CSCI 2100 Tutorial 6

WU Hao

Outline

- Counting sort again – a linked list version

- Dynamic array vs linked list

- Dynamic array: space and update tradeoff

Multi-set Sorting Problem (Review)

- Problem input:
  - An array containing n key-value pairs, where each key is an integer from [1, U].

    E.g.: (93, 1155123456)

- Goal:
  - An array storing all pairs in nondescending order of key.

Multi-set Sorting Problem

- Input: $\{\{9, v1\}, \{7, v2\}, \{2, v_3\}, \{6, v_4\}, \{2, v_5\}, \{7, v_6\}, \{1, v_7\}, \{2, v_8\}\}$
- Initially we will have the following array

Input Array

k_1	v_1	k_2	v_2	k_3	v_3	k_4	v_4	k_5	v_5	k_6	v_6	k_7	v_7	k_8	v_8
9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

- Rearrange the elements so that their keys are sorted:

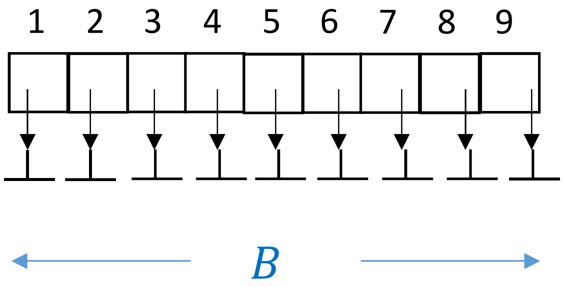
Sorted Array

1	v_7	2	v_3	2	v_5	2	v_8	6	v_4	7	v_2	7	v_6	9	v_1

Multi-set Sorting Problem

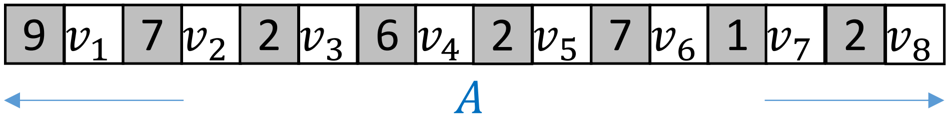
Today we will learn a simple variant of counting sort based on linked lists. The new algorithm also achieves the time complexity $O(n + U)$.

Counting Sort (Linked List Ver.)



