**7**	0	0/1	1/1	2/0	2/-1	2/-1	2/0	2/1	2/1	2/1	3/0	3/-1	3/-1	3/-1	3/-1
i	**6**	0	0/1	1/1	1/1	1/1	1/1	1/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1	2/1
r	**5**	0	0/1	1/1	1/1	1/1	1/1	1/1	2/0	2/-1	2/-1	2/-1	2/-1	2/0	2/-1	2/-1
o	**4**	0	0/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
g	**3**	0	0/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
l	**2**	0	0/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
a	**1**	0	0/1	1/0	1/-1	1/0	1/-1	1/-1	1/-1	1/-1	1/-1	1/-1	1/-1	1/-1	1/-1	1/-1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
			d	a	t	a	s	t	r	u	c	t	u	r	e	s

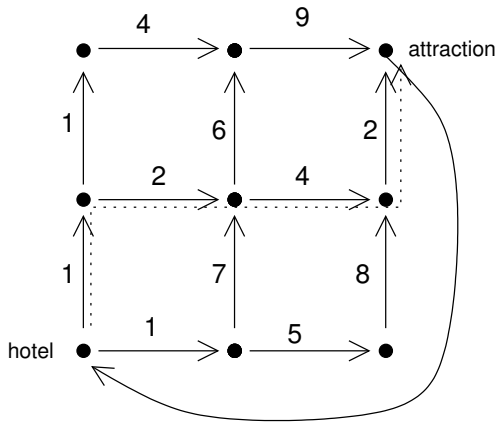





i
j
0 6
867/3
1 5
597/3 582/3
2 4
441/3 558/3 600/3
3 3
402/3 354/3 324/3 420/0
4 2
297/5 351/3 156/3 180/1 600/1
5 1
270/5 162/4 126/3 84/2 224/1 320/0
6 0
0 0 0 0 0 0 0

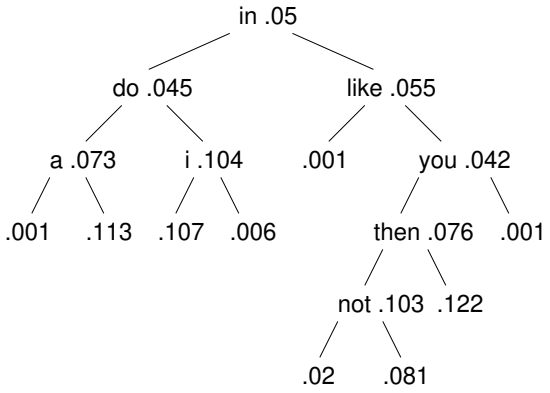

1 4 10 3
6 3 2
8 2 4 6

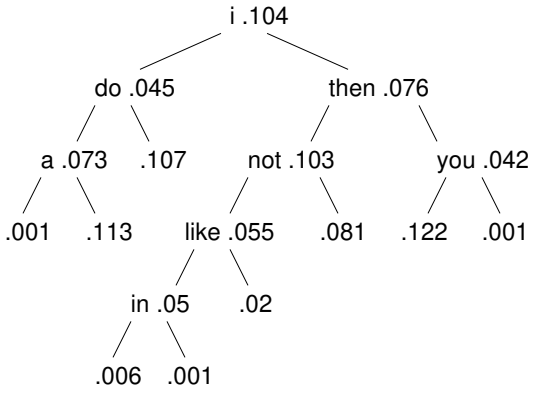

0
0
m

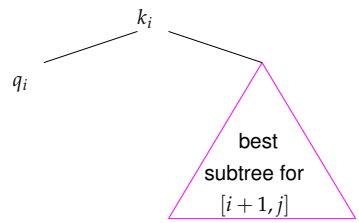


6.5 *Optimal binary search trees*

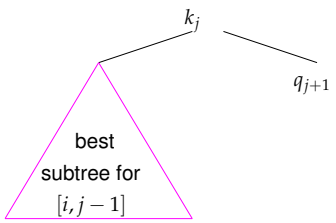
Key or miss event	combined frequency
{ }	0
a	59
{ am and anywhere are be boat box car could dark }	92
do	36
{ eat eggs fox goat good green ham here house }	86
i	84
{ if let }	5
in	40
{ }	0
like	44
{ may me mouse }	16
not	83
{ on or rain same say see so thank that the }	65
then	61
{ there they train tree try will with would }	99
you	34
{ }	0



