Třídící algoritmy

Obsah

Algoritmy

Selection sort

Průchodem hledáme největší nebo nejmenší prvek. Když ho najdeme tak ho zařadíme na začátek nebo konec a o tuto část zmenšíme příští hledání. Opakujeme dokud pole není seřazené.

Vlastnosti

- Jednoduchá implementace
- Ideální pro malé kolekce
- Stabilní
- Vázne u velkých kolekcí, i přes krásnou prostorovou složitost

```
public static void SelektioSort<T>(T[] array) where T : IComparable<T>
{
    for (int i = array.Length - 1; i >= 1; i--)
    {
        // najdeme maximum
        int maxi = 0;
        for (int ii = 1; ii <= i; ii++)
        {
            if (array[maxi].CompareTo(array[ii]) == -1)
            {
                maxi = ii;
```

```
            }
        }

        // pošleme na konec
        T tmp = array[i];
        array[i] = array[maxi];
        array[maxi] = tmp;
    }
}
```

Zdroj

Bubble sort

Procházíme dokud není seřazeno. Vždy vememe `n`-tý a `n+1`-tý prvek a porovnáme je. Pokud zjistíme, že nám pořadí nevyhovuje, tak je vyměníme. Lehčí prvky nám tedy probublávají na konec.

Vlastnosti

- Implementace možná i jednodušší než u Selection sortu
- Také stabilní
- Znovu nedostatečný pro velké kolekce, i přes krásnou prostorovou složitost

```
public static void BaubySort<T>(T[] array) where T : IComparable<T>
{
    for (int thei = 0; thei < array.Length; thei++)
    {
        // procházíme pole na druhou aby měl každý prvek šanci probublat
        for (int j = 1; j < array.Length - thei; j++)
        {
            // máme posouvat číslo vpřed?
            if (array[j].CompareTo(array[j - 1]) == -1)
            {
                T tmp = array[j];
                array[j] = array[j - 1];
                array[j - 1] = tmp;
            }
        }
    }
}
```

Zdroj

Insertion sort

Na každém prvku zkontroluje jestli by mohl sedět v řadě. Pokud ne, tak ťapeme zpátky dokud nenajdeme vhodné místo pro vložení vybraného prvku.

Vlastnosti

- Malé kolekce jsou pro něj hračka
- Stabilita není problém
- Tudy cesta velkým kolekcím nevede, i přes krásnou prostorovou složitost

```
public static void HateInsertKeySort<T>(T[] array) where T : IComparable<T>
{
    for (int i = 1; i < array.Length; i++)
    {
        // když najdeme menší prvek, tak ho pošleme dozádu
        if (array[i - 1].CompareTo(array[i]) == 1)
        {
            SendBack(array, i);
        }
    }
}

static void SendBack<T>(T[] array, int index) where T : IComparable<T>
{
    int lookback = index - 1;
    while (lookback >= 0)
    {
        // koncová podmínka
        if (array[lookback].CompareTo(array[index]) != 1)
        {
            break;
        }

        // cestou zpátky vymněňujeme
        T tmp = array[index];
        array[index] = array[lookback];
        array[lookback] = tmp;

        lookback--;
        index--;
    }
}
```

Zdroj

Heap sort

V první části prvky uspořádáme do haldy (binární). Následně můžeme vzít prvek na první pozici (kořen stromu) a přesunout ho na konec řady. Při příštím opakování bereme v potaz o prvek méně.

