Vyhledávací algoritmy

Sekvenční hledání, binární hledání, indexace klíče, BST, hashing.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Vyhledávání

Společně s třídícími algoritmy jedny z nejčastěji používaných.
Postup, při kterém ověřujeme, zda hledaný prvek y leží v množině X

$$y \in X?, \qquad X = \{x_1, ..., x_n\},$$

Všechny prvky x_i zpravidla stejného datového typu.

Aktuální téma, obrovské množství dat, jak je prohledávat?
Požadavek na rychlost, je funkcí n.
Nad setříděnými daty lze provádět efektivněji.

Použití v databázových strojích.
Význam v geoinformatice: efektivní práce s prostorovými daty.
Součástí řady algoritmů výpočetní geometrie.

2. Dělení vyhledávacích algoritmů

Elementární metody vyhledávání

Používají porovnání, pro obecná data nejlépe log n porovnání.

- neuspořádané pole,
- uspořádané pole,
- neuspořádaný seznam,
- uspořádaný seznam,
- binární hledání.

Pokročilé metody

Mohou používat i jiné techniky než porovnání.

- indexování klíče,
- BST stromy,
- RB stromy,
- trie,
- hashování.

3. Efektivita vyhledávacích algoritmů

Kritéria efektivity:

- počet porovnání při úspěšném hledání ($y \in X$),
- počet porovnání při neúspěšném hledání ($y \notin X$),
- režie spojená s vytvořením datové struktury (Insert).

Testování prováděno pro:

- Worst Case,
- Average Case,
- Best Case nemá smysl posuzovat (konstantní).

Efektivita roste při práci nad setříděnou posloupností.
Ovlivňuje zejména počet porovnání pro neúspěšné hledání.

4. Porovnání efektivity jednotlivých metod

Metoda	Worst Case		Average Case		
	Insert	Search	Insert	Search $y \in X$	Search $y \notin X$
Neuspořádané pole	1	n	1	$n/2$	n
Neuspořádaný seznam	1	n	1	$n/2$	n
Uspořádané pole	n	n	$n/2$	$< n/2$	$n/2$
Uspořádaný seznam	n	n	$n/2$	$< n/2$	$n/2$
Binární hledání (uspoř.)	n	$\log n$	$n/2$	$\log n$	$\log n$
Indexování klíčů	1	1	1	1	1
BST stromy	n	n	$\log n$	$\log n$	$\log n$
RB stromy	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Hashing	1	n	1	1	1

5. Popis metody

Lze aplikovat jak na setříděnou tak na nesetříděnou posloupnost.
Aplikuje hrubou sílu, postupné porovnání y se všemi $x_i \in X$.

Na setříděné posloupnosti dosahuje lepších výsledků.
Porovnává pouze prvky $x_i \leq y$ (setříděná).
V případě nenalezení prvku nemusí projít celou X.

Nejlepší případ: hledaný prvek je na 1. pozici.
Nejhorší případ: hledaný prvek na n. pozici.
Průměrný případ: projdeme $n/2$ prvků ($y \in X$), n prvků ($y \notin X$).
U setříděné posloupnosti a $y \notin X$ projdeme pouze $n/2$ prvků.

Odhady složitosti: $O(n)$, $\Omega(1)$, $\Theta(n)$.

6. Sekvenční hledání, nesetříděná data

```
boolean sSearch(int y, int [] X)
{
    boolean result = false;
    for (int i = 0; i < x.length; i++)
    {
        if (y == X[i])
            result = true;
    }
    return result;
}
```

$C_{worst}(n) = 1 + 1 + n + n - 1 + n + 1 + 1 = 3n + 3$, $y \in X$.
$C_{worst}(n) = 1 + 1 + (n + 1) + n - +n + 1 = 3n + 4$, $y \notin X$.
$C_{best}(n) = C_{worst}(n) = C_{aver}(n)$.
$O(n)$, $\Omega(1)$, $\Theta(n)$.

6. Sekvenční hledání, nesetříděná data

```
boolean sSearch(int y, int [] X)
{
    for (int i = 0; i < x.length; i++)
    {
        if (y == X[i])
            return true;
    }
    return false;
}
```

$C_{worst}(n) = 1 + n + n - 1 + n + 1 = 3n + 1, y \in X.$
$C_{worst}(n) = 1 + (n + 1) + n + n + 1 = 3n + 3, y \notin X.$
$C_{best}(n) = 1 + 1 + 1 + 1 = 4.$
$C_{aver}(n) = 0.5(C_{worst}(n) + C_{best}(n)) = 1.5n + 2.$
$O(n), \Omega(1), \Theta(n).$

7. Hledání se zarážkou, nesetříděná data

Hledaný prvek zkopírován “za” pole Y na pozici n.
Prohledávané pole má $n+1$ prvků.

```
boolean sSearch(int y, int [] X)
{
    X[n] = y;
    int i = 0;
    while (X[i]!=y) i++;

    if (i < n) return true;
    else return false;
}
```

$C_{worst}(n) = 1+1+n+n-1+1 = 2n+2,\ y \in X.$
$C_{worst}(n) = 1+1+(n+1)+n+1 = 2n+4,\ y \notin X.$
$C_{best}(n) = 1+1+1+1 = 4.$
$C_{aver}(n) = 0.5(C_{worst}(n)+C_{best}(n)) = n+4.$
$O(n),\ \Omega(1),\ \Theta(n).$

8. Sekvenční hledání, setříděná data

```
boolean sSearchSorted(int y, int [] X)
{
    int i = 0;
    while (i < X.length)
    {
        if (X[i] < y) i++;
        else
        {
          if (X[i] == y) return true;
          else return false;
        }
     }
     return false;
}
```

$C_{worst}(n) = 1 + 1 + n + n - 1 + 1 = 2n + 2, y \in X.$

$C_{worst}(n) = 1 + 1 + (n + 1) + n + 1 = 2n + 4, y \notin X.$

9. Shrnutí

Výhody:

- Snadná implementace.

Nevýhody:

- Nevhodné již pro středně velká data.
- Nutnost předzpracování (setřídění, $O(n \cdot \log n)$).
- Ani po setřídění není efektivita výrazně vyšší.
- Při neúspěšném hledání projdeme 1/2 vs. celé pole.
- Přidávání do setříděné množiny: $O(n)$.

Pro jednorázové vyhledání se nevyplatí množinu X setřídit.
V praxi není používáno, popř. pouze pro malá n.

10. Vyhledávání v nesetříděném seznamu

Stejné vlastnosti jako u nesetříděné posloupnosti.
Postupné procházení seznamu od hlavy a porovnání y se všemi $x_i \in X$.

Nejlepší případ: hledaný prvek je na 1. pozici.
Nejhorší případ: hledaný prvek na n. pozici.
Průměrně provedeme $n/2$ porovnání.

Odhady složitosti: $O(n)$, $\Omega(1)$, $\Theta(n)$.

Vhodné pouze pro malá data nebo jednorázové vyhledání.
Opakované hledání je neefektivní.

11. Sekvenční hledání, seznam

```
public Uzel najdi(String data)
{
   Uzel p = this.prvni;
   while(p != null)
   {
      if (p.data == data)
      {
         return p;
      }
      p = p.dalsi;
   }
   return null;
}
```

Pokud prvek není nalezen, vrací NULL.

12. Binární vyhledávání

Oproti sekvenčnímu přístupu výrazně rychlejší.
Nejefektivnějších elementární metoda založená na porovnání.
Lze aplikovat pouze na setříděné posloupnosti.
Aplikace přístupu Divide & Conquer (rekurze/iterace).

Princip:
Opakované dělení množiny X na dvě části X_1, X_2.
Určíme, ve které části prvek y leží.
Tuto část opět rozdělíme na 2 a postup opakujeme.

Maximální počet porovnání: $\log n + 1$.
Nutnost předzpracování: $O(n \cdot \log n)$.
Pro jednorázové vyhledání se nevyplatí množinu X setřídit.

13. Další vlastnosti binárního hledání

Nutnost udržovat množinu X setříděnou.
Operací Insert je nový prvek zařazen na správnou pozici.
Pokud přidáváme na konec X, pak $O(n)$.

Režie s udržením setříděné množiny větší než s vlastním hledáním.
Existuje mnoho datových struktur, které udržují prvky setříděné.

Technika není účinná ve spojení s lineárními seznamy.
Lineární seznamy neumožňují přímý přístup.
Díky sekvenčnímu přístupu nutno ke $k-$tému prvku projít $k-1$ předcházejících prvků.

14. Ilustrace binárního hledání

$X = \{8, 10, 13, 18, 26, 43, 55, 63, 77, 82, 96\}$, $y = 82$,
$y \in X$?

i	0	1	2	3	4	**5**	6	7	8	9	10
x_i	8	10	13	18	26	**43**	55	63	77	82	96

i	0	1	2	3	4	5	6	7	**8**	9	10
x_i	8	10	13	18	26	43	55	63	**77**	82	96

i	0	1	2	3	4	5	6	7	8	**9**	10
x_i	8	10	13	18	26	43	55	63	77	**82**	96

15. Binární hledání, rekurze

```
int bSearch( int num, int [] numbers, int l, int p)
{
    int m = (l + p) / 2;  //Index prostredniho prvku

    //Prazdny interval
    if ( l > p ) return -1;

    //Cislo nalezno
    if ( num == numbers[m] ) return m;

    //Leva podmnozina
    if ( num < numbers[m] )}
        return bSearch(num, numbers, l, m - 1);

    //Prava podmnozina
    else
        return bSearch(num, numbers, m + 1, p);

}
```

16. Binární hledání, iterace

```
int bSearch( int num, int [] numbers)
{
    int l = 0, p = numbers.length;

    do {
        //Index prostredniho prvku
        int m = (l + p)/ 2;

        // Inkrementuj levy index
        if (num > numbers[m])  l = m + 1;


        //Dekrementuj pravy index
        else p = m - 1;

        } while (l < p && num != numbers[m]);
    }

    //Byl prvek nalezen?
    if ( num == numbers[m])  return m;
```

17. Interpolační vyhledávání

Obdoba binárního vyhledáváním, upravená metoda dělení intervalu.
Místo půlení

$$m = (l + p)/2$$

použita lineární interpolace

$$\begin{aligned} \frac{m - l}{(y - x_l)} &= \frac{(p - l)}{x_p - x_l}, \\ m &= l + \frac{(y - x_l)(p - l)}{x_p - x_l}. \end{aligned}$$

Hledán index m odpovídající hledanému prvku s hodnotou y.
Předpokladem efektivity lineárně rostoucí hodnoty x na intervalu $[l, p]$.
Pak složitost $O(\log(\log(n)) + 1)$, která je téměř konstantní!
V jiných případech méně efektivní než binární hledání.

17. Pokročilé metody hledání

Všechny techniky založené na porovnání mají teoretické omezení.
Pro obecná data počet porovnání nemůže být menší než $\log n$ (binární hledání).
Pro zefektivnění nutno použít jiný přístup než vzájemné porovnávání.

Data uchovávána ve specializovaných strukturách, tabulkách.
Nejjednodušší technika využívá indexaci, složitější hashing.
Výhodou konstantní časová složitost nezávislá na n.
Časté použití v databázových strojích.

Nevýhodou omezená velikost zpracovávané množiny.
Nutnost alokace velkého množství paměti.
Lze zpracovat pouze omezený rozsah numerických hodnot-> transformace.
Může docházet ke kolizím.

I8. Vyhledávání založené na indexaci

Velmi rychlá metoda vyhledávání, přímý přístup do pole.
Konstantní časová složitost bez ohledu na vstupní data $O(1)$.
Dodatečné paměťové nároky.

Předzpracování (inicializace struktur) v čase $O(n)$.
V pomocné struktuře K (pole, tabulka) uchováván pár

```
[klíč, hodnota]
```

Klíč k_i je unikátní, celočíselný, kladný; hodnota odpovídá x_i:

$$K(k_i) = x_i.$$

Fáze:
1) Inicializace, vytvoření pole K,
2) Přiřazení klíčů $k_i \in K$ pro jednotlivé hodnoty x_i.
3) Naplnění struktury (Insert): $K(k_i) \leftarrow x_i$, $i = 1, ..., n$.
4) Vlastní hledání: $x_i = K(k_i)$.

