Adatszerkezetek

15. hét – Bináris keresőfa, kiegyensúlyozottság, kupac; a kurzus összefoglalása

Dr. Koczka Ferenc · Eszterházy Károly Egyetem · 2026

Ismétlés

- **Bináris fa:** legfeljebb két gyerek; láncolt és tömbös ábrázolás
- **Bejárások:** preorder, inorder, postorder (rekurzívan), szintenként (sorral)
- **Bináris keresés** (2. hét): rendezett tömbben O(log n) – de a beszúrás O(n)

A mai óra kérdései

- Hogyan lehet a keresést **és** a beszúrást is gyorssá tenni?
- Miért fontos, hogy a fa **kiegyensúlyozott** legyen?
- Hogyan ad a **kupac** gyors prioritási sort?

Bináris keresőfa

A bináris keresőfa

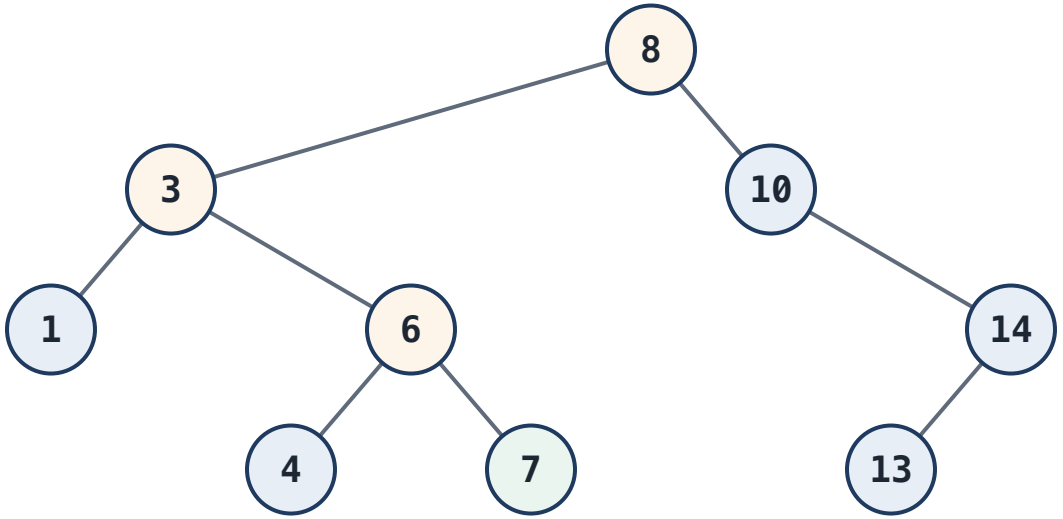
Bináris keresőfa (BST, **binary search tree**): olyan bináris fa, amelyben **minden** csomópontra igaz:

- a **bal** részfájában minden kulcs **kisebb**,
- a **jobb** részfájában minden kulcs **nagyobb**, mint a csomópont kulcsa.

Fontos következmény: az **inorder bejárás** a kulcsokat **növekvő sorrendben** adja!
Példa: 8, 3, 10, 1, 6, 14, 4, 7, 13 beszúrása után az inorder: **1 3 4 6 7 8 10 13 14**

Nem elég, hogy minden csomópont a **közvetlen** gyerekeinél nagyobb/kisebb – a feltétel a **teljes** részfákra vonatkozik!

Keresés



Keresés: x = 7

```
7 < 8 → balra
7 > 3 → jobbra
7 > 6 → jobbra
7 = 7 → megtaláltuk ✓
```

Minden lépés egy szinttel lejjebb visz:

lépésszám = O(magasság)

Keresés – algoritmus

Iteratívan

```
Függvény Keres(Gyökér, x)
  P ← Gyökér
  Ciklus amíg P ≠ Nil és Elem[P].Kulcs ≠ x
    Ha x < Elem[P].Kulcs akkor
      P ← Elem[P].Bal
    különben
      P ← Elem[P].Jobb
    Elágazás vége
  Ciklus vége
  Vissza: P            // Nil, ha nincs benne
Függvény vége
```