Vlastnosti

- Konstatní časová komplexita napříč případy
- Postrádá stabilitu

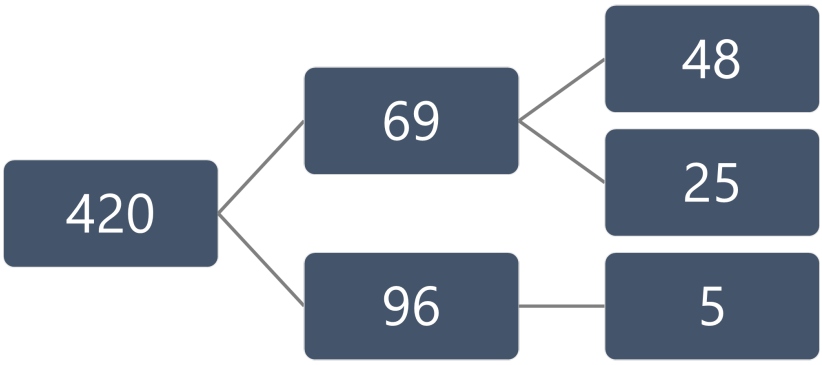
- Konečně něco co toho hodně schrousnte

Halda

Stromová struktura, která splňuje, že každý potomek má hodnotu menší nebo stejnou jako rodič. Z toho vyplívá, že kořen je prvek s nejvyšší hodnotou.

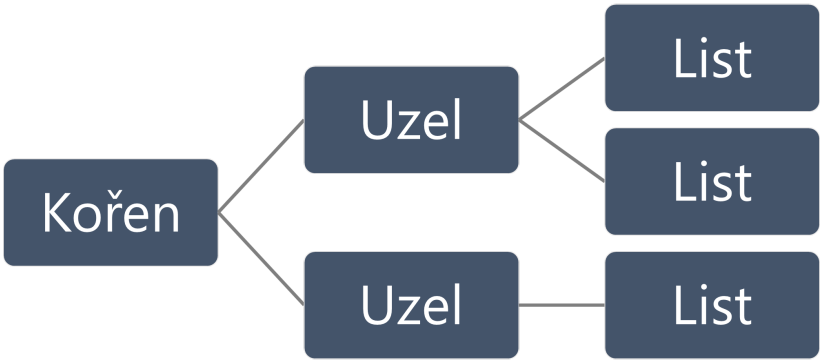
Binární halda

Halda, která splňuje podmínky binárního stromu.



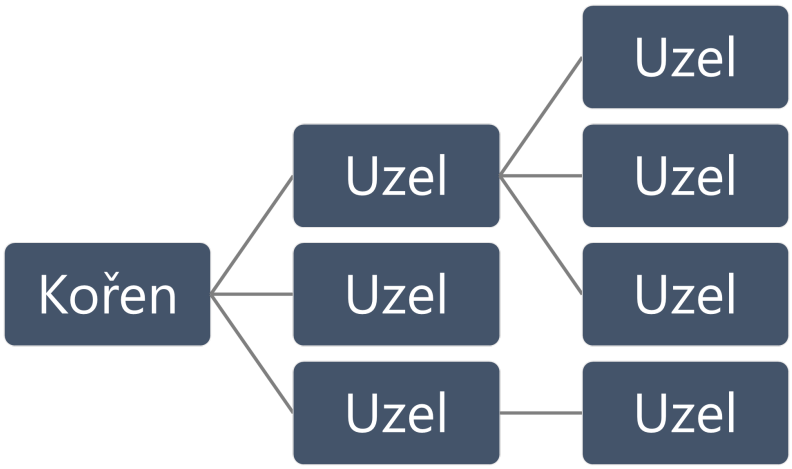
Binární strom

Strom, který splňuje, že jeho počet dětí je právě ≤2.



Strom

Datová struktura, jejíž graf se podobá opravdovému živému stromu. Každý uzel může mít své potomky. Pokud žádné nemá nazýváme ho *list*. Uzel, který nemá žádného rodiče, je *kořen*.




```
public static void HaldaSort<T>(T[] array) where T : IComparable<T>
{
    int index = array.Length - 1;
    // procházíme pozadu
    while (index > 0)
    {
        // složíme binary heap
        Heapify(array, index + 1);

        // pošleme maximum na konec
        T tmp = array[0];
        array[0] = array[index];
        array[index] = tmp;
        index--;
    }
}

static void Heapify<T>(T[] array, int count) where T : IComparable<T>
{
    // necháme všechny uzly probublat k maximu
    for (int i = 0; i < count; i++)
        Bubbly(array, i);
}

static void Bubbly<T>(T[] array, int index) where T : IComparable<T>
{
    while (index != 0)
    {
        int parent = IxParent(index);

        // prohodíme s rodičem pokud potřeba
        if (array[parent].CompareTo(array[index]) == -1)
        {
            T tmp = array[parent];
            array[parent] = array[index];
            array[index] = tmp;
            index = parent;
        }
        else
```

```
            return;
        }
    }

    // pomocné vzorce
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    static int IxParent(int i) => (i - 1) / 2;
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    static int IxLeftChild(int i) => 2 * i + 1;
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    static int IxRightChild(int i) => 2 * i + 2;
```