19. Vytvoření pomocné struktury

$X = \{8, 10, 13, 18, 26, 43, 55, 63, 77, 82, 96\}$

Vytvoření párů $[k_i, x_i]$

k_i	2	3	5	7	8	9	11	12	13	15	16
x_i	8	10	13	18	26	43	55	63	77	82	96

Vytvoření pole S

S_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_i	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Naplnění pole, indexace

S_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
x_i	.	.	8	10	.	13	.	18	26	43	.	55	63	77	.	82	96

20. Inicializace pole

Vytvoření pole $K = [k_0, ..., k_{n-1}]$.
Inicializace všech prvků $k_i = NULL$.

```
void init (int n)
{
    int [] K = new int [n];
    for (int i = 0; i < n; i++)
    {
       K[i] = -1;
    }
}
```

Dodatečné časové i paměťové náklady na indexaci $O(n)$.
Tato fáze výrazně delší než vlastní hledání.
Nemá smysl pro jednorázové vyhledání y.

21. Naplnění pole indexů

Do pole postupně přidány všechny prvky, pole nemusí být setříděné.
Index pole i odpovídá klíči k_i, $K(k_i)$ hodnotě prvku x_i

$$K(k_i) = x_i.$$

Některé prvky zůstanou prázdné.
Pozor na duplicitní duplicitní klíče -> přemazání hodnot.

```
public class Item
{
   private int key, val;
   ...
}

void insert (Item item)
{
    k(item.key) = item.val;
}
```

22. Vlastní hledání

Nejrychlejší možné, přímý přístup k prvkům pole.
Neprobíhá žádná porovnávací operace.
Nezávislost na n, složitost $O(1)$.
Výhodné pro rozsáhlé množiny.

Vyhledání probíhá na základě hodnoty klíče k_i

$$x_i = K(k_i).$$

Pro vyhledávání dle hodnoty volba klíče $k_i = x_i$.

```
int search(int key)
{
    return k(key);
}
```

Obtížně použitelné pro velký rozsah klíčů.
Efektivnější volba klíčů vychází z hashování.

23. Binární vyhledávací stromy

BST používány velmi často.
Přidávání prvků do polí seznamu náročné $O(n)$, $\Theta(n)$.
U BST v průměru lepší; $O(n)$, ale $\Theta(\log n)$.
RB stromy ještě lepší $O(\log n)$, $\Theta(\log n)$.

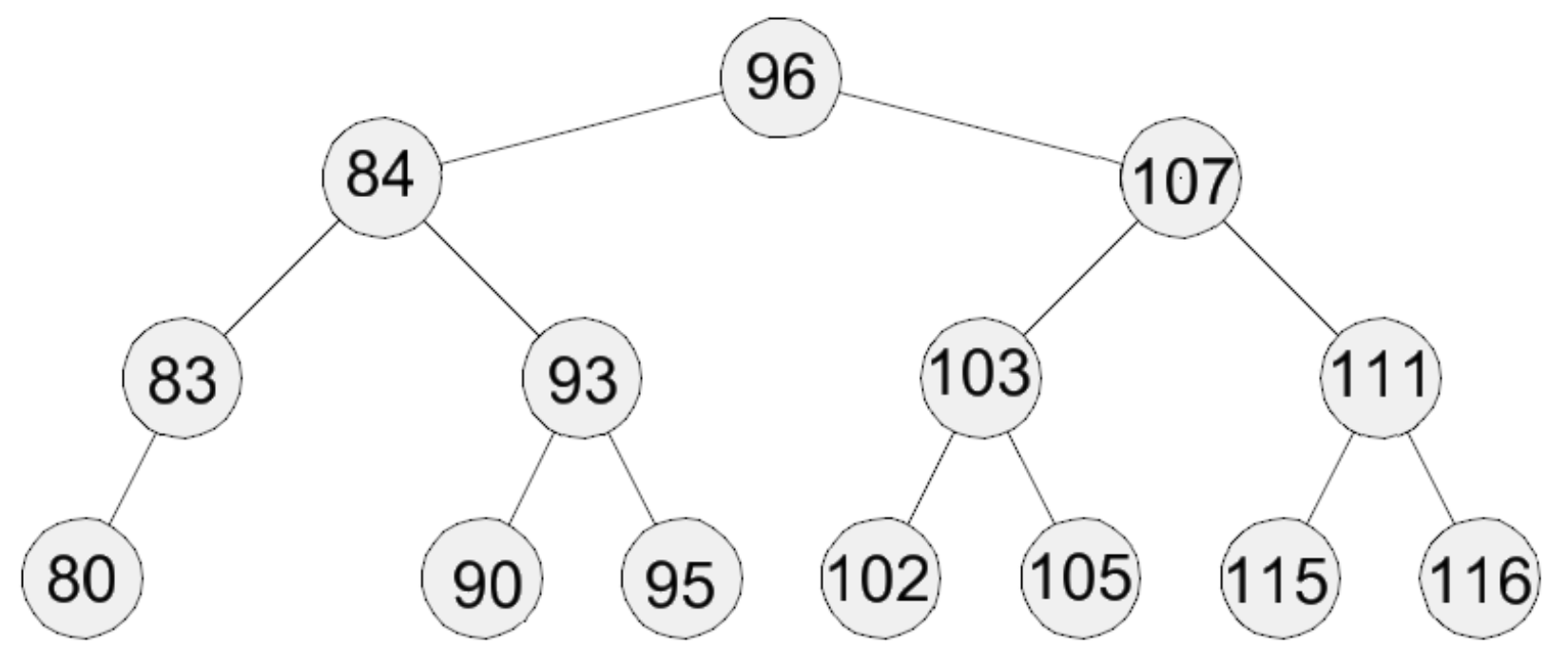
Vyhledávací algoritmus málo citlivý vůči n, složitost $O(\log n)$.
Použití i pro rozsáhlá data.

Při hledání je procházen BST od kořene k uzlům.
Rekurzivní řešení.
Nejprve zpracován levý podstrom, pak pravý.
Vlastní procedura implementačně snadná.

24. Binární vyhledávací strom

Základní vlastnost BST

$$U \geq U_L \wedge U \leq U_P.$$



24. Nalezení prvku v BST

Hledání probíhá od kořene k listům.
Obdoba binárního hledání, zužování intervalu z leva a z prava.

Postup hledání:

1. Pokud je strom prázdný, ukončíme prohledávání.
2. Jinak testujeme shodu uzlu U a hledané položky y.
   V případě shody ukončíme prohledávání.
3. Pokud y menší než hodnota $U.L$, hledáme v L podstromu.
4. Jinak prohledáme P podstrom.

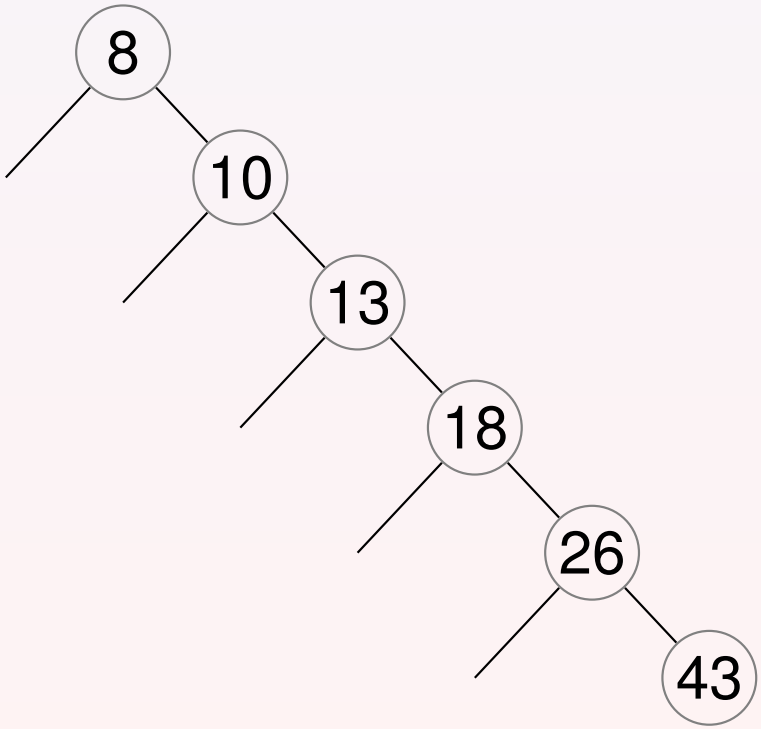
Složitost $C_{aver} = 1.4 \log(n)$ pro náhodně uspořádané klíče.
Pozor na degenerované stromy, strom přejde v seznam!
Složitost $O(n)$ odpovídá hledání v poli/seznamu.

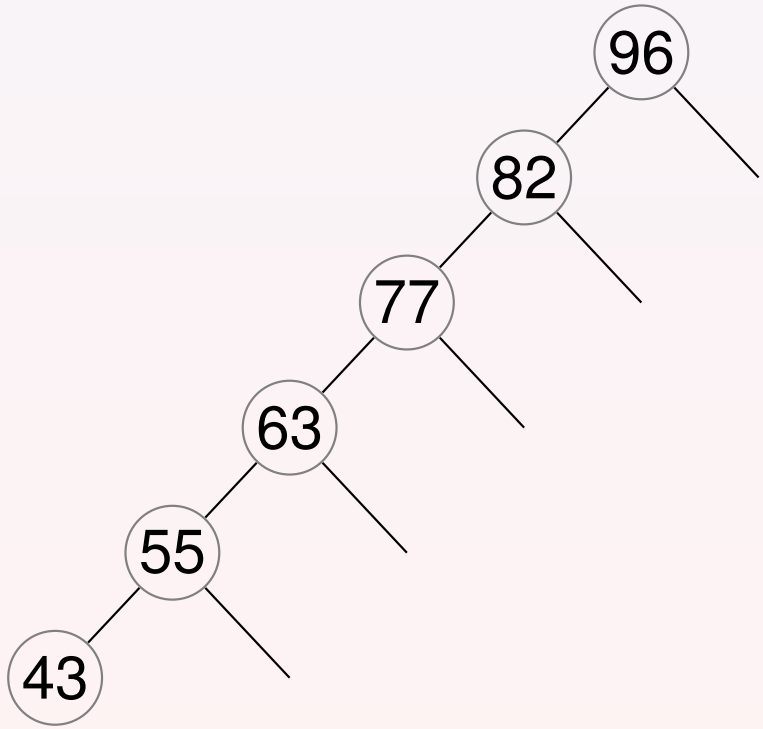
25. Nalezení prvku v BST, zdrojový kód

```
private Uzel search(int data, Uzel u)
{
   //Prazdny strom
   if (u==null) return null;

   //Rovnost s prvkem
   if (data==u.data) return u; //Vraceni uzlu

   //Prohledavani L podstromu
   if (data<u.data) return najdi (data,u.levy);

   //Prohledavani P podstromu
   else return najdi(data,u.pravy);
}

public void searchTree(int data) {  //Nerekurzivni metoda
   search (data, koren);  //Volani privatni rekurzivni metody
}
```

26. Problém 1, uspořádaná posloupnost

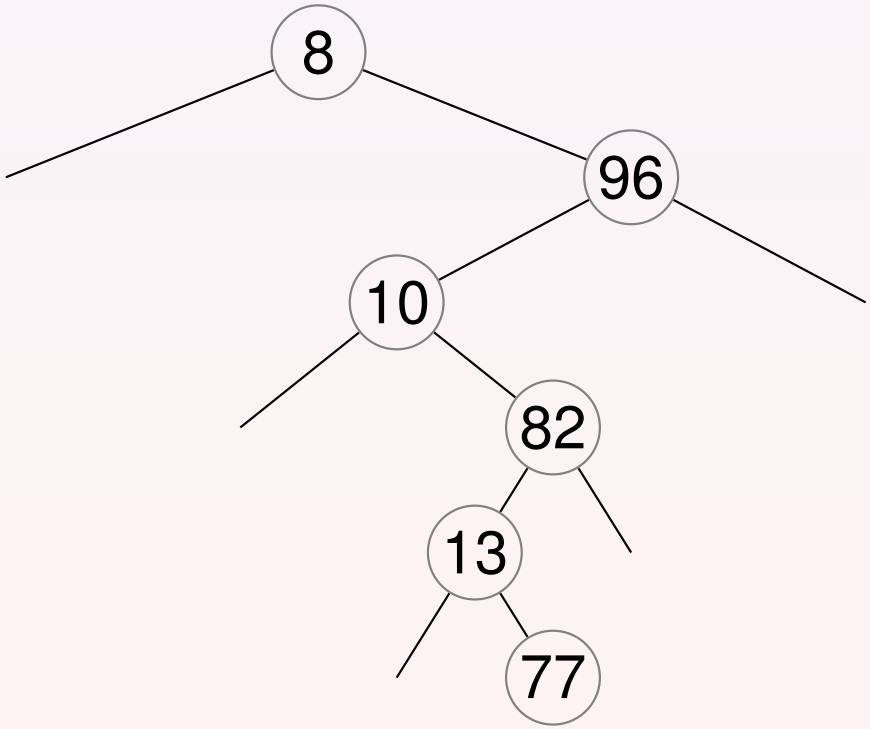
Vzestupně/sestupně uspořádané prvky => degenerovaný BST.
Vede na tvar podobný lineárnímu seznamu s hloubkou *n*.
Efektivita operací stejná jako v uspořádaném seznamu (poli).

