Robert Haken [MVP ASP.NET/IIS, MCT]
software architect, HAVIT, s.r.o.
haken@havit.cz, @RobertHaken

Základní algoritmy v praxi

Agenda

Intro

Řazení

Vyhledávání

Datové struktury

LINQ to Objects

Intro

Asymptotická složitost algoritmů

O-notace	SLOŽITOST	N = 10	N = 100	N = 1 000
O(1)	konstantní	1	1	1
O(log n)	logaritmická	3	7	10
O(n)	linární	10	100	1 000
O(n log n)	lineárně logaritmická	33	664	9 965
$O(n^2)$	kvadratická	100	10 000	1 000 000
$O(c^n)$	exponenciální (např. c=2)	1 024	$1{,}27 \cdot 10^{30}$	$1{,}1 \cdot 10^{301}$

paměťová
časová/operační

Pole jako základ všeho

lineární

- souvislý paměťový blok

homogenní

- prvky stejného typu (popř. reference/potomci)

s přímým přístupem

- přímé adresování paměti indexem - O(1)

System.Array v .NET

referenční datový typ

- deklarace nealokuje paměť

```
int[] pole;
```

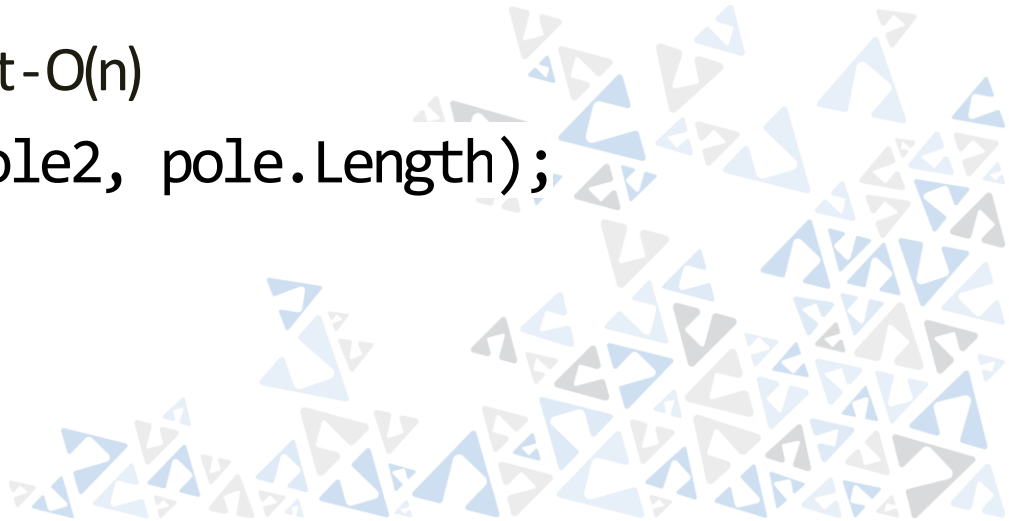
- instance se vytváří na heapu

```
pole = new int[20];
```

non-resizable

- resize = nové pole + kopie dat - O(n)

```
Array.Copy(pole, pole2, pole.Length);
```



System.Array, Windows Debugger

DEMO

List<T>

interně ukládá prvky do pole System.Array

automatické zvětšování pole

- výchozí kapacita 4 (není-li určena explicitně)
- při naplnění se zdvojnásobuje – kopírování O(n)
- přidání prvku typicky O(1), nejhůře O(n)

samo se nezmenšuje

- stálo by kopii O(n) do menšího
- `list.Capacity = list.Count;`
- `list.TrimExcess(); // Treshold 90%`

odebrání prvků = setřesení pole O(n)

