

Binární halda (= binary heap)

- je binární strom, kde platí, že každý potomek vrcholku má nižší (vyšší) nebo stejnou hodnotu nežli vrcholek sám
- **vlastnost tvar** = má tvar stromu, který je buď úplně vyváženým binárním stromem nebo, pokud je poslední úroveň stromu nekompletní, uzly se plní zleva do prava
- **vlastnost h** = "býti haldou" = každý uzel je menší/větší nebo roven svým potomkům
- jde o nejjednodušší typ haldy, tzn. MIN nebo MAX je na vrcholu
- v případě, že není poslední hladina haldy zaplněna, jsou uzly ukládány do haldy zleva
- je in place
- složitost nejhorší = $O(n \log n)$
- nejmenší strom = pouze kořen
- pokud indexujeme (od 0) prvky haldy od shora dolů, zleva doprava a rodič je na indexu i , pak potomci jsou na indexech
 - levý syn = $L(i) = 2i + 1$
 - pravý syn = $R(i) = 2i + 2$
 - rodič = $P(i) = \lfloor (i-1)/2 \rfloor$ = dolní celá část např. $9/2=4$

tato vlastnost je zajištěna tím, že v haldě *nevynecháváme mezery*. Graficky si tedy haldou můžeme představit jako pyramidu s useknutou základnou.

- vlastnost **býti haldou** je rekurzivní - všechny podstromy haldy jsou také haldy. Díky této vlastnosti máme zajištěno, že se halda chová jako **prioritní fronta** - na vrcholek haldy vždy musí vystoupit prvek s MIN/MAX prioritou (čehož se využívá u heapsortu = řazení haldou).
- pokud považujeme za vyšší prioritu nižší hodnotu klíče, hovoříme o *min-heap*, při opačném uspořádání o *max-heap*

Funkce heapify - volá se na jeden prvek, který umístí na správné místo a přitom dodrží **h vlastnost**

- umístí zadaný prvek na správné místo v haldě (resp. ještě ve stromu, protože v danou chvíli ještě strom nemusí být haldou) – tzv. zajišťuje vlastnost **býti haldou**
- rekurzivní funkce, která opravuje vnitřní strukturu haldy
- algoritmus začíná od shora
- vždy porovná otce s oběma potomky, a pokud má otec nižší prioritu než některý ze synů (případně než oba synové), tak jej prohodí s vyšším z nich
- dál postupuje větví, kde se prohazovalo a pokud dojde k přenesení nerovnováhy o úroveň níže, tak algoritmus musí opravit i tuto podhaldou. Celá procedura terminuje v okamžiku, kdy buď příslušný otec již nemá žádné potomky, nebo je vlastnost **býti haldou** zaručena (tzn. otec má vyšší prioritu než oba synové).
- složitost $O(\log n)$

Funkce build heap - procedura, která zkonstruuje v zadaném poli haldou tím, že od zdola

- zavolá na všechny vnitřní uzly funkci heapify
- rekurzivní fce
- pro vytvoření haldy se volá na každý uzel
- první krok haldového řazení spočívá v postupném prodlužování haldy z rozsahu $(n/2 - n)$ na $(1 - n)$. Halda je vytvořena po provedení $n/2$ kroků.
- začíná se vždy od spodu zprava, aby se nahoru dostal max/min prvek
- nejhorší složitost = $O(n)$

Tato metoda je volána na začátku při vytvoření původní struktury **halda**. To se děje rychle, protože struktura halda je poněkud vágní. Jedinou podmínkou je, že každý kořenový uzel musí být větší než podřízené uzly.

Funkce heapsort = řazení haldou

- jeden z nejlepších obecných algoritmů řazení založený na porovnání prvků
- je nestabilní
- nahore je max, dole min = znám první a poslední prvek
- je in place
- nejhorší složitost = $O(n \log n)$
- složitost je zaručena, proto je lepší než rychlejší quicksort s $O(n^2)$

Princip

Základem heapsortu je **binární halda**, jejíž základní vlastností je, že se chová jako prioritní fronta.