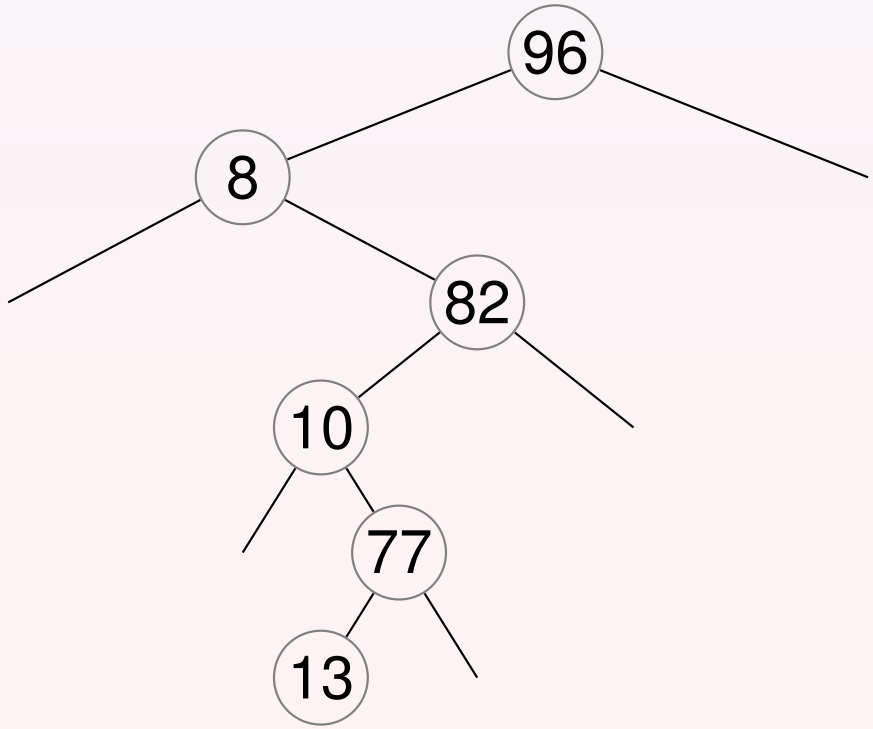




27. Problém 2, CIK-CAK posloupnost

CIK-CAK posloupnost => degenerovaný BST.
Ze setříděné posloupnosti odebírány prvky zleva, zprava.
Vede na tvar podobný lineárnímu seznamu s hloubkou *n*.
Efektivita operací stejná jako v uspořádaném seznamu (poli).





28. Hashing

Hash = otisk.
Zobecnění a rozšíření vyhledávání s indexovanými klíči.
Nepoužívá porovnávací operace, konstantní časová složitost.
Distribuce indexů klíčů po intervalu prováděna inteligentněji.
Jedna z nejefektivnějších technik vyhledávání.
Používána při práci s rozsáhlými daty.

Index prvku x_i určován hashovací funkcí h.
Transformuje klíč k_i na index (adresu) a_i při splnění dalších požadavků.

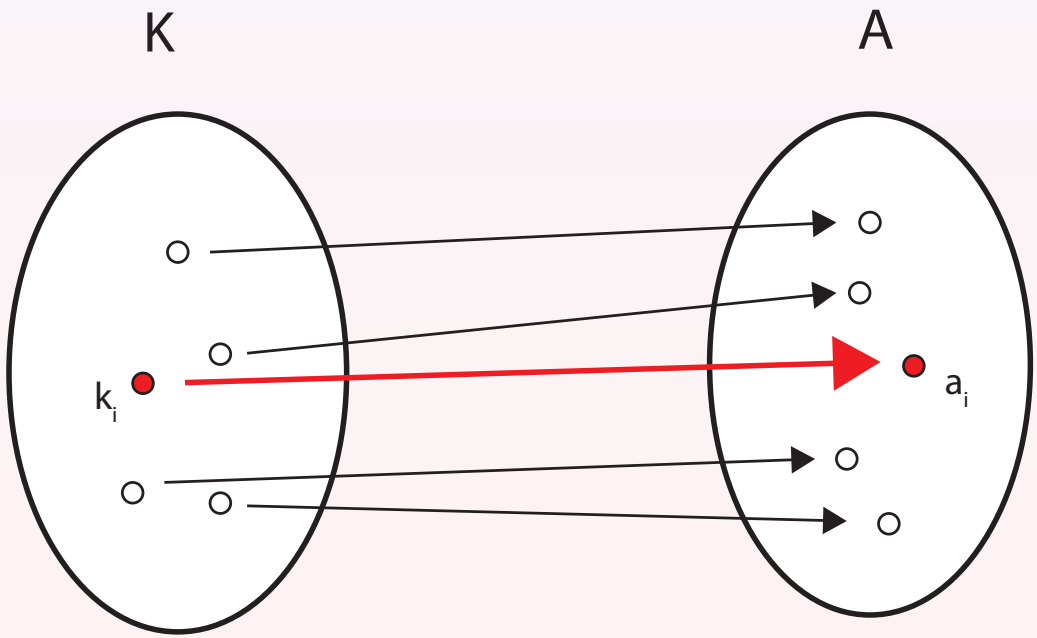
Nevýhodou vznik kolizí, různé klíče transformovány na stejnou adresu.
Lze obejít dodatečnými úpravami (řetězení).
Efektivita závisí na délce klíče (< klíč, > kolizí).
Nepodporuje některé důležité operace: sort.

29. Hashovací funkce

Hashovací funkce

$$h : K \to A, \qquad h(k) = a, \qquad k \in K, a \in A,$$

převádí posloupnost bitů k (klíč) na posloupnost bitů pevné délky a (adresa), $|a| < |k|$.



Prostor klíčů K zobrazen do prostoru adres A.

30. Kolize hashovací funkce

Kolize hashovací funkce $h : K \to A$ nastává pro $k_1, k_2 \in K$, jestliže platí

$$h(k_1) = h(k_2), \qquad k_1 \neq k_2.$$

Důsledek: více klíčů může mít stejné adresy.

Problém kolize není možné odstranit, $|a| < |k|$.
Vhodnou volbou h lze snížit pravděpodobnost výskytu kolize.
U některých problémů (kryptografie) vede k selhání metody.

Požadavky na hashovací funkci:
1) jednoduchost výpočtu: $h(k)$ polynomiální čas,
2) malá časová/paměťová režie,
3) aproximace náhodné funkce (rovnoměrné rozložení a_i pro k_i),
4) rezistence vůči kolizím,
5) rezistence vzoru: $k \neq h^{-1}(a)$,
6) deterministická.

31. Ukázky hashovacích funkcí (1/2)

Modulo hashování:

$$h(k) = k\%M.$$

Volba M: $M \neq 2^p$, lépe prvočíslo, např. $M = 31, 97$.

k	$(k)_2$	$k\%100$	$k\%16$	$k\%97$
12873	11001001001001	73	$2^3 + 2^0 = 9$	69
1073	10000110001	73	$2^0 = 1$	6
173	10101101	73	$2^3 + 2^2 + 2^0 = 13$	76
135	10000111	35	$2^2 + 2^1 + 2^0 = 7$	38
263	100000111	63	$2^2 + 2^1 + 2^0 = 7$	69
247	11110111	47	$2^2 + 2^1 + 2^0 = 7$	53

Při převodu $(k_{10}) \rightarrow (k_2)$, kdy $M = 2^p$ je modulem posledních p bitů!

32. Ukázky hashovacích funkcí (2/2)

Mildsquare hashing

$$h(k) = \frac{M}{c}(k^2 \% M), \qquad \frac{c}{M} = 2^{c-p}, M = 2^p.$$

Multiplication hashing

$$\begin{aligned} h(k) &= (int)(c * (float)k)\%M, \qquad c = 0.616161, \\ h(k) &= (c * k)\%M, \qquad c = 16161, \\ h(k) &= \frac{M}{c}((d \cdot k)\%M), \qquad d = 2654435769. \end{aligned}$$

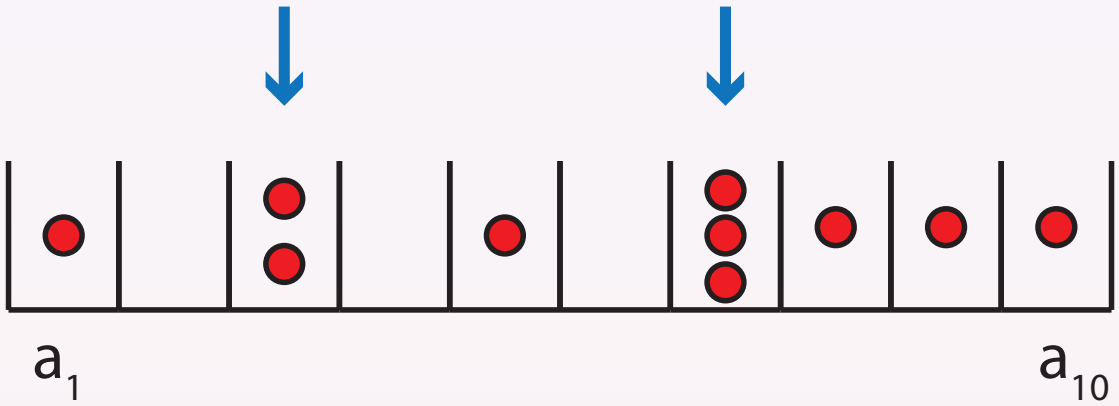
String hashing

$$h(k) = (c \cdot h(k) + h[i])\%M, \qquad h(k) = 0, c = 128.$$

Neexistuje univerzální hashovací funkce.
Pro liché k liché $h(k)$ a naopak -> stejná pravděpodobnost.

33. Odstranění kolizí

Kolizím se při hashování nelze vyhnout.
Jejich vliv na data lze částečně potlačit.



2 základní přístupy:

- akceptace kolize,
- minimalizace počtu kolizí.

Metody:

- Separate Chaining,
- Linear Probing,
- Double Hashing.

34. Separate chaining

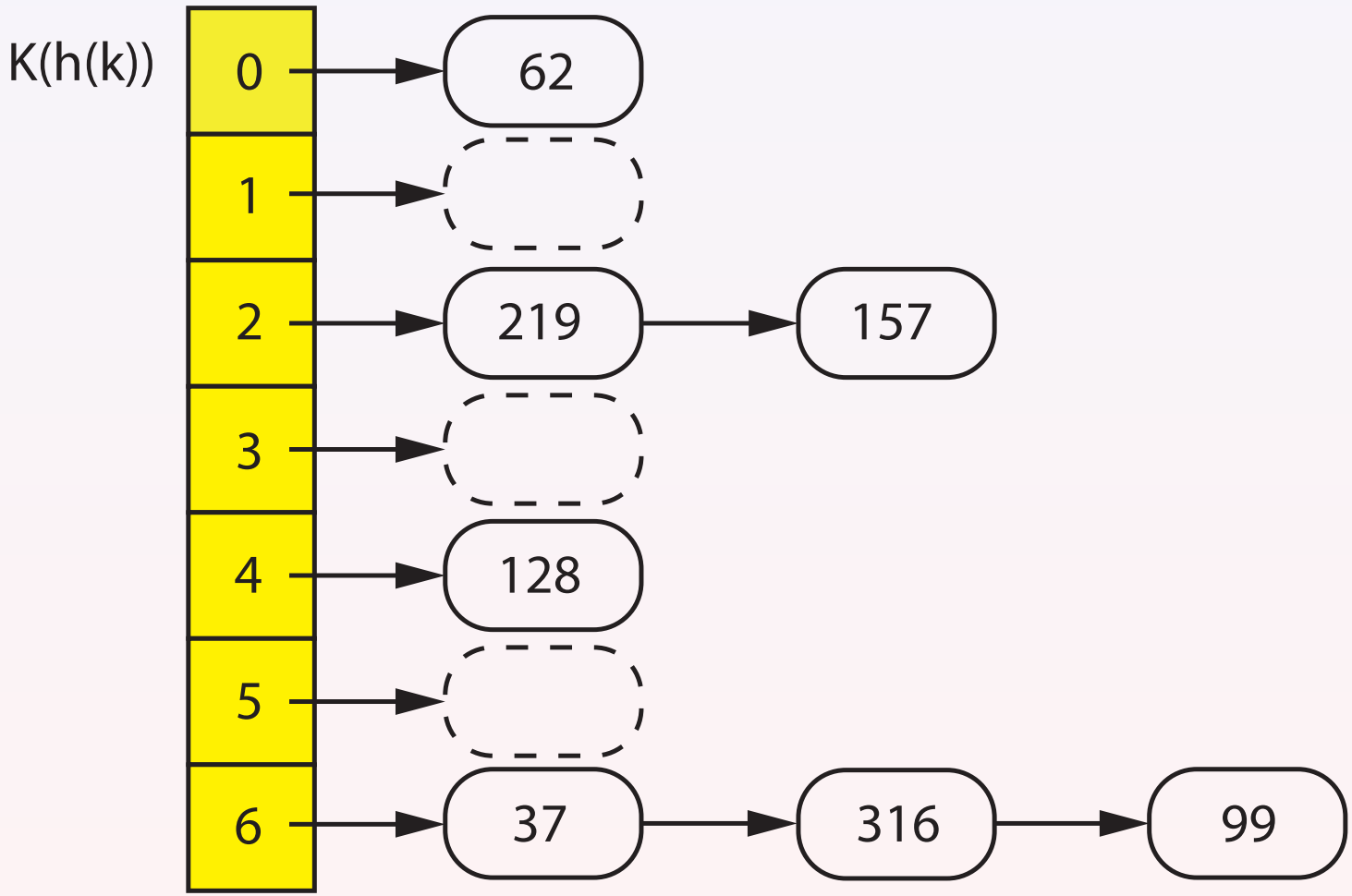
Akceptace kolizí a jejich ošetření.
Úprava datového modelu, kolidující buňka obsahuje lineární seznam.
Dva páry $[k_1, v_1]$, $[k_2, v_2]$ s kolidujícími klíči $k_1 \neq k_2$, $h(k_1) = h(k_2)$,

$$K(h(k_1)) = [v_1, v_2].$$

Celkem M seznamů, $M < N$.
Průměrná délka seznamu $\alpha = N/M$.