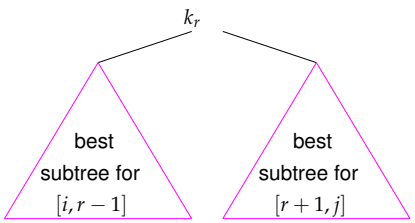




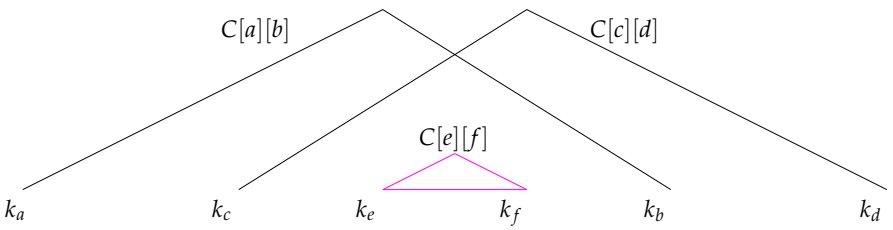
$q_i + C[i+1][j]$

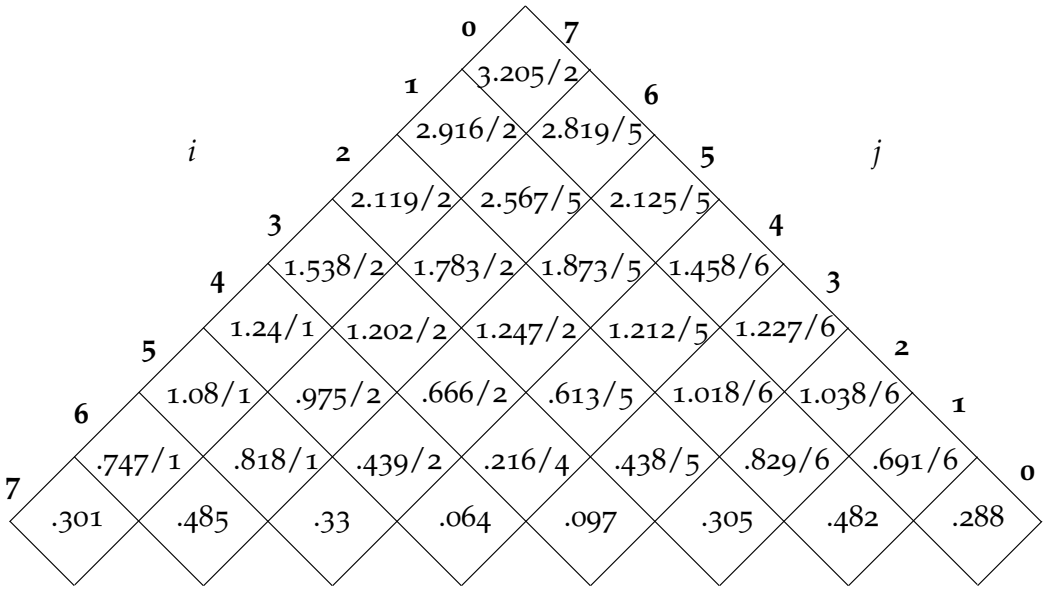


$C[i][j-1] + q_{j+1}$

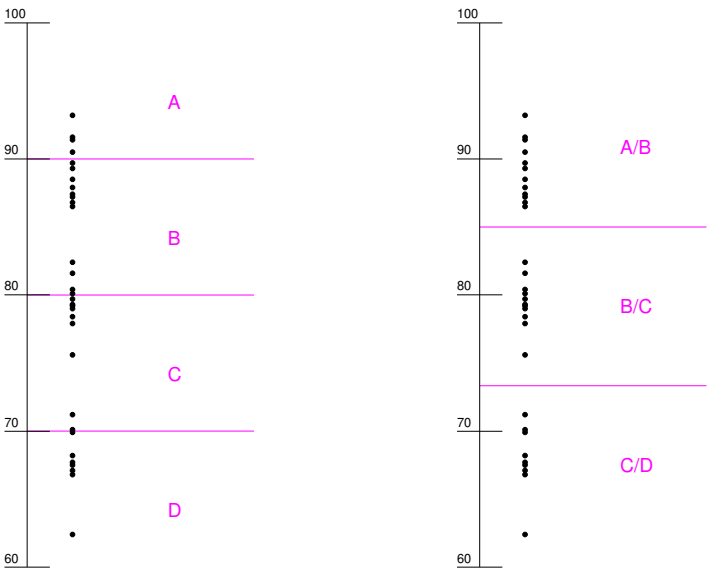


$C[i][r-1] + C[r+1][j]$

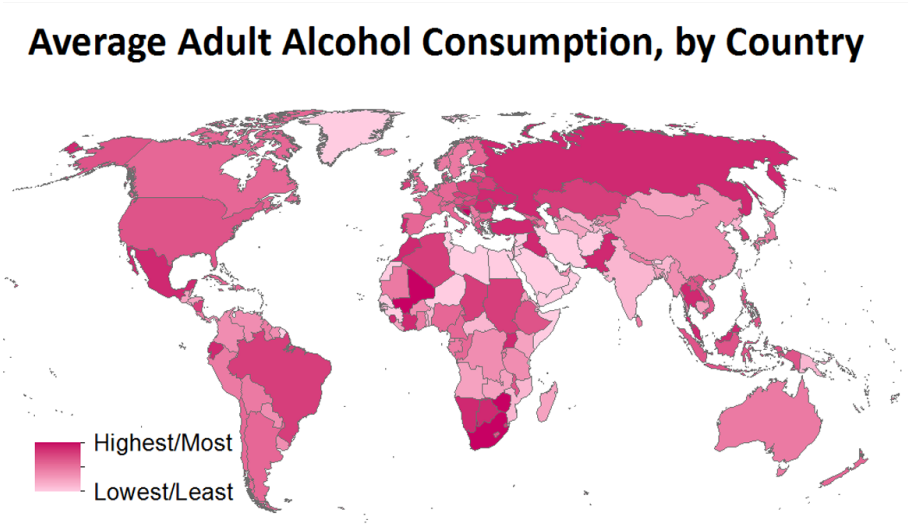


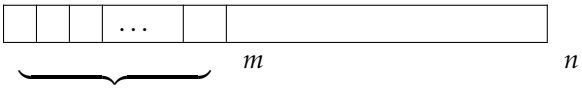


6.6 *Natural-breaks classification*

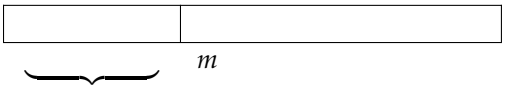


http://www.science.smith.edu/sal/2014/02/19/february-mystery-map-solution-revealed/

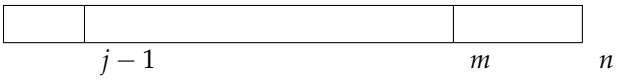




m points, each in their own class



m points, all in one class



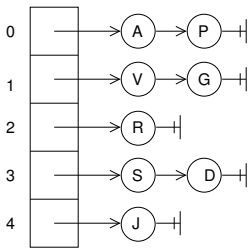
candidate positions to break interval

4				0.0/4	0.1/4	1.6/4	7.0/6	8.4/6	10.0/6	11.8/6	22.7/10	23.9/10
3			0.0/3	0.1/3	5.2/3	7.0/4	16.4/5	19.4/6	21.0/6	22.7/6	45.6/6	
2		0.0/2	0.1/2	5.5/3	10.7/3	17.9/3	41.4/3	66.1/4	79.6/4	91.5/4		
1	0.0/0	0.1/0	5.5/0	37.2/0	89.4/0	144.9/0	234.1/0	329.5/0	415.5/0			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
	72.7	73.2	75.8	80.4	83.6	85.3	88.7	90.4	91.1	91.6	95.8	97.3