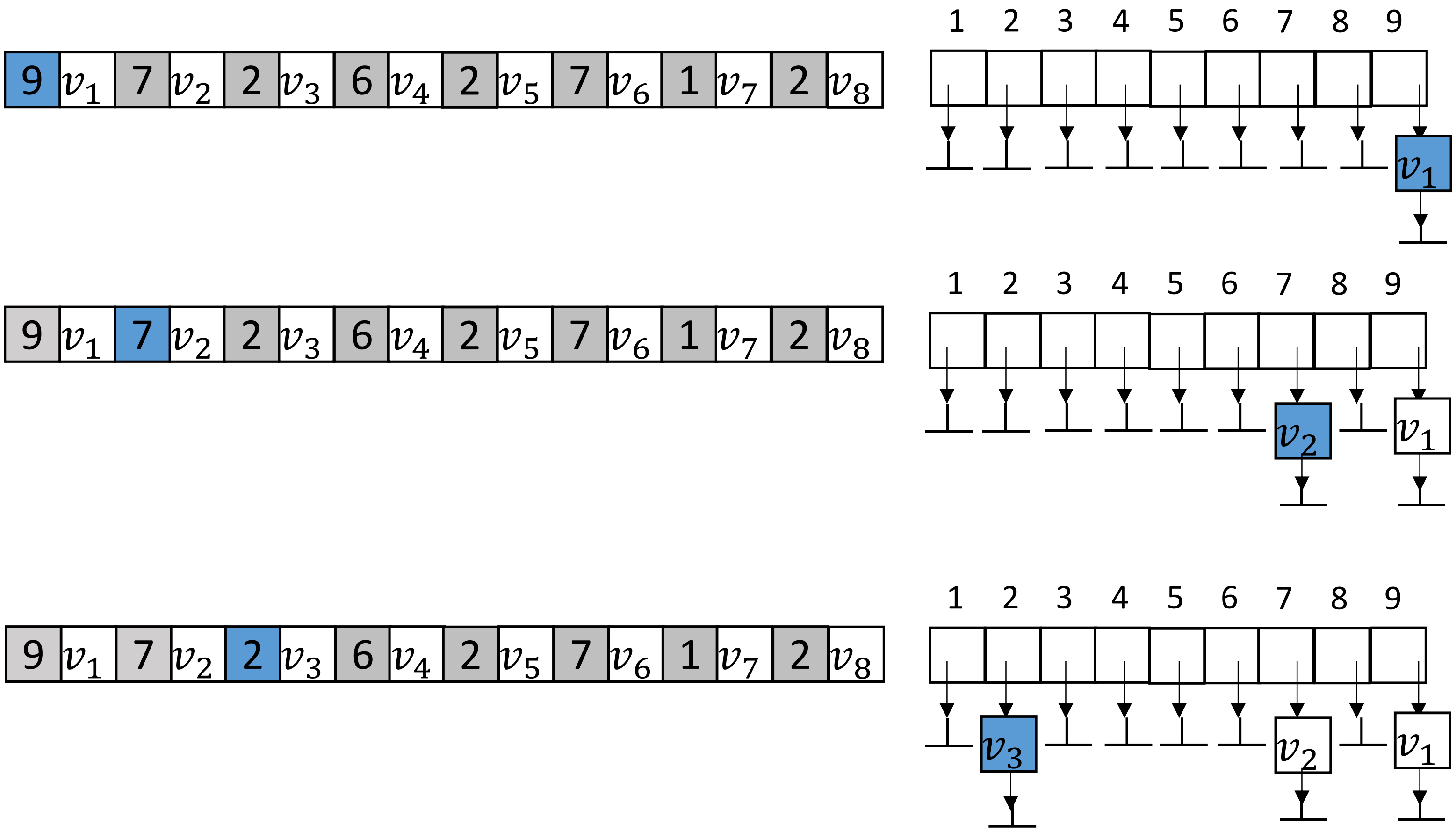
⊥: This means a null pointer

Compute B

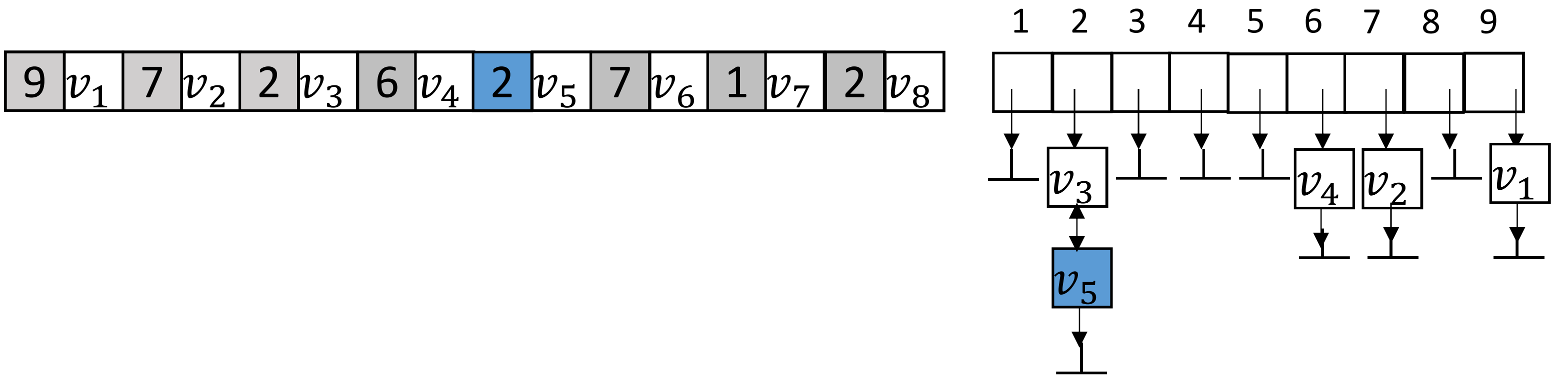