Seznam zpravidla neuspořádaný, prochází se sekvenčně.
Aby byla metoda efektivní, seznamy musí být krátké.
Hashovací funkce musí produkovat náhodné hodnoty!

35. Ukázka Separate Chainingu



$$h(k) = k\%M, \qquad M = 31.$$

36. Výkonostní charakteristiky Separate Chainingu (1/2)

Průměrná délka seznamu

$$\alpha = \frac{N}{M}.$$

Režie hledání přímo úměrná délce seznamu.
Nejhorší případ: všechny $h(k)$ směřují do jednoho seznamu, $\alpha = N$.

Příliš malé M: vznik dlouhých seznamů, sekvenční hledání pomalé.
Příliš velké M: vznik prázdných buněk v K.

Pravděpodobnost p, že seznam má více než $t\alpha$ položek

$$p = \left(\frac{\alpha e}{t}\right)^t e^{-\alpha},$$

pro $\alpha = 10$, $t = 2$, $p = 0.0084$.

37. Výkonostní charakteristiky Separate Chainingu (1/2)

Prvek nenalezen, očekávaný počet pokusů není horší než

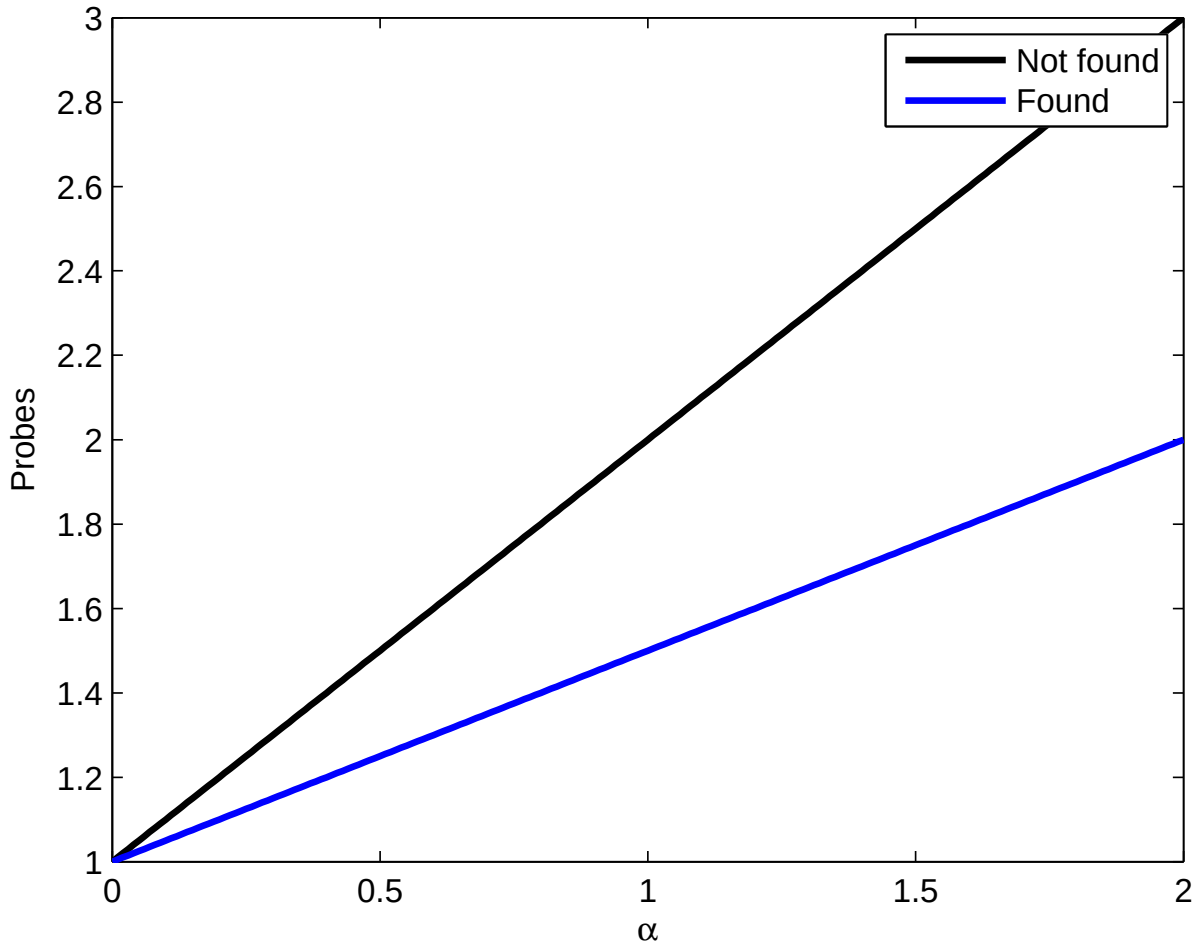
$$n_{prob}(\alpha) = 1 + \alpha.$$

Prvek nalezen, očekávaný počet pokusů není horší než

$$n_{prob}(\alpha) = 1 + \frac{\alpha}{2}.$$

V závislosti na α roste lineárně.
Nemá smysl délku pole příliš prodlužovat, zlepšení není významné.

38. Ilustrace Separate Chainingu



39. Modifikace třídy Uzel

```
private class Uzel
{
      int klic;
      int hod;
      Uzel dalsi;
}

Uzel(int klic_, int hod_, Uzel dalsi_)
{
   klic = klic_;
   hod = hod_;
   dalsi = dalsi_;
}
```

40. Implementace Separate Chainingu (1/3)

Vytvoření pole uzlů.
Každý prvek inicializován na null.
Provede se v lineárním čase $O(n)$.

```
void init (int n)
{
    Uzel [] K = new Uzel [n];
    for (int i = 0; i < n; i++)
    {
       K[i] = null;
    }
}
```

41. Implementace Separate Chainingu (2/3)

Pro daný klíč k spočteme hash $h(k)$.
Test, zda v seznamu již není prvek se stejným klíčem.
Prvek přidán do seznamu na poslední pozici.

```
public void insert(int klic, int hod)
{
    int i = hash(klic);
    if (K[i] == null) //Prvek s timto hashem neni v seznamu
         K[i] = new UzelSC(klic, hod);
    else
    {
        //Test stejnych hodnot (nepovinny)
        UzelSC u = K[i];
        while ((u.dalsi != null) && (hod != u.hod))  u = u.dalsi;

        //Stejna hodnota
        if (hod == u.hod) return;

        //Rozdilna hodnota, pridan jako dalsi prvek
        else u.dalsi = new UzelSC(klic, hod);
    }
```

42. Implementace Separate Chainingu (3/3)

Pro daný klíč k spočteme hash $h(k)$.
Porovnáme klíč k se všemi prvky v seznamu.

```
public int search(int klic)
{
     int i = hash(klic);
     if (K[i] == null)  //Prvek s timto hashem neni v seznamu
          return -1;
     else
     {
         //Test stejnych hodnot
         UzelSC u = K[i];
         while ((u.dalsi != null) && (hod != u.hod))  u = u.dalsi;

         //Stejna hodnota
         if (hod == u.hod) return u.hod;

         //Rozdilna hodnota
         else return -1;
      }
  }
```

43. Linear Probing

Nejvýznamnější hashovací technika, neakceptuje kolize.
Snaží se jim vyhnout vhodným rozmístěním klíčů.
Analogie Car Parking Problem.
Nepoužívá žádné další pomocné datové struktury.
Méně náročná na paměť, ale pomalejší než řetězení

$$h(k) = (h(k) + i)\,\%M.$$

Počet prvků pole $M > N$, pak $\alpha < 1$.
Do prázdného místa ukládány duplicitní klíče.

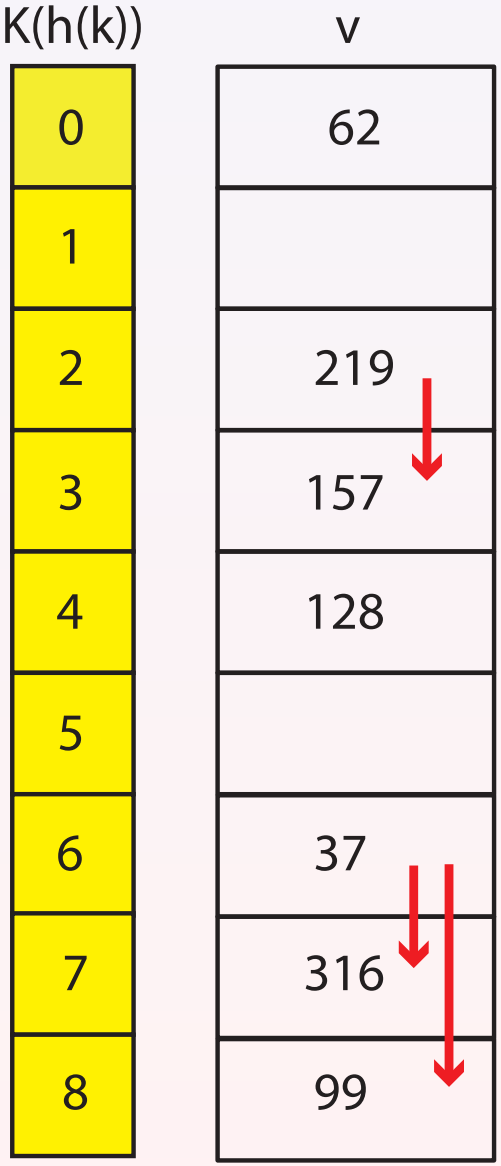
Přidávaný prvek k_i, hash $h(k_i)$.
Pokud $K(h(k_i))$ je již obsazen nějakým k_j, tj. platí

$$h(k_i) = h(k_j) \wedge k_i \neq k_j,$$

umístíme $h(k_i)$ na první volnou pozici k_l.

Velikost pole volena nejčastěji $M = 2N$, $\alpha = 0.5$.

44. Ukázka Linear Probingu



45. Výkonnostní charakteristiky Linear Probingu (1/2)

Položky se stejným $h(k)$ seskupovány do skupin (clusterů).
Délka clusteru

$$\alpha = \frac{N}{M}.$$

Výkonnost LP výrazně závisí na α.
Pro malá α (řídká tabulka) stačí pár posunů
Pro $\alpha \to 1$, zbytečně mnoho skoků, nebezpečí nekonečného cyklu.

Nejlepší případ: každý klíč k hashován do jiného $h(k)$

$$\widehat{n}_{prob} = \frac{1 + 1 + ... + 1}{N} = \frac{N}{N} = 1.$$

Nejhorší případ: všechny klíče hashovány do stejného $h(k)$

$$\widehat{n}_{prob} = \frac{1 + 2 + ... + (N-1)N}{N} = \frac{(1+N)N}{2N} \approx \frac{N}{2}.$$

Počet přesunů $\widehat{n}_{shifts} = \widehat{n}_{prob} - 1$ roste lineárně, neefektivní!