Řazení

Řadící algoritmy

stabilní / nestabilní

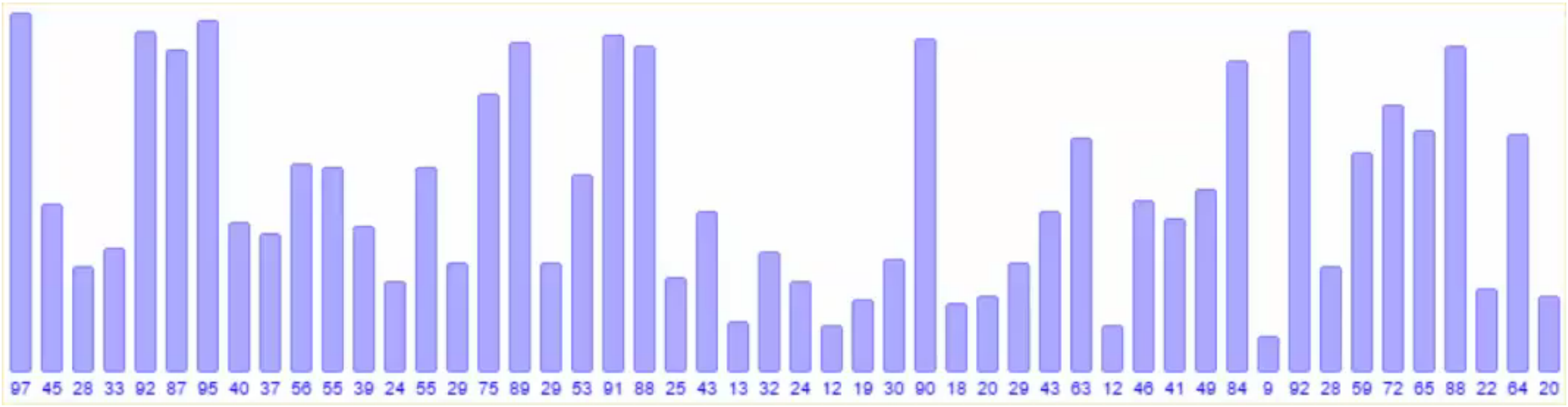
s kvadratickou složitostí – $O(n^2)$

- Bubble-Sort, Shaker-Sort, Select-Sort , Shell-Sort
- Insert-Sort
- jednotlivé průchody na sobě nezávislé

s lineárně logaritmickou složitostí – $O(n \log_2 n)$

- výběrem z binárního stromu = > Heap-Sort
- Quick-Sort
- Merge-Sort

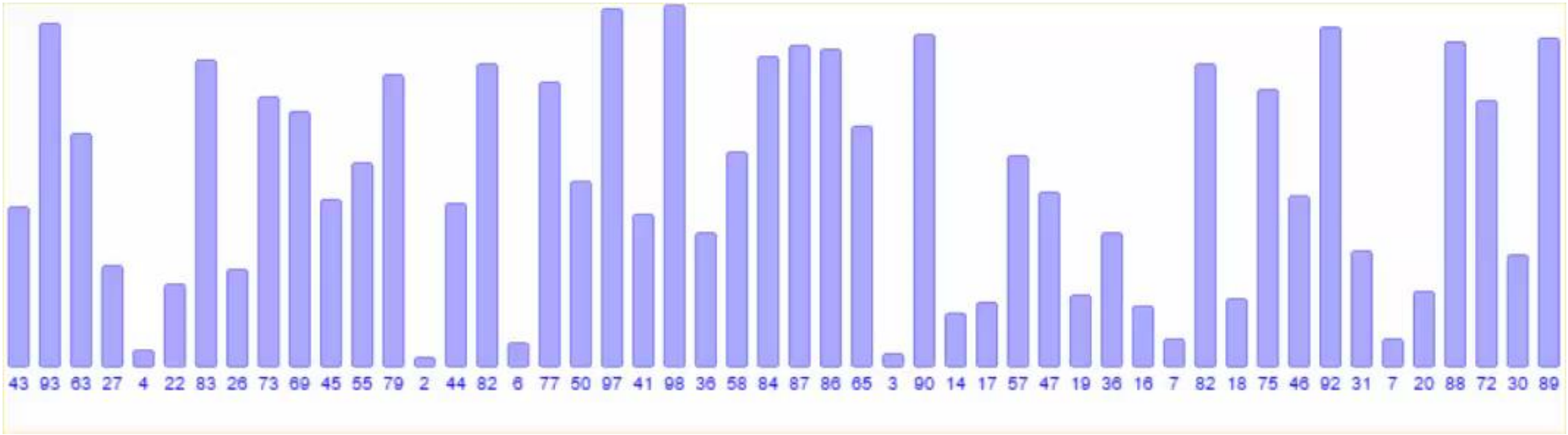
Insert Sort – $O(n^2)$



v každém průchodu zařadím jeden prvek
na své místo do již seřazené části

lze vylepšit pomocí BinnarySearch nad seřazenou částí

Quick Sort – O(n log n)