6.7 *Chapter summary*

7
Hash tables

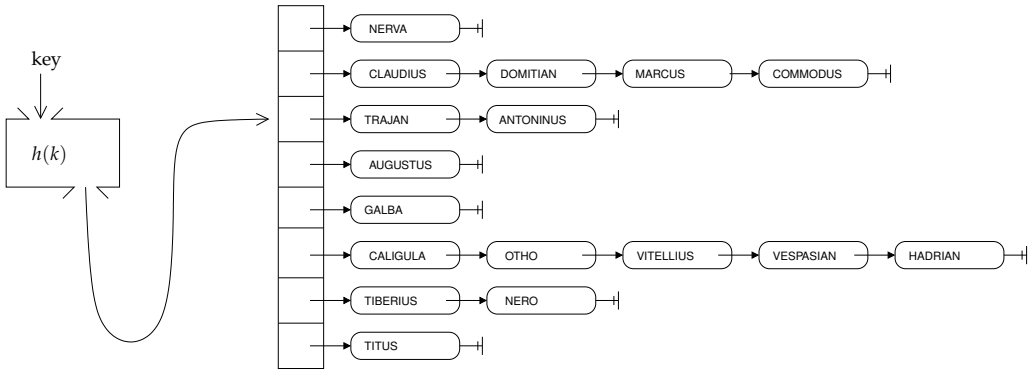
7.1 Hash table basics

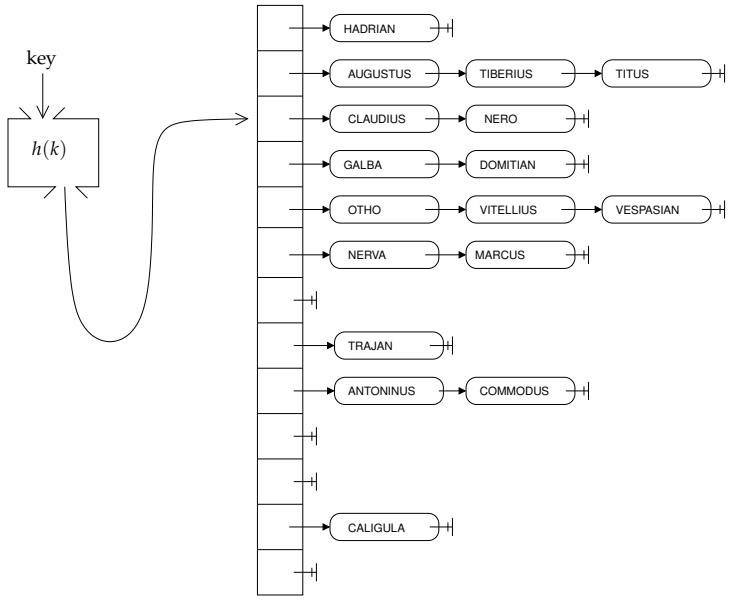


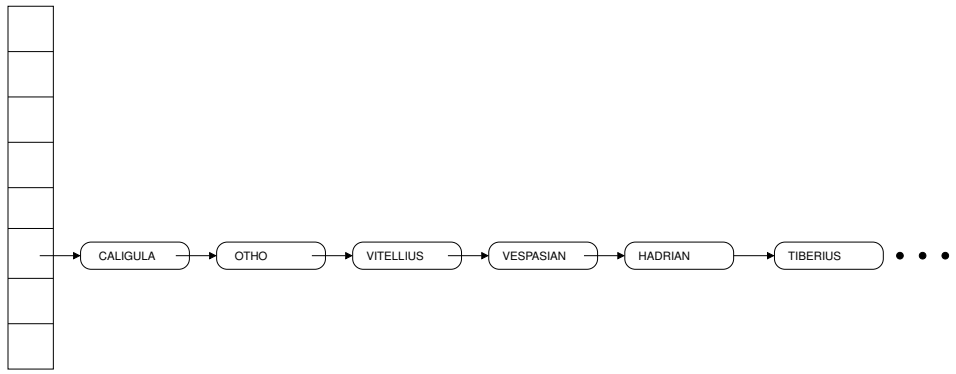
S		J	V		A			D	P	R	G

A		P	D	R	S	G		V	J			

7.2 Separate chaining

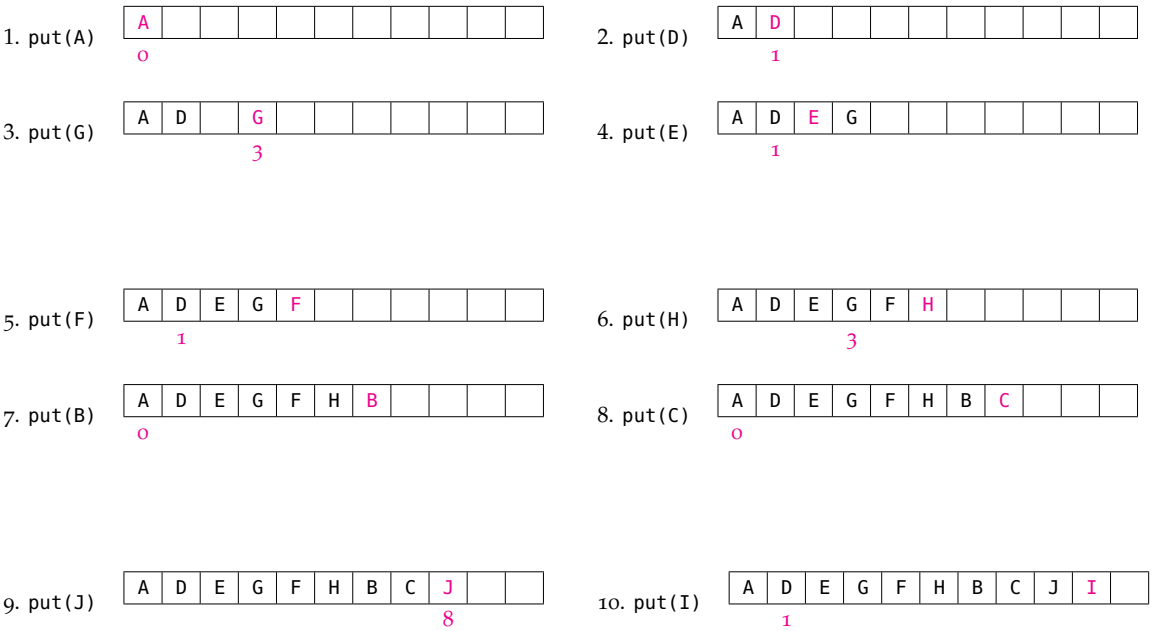


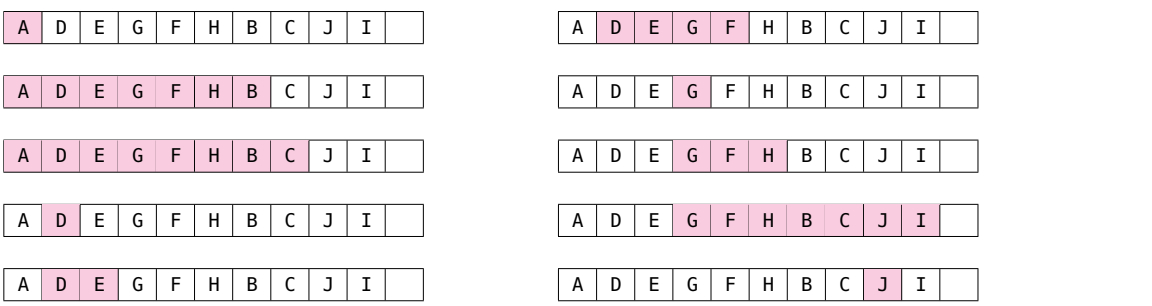


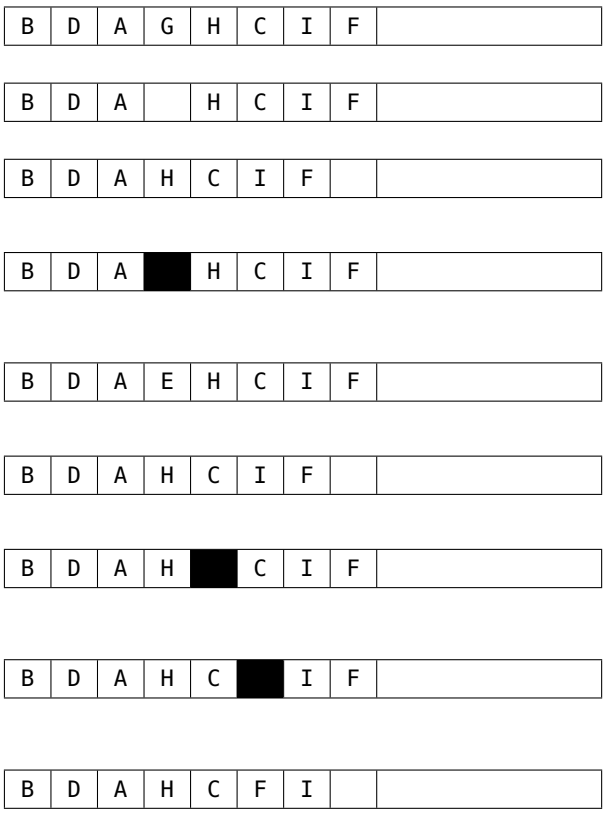


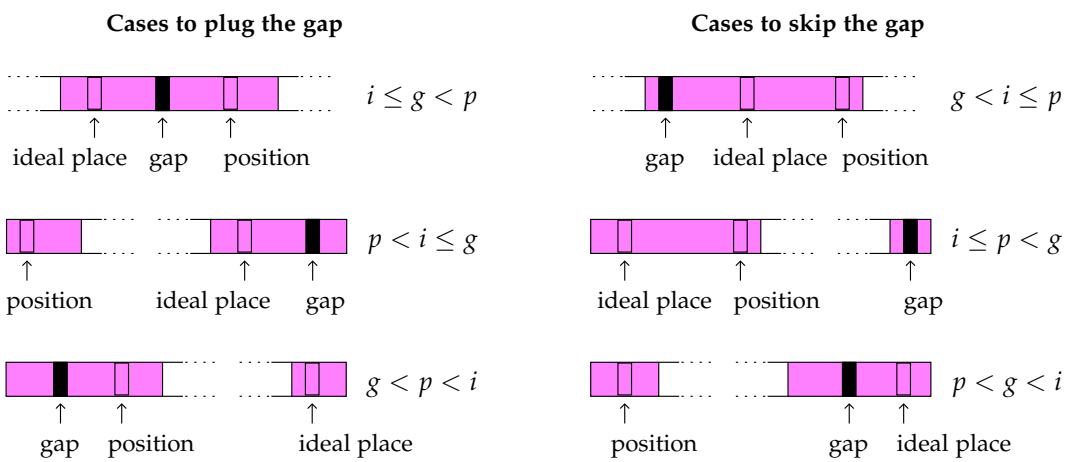
$$
\begin{array}{lll}
 & O(1) & c_0 \\
 & O(1) & c_0 \\
 & O(1) & c_0 \\
 & \vdots & \\
 & O(1) & c_0 \\
\text{rehash} \longrightarrow & O(n) & c_1 + c_2 n \\
 & O(1) & c_0 \\
 & \vdots &
\end{array}
\qquad
\begin{array}{rcl}
T(n) & = & (n-1)c_0 + c_1 + c_2 m + c_3 n \\
 & = & (n-1)c_0 + c_1 + c_2 \frac{n}{\alpha} + c_3 n \\
 & = & d_0 + d_1 n \\
 & = & \Theta(n)
\end{array}
$$

7.3 *Open addressing*

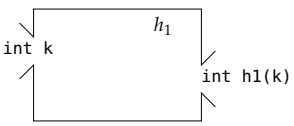


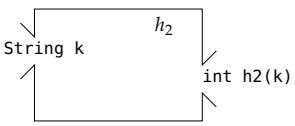


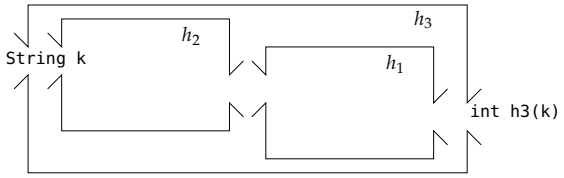

B D A G H C I F
B D A H C I F
B D A H C I F
B D A H C I F
B D A E H C I F
B D A H C I F
B D A H C I F
B D A H C I F
B D A H C F I