46. Výkonnostní charakteristiky Linear Probingu (1/2)

Odvození velmi složité (Knuth, 1962)
Prvek nenalezen, očekávaný počet pokusů není horší než

$$n_{prob}(\alpha) = 0.5 \left(1 + \frac{1}{(1-\alpha)^2}\right).$$

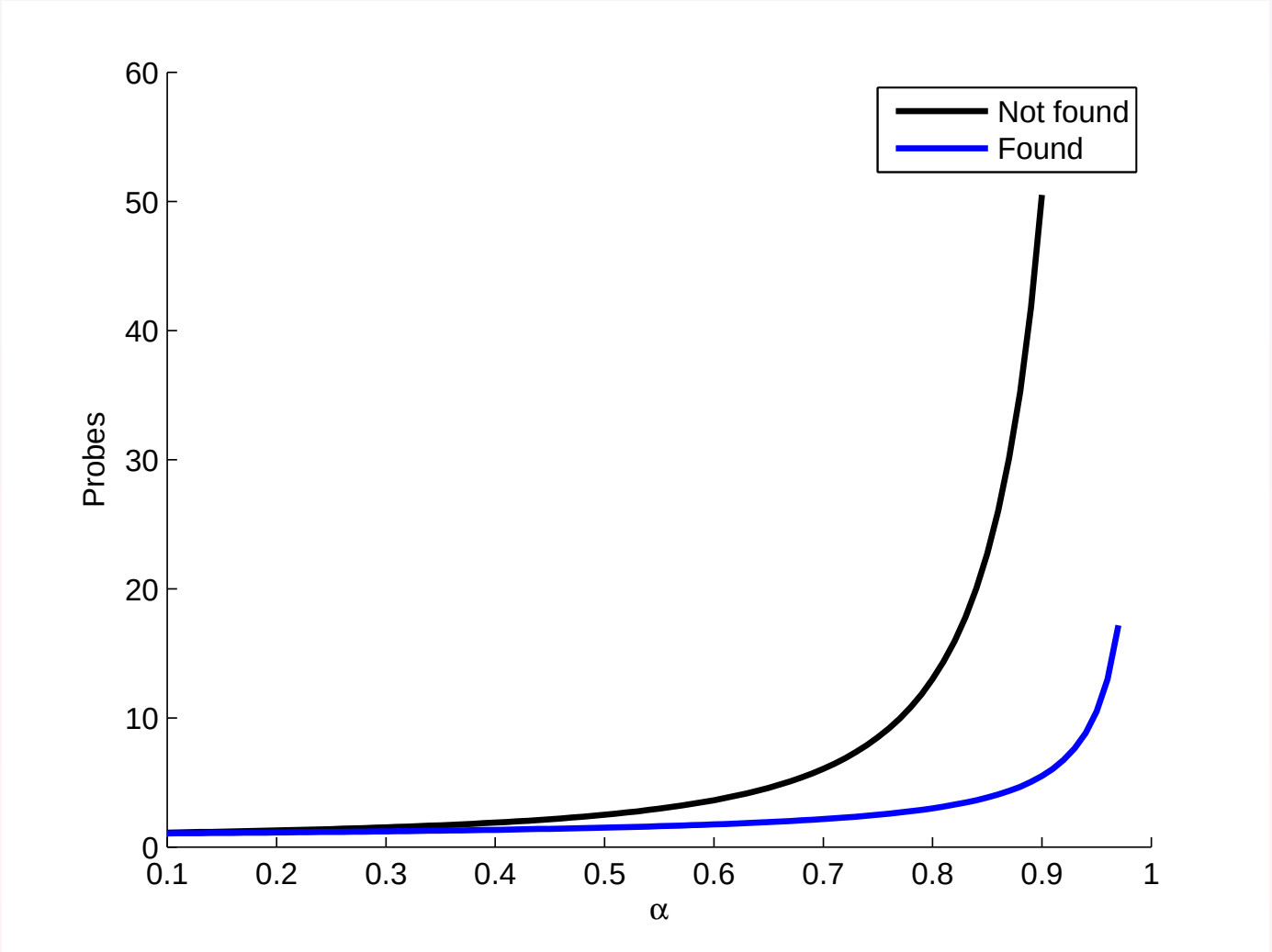
Prvek nalezen, očekávaný počet pokusů není horší než

$$n_{prob}(\alpha) = 0.5 \left(1 + \frac{1}{1-\alpha}\right).$$

Pro $\alpha \to 1$ se metoda stává neefektivní.
Neměla by být používána pro “téměř” zaplněnou tabulku.
Rozumné hodnoty zhruba $\alpha = 0.5$.

Poněkud problematické odstranění prvku, zůstane mezera, zaplnění příznakem -2.

47. Ilustrace výkonosti Linear Probingu



48. Inicializace pole klíčů a hodnot

Při inicializaci vytvořeno pole klíčů a hodnot.
Počet prvků nastaven jako M.
Všechny klíče inicializovány jako nepoužité na -1.

```
void init (int M)
{
    int [] K = new int [M];
    int [] V = new int [M];
    for (int i = 0; i < M; i++)
    {
       K[i] = -1;
    }
}
```

49. Přidání položky

```
void insert(int key, int val)
{
   int i = hash(key);

   //Projdi vsechny obsazene klice
   for (; K[i] != -1; i = (i+1) % M)
   {
      if (key == K[i])
         break;
   }

   //Uloz na prvni neobsazenou pozici
   V[i] = val;
   K[i] = key;
}
```

50. Vlastní hledání

Snadná implementace.
Vypočten hash $h(k)$ klíče k.
Od této pozice procházej pole klíčů, dokud není nalezen prvek se stejným klíčem nebo první neobsazený prvek.

```
 int search(int key)
{
   for (int i = hash(key); K[i] != -1; i = (i+1) % M)
   {
      //Klic se shoduje s hledanym klicem
      if (key == K[i])
        return V[i];
   }
   return -1;
}
```

51. Double Hashing

Odstraňuje nevýhody LP, nerovnoměrnou dobu hledání.
Díky clusteringu některé posuny malé, jiné příliš velké.
Hashovací funkce

$$h(k) = (h(k) + h_2(k))\,\%M,$$

eliminuje vznik clustrů.

DH provádí dvojí hashování:
1] První hashování $h(k)$ stejné jako LP.
2] Druhé hashování $h_2(k)$ spočítá přírůstek v adrese (místo sekvenčního procházení K).
Na rozdíl od LP není K sekvenčně procházen s krokem 1, ale s $h_2(k)$.
Takové prohledávání bude zřejmě rychlejší.
V porovnání s LP -30-50% testů.

Hashovací funkce h_2 má specifické vlastnosti.

52. Požadavky na h_2

Snadnost výpočtu.
Hashovací funkce $h_1(k) \in [0, M-1]$.

Hodnoty hashovací funkce $h_2(k) \in \mathbb{N}$, tj. $h_2(k) > 0$.
Nejlépe, pokud $h_2(k)$, M jsou prvočísla, nesmí být soudělná (co nej < pokusů, celá tabulka).

Nejčastější varianta hashovací funkce

$$h_2(k) \quad = \quad 1 + k\%c,$$

kde c prvočíslo, $c < M$, např. $c = 97$.
Alternativně

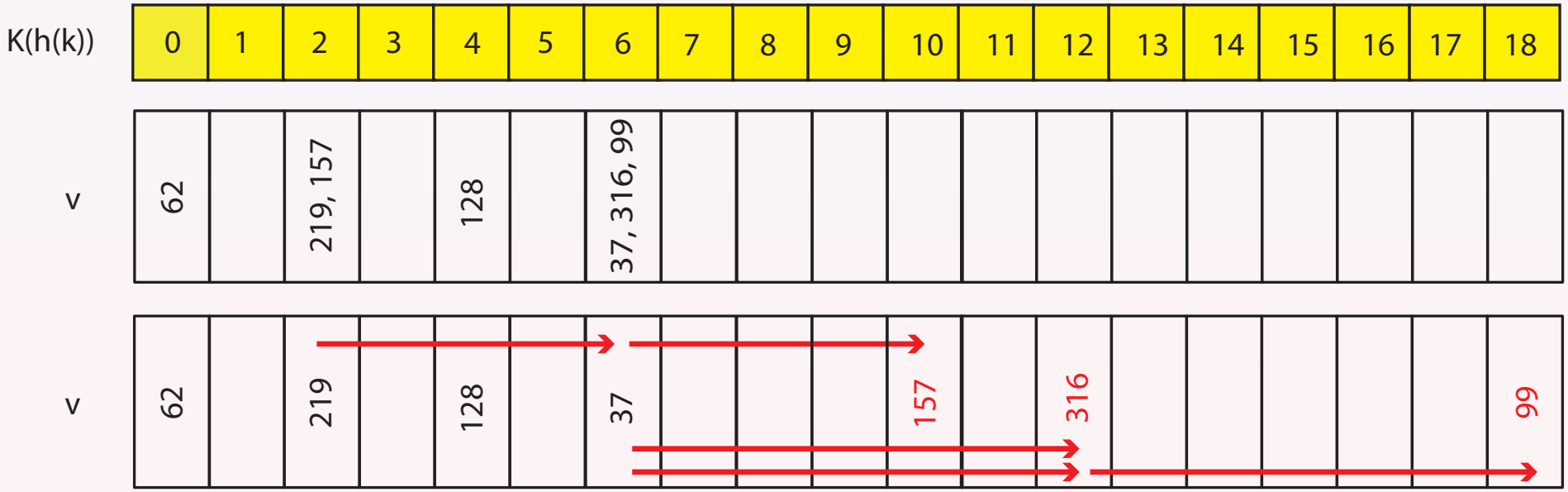
$$h_2(k) \quad = \quad q - k\%q,$$

kde q prvočíslo, $q < M$, např. $c = 7$.

53. Ilustrace Double Hashingu

Hashovací funkce:

$$h(k) = (h(k) + h_2(k))\,\%M, \quad h(k) = k\%31, \quad h_2(k) = 7 - k\%7.$$



$h_2(157) = 4,$
$h_2(316) = 6,$
$h_2(99) = 6.$

54. Výkonostní charakteristiky Double Hashingu

Prvek nenalezen, očekávaný počet pokusů není horší než

$$n_{prob}(\alpha) = \frac{1}{1-\alpha}.$$

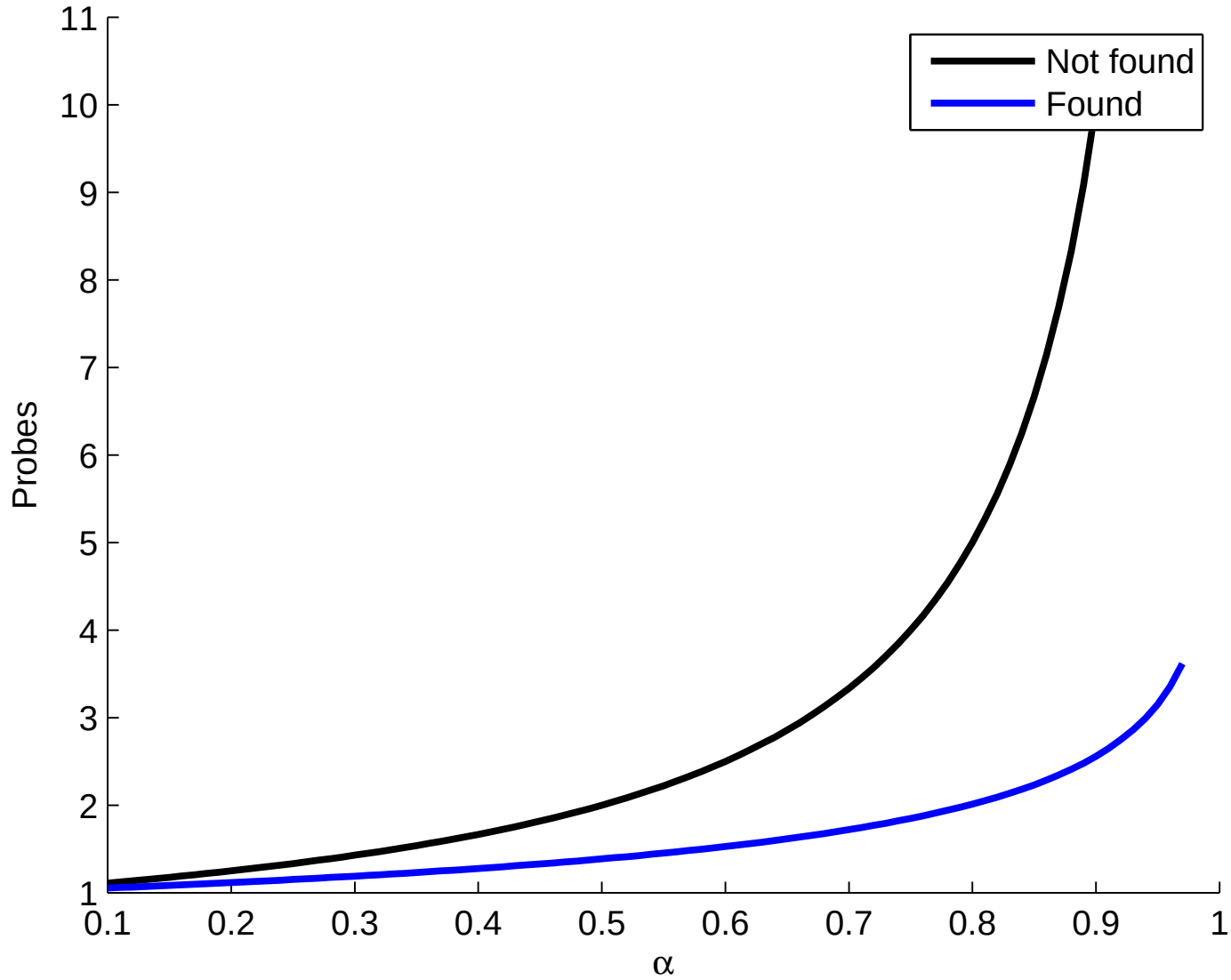
Prvek nalezen, očekávaný počet pokusů není horší než

$$n_{prob}(\alpha) = \frac{1}{\alpha} \ln \frac{1}{1-\alpha}.$$

Pro $\alpha \to 1$ se metoda stává méně efektivní.
Vliv zaplněnosti není tak významný jako u LP.
Neúspěšná hledání mají stále větší režii než úspěšná.
Lze použít menší tabulku.
Rozumné hodnoty zhruba $\alpha = 0.7$.