Minimum: mindig balra, amíg lehet

Maximum: mindig jobbra, amíg lehet

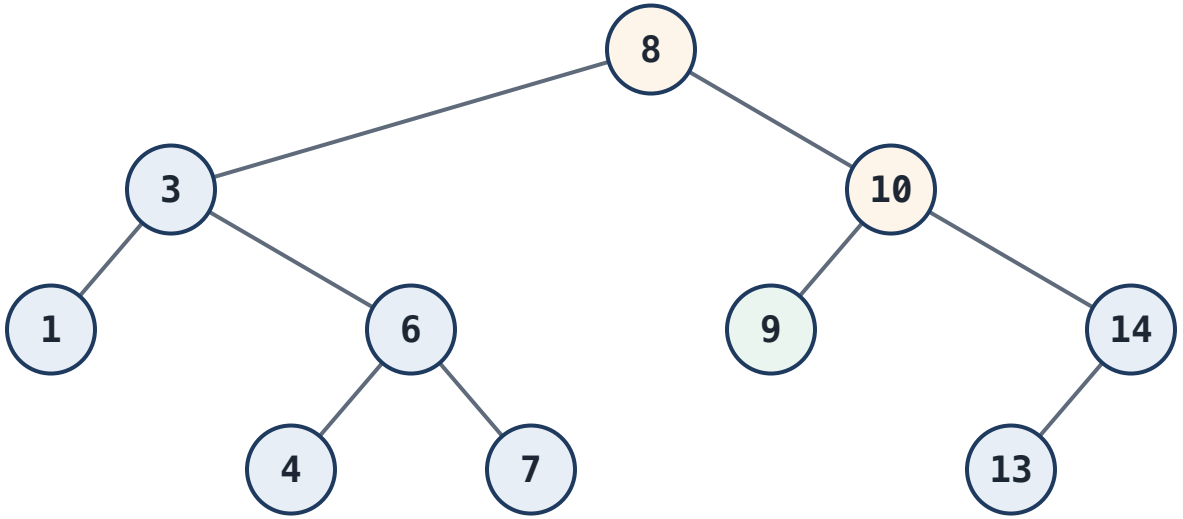
Java

```java
boolean contains(int x) {
    Node p = root;
    while (p != null && p.key != x)
        p = (x < p.key) ? p.left : p.right;
    return p != null;
}
```

Go

```go
func (t *BST) Contains(x int) bool {
    p := t.Root
    for p != nil && p.Key != x {
        if x < p.Key {
            p = p.Left
        } else {
            p = p.Right
        }
    }
    return p != nil
}
```

Beszúrás



Beszúrás: x = 9

```
9 > 8 → jobbra
9 < 10 → balra
üres hely → ide (új levél)
```

Az új elem mindig levél lesz.

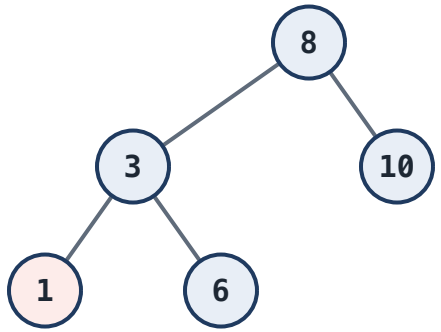
Beszúrás – algoritmus

```
Függvény Beszúr(P, x)        // az új részfát adja
  Ha P = Nil akkor
    Lefoglal(Q) ; Elem[Q].Kulcs ← x
    Elem[Q].Bal ← Nil ; Elem[Q].Jobb ← Nil
    Vissza: Q
  Ha x < Elem[P].Kulcs akkor
    Elem[P].Bal ← Beszúr(Elem[P].Bal, x)
  különben ha x > Elem[P].Kulcs akkor
    Elem[P].Jobb ← Beszúr(Elem[P].Jobb, x)
  Vissza: P
Függvény vége
```

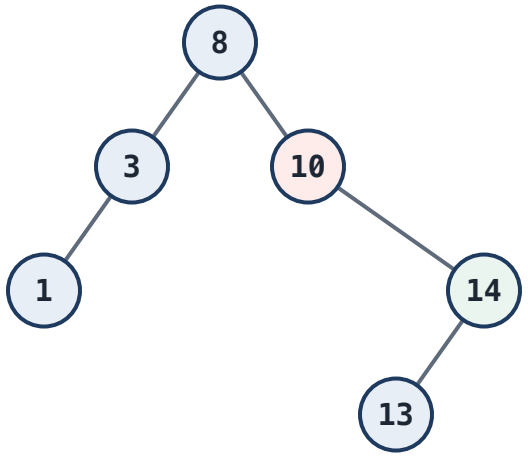
```java
Node insert(Node p, int x) {
    if (p == null) return new Node(x);
    if (x < p.key)
        p.left = insert(p.left, x);
    else if (x > p.key)
        p.right = insert(p.right, x);
    return p;
}
```