Cases to plug the gap
Cases to skip the gap
$i \le g < p$
ideal place gap position
$g < i \le p$
gap ideal place position
$p < i \le g$
position ideal place gap
$i \le p < g$
ideal place position gap
$g < p < i$
gap position ideal place
$p < g < i$
position gap ideal place

7.4 *Hash functions*

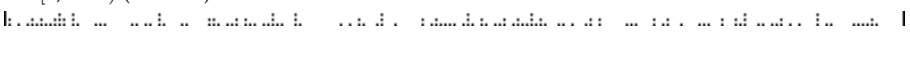
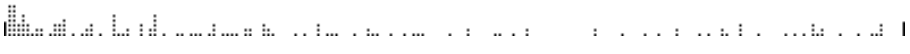
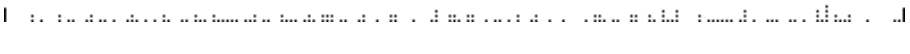
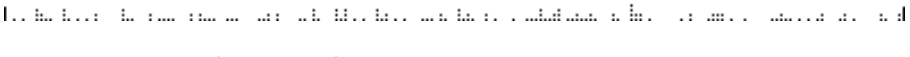
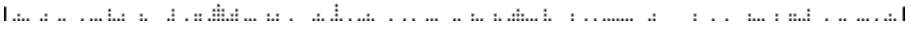
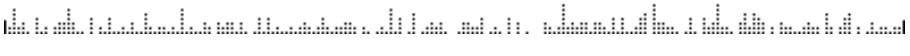
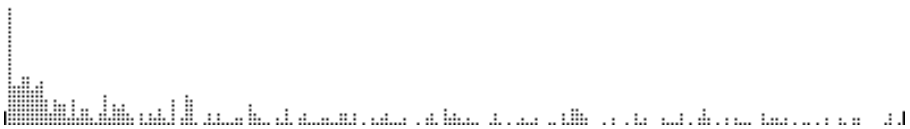
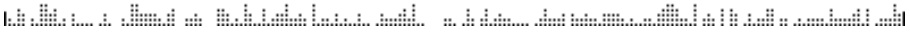
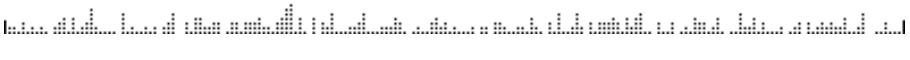
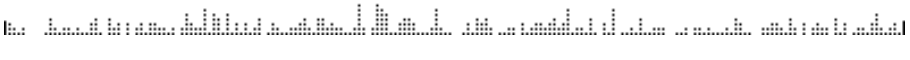