Pokud z prioritní fronty postupně odebíráme prvky, tak je zřejmé, že tím dochází k jejich řazení. Celý postup se skládá z následujících kroků:

1. Postavme haldou nad zadaným polem.

2. Utrhněme vrchol haldy (prvek s nejvyšší prioritou - nejvyšší nebo nejnižší prvek dle způsobu řazení).
3. Prohodíme utržený prvek s posledním prvkem haldy.
4. Zmenšíme haldy o 1 (prvky řazené dle priority na konci pole jsou již seřazené).
5. Opravme haldy tak, aby splňovaly požadované vlastnosti (přestaly platit v momentě prohození prvků).
6. Dokud má halda prvky **GOTO: 2**.
7. Pole je seřazené v opačném pořadí, než je prioritita prvků.

Funkce delete - mazání prvku

- složitost $O(\log n)$
 - z vrcholu (= z kořene) - vezmu předposlední prvek a vyměním s kořenem
 - provedu heapify, aby zase byla halda
 - odjinud (= z vnitřního uzlu) - uzel zvětšíme na hodnotu vyšší než je kořen (tzn. uzel)
- prohodíme kořen a
- dál viz delete z kořene

Vložení

Operace vkládání (*insert*) funguje přesně naopak než operace *repairTop*. Nový prvek je přidán na konec haldy a probublává vzhůru tak dlouho, dokud má vyšší prioritu než jeho otec. Asymptotická složitost vkládání je $O(\log n)$.

Top

Operace top vrátí hodnotu prvku s MIN/MAX prioritou – tj. vrcholu. Jelikož se jedná o návrat prvního prvku pole, tak má operace top konstantní asymptotickou složitost $O(1)$.

Merge

Operace *merge* sloučí dvě haldy do jedné – vytvoří nové pole, do něž nakopíruje obsah obou hald a na toto nové pole zavolá operaci *heapify*. Asymptotická složitost tohoto postupu je $O(m + n)$, kde *m* je velikost první haldy a *n* je velikost druhé haldy.

/ PrioritniFrontaImplementovanaHeapem*

`public class PrioritniFrontaImplementovanaHeapemOdNuly {`

```

    /* Binarni halda (min-heap)
    // pole prvku fronty-haldy,
    private double[] array; //
    // index posledniho prvku v poli
    private int lastElemIndex = -1;
  
```

```

    *****
    // Verejne metody, ktere vidi uzivatel prioritni fronty
    *****
  
```

```

    /* Konstruktor vytvori prazdnou frontu-haldu pripravenou pojmout az
    maxSize prvku
  
```

```

    public PrioritniFrontaImplementovanaHeapemOdNuly(int maxSize) {
        this.array = new double[maxSize];
    }
  
```

```

    public boolean isEmpty() {
        return size() == 0;
    }
  
```

```

    public boolean isFull() {
        return size() == array.length;
    }
  
```

```

    public final void clear() {
        lastElemIndex = -1;
    }
  
```

```

    public int size() {
        return lastElemIndex + 1;
    }
  
```

```

/* Vlozi prvek do fronty-haldy
public void insert(double element) {
    if (isFull()) {
        throw new IllegalStateException("fronta-halda je plna");
    }

    int slotIndex = ++lastElemIndex; // index volneho mesta
    int parentSlotIndex = parentIndex(slotIndex);
        // index rodice volneho mesta
    while (parentSlotIndex < slotIndex && // dokud nejsem na vrcholu
        element < array[parentSlotIndex]) {
        //dokud je vkladany prvek mensi nez otec slotu
        array[slotIndex] = array[parentSlotIndex];
        //presun rodice o uroven niz
        slotIndex = parentSlotIndex; // tim se slot posune vis
        parentSlotIndex = parentIndex(slotIndex);
    }
    array[slotIndex] = element;
}

/* vrati, ale nevyjme, prvni (nejmensi) prvek
public double first() { //
    if (isEmpty()) {
        throw new IllegalStateException("fronta-halda je prazdna");
    }
    return array[0];
}

/*vyjme a vrati prvni (nejmensi) prvek
public double removeFirst() {
    if (isEmpty()) {
        throw new IllegalStateException("fronta-halda je prazdna");
    }
    double min = array[0];
    array[0] = array[lastElemIndex]; // posledni prvek na vrchol
    lastElemIndex--;
    repairTop(0, array, lastElemIndex); // opravi haldu
    return min;
}

/*Slouzeni tuto frontu-haldu s jinou
public void merge(PrioritniFrontaImplementovanaHeapemOdNuly heap) {
    double[] newArray = new double[size() + heap.size()];
    System.arraycopy(array, 0, newArray, 0, size());
    System.arraycopy(heap.array, 0, newArray, size(), heap.size());
    array = newArray;
    lastElemIndex = newArray.length - 1;
    buildHeap(array);
}

@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i <= lastElemIndex; i++) {
        builder.append(array[i]).append(" ");
    }
    return builder.toString();
}