Hívás: `root = insert(root, x);`

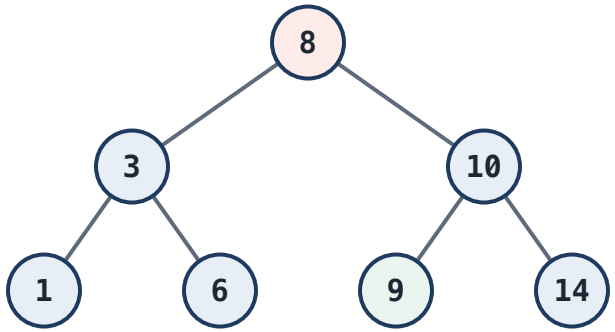
Törlés – három eset



1. levél:
egyszerűen töröljük



2. egy gyerek:
a gyerek a helyére lép



3. két gyerek:
az inorder rákövetkező értéke

- **Levél:** egyszerűen töröljük
- **Egy gyerek:** a gyerek (a teljes részfájával) a törölt helyére lép
- **Két gyerek:** a jobb részfa **legkisebb** elemét (az **inorder rákövetkezőt**) a helyére másoljuk, majd azt töröljük – annak legfeljebb egy gyereke van

Törlés – kód

Java

```java
Node delete(Node p, int x) {
    if (p == null) return null;
    if (x < p.key) p.left = delete(p.left, x);
    else if (x > p.key) p.right = delete(p.right, x);
    else {
        if (p.left == null) return p.right;   // 0 vagy 1 gyerek
        if (p.right == null) return p.left;
        Node s = p.right;                     // 2 gyerek:
        while (s.left != null) s = s.left;    // inorder rákövetkező
        p.key = s.key;
        p.right = delete(p.right, s.key);
    }
    return p;
}
```

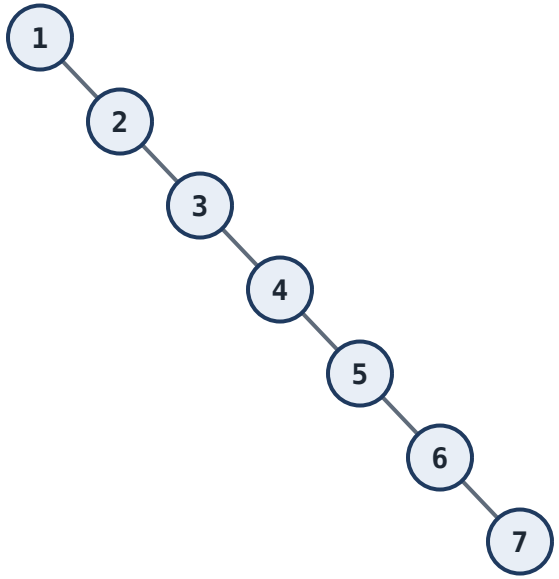
Go – a két gyerekes eset

```go
default:
    if p.Left == nil {
        return p.Right
    }
    if p.Right == nil {
        return p.Left
    }
    s := p.Right              // inorder rákövetkező
    for s.Left != nil {
        s = s.Left
    }
    p.Key = s.Key
    p.Right = del(p.Right, s.Key)
```

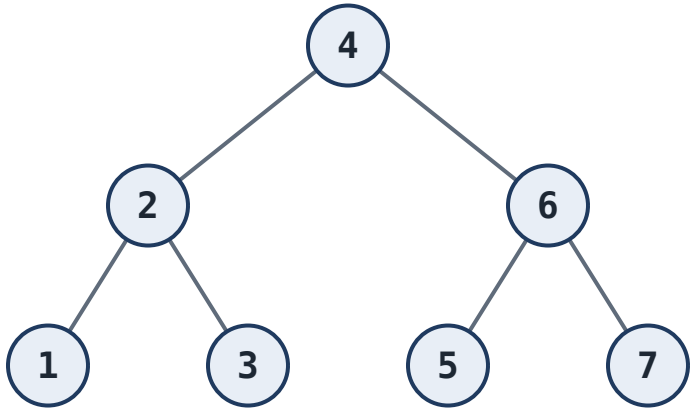
A teljes kód: `h15/bst`

Kiegyensúlyozottság

A keresőfa hatékonysága a magasságon múlik



1, 2, …, 7 sorrendben beszúrva: magasság 6



ugyanezek az elemek kiegyensúlyozva: magasság 2

- Keresés, beszúrás, törlés: **O(h)**, ahol **h** a fa magassága
- **Kiegyensúlyozott** fa: $h \approx \log_2 n \rightarrow$ **O(log n)** · **degenerált** fa: $h = n - 1 \rightarrow$ **O(n)** (lista!)
- **Mérés:** 1000 elem rendezett sorrendben → magasság **999** · véletlen sorrendben → magasság **18**

Kiegyensúlyozott keresőfák

Kiegyensúlyozott fa: minden csomópontban a bal és a jobb részfa magassága **legfeljebb 1-gyel** tér el (AVL-feltétel).