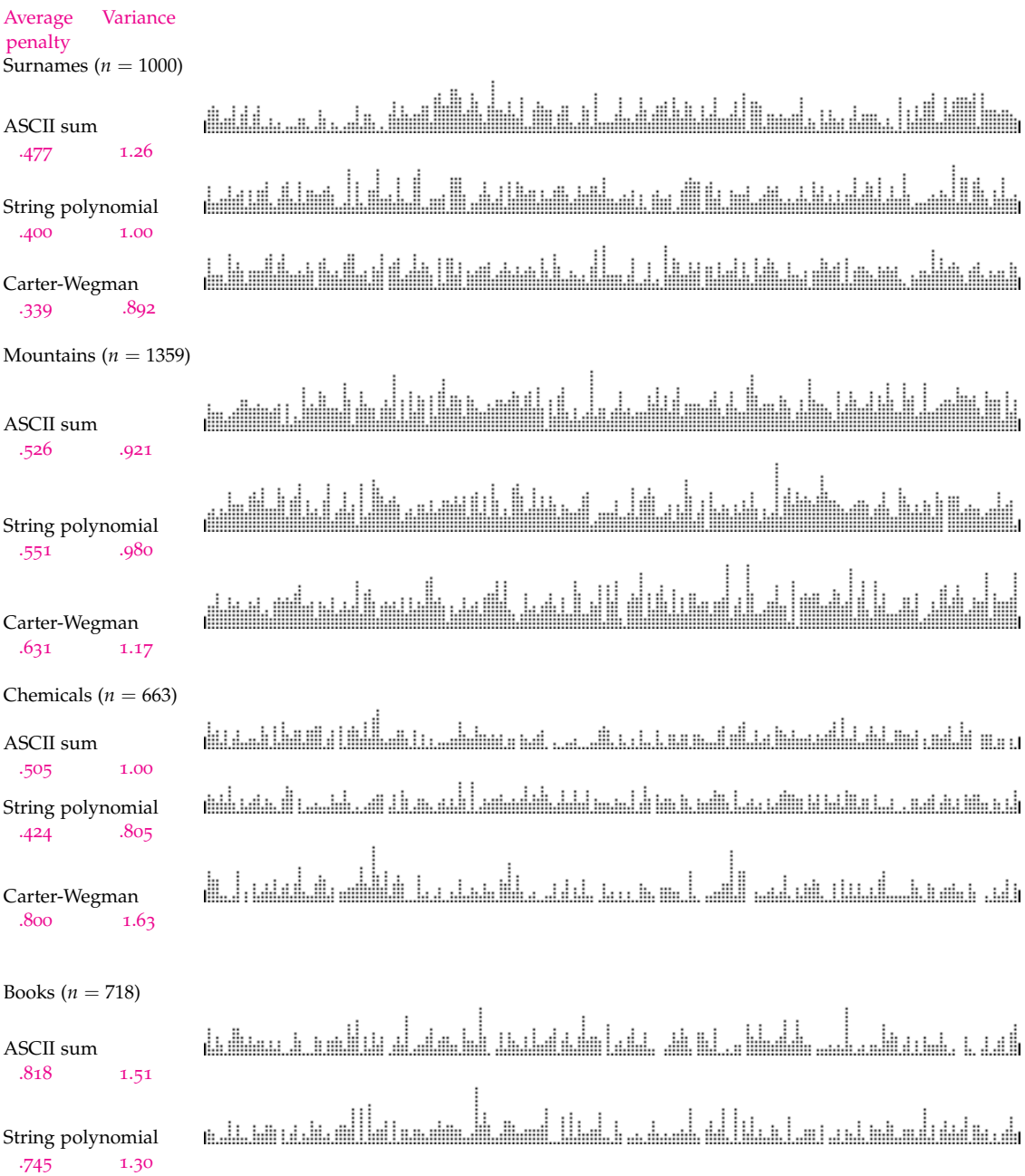


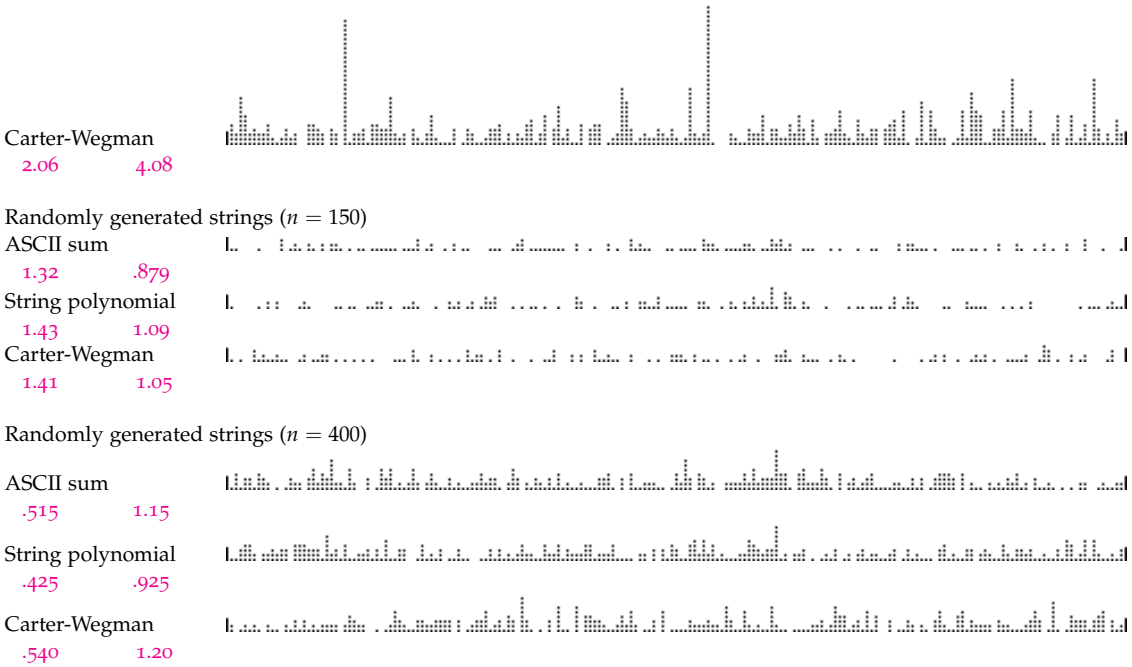


Penalty	0	0	1	2	3

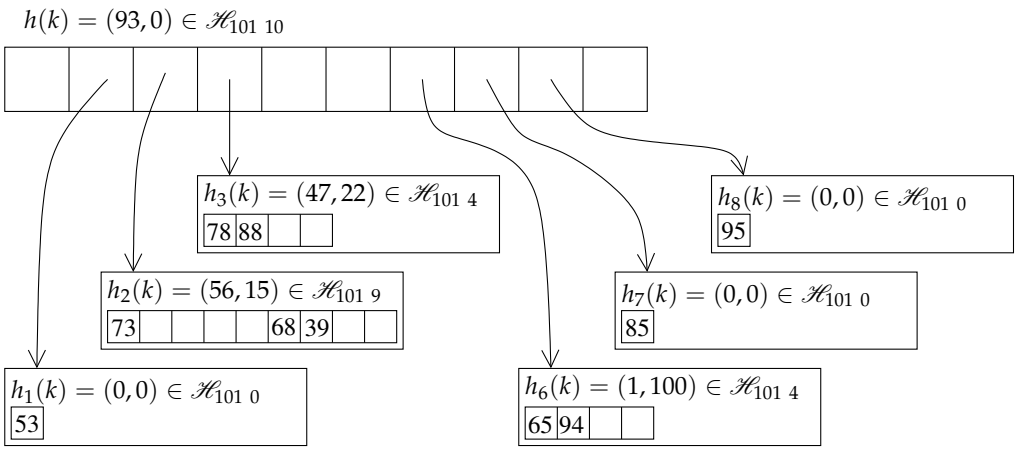

Average penalty
Variance
Area codes (n = 303)
Division
.673
.808
Mid square
1.09
1.64
Multiplicative
.508
.478
Fibonacci
.617
.696
Universal
.578
.617
Book ISBNs (n = 718)
Division
.618
1.05
Mid square
.812
1.48
Multiplicative
.565
.954
Fibonacci
.544
.873
Universal
.667
1.15
Mountain heights in meters (n = 1359)
Division
.577
1.03
Mid square
.857
1.80
Multiplicative
.669
1.30
Fibonacci
.577
1.05
Universal
.662
1.30

Average penalty	Variance	
Randomly generated from $[0, 1000)$ ($n = 150$)		
Division		
1.36	.958	
Mid square		
1.86	1.96	
Multiplicative		
1.34	.919	
Fibonacci		
1.41	1.07	
Universal		
1.39	1.02	
Randomly generated from $[0, 1000)$ ($n = 400$)		
Division		
.518	1.16	
Mid square		
1.73	3.68	
Multiplicative		
.405	.930	
Fibonacci		
.448	.980	
Universal		
.488	1.08	


Average penalty
Variance
Surnames ($n = 1000$)
ASCII sum
.477
1.26
String polynomial
.400
1.00
Carter-Wegman
.339
.892
Mountains ($n = 1359$)
ASCII sum
.526
.921
String polynomial
.551
.980
Carter-Wegman
.631
1.17
Chemicals ($n = 663$)
ASCII sum
.505
1.00
String polynomial
.424
.805
Carter-Wegman
.800
1.63
Books ($n = 718$)
ASCII sum
.818
1.51
String polynomial
.745
1.30