Zdroj

Merge sort

Rekurzivně dělíme prvky až se dostaneme k jednoduchému porovnávání. Cestou zpátky z rekurze pak skládáme (sléváme) zpátky. Už stačí jen opakovat a pokud prvky zdrojů byly seřazené tak na koneci skončíme s jednou sadou seřazených prvků.

Vlastnosti

- Časová komplexita se nemění
- Drží stabilitu
- Zvládne toho hodně, ale přidá na účet paměti

Složitost

Časová složitost merge sortu je stálá napříč případy a to příjemná `nlog(n)`. Mezitím prostorová nemusí být ideální pro každé využití a je `O(n)`.

Princip slévání

Máme dvě kolekce prvků, které potřebujeme slít. Držíme si tedy pozici z první, z druhé a pozici pro destinaci. Porovnáme prvky na kterých stojíme v prvním a druhém zdroji a vložíme do cíle. Pozice samozřejmě příslušně změníme.

```
public static void MargeSort<T>(T[] array) where T : IComparable<T>
{
    Recursive(array, 0, array.Length - 1);
}

static void Recursive<T>(T[] array, int start, int end) where T : IComparable<T>
{
    // base case rekurze
    if (start >= end)
        return;

    int midbeast = (start + end) / 2;
```

```
    // rekurzivní část
    Recursive(array, start, midbeast);
    Recursive(array, midbeast + 1, end);

    // složení
    Merge(array, start, midbeast, end);
}

static void Merge<T>(T[] array, int start, int mid, int end) where T :
IComparable<T>
{
    // do polí si hodíme to, co budeme třídit
    T[] left = new T[mid - start + 1];
    T[] right = new T[end - mid];
    for (int i = 0; i < left.Length; i++)
        left[i] = array[start + i];
    for (int i = 0; i < right.Length; i++)
        right[i] = array[mid + 1 + i];

    // skládání zpět
    int ai = start;
    int ri = 0, li = 0;
    while (li < left.Length && ri < right.Length)
    {
        // co hodíme první?
        if (left[li].CompareTo(right[ri]) != 1)
        {
            array[ai] = left[li];

            // posun v levém poli
            li++;
        }
        else
        {
            array[ai] = right[ri];

            // posun v pravém poli
            ri++;
        }
        // posun v destinaci
        ai++;
    }

    // dohození zbytku
    while (li < left.Length)
    {
        array[ai] = left[li];

        // posun v levém poli
        li++;
        // posun v destinaci
        ai++;
    }
```

```
    while (ri < right.Length)
    {
        array[ai] = right[ri];

        // posun v pravém poli
        ri++;
        // posun v destinaci
        ai++;
    }
}
```

Zdroj

Quick sort

Vybereme si jeden prvek (klidně náhodný). Podle tohoto prvku nyní budeme rozdělovat zbytek na menší a větší. Stejně se pak chováme k oboum polovinám.