Po odstraněném prvku zůstane mezera, zaplnění příznakem.

55. Ilustrace výkonosti Double Hashingu



56. Přidání položky

```
void insert(int key, int val)
{
    int i = hash(key);
    int j = hash2(key);

    //Projdi vsechny obsazene klice
    for (; K[i] != -1; i = (i+k) % M)
    {
        if (key == K[i])
            break;
    }

    //Uloz na prvni neobsazenou pozici
    V[i] = val;
    K[i] = key;
}
```

57. Vlastní hledání

Vypočteny hashe $h(k)$, $h_2(k)$.
Od této pozice procházej pole klíčů, dokud není nalezen prvek se stejným klíčem nebo první neobsazený prvek.

```
 int search(int key)
{
   int i = hash(key);
   int k = hash2(key)
   for (; K[i] != -1; i = (i+k) % M)
   {
      //Klic se shoduje s hledanym klicem
      if (key == K[i])
        return V[i];
   }
   return -1;
}
```

58. Hashování vs. stromy

Binární stromy:

- Zaručená asymptotická složitost
- Menší závislost na datech.
- Podpora více operací (např. třídění).
- Výrazně složitější implementace.

Hashování:

- Ve většině případů výrazně rychlejší.
- Závislost na datech a velikosti tabulky.
- Závislost na hashovací funkci.
- Jednoduchá implementace.

59. Spatial Hashing

Často používáno v počítačové grafice (detekce kolizí).
V geoinformatice vhodné pro dělení prostoru na podmnožiny.
Prvky se stejným hashem leží uvnitř jednoho intervalu

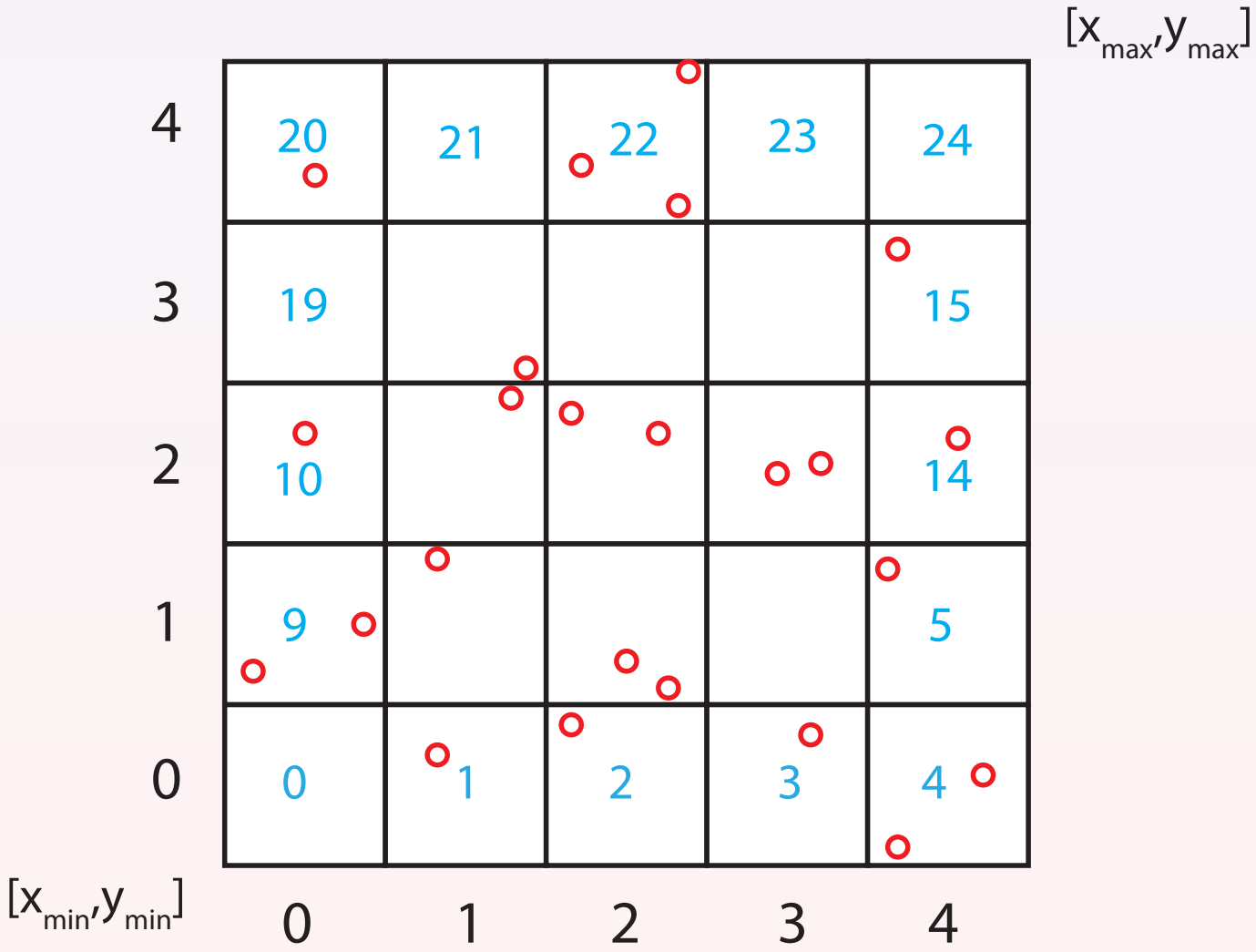
$$[x_{min}, x_{max}] \times [y_{min}, y_{max}] ,$$

tvořícího "dlaždici".

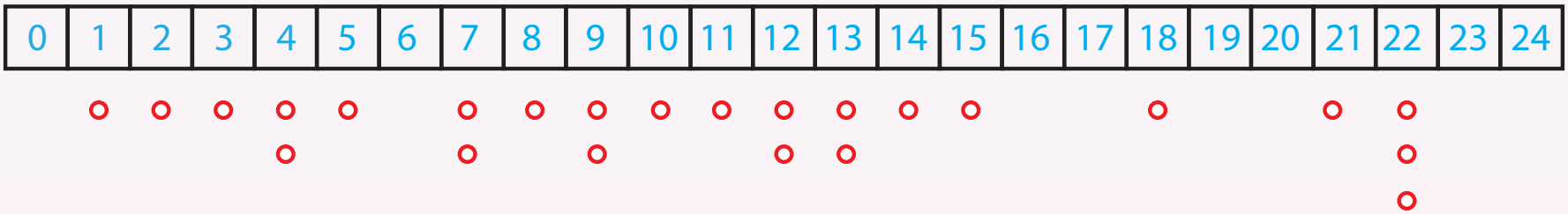
Není nutno pracovat celou množinu, ale pouze s její částí.
Prvky ležící uvnitř dlaždice jsou vzájemně blízké.
Pozor, jeden prvek může zabírat více dlaždic.

Klíčový význam pro úlohy z analytické geometrie: rychlé hledání průniků, průsečíků prvků.
Hashovací funkce nemusí splňovat některé požadavky, např. náhodnost.
Cílem není nalezení prvku samotného, ale dlaždice.

60. Ilustrace Spatial hashingu: ZIG-ZAG



61. Ilustrace Spatial hashingu

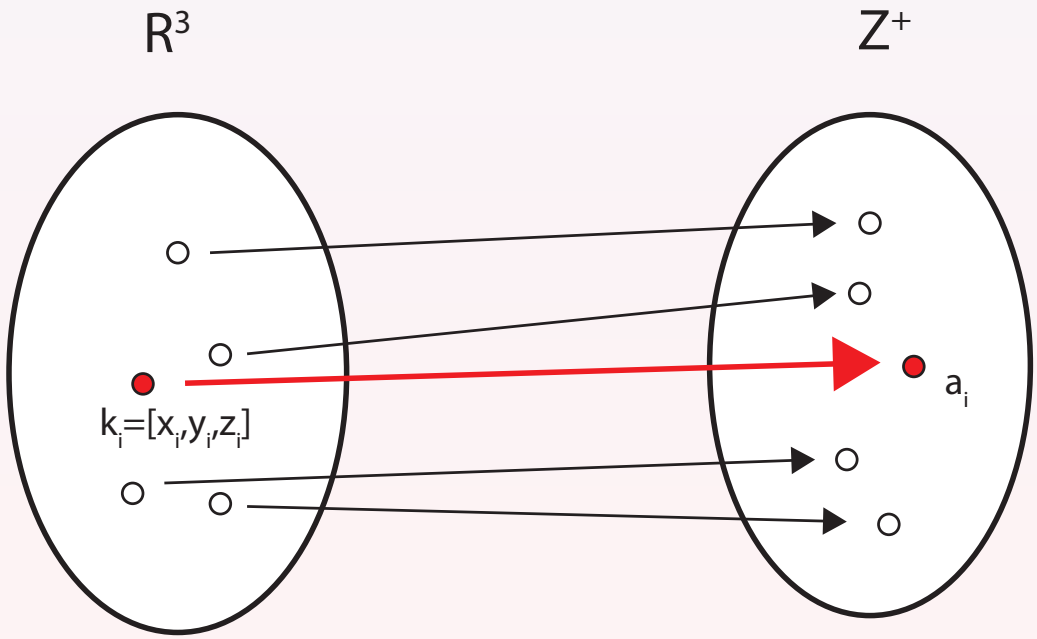


62. Prostorová hashovací funkce

Prostorová hashovací funkce

$$h : K \to A, \qquad h(k) = a, \qquad k \in K, a \in A, \qquad k \in \mathbb{R}^3, a \in \mathbb{Z}^+,$$

převádí klíč $k = [x, y, z]$ na adresu a.



Na rozdíl od běžného hashingu adresa nemusí mít pevnou délku.