A < x < B < y < C – az inorder sorrend (és a keresőfa-tulajdonság) megmarad

- Beszúrás és törlés után a fát **forgatásokkal** helyreállítjuk – egy forgatás O(1)
- **AVL-fa** (1962), **piros-fekete fa** (Java `TreeMap`, `TreeSet`), **B-fa** (adatbázisok, fájlrendszerek)
- Garantáltan **O(log n)** keresés, beszúrás, törlés

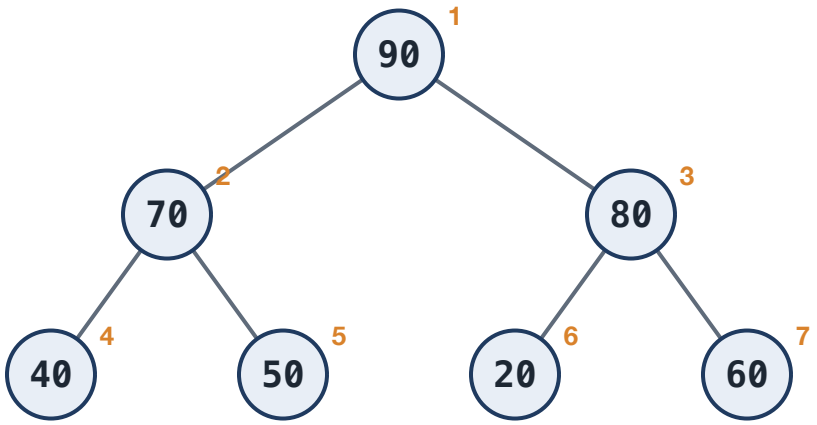
Keresőfa, rendezett tömb vagy hash tábla?

Művelet	Rendezett tömb	Hash tábla	Kiegyensúlyozott keresőfa
Keresés	O(log n)	**O(1)** átlagosan	O(log n)
Beszúrás, törlés	O(n)	**O(1)** átlagosan	O(log n)
Minimum, maximum	**O(1)**	O(n)	O(log n)
Rendezett bejárás	O(n)	O(n log n)	O(n)
Intervallum [a, b]	O(log n + k)	O(n)	O(log n + k)

Ha csak keresni kell: **hash tábla**. Ha a **sorrend** is számít (legkisebb, következő, intervallum): **keresőfa**.

Kupac

A kupac



A[1..7] – teljes fa, tömbben (14. hét)

90	70	80	40	50	20	60
1	2	3	4	5	6	7

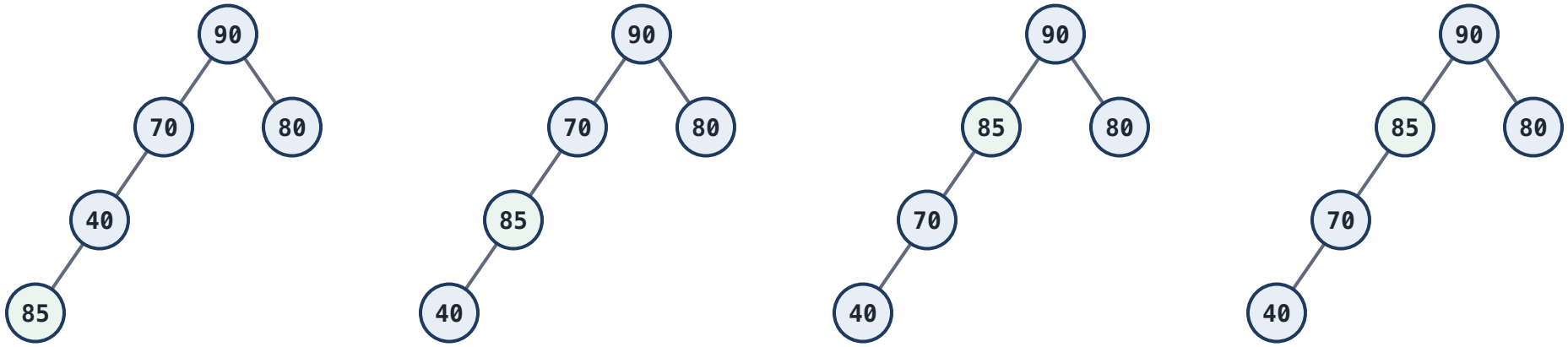
Maximum-kupac: minden szülő ≥ gyerekei

`A[i] ≥ A[2i] és A[i] ≥ A[2i + 1]`

a maximum mindig a gyökérben: A[1]