```

```

*****
// Pomocne metody
*****
private static void swap(double[] array, int i, int j) {
    double tmp = array[i];
    array[i] = array[j];
    array[j] = tmp;
}

```

```

}

private static int parentIndex(int childIndex) {
    return (childIndex - 1) / 2;
}

private static int leftChildIndex(int parentIndex) {
    return (parentIndex + 1) * 2 - 1;
}

/* Prohazi prvky pole array aby melo h-vlastnost. Metoda je staticka,
aby mohla byt volana ve staticke metode heapSort
private static void buildHeap(double[] array) {
    for (int parentIndex = parentIndex(array.length - 1); parentIndex
        >= 0; parentIndex--) {
        repairTop(parentIndex, array, array.length - 1);
        // resp. repairTopIterativne(parentIndex, array,
            lastElementIndex);
    }
}

/* Opravi haldu, jejiz koren neni na spravnem miste. Metoda je
staticka, aby mohla byt volana ve staticke metode buildheap
private static void repairTop(int parentIndex, double[] array, int
lastElementIndex) {
    int minChildIndex = leftChildIndex(parentIndex);
        // index mensiho potomka
    if (minChildIndex > lastElementIndex) {
        return; // nema potomky
    }
    int rightChildIndex = minChildIndex + 1; // index praveho potomka
    if (rightChildIndex <= lastElementIndex && // ma druheho potomka
        array[rightChildIndex] < array[minChildIndex]) {
        minChildIndex = rightChildIndex; // pravy potomek je mensi
    }
    if (array[parentIndex] > array[minChildIndex]) {
        // rodic je vetsi nez mensi potomek
        swap(array, parentIndex, minChildIndex);
        repairTop(minChildIndex, array, lastElementIndex);
        // oprava podstromu mensiho potomka
    }
}

private static void repairTopIterativne(int parentIndex, double[]
array, int lastElementIndex) {
    double tmp = array[parentIndex];
    int minChildIndex = leftChildIndex(parentIndex);
    if (minChildIndex > lastElementIndex) {
        return; // nema potomky
    }
    if (minChildIndex < lastElementIndex && array[minChildIndex] >
        array[minChildIndex + 1]) {
        minChildIndex++;
    }

    while (minChildIndex <= lastElementIndex && tmp >
        array[minChildIndex]) {
        array[parentIndex] = array[minChildIndex];
        parentIndex = minChildIndex;
        minChildIndex = leftChildIndex(minChildIndex);
        if (minChildIndex < lastElementIndex && array[minChildIndex] >
            array[minChildIndex + 1]) {
            minChildIndex++;
        }
    }
    array[parentIndex] = tmp;}

