A. Subalgoritm recursiv cu complexitatea $O(\log_2 n)$

```
Subalgoritm rec (s, m, v, e)
    {pre: m ∈ Intreg (dimensiunea vect. v) + s ∈ Intreg
          v ∈ Intreg[] - ordonat crescator (poz. de inceput)
          e ∈ Intreg
     post: 1 -> daca elementul e este gasit in v
           0 -> altfel
    }
    Daca s > m atunci:
        rec <- 0
    SfDaca
    Daca v[m] = e atunci:
        rec <- 1
    altfel
        m <- [(s+m) | 2]
        Daca v[m] < e atunci:
            s <- m+1
        altfel
            m <- m-1
        SfDaca
        rec(s, m, v, e)
    SfDaca
SR Subalgoritm
```

caz favorabil - când elem. căutat se află la mijloc $\Rightarrow \Theta(1)$

caz defavorabil - când elem. nu este în vector $\Rightarrow$

$T(n) = T(n|2) + 1 \quad \Theta(\log_2 n)$

$n = 2^k \Rightarrow k = \log_2 n$

$T(2^k) = T(2^{k-1}) + 1$

$T(2^{k-1}) = T(2^{k-2}) + 1$

$\vdots$

$T(1) = 1$

$\Rightarrow T(2^k) = k \Rightarrow T(n) = \log_2 n$

$\Rightarrow$ Complexitate generală $O(\log_2 n)$ (căutare eficientă)

B.

n -> nr. cheițe ce trebuie dispersate

m -> nr. de găleți din tabelă

$|U| > n * m$

Presupunem că $\nexists S \subset U$ o submulțime de dim n cu elem. care se dispersează în aceeași găleată =>

pe poz. 0 se pot dispersa maxim n-1 chei

pe poz. 1 se pot dispersa maxim n-1 chei

⋮

pe poz. m-1 se pot dispersa maxim n-1 chei

} =>

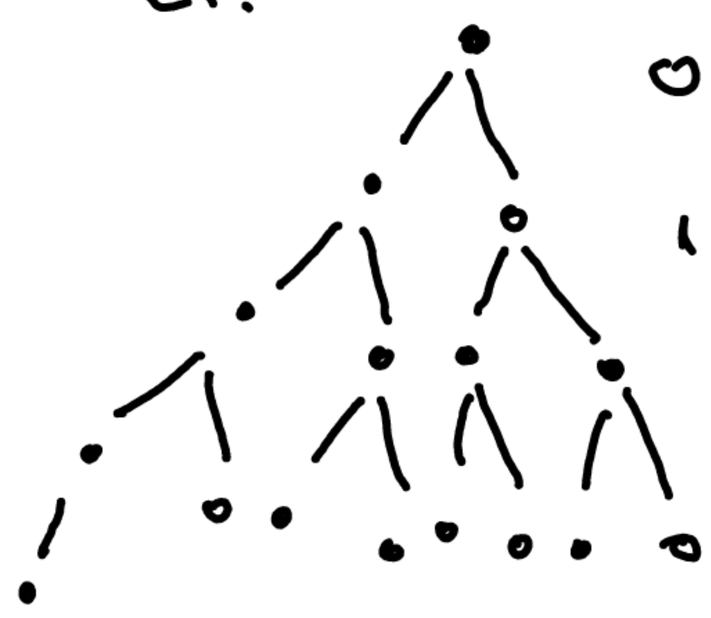
avem în total $m \cdot (n-1) = n * m - m$ chei

$=> n * m - m > n * m$

$-(m+n) > 0$

Dar $m, n > 0$ } => contradicție => $\exists$ S o submulțime a lui U ce conține n elem. care se dispersează în aceeași găleată

C1.



Cum avem arbore binar, fiecare nivel va avea 2^{nivel} noduri:

0 - 1 nod

1 - 2 noduri

2 - 4 noduri

3 - 8 noduri

4 - 16 noduri (maxim)

Dar se menționează că algoritmul este apropiat

pcin => ultimul nivel nu este complet | => ultimul

Se caută nr. minim de noduri

nivel va avea cu singur nod

Nr. noduri: 1 + 2 + 4 + 8 + 1 = 3 + (2+1 = 16 - c)

c2.

c) coada memorează conținutul capete ale structurii, stivele doar un capăt

Acest lucru se întâmplă deoarece coada funcționează pe politica FIFO (First In, First Out), adică nevoie să extragă capătul din față pt. a accesa elementele și pe cel din spate pentru a stoca elemente. Pe de altă parte, în stivă, primul element accesat este ultimul stocat, având ambele operații (de accesare și stocare) făcându-se la același capăt.

5.

Implementare eficientă

Nod

e : TElement

p : ↑Nod

st : ↑Nod

dr : ↑Nod

Arbore

r : ↑ Nod (rădăcina arborelui)

Subalgoritm max-min (n) este

{pre: n ∈ ↑Nod ; nod valid (aparține arborelui)}

{post: se returnează cel mai mare element mai mic decât cel nodului dacă există și -1 în caz contrar}

mmax ← [n].e

{se determină maximul subarborelui stâng, dacă acesta există}

Dacă [n].st ≠ NIL atunci

p ← [n].st

Cât Timp [p].dr ≠ NIL execută

p ← [p].dr

SCâtTimp

{în p avem cheia cu valoarea maximă din sub. st.}

mmax ← [p].e

SfDacă

```
{ vom parcurge și ascendenții nodului }
{ vom alege cea mai mare cheie, mai mică decât [n].e }
Dacă [n].p ≠ NIL atunci
    p ← [n].p
    Cât Timp p ≠ NIL execută
        Dacă [p].e > mmax ∧ [p].e < [n].e atunci
            mmax ← [p].e
        SF Dacă
        p ← [p].p
    SF Cât Timp
SF Dacă
{ am terminat de căutat => verificăm rădăcina }
Dacă [n].e = mmax atunci
    max - min ← -1
altfel
    max - min ← mmax
SF Dacă
SF SubalgoritmA
```

complexitate:

- de timp: $O(h)$, unde h reprezintă înălțimea arborelui inițial
- de spațiu: $\Theta(1)$ (spațiu utilizat în plus)