- **Teljes** bináris fa, **tömbben** tárolva (14. hét: gyerekek 2i és 2i + 1, szülő i div 2)
- **Kupac-tulajdonság:** minden szülő ≥ a gyerekei (maximum-kupac) – vagy ≤ (minimum-kupac)
- A testvérek között **nincs** sorrend – a kupac **nem** keresőfa!

Beszúrás: felszivárogtatás



85 a végére | 85 > 40 → csere | 85 > 70 → csere | 85 < 90 → kész

- Az új elemet a tömb **végére** tesszük (a fa következő szabad helyére)
- Amíg nagyobb a szülőjénél, **cseréljük** vele → legfeljebb **h** = $\log_2 n$ csere → **O(log n)**

A maximum kivétele: lesüllyesztés

```
Függvény MaximumKivétel(A, n)
  max ← A[1]
  A[1] ← A[n] ; n ← n − 1
  i ← 1
  Ciklus amíg 2·i ≤ n
    g ← 2·i                          // bal gyerek
    Ha g + 1 ≤ n és A[g + 1] > A[g] akkor
      g ← g + 1                      // a nagyobbik gyerek
    Ha A[i] ≥ A[g] akkor kilépés
    csere(A[i], A[g]) ; i ← g
  Ciklus vége
  Vissza: max
Függvény vége
```

1. A gyökérben a **maximum** – ezt adjuk vissza
2. Az **utolsó** elemet a gyökérbe tesszük
3. Amíg kisebb a nagyobbik gyerekénél, **lefelé cseréljük**

O(log n)

Prioritási sor	Betétel	Max kivétele
rendezetlen tömb	O(1)	O(n)
rendezett tömb	O(n)	O(1)
kupac	**O(log n)**	**O(log n)**

Java `PriorityQueue` · Go `container/heap`

Kupacrendezés

1. **Kupacépítés:** a tömbből helyben maximum-kupacot építünk – O(n)
2. Ismételjük **n** – 1-szer: a gyökeret (maximum) **kicseréljük az utolsó elemmel**, a kupacot eggyel kisebbnek tekintjük, és a gyökeret **lesüllyesztjük**

```
[40, 70, 20, 90, 50, 80, 60, 10]   kezdet
[90, 70, 80, 40, 50, 20, 60, 10]   kupac
[10, 20, 40, 50, 60, 70, 80, 90]   rendezett
```

Tulajdonság	Kupacrendezés
Lépésszám	**O(n log n)** – minden esetben
Segédtár	**O(1)** – helyben rendez
Stabil	nem

A kurzus összefoglalása

Adatszerkezetek – áttekintés

Adatszerkezet	Elérés	Keresés	Beszúrás	Törlés	Hét
tömb	**O(1)** index szerint	O(n) / O(log n) rendezett	O(n)	O(n)	3
dinamikus tömb	O(1)	O(n)	O(1)* a végére	O(n)	3
láncolt lista	O(n)	O(n)	**O(1)** ismert hely	**O(1)** ismert hely	9–10
verem, sor	csak a vég(ek)	–	O(1)	O(1)	7, 8, 11
halmaz (logikai vektor)	–	O(1)	O(1)	O(1)	6
hash tábla	–	**O(1)** átl.	**O(1)** átl.	**O(1)** átl.	11
keresőfa (kiegyens.)	–	O(log n)	O(log n)	O(log n)	15
kupac	max: O(1)	O(n)	O(log n)	max: O(log n)	15

* amortizáltan

Algoritmusok – áttekintés

Feladat	Algoritmus	Lépésszám
keresés	lineáris · strázsás · bináris	O(n) · O(n) · O(log n)
rendezés (egyszerű)	buborék · kiválasztásos · beszúrásos	$O(n^2)$
rendezés (hatékony)	összefésülő · gyors · kupac	O(n log n)
halmazműveletek	összefésülés rendezett tömbökön	O(n + m)
kifejezések	postfix kiértékelés veremmel	O(n)
mintakeresés	naiv	O(n · m)
fabejárás	pre-, in-, postorder · szintenként	O(n)