```

```

*****
// HeapSort
*****
    public static void heapSort(double[] array) {
        buildHeap(array);
        for (int i = array.length - 1; i >= 0; i--) {
            swap(array, 0, i);
            repairTop(0, array, i - 1);
        }
    }

// Testovani
public static void main(String[] args) {
    PrioritniFrontaImplementovanaHeapemOdNuly pf1 = new
        PrioritniFrontaImplementovanaHeapemOdNuly(1000);
    pf1.insert(1);
    pf1.insert(13);
    pf1.insert(10);
    PrioritniFrontaImplementovanaHeapemOdNuly pf2 = new
        PrioritniFrontaImplementovanaHeapemOdNuly(1000);
    pf2.insert(6);
    pf2.insert(8);
    pf2.insert(2);
    pf1.merge(pf2);
    System.out.println(pf1);
    while (!pf1.isEmpty()) {
        System.out.println(pf1.removeFirst()); // vypise: 1.0 6.0 8.0 13.0
    }

    double[] x = {10, 5, 2, 8, -3, 1000, 1, 3, 5, 9};
    heapSort(x);
    System.out.println(Arrays.toString(x));
    // vypise: [1000.0, 10.0, 9.0, 8.0, 5.0, 5.0, 3.0, 2.0, 1.0, -3.0]
}
}

```

Implementace binarni haldy (min-heap)

```

public class BinaryHeap {
    private int[] array;
    private int size; //velikost haldy...je treba mit na pameti, ze
    indexujeme od 1

    * Konstruktor
    * @param arraySize velikost haldy
    public BinaryHeap(int arraySize) {
        this.array = new int[arraySize + 1]; //na prvni miste nebude nic
        this.size = 0;
    }

    * Konstruktor
    * @param arraySize velikost haldy
    public BinaryHeap(int[] source) {
        this.array = new int[source.length + 1]; //na prvni miste
        nebude nic
        System.arraycopy(source, 0, array, 1, source.length);
        this.size = 0;
    }

    * Provede operaci sloucení hald
    * @param heap halda, která bude sloučena s touto haldou

```

```

public void merge(BinaryHeap heap) {
    int[] newArray = new int[this.size + heap.size + 1];
    System.arraycopy(array, 1, newArray, 1, this.size);
    System.arraycopy(heap.array, 1, newArray, this.size + 1, heap.size);
    size = this.size + heap.size;
    array = newArray;
    heapify(newArray);
}

* Vrati pole vseh prvku v halde
* @return pole vseh prvku v halde
public int[] getAll() {
    return Arrays.copyOfRange(array, 1, this.size + 1);
}

* Vlozi prvek do halde
* @param i prvek k vlozeni
public void insert(int i) {
    size++;
    int index = this.size;
    while (i < array[index / 2] && index != 0) {
        //dokud je nas prvek mensi nez
        //jeho otec
        array[index] = array[index / 2];
        //pak otce posuneme o uroven niz (cimz se nam mezera na vlozeni
        //posune o patro)
        index /= 2; //a opakujemo o uroven vys
    }
    array[index] = i;
    //o patro vys je jiz prvek nizsi, proto vložime prvek prave sem,
    //vlastnost halde byla obnovena
}

public int top() {
    if (getSize() == 0) {
        throw new IllegalStateException("halda je prazdna");
    }
    return array[1];
}

* Odstrani vrchol a vrati ho
* @return vraceny vrchol
public int returnTop() {
    if (getSize() == 0) {
        throw new IllegalStateException("halda je prazdna");
    }
    int tmp = array[1];
    array[1] = array[this.size];
    size--;
    repairTop(this.size, 1);
    return tmp;
}

* @return the size
public int getSize() {
    return size;
}

```

```

* Vytvor haldu ze zdrojoveho pole
* @param array pole
private void heapify(int[] array) {
    for (int i = array.length / 2; i > 0; i--) {
        repairTop(this.size, i);
    }
}

* Umisti vrchol haldy na korektni misto v halde (opravi haldu)
* @param bottom posledni index pole, na který se jeste smi sahnout
* @param topIndex index vrsku haldy
private void repairTop(int bottom, int topIndex) {
    int tmp = array[topIndex];
    int succ = topIndex * 2;
    if (succ < bottom && array[succ] > array[succ + 1]) {
        succ++;
    }
    while (succ <= bottom && tmp > array[succ]) {
        array[topIndex] = array[succ];
        topIndex = succ;
        succ = succ * 2;
        if (succ < bottom && array[succ] > array[succ + 1]) {
            succ++;
        }
    }
    array[topIndex] = tmp;
}

```