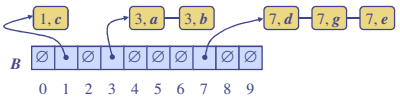
Tri par paquets



1

Définition:
Algorithme de Tri STABLE =
Un algorithme qui préserve l'ordre (avant le tri) des éléments ayant la même clef à la fin du tri

key object

Entrée : (1,x),(6,m),(4,k),(5,t),(2,s),(3,b),(1,a),(6,f),(6,b)
Exemple de résultat d'un tri stable
Sortie : (1,x),(1,a),(2,s) (3,b),(4,k),(5,t),(6,m),(6,f),(6,b)

Algorithmes de tri pas stables : tri par monceau, tri rapide

2

Question…

Q: Jusqu'à maintenant les meilleurs tris que nous avons vus sont exécutés en un temps O(n log n). Est-il possible de faire mieux?

R: Non, en général, la limite inférieure de problème est Ω(n log n)

…… mais dans certaines circonstances, O(n) est possible !

3

Tri par paquet

Imaginez trier 200 articles d'étudiants par ordre alphabétique selon la première lettre du nom de famille. Le tri par insertion (ou tri par sélection, ou tri par tri à bulle) essaierait de traiter le tas complet tout de suite. Le tri par fusion exigerait d'étendre tous les 200 papiers, de les comparer et de les rempiler dans l'ordre

Le tri par paquets place les 200 articles dans 26 paquets selon la première lettre du nom; Les paquets sont ensuite empilés dans l'ordre.

4

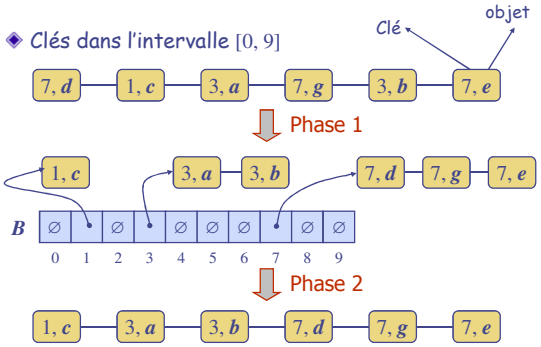
Tri par paquet (Bucket-Sort)

- Soit **S une séquence de n** entrées (clé, élément) avec les clés dans **[0, N - 1]**
- Le tri par paquets utilise les clés comme indices dans un tableau auxiliaire **B** de séquences (paquets)
- Phase 1: Vider la séquence **S en** déplaçant chaque entrée **(k, o)** dans son paquet **B[k]**
- Phase 2: Pour **i = 0, …, N - 1,** déplacer les entrées du paquet **B[i] à la fin de la séquence S**
- Analyse: Le tri par paquet prend un temps **O(n + N)**

Algorithm *bucketSort(S, N)*
  Input sequence S of (key, element) items with keys in the range $[0, N-1]$
  Output sequence S sorted by increasing keys
  $B \leftarrow$ array of N empty sequences
  while !*S.isEmpty*()
    $f \leftarrow$ *S.first*()
    $(k, o) \leftarrow$ *S.remove*(*f*)
    B[*k*].*insertLast*((*k*, *o*))
  for $i \leftarrow 0$ **to** $N-1$
    while !*B*[*i*].*isEmpty*()
      $f \leftarrow$ *B*[*i*].*first*()
      $(k, o) \leftarrow$ *B*[*i*].*remove*(*f*)
      S.insertLast((*k*, *o*))

Exemple

- Clés dans l'intervalle [0, 9]



Propriétés

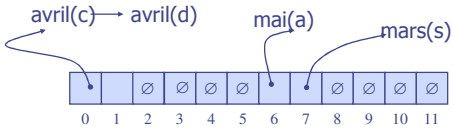
- Propriétés du type de clé
  - Les clés sont utilisées comme les indices dans un tableau et ne peuvent pas être des objets arbitraires
  - Aucun comparateur externe
- Propriété: tri stable
  - L'ordre relatif de n'importe quels deux entrées avec la même clé est conservé après l'exécution de l'algorithme
- Le tri par paquets est rapide surtout lorsque N <<<< n (plusieurs entrées avec des clés identiques); Quand N croit par rapport à n cet algorithme devient lent

Extensions

- Extensions
  - Clés entières entre **[a, b]**
    - Insérer une entrée **(k, o)** dans un paquet **B[k - a]**
- Clés en chaîne de caractères parmi **D** chaînes de caractères possibles où **D a** une taille constante (par ex., les noms des 50 états américains)
  - Trier D et calculer le rang r(k) de chaque chaîne de caractères k de D dans la séquence triée
  - Insérer (k, o) dans un paquet B[r(k)]

Ex: D ={janvier, février, mars,..... décembre}
Tri de D:
0.Apr
1.Aug
2.Feb
3.Dec
4.Jan
5.June
6.July
7.March
8.May
9.Nov
10.Oct
11.Sep

Séquence:
{mai(a), avril(c), avril(d), mars(s)}



9

Ordre lexicographique

- Un d-uplet est une séquence de d clés (k1, k2, …, kd), où la clé ki est la ième dimension du d-uplet
- L'ordre lexicographique de deux d-uplets est récursivement défini comme suit

$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$
$\Leftrightarrow$
$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$

- c.-à-d., les éléments des d-uplets sont comparés par la première dimension, ensuite par la deuxième dimension, etc