Vlastnosti

- Jménu se nedá věřit - může spomalit až na `O(n^2)`
- Na stabilitu kašle
- Hodně vykrmuje pamět

```
public static void QuickieSort<T>(T[] array) where T : IComparable<T>
{
    RecursiveKekel(array, 0, array.Length - 1);
}

static void RecursiveKekel<T>(T[] array, int start, int end) where T :
IComparable<T>
{
    if (start >= end)
        return;

    int pivot = start;

    // pivot hodíme na konec
    {
        T tmp = array[pivot];
        array[pivot] = array[end];
        array[end] = tmp;
    }

    int index = start;

    for (int i = start; i <= end - 1; i++)
    {
        // když je prvek menší než pivot prohodíme
        if (array[i].CompareTo(array[end]) == -1)
```

```
        {
            T tmp = array[i];
            array[i] = array[index];
            array[index] = tmp;

            index++;
        }
    }

    // prohodíme pivot s indexem
    {
        T tmp = array[end];
        array[end] = array[index];
        array[index] = tmp;
    }

    // třídíme *poloviny*
    RecursiveKekel(array, start, index - 1);
    RecursiveKekel(array, index + 1, end);
}
```

Zdroj

Counting sort

Vytvoříme si pole ve kterém budeme počítat všechny výskyty prvků. Hodnotu prvku budeme používat jako index v tomto poli. Všechny výskyty tedy spočítáme a pak nám už jen zbývá přečíst pole s počty výskytů v pořadí a máme seřazeno.

Vlastnosti

- Komplexitou září
- Potřebuje vědět hodnotu prvku - porovnání mu nestačí
- V základní podání alokuje dle rozdílu hodnot minima a maxima - numusí vyhovovat v každém případě

```
public static void CounterProductiveSort(int[] array)
{
    // najdeme si minimum, maximum
    int max = array.Max();
    int min = array.Min();
    // vypočítáme si rozmezí
    int range = max - min + 1;

    // pole pro počítání výskytů
    int[] count = new int[range];
    // pole pro výsledek
    int[] output = new int[array.Length];

    // sečteme počet všech prvcků
    for (int i = 0; i < array.Length; i++)
```

```
        {
            count[array[i] - min]++;
        }

        // odečteme
        for (int i = 1; i < count.Length; i++)
        {
            count[i] += count[i - 1];
        }

        // složíme výstup
        for (int i = array.Length - 1; i >= 0; i--)
        {
            output[count[array[i] - min] - 1] = array[i];
            count[array[i] - min]--;
        }

        // nakopírujeme zpátky
        for (int i = 0; i < array.Length; i++)
        {
            array[i] = output[i];
        }
    }
```

Zdroj

Porovnání složitostí

Sort	Časová	Prostorová
Selection	O(n^2)	O(1)
Bubble	O(n^2)	O(1)
Insertion	O(n^2)	O(1)
Heap	O(nlog(n))	O(1)
Merge	O(nlog(n))	O(n)
Quick	O(n^2)	O(n)
Counting	O(n+k)	O(k)

Kódy

- GitHub
  - Selection sort
  - Bubble sort
  - Insertion sort

    - Heap sort
    - Merge sort
    - Quick sort
    - Counting sort

Zdroje

- Selection Sort Algorithm [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/selection-sort/
- Bubble Sort Algorithm [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/bubble-sort/
- Insertion Sort [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/insertion-sort/
- Heap Sort [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/heap-sort/
- Lekce 4 - Heapsort [online]. [cit. 2023-02-21]. Dostupné z: https://www.itnetwork.cz/algoritmy/razeni/algoritmus-heap-sort-trideni-cisel-podle-velikosti
- Merge Sort Algorithm [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/merge-sort/
- Lekce 5 - Merge Sort [online]. [cit. 2023-02-21]. Dostupné z: https://www.itnetwork.cz/algoritmy/razeni/algoritmus-merge-sort-trideni-cisel-podle-velikosti
- QuickSort [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/quick-sort/
- Lekce 6 - Quick sort [online]. [cit. 2023-02-21]. Dostupné z: https://www.itnetwork.cz/algoritmy/razeni/algoritmus-quick-sort-razeni-cisel-podle-velikosti
- Counting Sort [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/counting-sort/x
- Time Complexities of all Sorting Algorithms [online]. [cit. 2023-02-21]. Dostupné z: https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/