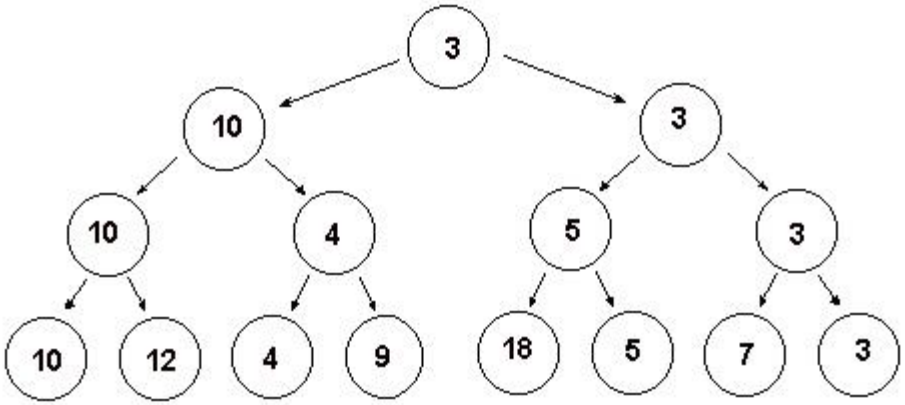


vyberu prvek, tzv. pivot
pole roztřídím na dvě části – menší a větší než pivot
opakuji pro každou takto vzniklou část (rekurze)
dokud nemám část s jedním prvkem

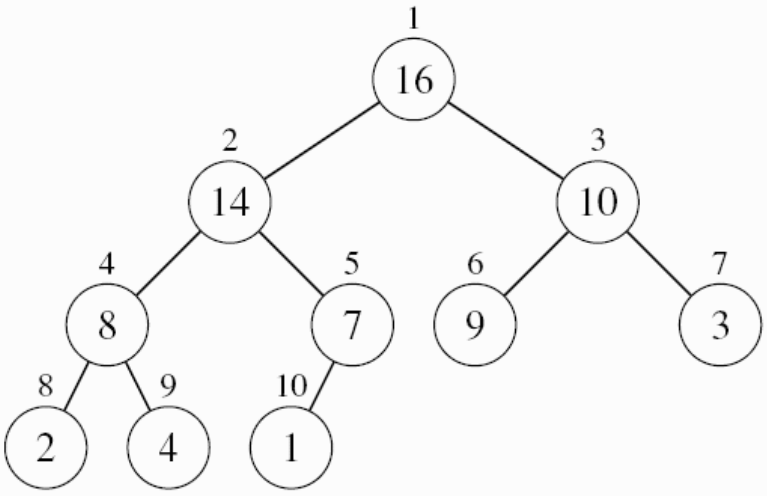
kritickým okamžikem je výběr pivota
v nejhorším případě vede na $O(n^2)$

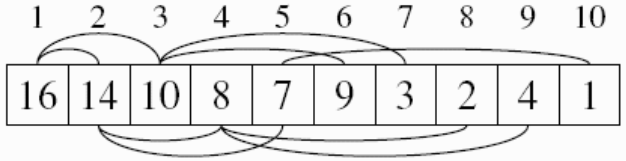
Heap Sort – O(n log n)

principiálně vychází z řazení výběrem z binárního stromu



binární strom reprezentován v poli hromadou = Heap





Heap Sort – O(n log n)

483	604	128	348	412	917	284	821	695	232	105	796	567	941	898	719	546	384	409	500	306	539	865	886	52	345	88	545	55	130	67
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