1,2,3 < 2,5,3
3,2,1 > 3,1,5

10

Tri Lexicographic

- Soit Ci le comparateur qui compare deux d-uplets par leur ième dimension c.-à-d. par ex.
  Pour C_3
  (x1,x2,x3)≤(y1,y2,y3) if x3≤y3.
- Soit stableSort(S, C) n'importe quel algorithme de tri stable qui utilise un comparateur C. Par ex., stableSort pourrait appliquer un tri par paquets en utilisant la ième dimension comme clé
- Le tri-lexicographique trie une séquence de **d -uplets** dans l'ordre lexicographique en exécutant **d fois** l'algorithme **stableSort = une fois par dimension**

11

Exemple:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

Trier les dimensions de droite à gauche

(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

12

Lexicographic-Sort

- Le tri procède de la dernière dimension vers la première, pourquoi?

Algorithme *lexicographicSort(S)*
Entrée: séquence *S de d-uplets*
Sortie: séquence *S triée dans l'ordre* lexicographique

for $i \leftarrow d$ *to 1*
  stableSort(S, Ci)

- Le tri Lexicographique s'exévute en temps $O(dT(n))$ où $T(n)$ est le temps d'exécution du *stableSort*

13

Tri par base (Radix-Sort) (1)

- Le tri par base est un cas particulier du Tri lexicographique qui utilise le tri par paquets comme algorithme de tri stable pour chaque dimension.
- appliquable lorsque les clés de chaque dimension sont des entiers $[0, N-1]$
- $O(d(n+N))$

Algorithme *radixSort(S, N)*
Entrée : séquence *S de d-uplets tels* que *(0, …, 0) ≤ (x1, …, xd)*
  et (x1, …, xd) ≤ (N - 1, …, N - 1) pour chaque d-uplet *(x1, …, xd)* dans *S*
Sortie : séquence *S triée dans l'ordre* lexicographique
for $i \leftarrow d$ *to 1*
  bucketSort(S, N)

14

Tri par base (Radix-Sort) (2)

- Considérer une séquence de **n** entiers avec une représentation binaire en **b bits**
- $x = x_{b-1} \ldots x_1 x_0$
- Nous représentons chaque élément comme un **b-uplet** d'entiers dans la gamme [0, 1] et appliquons le tri par base avec **N = 2**
- Exécuté dans un temps **O(bn);**
  - ex. trier les entiers Java (32bits) en O(n)

Algorithme *binaryradixSort(S)*
Entrée séquence *S d'entiers* de *b-bits*
Sortie séquence *S triée*

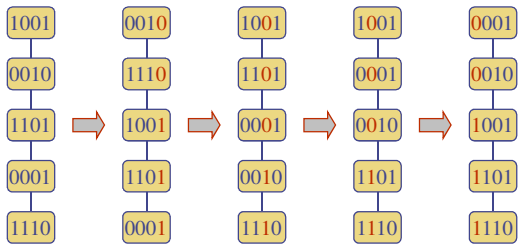
remplacer tout élément *x* de *S avec l'entrée (0, x)*

for $i \leftarrow 0$ *to b - 1*
  remplacer la clé *k de* chaque entrée *(k, x) de S* avec le bit x_i *de x*
  bucketSort(S, 2)

15

Exemple

- Tri d'une séquence d'entiers de 4-bits



16

Extensions

- Tri par base - variation 3
  - Les clés sont des entiers dans [0, N^2 - 1]
  - Nous représentons une clé comme un 2-uplet de chiffres dans [0, N - 1] et appliquons le tri par base, c.-à-d. nous l'écrivons en notation de base ***N:***
    - Exemple (***N = 10):***
      - 75 → (7, 5)
    - Exemple (***N = 8):***
      - 35 → (4, 3)
- Le temps d'exécution du tri par base est ***O(n + N)***
- Peut être étendu aux clés d'entier dans la gamme [0, N^d- 1]

17

Tri par base - Exemple

(N = 10) Les clés sont dans l'intervalle [0, 99]

75,12,54,33,14,87,45,17

75 = (7, 5) , 12 = (1,2), 54 = (5,4), 33 =(3,3), …

(7, 5) (1,2) (5,4) (3,3) (1,4) (8,7) (4,5) (1,7)

(1,2) (3,3) (5,4) (1,4) (7, 5) (4,5) (8,7) (1,7)

(1,2) (1,7) (1,4) (3,3) (4,5) (5,4) (7, 5) (8,7)

18

Tri par base - Extension

- Tri par base: variation avec une chaîne de caractères
  - Les clés sont des chaînes de caractères de d caractères chacune
  - Nous représentons chaque clé par un d-uplet d'entiers où la dimension i est la représentation ASCII (entier de 8-bits) ou Unicode (entier de 16-bits) du ième caractère et appliquons le tri par base variation 1.

19

Tri par base - Exemple

Utiliser la dernière variation pour trier les chaînes de caractères suivantes: "cart ", "core ", "fore", "cans", "cars", "bans"

Étape 1: convertir en 4-uplet

cart:	(3,1,18,20)	cans:	(3,1,14,19)
core:	(3,15,18,5)	cars:	(3,1,18,19)
fore:	(6,15,18,5)	bans:	(2,1,14,19)

Étape 2: Trier chaque dimension du d-uplet avec un tri par paquets en commençant par la dernière dimension

20