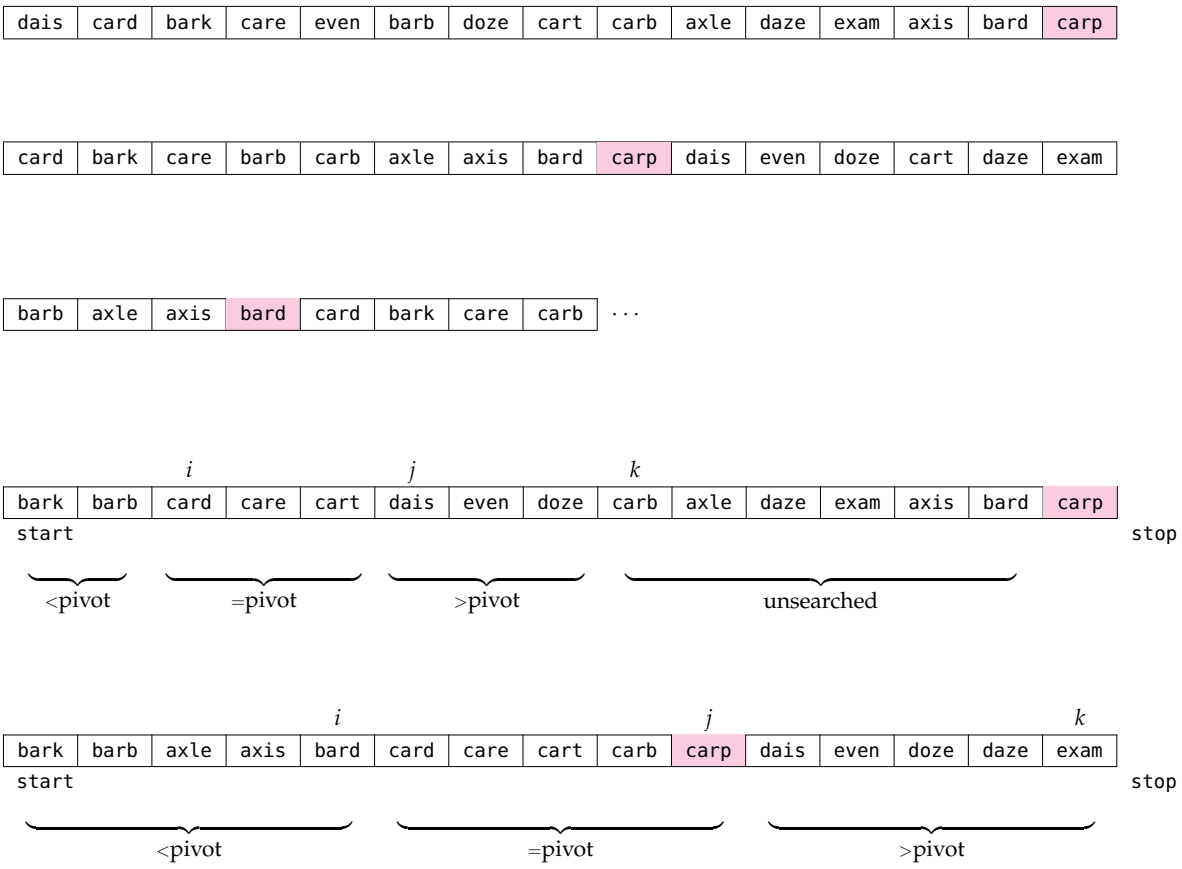
7.5 *Perfect hashing*

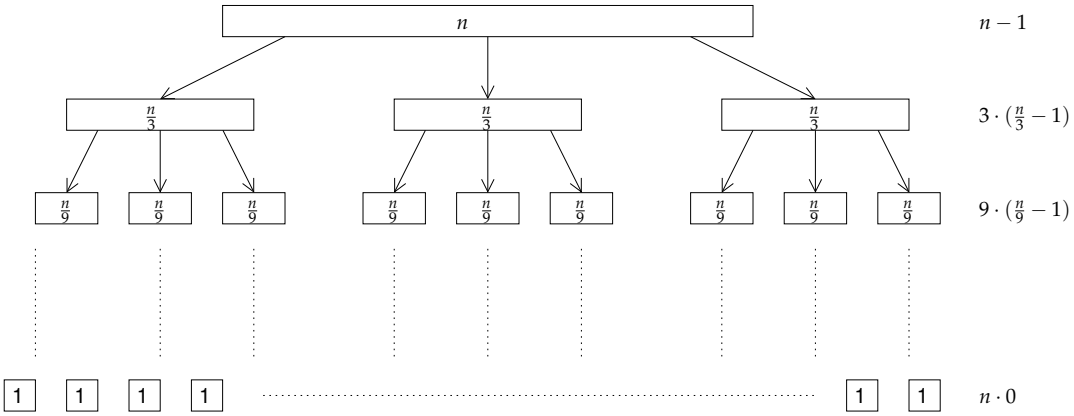


7.6 *Chapter summary*

8
String processing

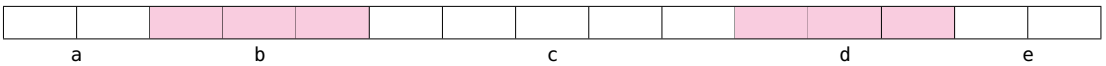
8.1 Sorting algorithms for strings





dais	card	bark	care	even	barb	doze	cart	carb	axle	daze	exam	axis	bard	carp

0	0	2	3	8	11	⋯
a	b	c	d	e		



axle	axis	bark	barb	bard	card	care	cart	carb	carp	dais	doze	daze	even	exam
a			b				c				d			e

dais	card	bark	care	even	barb	doze	cart	carb	axle	daze	exam	axis	bard	carp

barb	carb	card	bard	care	doze	axle	daze	bark	exam	even	carp	dais	axis	cart

exam	even	dais	axis	axle	barb	carb	card	bard	care	bark	carp	cart	doze	daze

dais	barb	carb	card	bard	care	bark	carp	cart	daze	doze	even	exam	axis	axle

axis	axle	barb	bard	bark	carb	card	care	carp	cart	dais	daze	doze	even	exam

beach	event	can	core	hope	any	front	ball	done	a	frond	an	i	give	eve

can	core	hope	any	ball	done	a	an	i	give	eve	frond	beach	event	front

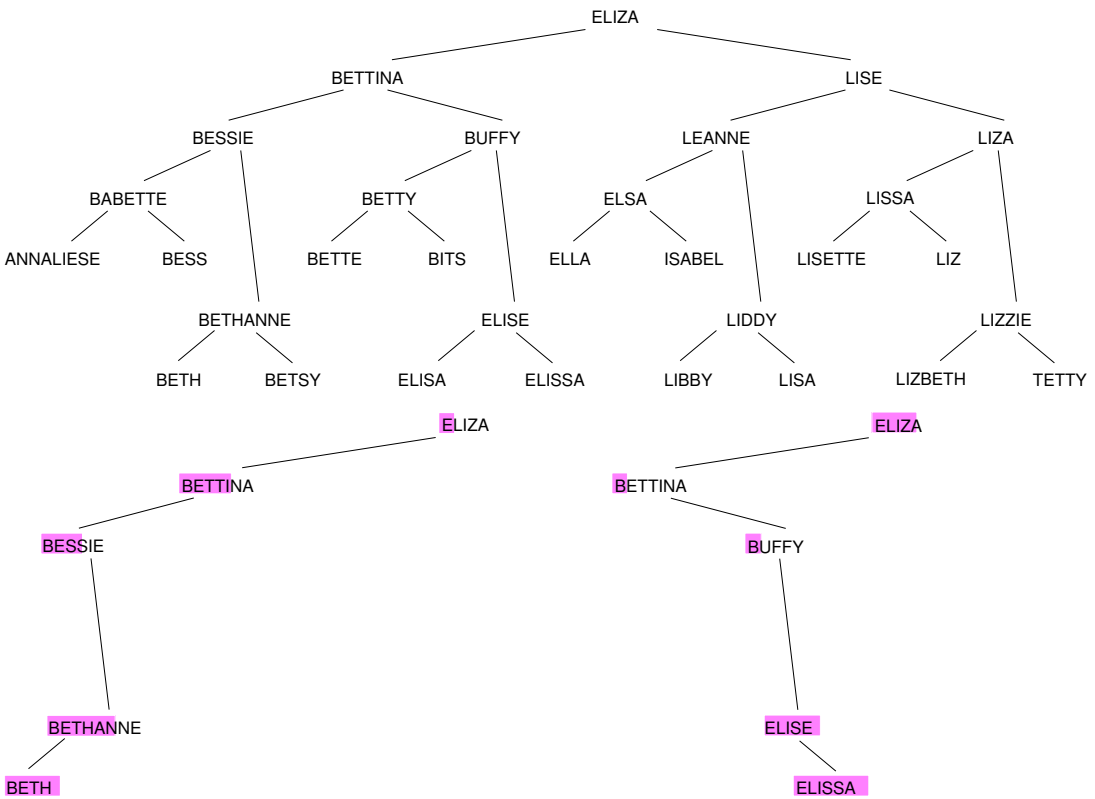
can	any	a	an	i	eve	beach	core	hope	done	give	ball	frond	event	front

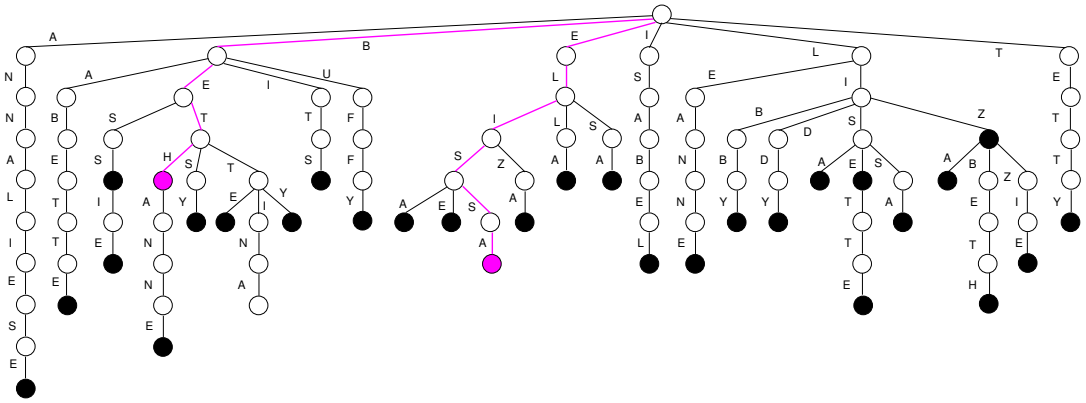
a	an	i	beach	eve	event	ball	can	done	frond	front	hope	core	give	any

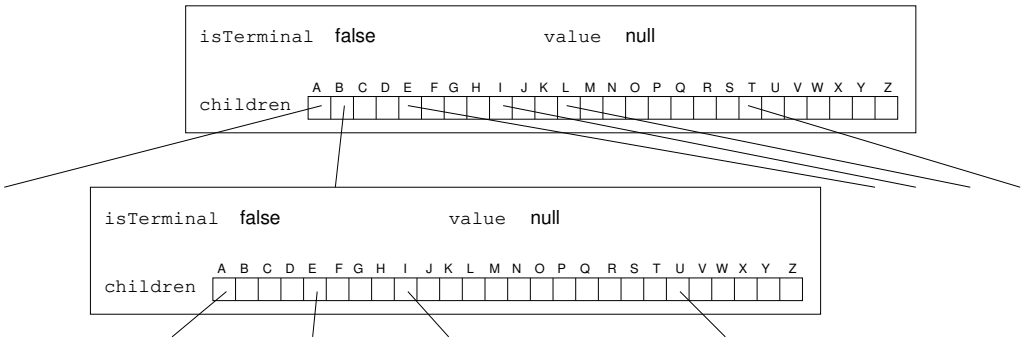
a	i	ball	can	beach	give	an	any	done	hope	core	frond	front	eve	event