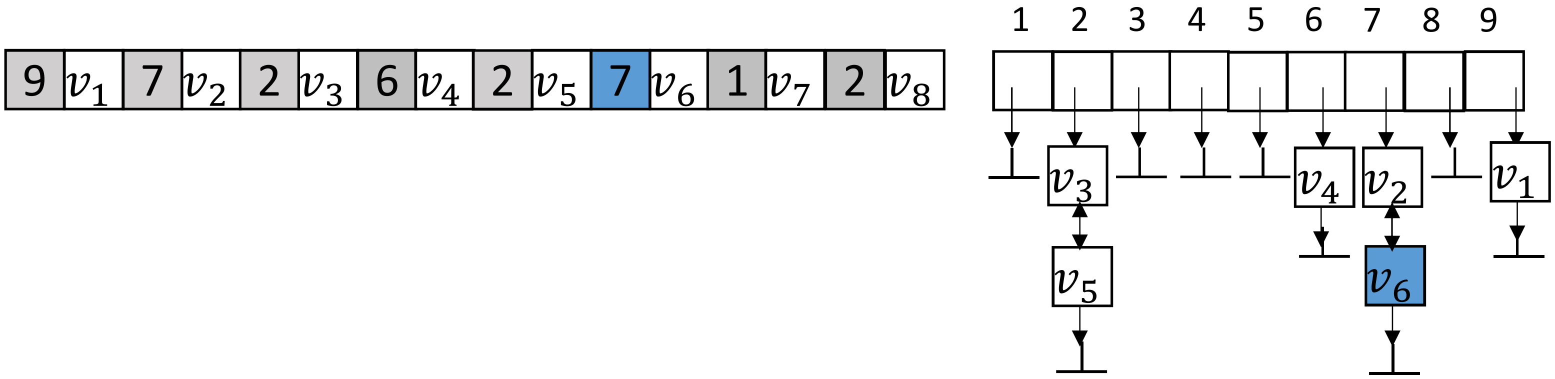
9	v_1	7	v_2	2	v_3	6	v_4	2	v_5	7	v_6	1	v_7	2	v_8

Counting Sort (Linked List Ver.)