63. Parametry dlaždice, ZIG-ZAG (1/2)

Celkem n obdélníkových dlaždic

$$n = \sqrt{N}.$$

Počet dlaždic n_y v řádku a n_x ve sloupci

$$n = n_x n_y, \qquad \frac{n_x}{x_{max} - x_{min}} = \frac{n_y}{y_{max} - y_{min}},$$

pak

$$n = \left(n_y \frac{x_{max} - x_{min}}{y_{max} - y_{min}} \right) n_y, \qquad n_y = \sqrt{n \frac{y_{max} - y_{min}}{x_{max} - x_{min}}}.$$

Velikost dlaždice

$$dx = \frac{x_{max} - x_{min}}{n_x} = \frac{y_{max} - y_{min}}{n_y} = dy.$$

Řádkový index dlaždice

$$i = (int) \frac{y - y_{min}}{dy}.$$

64. Parametry dlaždice, ZIG-ZAG (2/2)

Sloupcový index odlišný podle toho, zda je index i lichý

$$j = (int)\frac{x - x_{min}}{dx},$$

nebo sudý

$$j = n_x - (int)\frac{x - x_{min}}{dx}.$$

Hashovací funkce pro i liché

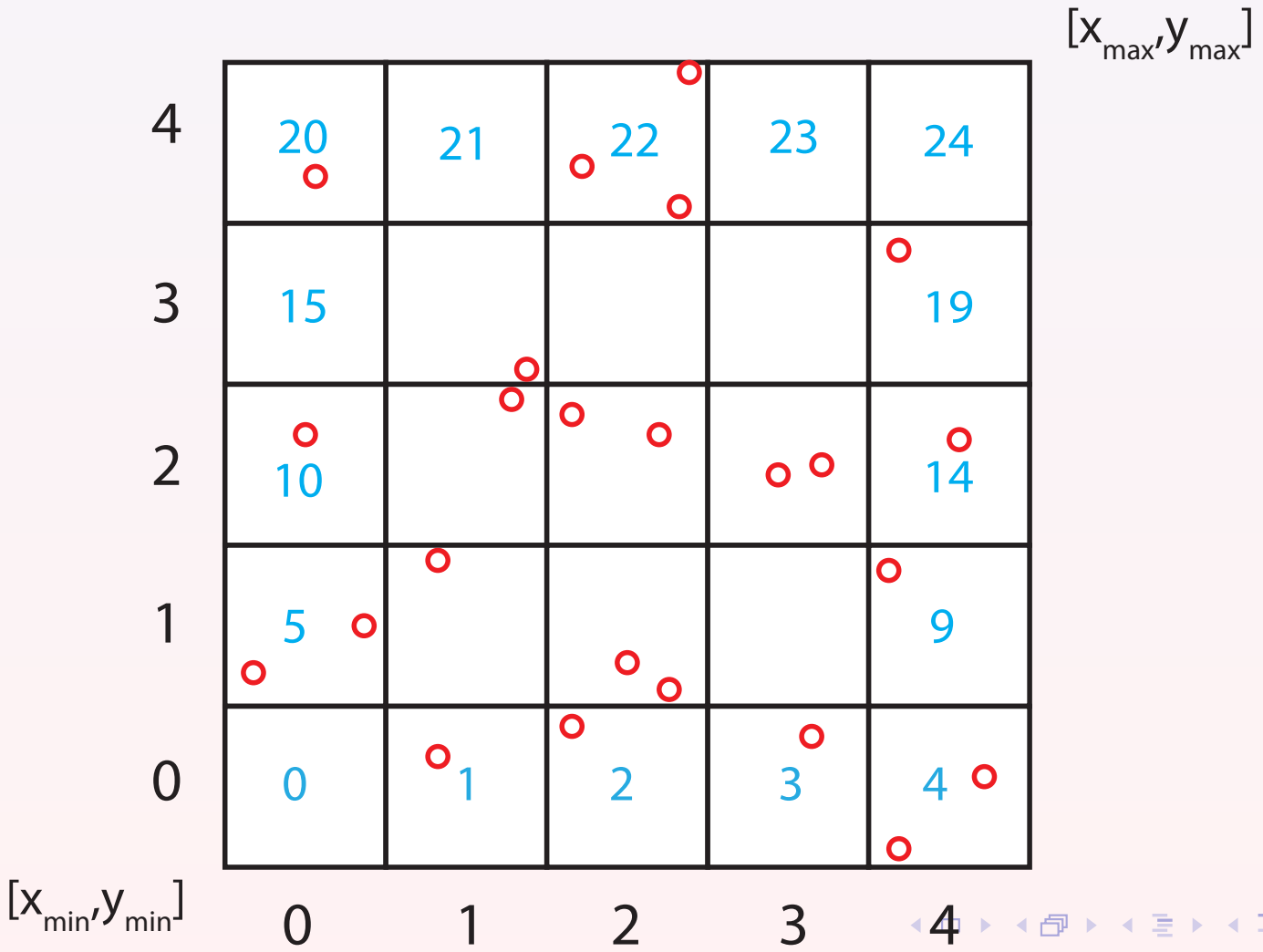
$$h(k) = i \cdot n_x + j,$$

pro i sudé

$$h(k) = i \cdot n_x + n_x - j - 1.$$

65. Spatial hashing: Ordered

Úprava indexace, procházení dlaždic v jednom směru $h(k)$.



66. Spatial hashing: Ordered

Řádkový index dlaždice

$$i = (int)\frac{y - y_{min}}{dy}.$$

Sloupcový index odlišný podle toho, zda je index i lichý

$$j = (int)\frac{x - x_{min}}{dx}.$$

Hashovací funkce

$$h(k) = i \cdot n + j.$$

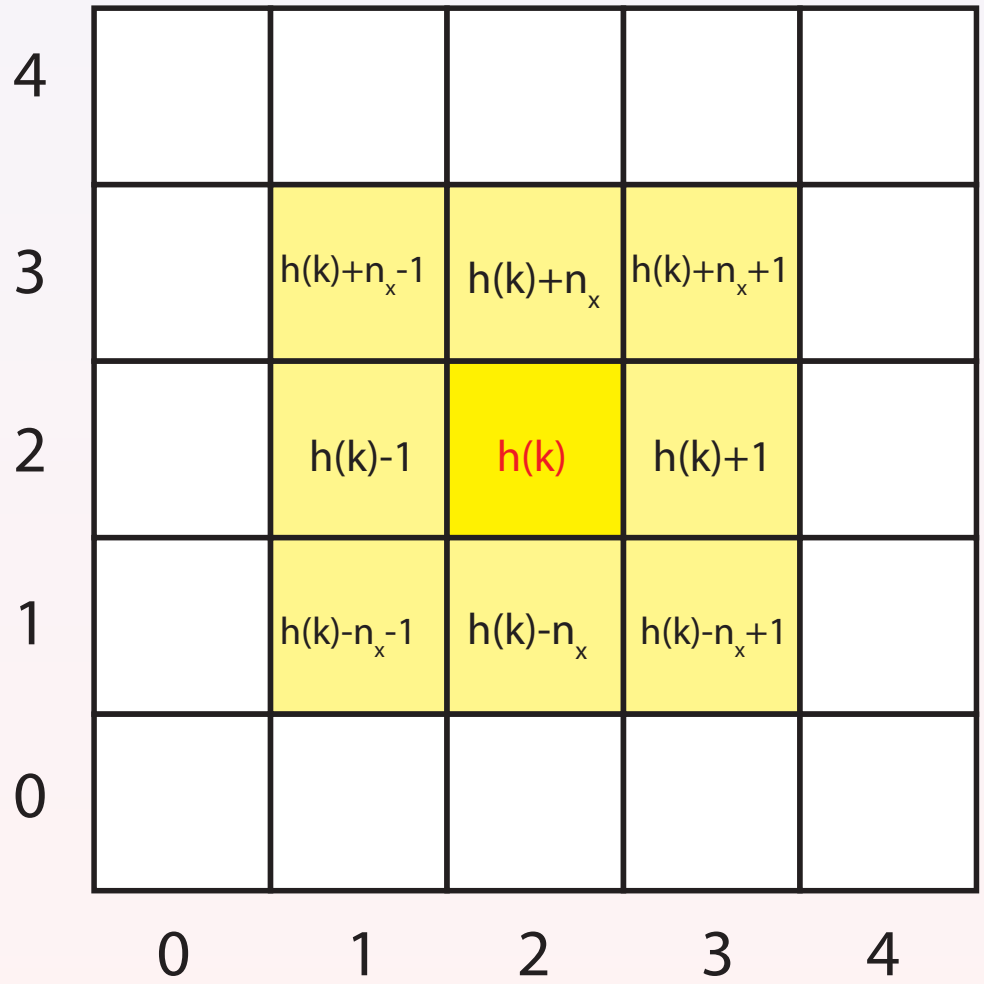
Snadnější určení hashe sousedních dlaždic (4-spojitost):

$$[h(k) + n_x, h(k) + 1, h(k) - n_x, h(k) - 1],$$

8-spojitost:

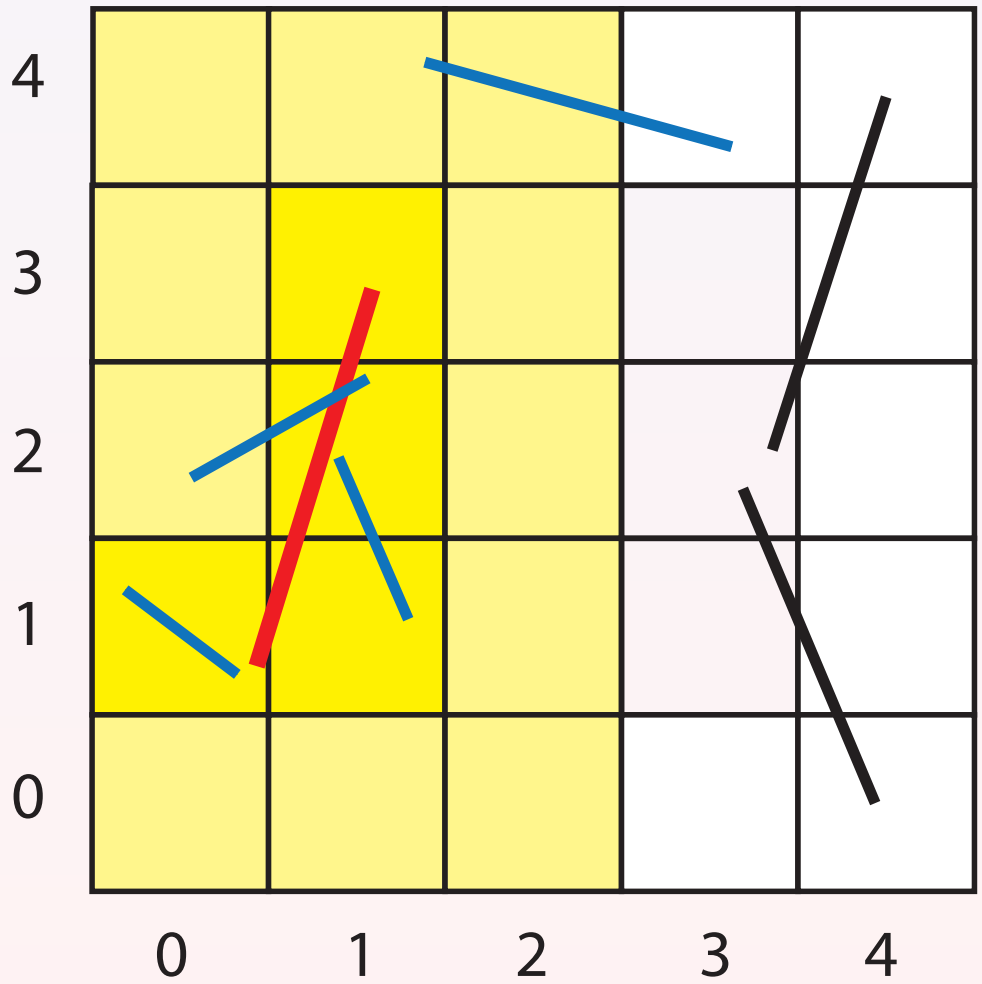
$$[h(k) + n_x - 1, h(k) + n_x, h(k) + n_x + 1, h(k) + 1],$$
$$[h(k) - n_x + 1, h(k) - n_x, h(k) - n_x - 1, h(k) - 1].$$

67. Ilustrace 8-spojitosti

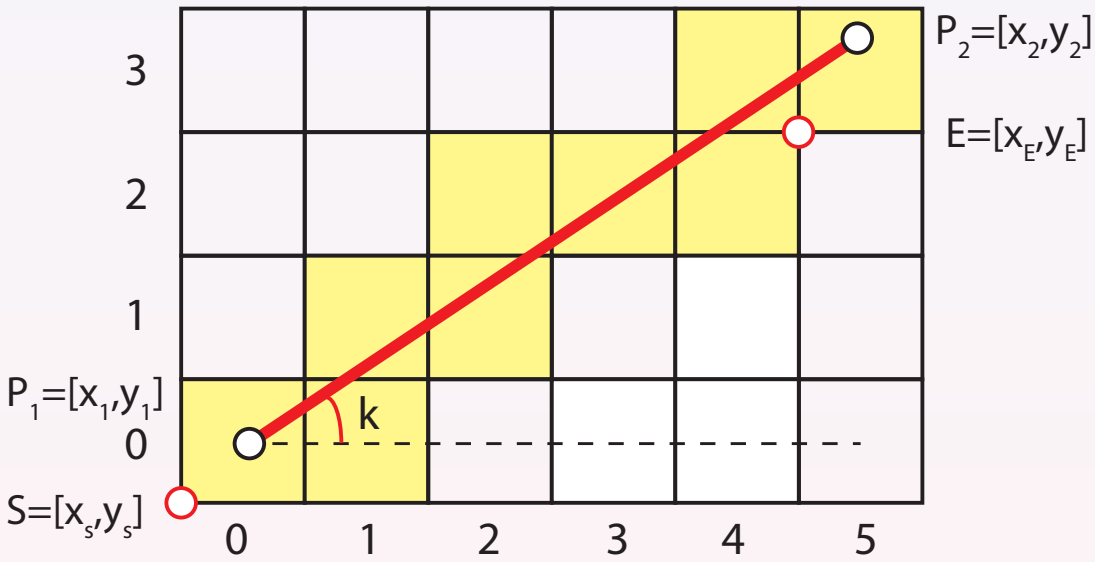


68. Použití 8-spojitosti

Při detekci průsečíků linie analyzovány pouze objekty v 8-spojitých dlaždicích.
Úspora, neprochází se celá scéna (zejména pro “krátké” prvky).