a	an	any	ball	beach	can	core	done	eve	event	frond	front	give	hope	i

8.2 *Tries*

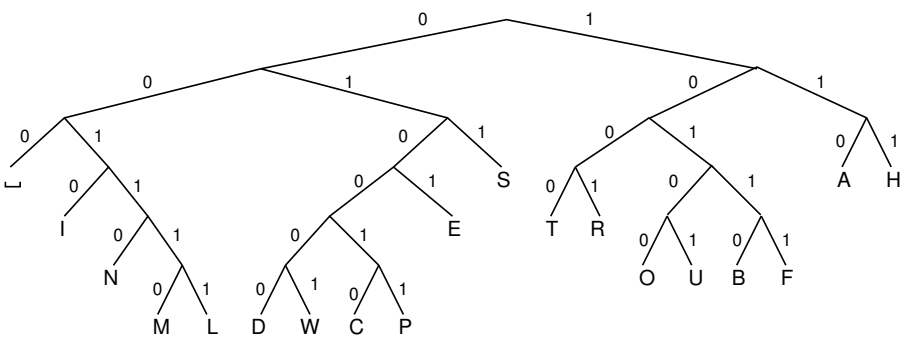




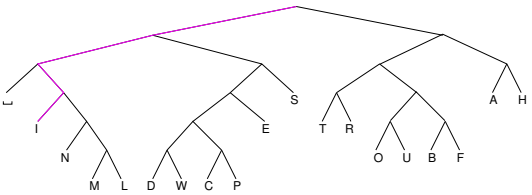


8.3 *Huffman encoding*

I F ␣ T H I S ␣ A ␣ ...
00101011100010011110010011000110000

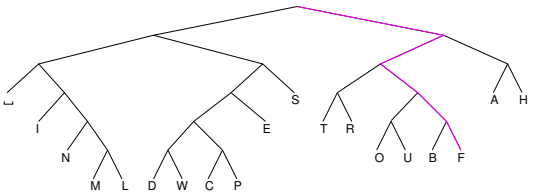


0010101110001001111001001100011000 0...



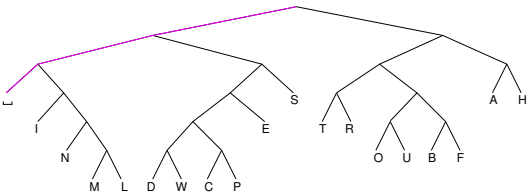
I

0010101110001001111001001100011000 0...



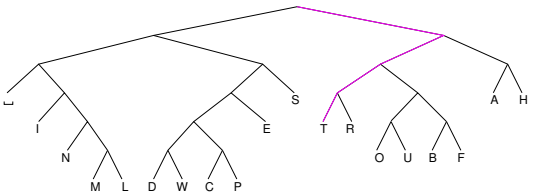
IF

0010101110001001111001001100011000 0...



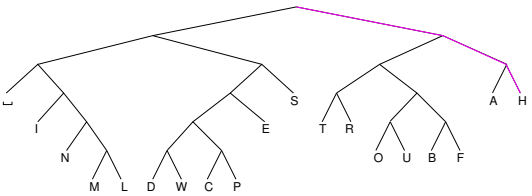
IF␣

0010101110001001111001001100011000 0...



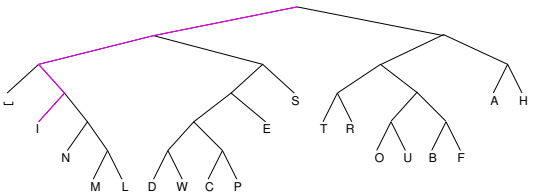
IF␣T

0010101110001001111001001100011000 0...

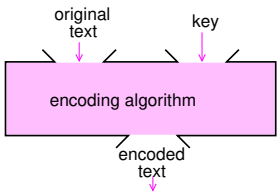


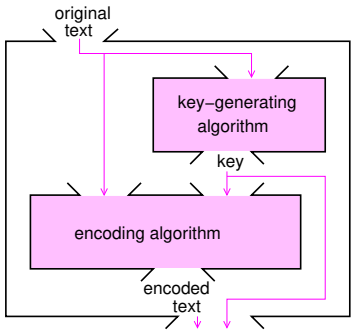
IF␣TH

0010101110001001111001001100011000 0...

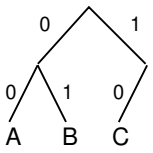


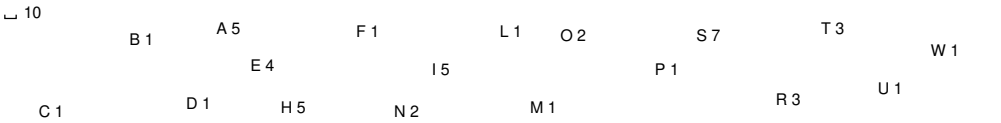
IF␣THI

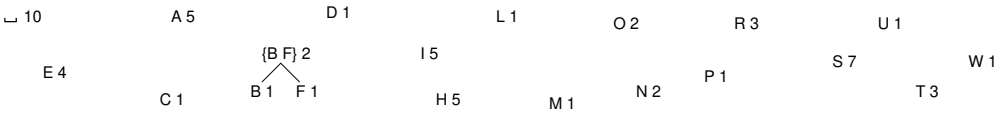


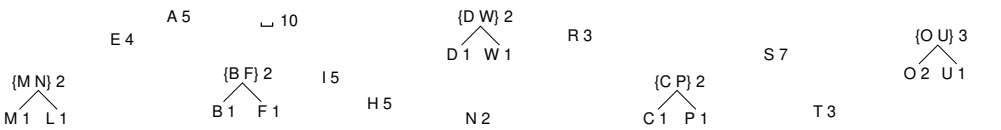


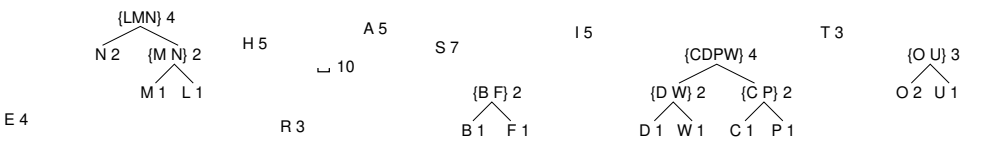

```
A   00
B   01
C   10
```

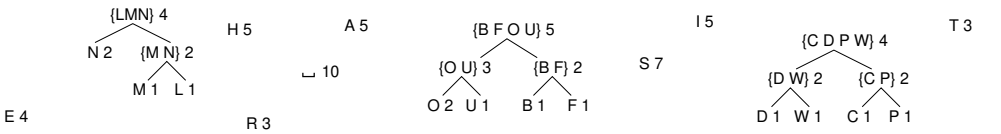


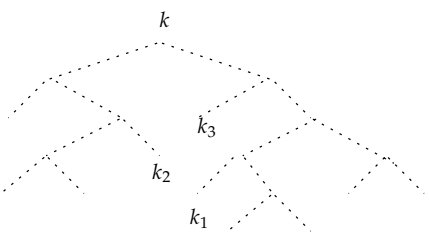

␣ 10
B 1
A 5
F 1
L 1
O 2
S 7
T 3
W 1
E 4
I 5
P 1
U 1
C 1
D 1
H 5
N 2
M 1
R 3


␣ 10
A 5
D 1
L 1
O 2
R 3
U 1
E 4
{B F} 2
I 5
S 7
W 1
P 1
C 1
B 1
F 1
H 5
N 2
T 3
M 1

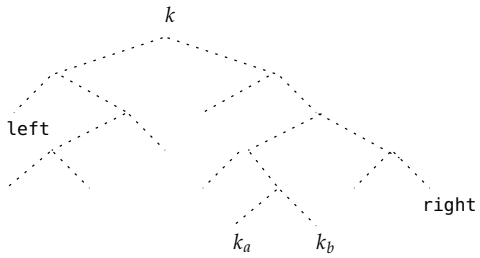

A 5
␣ 10
{D W} 2
R 3
E 4
S 7
{O U} 3
{M N} 2
{B F} 2
I 5
{C P} 2
M 1
L 1
B 1
F 1
H 5
D 1
W 1
N 2
C 1
P 1
T 3
O 2
U 1


{LMN} 4
H 5
A 5
S 7
I 5
T 3
{CDPW} 4
{O U} 3
N 2
{M N} 2
␣ 10
{B F} 2
{D W} 2
{C P} 2
E 4
M 1
L 1
R 3
B 1
F 1
D 1
W 1
C 1
P 1
O 2
U 1

