1. Într-o secvență de n operații Insert, fiecare clădire i este adăugată în stivă fix o dată, și eliminată din stivă maxim o dată. În plus, într-o inserare în care se efectuează k operații pop() se vor efectua maxim k+1 operații top(). Prin urmare, alocăm costurile amortizate:
   $\hat{C}_{push} = 4$, $\hat{C}_{pop} = \hat{C}_{top} = 0$, care vor putea mereu acoperi costurile reale ($c_{push} = c_{pop} = c_{top} = 1$).
   ($\hat{C}_{push} = 4$ plătește pentru 1 push(), 1 pop(), 2 top()-uri)

   $\Rightarrow T(n) = \sum_{i=1..n}(c_{Insert}) \le \sum_{i=1..n}(\hat{c}_{Insert}) \le 4n \Rightarrow$ costul mediu = T(n)/n = O(1).

2. O(n*k) – este posibil să ajungem în situația în care toți biții contorului sunt setați pe 1 și să efectuăm încontinuu câte o incrementare și câte o decrementare, modificând de fiecare dată fiecare bit.

3. Codul inversează un vector v de dimensiune n.
   Invariant: Înainte de iterația (i, j), v[i+1..j-1] conține elementele **aflate inițial** în v[i+1..j-1], în ordine inversă.
   Inițializare: Înainte de prima iterație, i=(n-1)/2, j=n/2. Pentru n impar => i = j, pentru n par => i+1 = j. În ambele cazuri, v[i+1..j-1] este un vector vid (indicele de start mai mare decât cel de stop), și conține cele 0 elemente aflate inițial în v[i+1..j-1], în ordine inversă.
   Menținere: Presupunem că înainte de iterația (i, j) v[i+1..j-1] conține elementele aflate inițial în v[i+1..j-1], în ordine inversă. După iterația (i, j) (înainte de iterația (i-1, j+1)), se modifică (prin interschimb) doar elementele de pe pozițiile i și j, deci v[i-1+1..j+1-1] (adică v[i..j]) va conține elementele aflate inițial în v[i..j], în ordine inversă.
   Terminare: La ieșirea definitivă din buclă (înainte de iterația (i=-1, j=n)), v[0..n-1] conține elementele aflate inițial în v[0..n-1], în ordine inversă.

4. d[v] este drumul minim de la source la v care trece prin maxim i-1 muchii. Acesta este principiul de funcționare al algoritmului Bellman-Ford (celelalte variante nu au legătură cu ideea algoritmului). Există o mică problemă cu implementarea oferită, care permite ca, pe un drum, în aceeași iterație, să se efectueze mai multe relaxări de muchii (să avansăm mai mult decât cu câte o muchie în fiecare pas). Am putea împiedica aceasta dacă d ar fi o matrice, nu un vector. (Am scrie if d[i-1][v] > d[i-1][u] + w { d[i][v] = ... })

5. (0, 1) : -> NConsPair // generează perechea (0, 1)
   succ : NConsPair -> NConsPair // pe baza perechii (n, n+1), obține perechea (n+1, n+2)

6. (V1) v(nil) = 0
   (V2) v(node(l, x, r)) = 1 + v(l) + v(r)
   (E1) e(nil) = 0
   (E2) e(node(l, x, r)) = (nil?(l) ? 0 : 1) + (nil?(r) ? 0 : 1) + e(l) + e(r)

   P: not(nil?(t)) => v(t) = e(t) + 1

   Cazul de bază: t = nil
   not(nil?(t)) => ceva <=(N1)=>
   false => ceva.

Pasul de inducție: t = node(l, x, r)

Ipoteza inductivă: not(nil?(l)) => v(l) = e(l) + 1 ;   not(nil?(r)) => v(r) = e(r) + 1

not(nil?(node(l, x, r))) => v(node(l, x, r)) = e(node(l, x, r)) + 1 <=(N2/V2, E2)=>

True => 1 + v(l) + v(r) = 1 + (nil?(l) ? 0 : 1) + (nil?(r) ? 0 : 1) + e(l) + e(r)

Există 4 posibilități:

a) l = r = nil

1 + v(nil) + v(nil) = 1 + (nil?(nil) ? 0 : 1) + (nil?(nil) ? 0 : 1) + e(nil) + e(nil) <=(V1/N1, E1)=>

1 + 0 + 0 = 1 + 0 + 0 + 0 + 0.

b) l = nil, r ≠ nil

1 + v(nil) + v(r) = 1 + (nil?(nil) ? 0 : 1) + (nil?(r) ? 0 : 1) + e(nil) + e(r) <=(V1/N1, E1)=>

1 + v(r) = 1 + 1 + e(r) <=(II)=> True.

c) r = nil, l ≠ nil

Analog cu b).

d) l ≠ nil, r ≠ nil (adică not(nil?(l)), not(nil?(r)))

1 + v(l) + v(r) = 1 + (nil?(l) ? 0 : 1) + (nil?(r) ? 0 : 1) + e(l) + e(r) <=>

1 + v(l) + v(r) = 1 + 1 + 1 + e(l) + e(r) <=(II)=> True.