Řazení v .NET Framework

`Enumerable.OrderBy()`

- stabilní verze QuickSort, pivotem prvek z prostředka partition

`Array.Sort()`, `List<T>.Sort()`

- nestabilní
- prosté pole, primitivní číselné typy a default Comparer => „SZSort“

  nativní implementace QuickSort, pivotem prvek z prostředka partition (jde o obejití CompareTo() ve prospěch rychlejších přímých číselných porovnání), jinak
- NET1.0 – prostý QuickSort, pivotem prvek z prostředka partition
- NET2.0 – pivotem medián (prostřední hodnota) z prvního, prostředního a posledního prvku partition
- NET4.5 – významné optimalizace, O(n log n) i pro worst-case

Array.Sort() v .NET 4.5 – O(n log n) worst

dle `BinaryCompatibility.TargetsAtLeast_Desktop_V4_5`

- false => Depth Limited Quick Sort
  - hloubka rekurze omezena na 32, poté se partition seřadí HeapSortem
  - pivot se volí jako prostřední hodnota z prvního, posledního a prostředního prvku partition
- true => Introspective Sort (1997)
  - Quick Sort do hloubky 2·Floor(log n), pak HeapSort
  - partition 4-16 prvků se řadí Insert Sortem

zarážka rekurze pro partition velkosti 1, 2 a 3 - nedělí se, ale porovná přímo (obě verze)

Vyhledávání

Vyhledávací algoritmy prvku v množině

sekvenční průchod – O(n)

binární půlení seřazené množiny – O(log n)

+ vyhledávací stromy

hashovací tabulka (rozptylování) – O(1)

přímý přístup přes index – O(1)

Hashovací tabulka

pozice prvku v poli se spočítá

index = f(hash_code)

- hashovací (rozptylovací) funkce, např. `H(key) = key mod n`

kolizní strategie

- chaining
  - spojový seznam
  - další pole
- open addressing
  - linear probing - sekvenčně se hledá volné místo
  - quadratic probing - postupně se krok kvadraticky zvyšuje (1., 2., 4., 8., 16. pozice atd.)
  - rehashing/double hashing - další hashovací funkce spočte další pozici nebo posun

.NET Hashtable

hashovací funkce

```
H(HashCode) = [HashCode + 1 + (((HashCode >> 5) + 1) % (HashSize – 1))] % HashSize
```

kolizní strategie = rehashing (double hashing)

$$H_k(HashCode) = [HashCode + k * (1 + (((HashCode >> 5) + 1) \% (HashSize - 1)))] \% HashSize$$

zvětšování při zaplnění

- load factor = 72% (odpovídá 1.0 ctoru!)

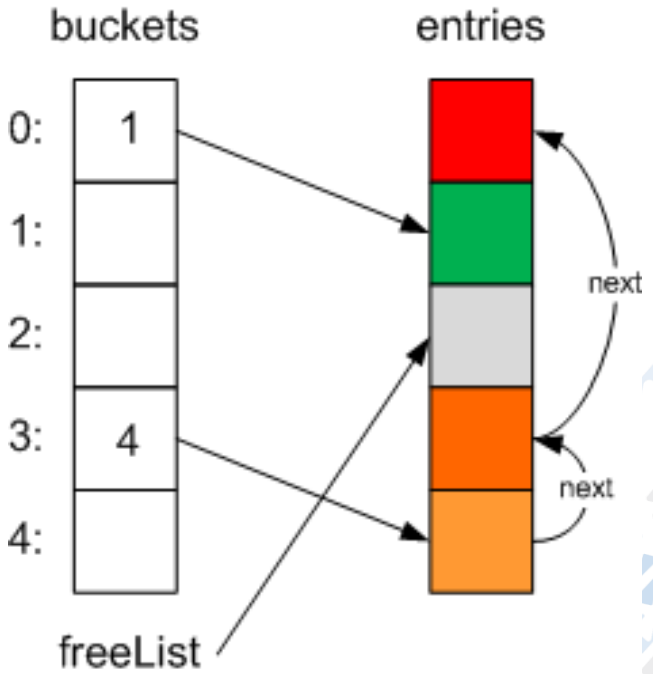
thread-safe – multi-read, single-write

Dictionary<TKey, TValue>

jiná collision strategy – chaining

- spojový seznam v poli (pomocí ofsetů) + FreeList pro díry

```
private struct Entry
{
        public TKey key;
        public TValue value;
        public int hashCode;
        public int next;
}
```



není thread-safe!