69. Nalezení dlaždic protnutých prvkem $P_1 = [x_1, y_1], P_2 = [x_2, y_2]$,



Počáteční dlaždice $S = [x_S, y_S]$, koncová dlaždice $E = [x_E, y_E]$,

$$x_S = x_1 - x_1 \% dx, \qquad y_S = y_1 - y_1 \% dy,$$
$$x_E = x_2 - x_2 \% dx, \qquad y_E = y_2 - y_2 \% dy,$$

Směrnice

$$k = \frac{y_2 - y_1}{x_2 - x_1}, \qquad x_2 - x_1 \neq 0.$$

70. Nalezení dlaždic protnutých prvkem

1 dlaždice v horizontálním směru $\Rightarrow$ k dlaždic ve vertikálním směru.
Přírůstky počtu dlaždic

$$\Delta n_x = k, \qquad \Delta n_y = 1.$$

Přírůstky x, y dle kvadrantu

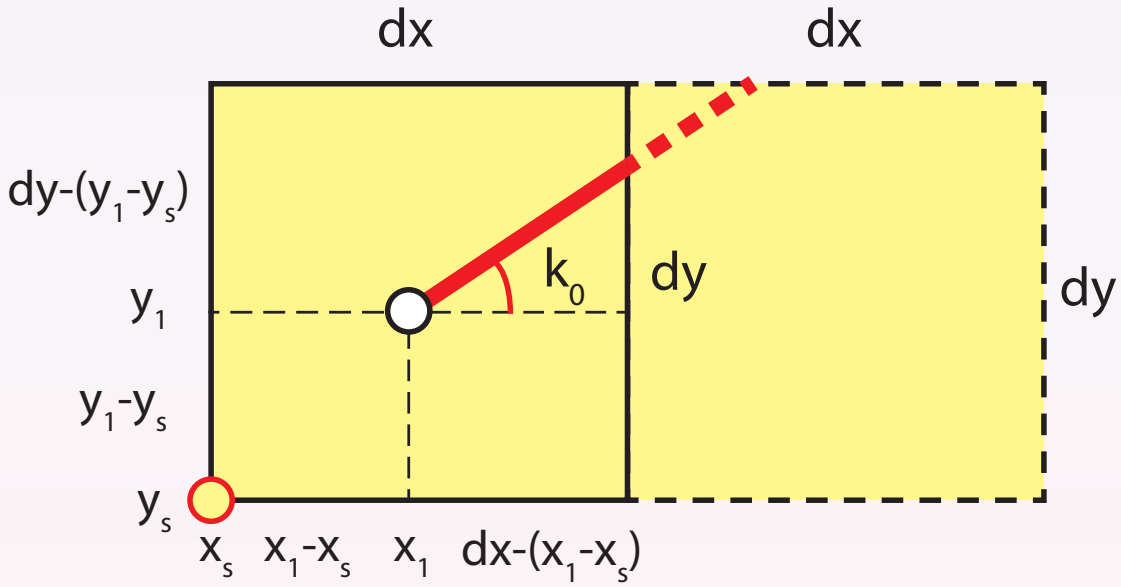
$$\Delta x = \begin{cases} dx, & x_2 > x_1, \\ -dx, & x_2 \leq x_1, \end{cases} \qquad \triangle y = \begin{cases} dy, & y_2 > y_1, \\ -dy, & y_2 \leq y_1, \end{cases}$$

Souřadnice levého dolního rohu dlaždice

$$[x_0, y_0] = \begin{cases} [x_0 + \Delta x, y_0], \Delta n_x = \Delta n_x + k, & \Delta n_x \leq \Delta n_y, \\ [x_0, y_0 + \Delta y], \Delta n_y = \Delta n_y + 1, & \Delta n_x > \Delta n_y, \end{cases}$$

kde $x_0 = x_S$, $y_0 = y_S$.

71. Problém: úvodní dlaždice protnuta částečně



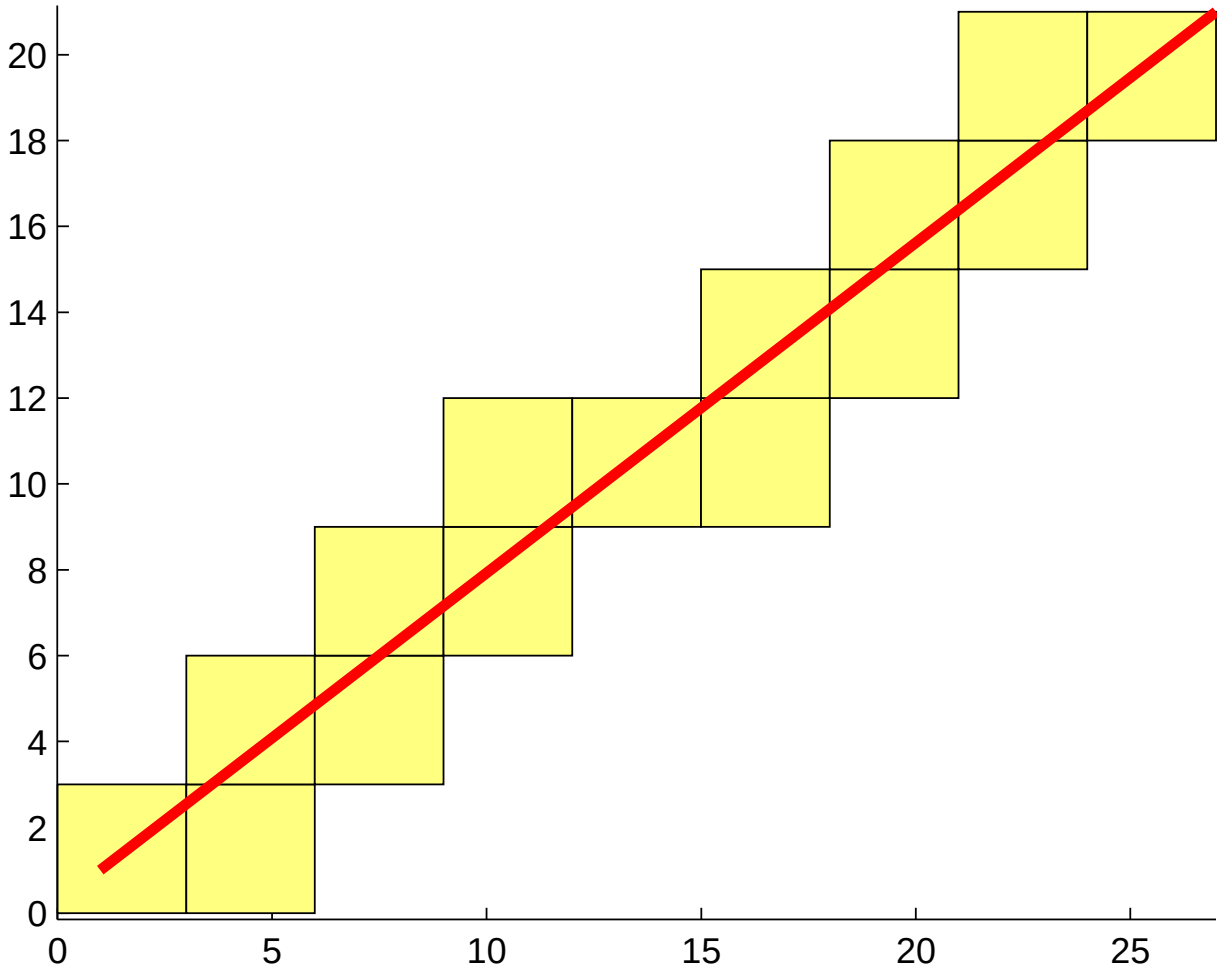
Průměty částí úvodní dlaždice protnuté úsečkou

$$dx - (x_1 - x_s), \qquad dy - (y_1 - y_s).$$

Fragmenty přírůstků

$$\Delta n_x^0 = \frac{dx - (x_1 - x_s)}{dx} k, \qquad \Delta n_y^0 = \frac{dy - (y_1 - y_s)}{dy}.$$

72. Ukázka činnosti rasterizačního algoritmu



73. Rasterizační algoritmus (1/2)

```
void rasterization(double x1, double y1, double x2, double y2,
                   double dx, double dy){
   double xs = x1-x1%dx;
   double ys = y1-y1%dy;
   double xe = x2-x1%dx;
   double ye = y2-y1%dy;

   //Smernice
   double k = (y2-y1)/(x2-x1);

   //Prirustky
   double deltax = (x2 > x1 ? dx: -dx);
   double deltay = (y2 > y1 ? dy: -dy);

   //Souradnice vychozi dlazdice
   double x0 = xs, y0 = ys;
```

74. Rasterizační algoritmus (2/2)

```
    //Vychozi prirustky pro 1. dlazdici
    double dnx0 = ((dx - abs(xs-x1))/dx)*k;
    double dny0 = ((dy - abs(ys-y1))/dy);
    double dnx = dnx0,  dny = dny0;


   //Vykresli vsechny dlazdice
    while (x0 < x1) && (y0 < y1) {
       //Nakresli obdelnik
       drawRectangle(x0, y0, dx, dy);

       //Posun dlazdici horizontalne
       if (dnx <= dny) {
           x0 += deltax;
           dnx += k;
      }

       //Posun dlazdici vertikalne
       else {
           y0 += deltay;
           dny += 1;
       }
}
```

75. Implementace Spatial Hashingu (1/5)

Třída uchovává 2 klíče, souřadnice x, y počátečního bodu grafického objektu.

```
class Uzel {

    double  k1, k2; //Dva klice - x,y
    GO hod;
    Uzel dalsi;

    public Uzel(double k1_, double k2_, GO hod_)
    {
        k1 = k1_; k2 = k2_;
        hod = hod_;
        dalsi = null;
    }
}
```

76. Implementace Spatial Hashingu (2/5)

Pro každý prvek uchováván seznam dlaždic, kterými prochází.

```
class GO {

    double x1, y1; //Pocatecni bod
    double x2, y2; //Konc. bod (napr. usecka)
    public int [] dlazdice;

    public GO(double x1_, double y1_, double x2_,
              double y2_) {
        x1 = x1_; y1 = y1_;
        x2 = x2_; y2 = y2_;
        dlazdice = null;
    }
}
```

Model lze rozšířit: každá dlaždice uchovává seznam prvků, které ji protínají.

77. Implementace Spatial Hashingu (3/5)

```
public void insert(double k1, double k2, GO hod) {
        int i = hash(k1, k2);

        if (K[i] == null)
             K[i] = new UzelSH(k1, k2, hod);
        else
        {
            //Test stejnych hodnot (nepovinny)
            Uzel u = K[i];
            while ((u.dalsi != null) && (hod != u.hod))  u = u.dalsi;

            //Stejna hodnota
            if (hod == u.hod)
                return;

            //Rozdilna hodnota, pridan jako dalsi prvek
            else
                u.dalsi = new Uzel(k1, k2, hod);
        }
    }
```

78. Implementace Spatial Hashingu (4/5)

```
public GO search(double k1, double k2, GO hod) {
        int i = hash(k1, k2);
        if (K[i] == null)
             return null;
        else
        {
            //Test stejnych hodnot (nepovinny)
            Uzel u = K[i];
            while ((u.dalsi != null) && (hod != u.hod))  u = u.dalsi;

            //Stejna hodnota
            if (hod == u.hod)
                return u.hod;

            //Rozdilna hodnota
            else
                return null;
        }
    }
```

```
public int hash(double x1, double y1) {

    int i = (int)((x1 - xmin)/dx);
    int j = (int)((y1 - ymin)/dy);

    return i * n_dlazdic + j;
}

public static void main(String[] args)
{
    SpatialHashing h = new SpatialHashing(10, 3, 3);

    GO hod1 = new GO(1,1,27, 21);
    hod1.tiles = h.rasterization(hod1);

    GO hod2 = new GO(1,1,27, 23);
    hod2.tiles = h.rasterization(hod2);

    h.insert(hod1.x1, hod1.y1, hod1);
    h.insert(hod2.x1, hod2.y1, hod2);
}
```