Felkészülés a 16. hétre

Záró dolgozat (60 perc, papíron)

- Igaz/hamis és feleletválasztós kérdések a fogalmakról és a hatékonyságról
- Kódkövetés: mit ír ki egy verem vagy egy sor?
- Pszeudokód kiegészítése (hiányzó kifejezések, relációjelek): lista-, strázsás lista- és keresőfa-műveletek
- Papíron végrehajtandó algoritmusok: rendezés, bináris keresés, bejárás, keresőfa-beszúrás

Gyakorlati jegy (gépes feladat)

- Egy adatszerkezet megvalósítása és alkalmazása **Java vagy Go** nyelven

> Ismételd át minden hét **ellenőrző kérdéseit** és **feladatait** – a dolgozat ezekhez hasonló!

Összefoglalás

- **Bináris keresőfa:** bal < gyökér < jobb; az inorder bejárás rendezett
- Keresés, beszúrás, törlés **O(h)** – törlésnél három eset (levél, egy gyerek, két gyerek)
- **Kiegyensúlyozott** fa: h = O(log n); rendezett beszúrás → degenerált fa
- AVL, piros-fekete fa: **forgatásokkal** tartják egyensúlyban a fát
- **Kupac:** teljes fa tömbben, szülő ≥ gyerek; beszúrás és maximum-kivétel O(log n)
- **Kupacrendezés:** O(n log n), helyben

Jövő héten: záró dolgozat és gyakorlati jegy. Sok sikert!

Ellenőrző kérdések

1. Szúrd be üres keresőfába: 50, 30, 70, 20, 40, 60, 80, 35! Rajzold le, és add meg az inorder bejárást!
2. Töröld az 1. kérdés fájából a 30-at! Hogyan néz ki a fa?
3. Igaz vagy hamis: a bináris keresőfa mindig kiegyensúlyozott bináris fa.
4. Igaz vagy hamis: a keresés egy bináris keresőfában lehet ugyanolyan lassú, mint egy láncolt listában.
5. Szúrd be üres maximum-kupacba: 15, 40, 30, 50, 10! Add meg a tömböt minden lépés után!
6. Miért nem alkalmas a kupac egy tetszőleges elem gyors megkeresésére?

Gyakorlat

✎ Bemutató programok

Program	Java	Go	Mit mutat?
Keresőfa	`BST.java`	`h15/bst`	keresés, beszúrás, a törlés három esete, degenerált fa
Kupac	`MaxHeap.java`	`h15/maxheap`	felszivárogtatás, lesüllyesztés, beépített prioritási sor
Kupacrendezés	`HeapSort.java`	`h15/heapsort`	kupacépítés, rendezés, időmérés

✎ Feladatok

1. **Keresőfa-e?** Döntsd el egy bináris fáról, hogy keresőfa-e! (Vigyázat: nem elég a közvetlen gyerekeket nézni!)
2. **k-adik legkisebb:** keresd meg a keresőfa **k**-adik legkisebb kulcsát!
3. **Intervallum:** írd ki az [a, b] közé eső kulcsokat úgy, hogy csak a szükséges részfákba lépsz be!
4. **Közös ős:** keresd meg két kulcs legközelebbi közös ősét a keresőfában O(h) lépésben!
5. **Alsó egészrész:** keresd meg a legnagyobb kulcsot, amely ≤ **x**!
6. **Top k:** keresd meg egy tömb **k** legnagyobb elemét egy **k** méretű minimum-kupaccal! Mi a lépésszám?

Megoldások: `Exercises15.java`, `h15/exercises`

A gyakorlat második felében: **gyakorló feladatok a záró dolgozathoz** (a hetek ellenőrző kérdései alapján).

✎ Házi feladat (felkészülés)

1. Egészítsd ki a `BST` osztályt egy `successor(x)` metódussal, amely az **x**-nél nagyobb legkisebb kulcsot adja vissza!
2. Valósíts meg egy **telefonkönyvet** keresőfával (név → szám), amely névsorban is ki tudja írni a bejegyzéseket!
3. Valósítsd meg a **minimum-kupacot**, és használd egy egyszerű **feladatütemezőhöz** (a legkisebb határidejű feladat jön először)!
4. Ismételd át az összes hét ellenőrző kérdését, és oldd meg papíron a régi dolgozat mintafeladatait!