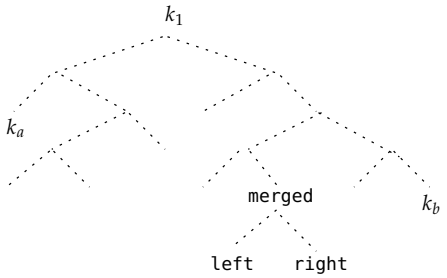

{LMN} 4
H 5
A 5
{B F O U} 5
I 5
T 3
{C D P W} 4
N 2
{M N} 2
{O U} 3
{B F} 2
S 7
{D W} 2
{C P} 2
E 4
M 1
L 1
␣ 10
R 3
O 2
U 1
B 1
F 1
D 1
W 1
C 1
P 1


k
k_3
k_2
k_1

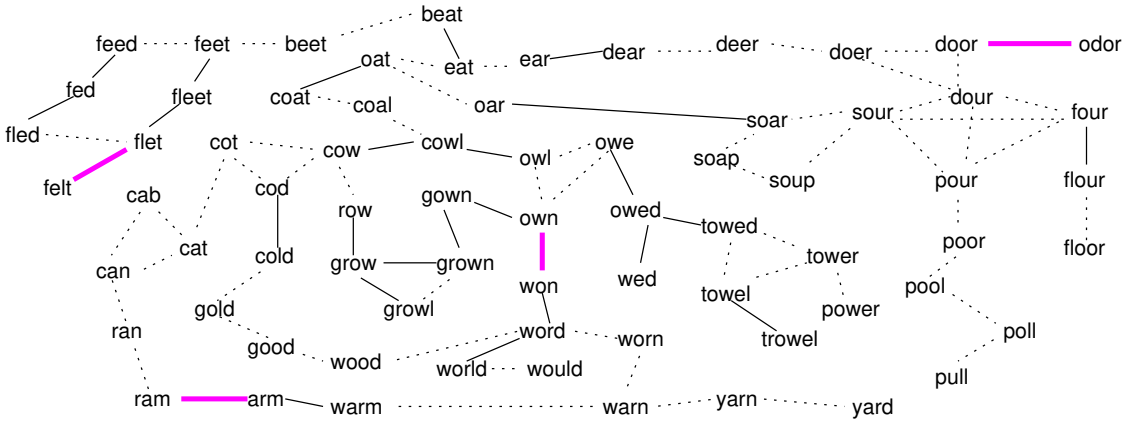

… k_1 … k_2 … k_3 …



left	right	...	k_a	...	k_b	...

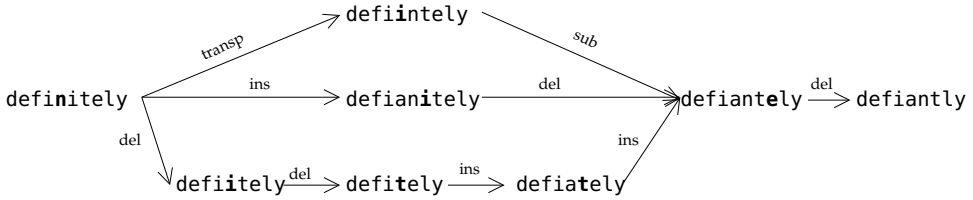


8.4 *Edit distance*



rec**i**eve $\xrightarrow{\text{del}}$ rec**e**ve $\xrightarrow{\text{ins}}$ receive versus rec**ie**ve $\xrightarrow{\text{transp}}$ receive

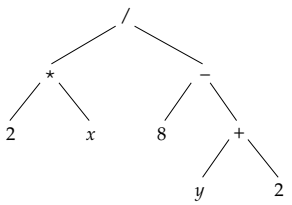
sep**e**rate $\xrightarrow{\text{del}}$ sep**r**ate $\xrightarrow{\text{ins}}$ separate versus sep**e**rate $\xrightarrow{\text{sub}}$ separate

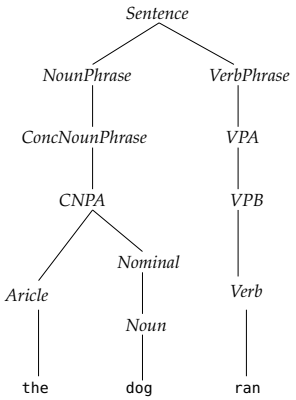


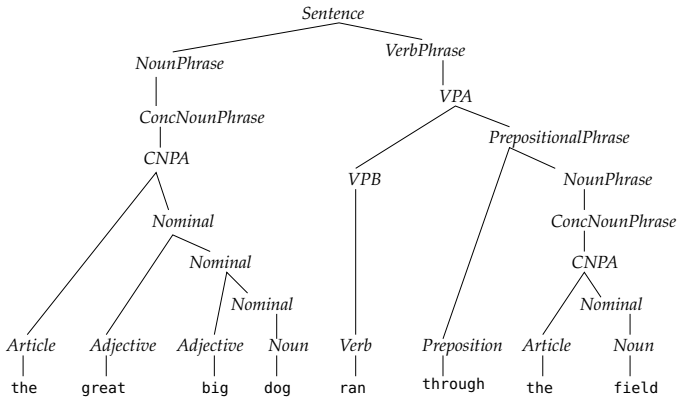
n	6/ins-all	5/ins	4/ins	4/ins	3/ins	3/ins	2/nop	3/del
e	5/ins-all	4/ins	3/ins	3/ins	2/ins	2/sub	3/del	4/del
v	4/ins-all	3/ins	2/ins	2/ins	1/nop	2/del	3/del	4/del
a	3/ins-all	2/ins	1/nop	1/transp	2/del	3/del	4/del	5/del
r	2/ins-all	1/ins	1/sub	1/nop	2/del	3/del	4/del	5/del
c	1/ins-all	0/nop	1/del	2/del	3/del	4/del	5/del	6/del
	0/del-all	1/del-all	2/del-all	3/del-all	4/del-all	5/del-all	6/del-all	7/del-all
		c	a	r	v	i	n	g

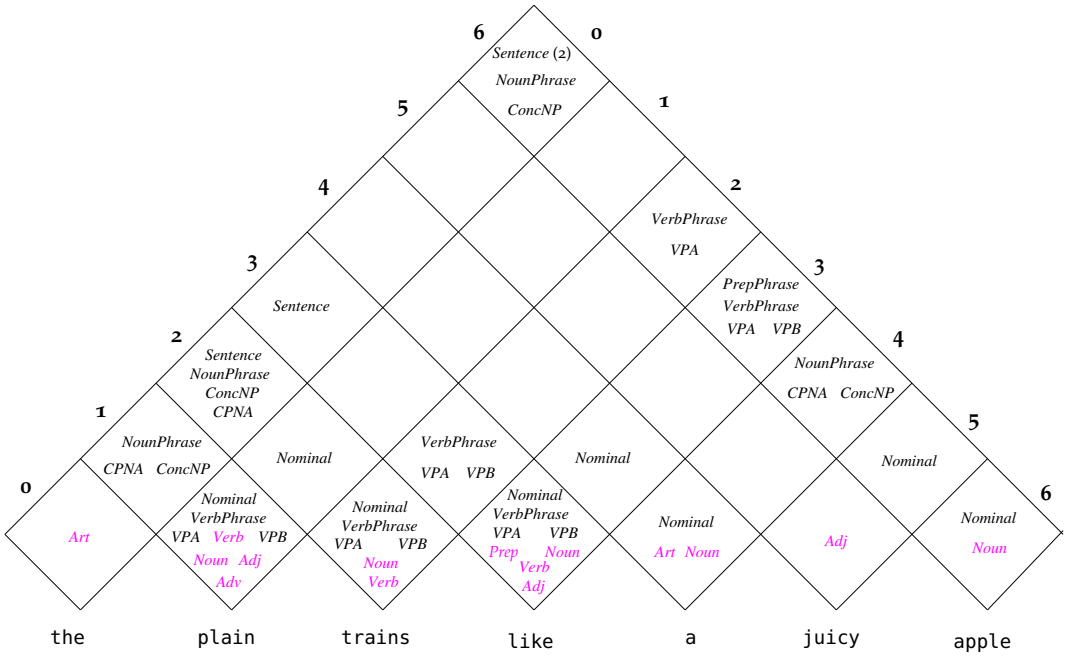
8.5 *Grammars*

```
((2*x)/(8*(y+2)))
```









8.6 Chapter summary