rychlejší pro value-types

virtual int Object.GetHashCode()

musí splňovat

- pro stejné objekty (Equals) musí vracet stejný kód
  - pro různé nemusí vracet různý
- dokud se obj nezmění, musí vracet stejný kód (v čase per app)
- dobrá distribuce kódu (i malá změna obj, velká změna kódu)
- rychlý výpočet
- nevyhazovat výjimky

obvyklá implementace – využít existující .NET implementaci

```
public override int GetHashCode()
{
    return this.ID.GetHashCode();
}
```

Operace nad .NET datovými strukturami

	Přidání na konec	Odebrání z konce	Vložení doprostřed	Odebrání z prostředka	Přístup podle indexu	Sekvenční přístup	Vyhledávání elementu	Poznámky
Array	O(n)	O(n)	O(n)	O(n)	O(1)	O(1)	O(n) seřaz. O(log N)	Nejefektivnější využití paměti; vhodné v případě neměnného počtu položek.
List<T>	většinou O(1); nejhůře O(n)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n) seřaz. O(log N)	Vnitřně implementováno pomocí pole, při zaplnění se 2x zvětší. Přidávání nebo odebírání z prostředka či začátku jsou pomalé.
LinkedList<T>	O(1)	O(1)	O(1)*	O(1)*	O(n)	O(1)	O(n)	Obousměrný spojový seznam. Pomalé přístupy doprostřed, rychlé přidávání na začátek a na konec. *) pokud už mám referenci na příslušnou položku
Stack<T>	většinou O(1); nejhůře O(n)	O(1)	N/A	N/A	N/A	N/A	N/A	Zásobník, vhodné pro implementaci různých algoritmů. Last in, first out.
Queue<T>	většinou O(1); nejhůře O(n)	O(1)	N/A	N/A	N/A	N/A	N/A	Fronta, vhodná pro implementaci různých algoritmů. First in, first out.
Dictionary<K,T> **HashTable**	většinou O(1); nejhůře O(n)	O(1)	většinou O(1); nejhůře O(n)	O(1)	O(1)*	O(1)*	O(1)	Vnitřně implementováno pomocí hashovací tabulky, vhodné pro rychlé vyhledávání podle klíče. *) postavení pole Keys a Values má složitost O(n), ale stačí ho postavit jen jednou

LINQ to Objects optimalizace

nespoléhá jen na `IEnumerable<T>` - sekvenční přístup `MoveNext()`

- zkouší různá rozhraní, která by mohla pomoci

indexed access – `ElementAt`, `Skip`, `Last`, `LastOrDefault`

- testuje implementaci rozhraní `IList<T>`, převod z O(n) na O(1)

`Count` zkouší `ICollection` pro O(1) místo O(n)

`Distinct`, `GroupBy`, `Join` používá hashing pro O(n) místo $O(n^2)$

`Contains` zkouší `ICollection`, což např. pro `HashSet` je O(1)

`OrderBy` používá stabilní `QuickSort`

Asymptotická časová složitost algoritmů

O-notace	SLOŽITOST	PŘÍKLADY
O(1)	konstantní	Přímý přístup, HashTable, Dictionary, HashMatch (SQL)
O(log n)	logaritmická	BinnarySearch (půlení intervalu), vyhledávání ve stromu, IndexSeek
O(n)	linární	Sekvenční průchod množinou – vyhledávání v neseřazeném poli, přesuny dat, atp. IndexScan
O(n log n)	lineárně logaritmická	Quick-Sort, Heap-Sort, Merge-Sort, Intro-Sort
$O(n^2)$ $O(n \cdot m)$	kvadratická (m – druhý vstup)	Bubble-Sort, Insert-Sort, Shell-Sort, Select-Sort nested-loops
$O(c^n)$	exponenciální	Prohledávání grafu možných řešení, problém obchodního cestujícího (nalezení nejkratší možné cesty procházející všemi zadanými body na mapě)