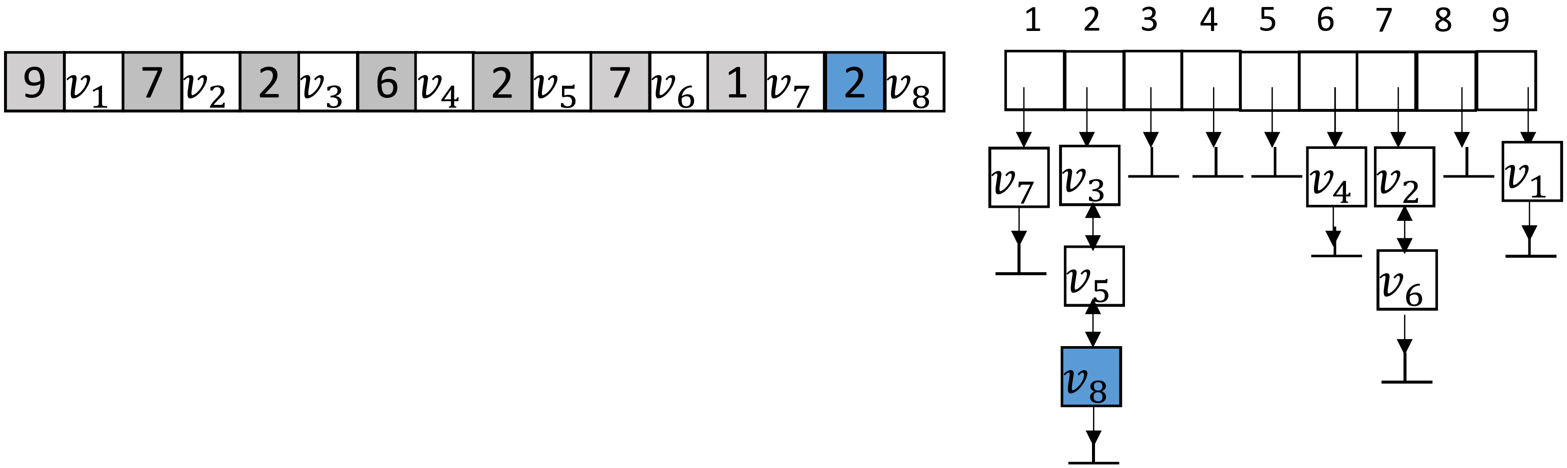


Counting Sort (Linked List Ver.)

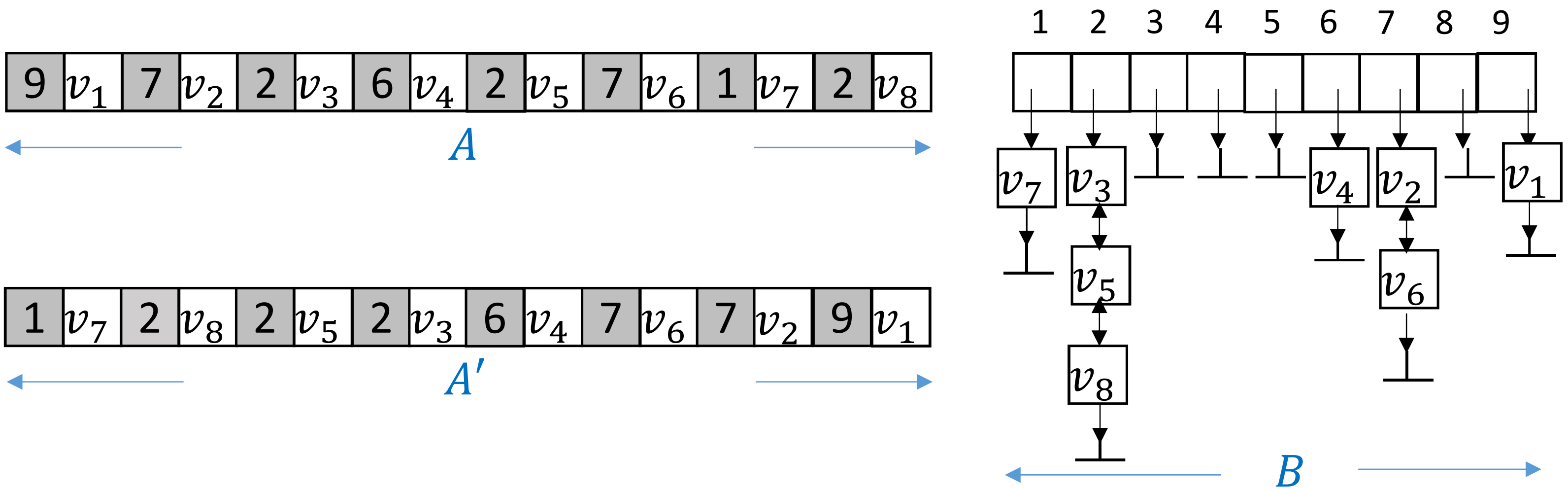




Counting Sort (Linked List Ver.)



Counting Sort (Linked List Ver.)



How do we produce the sorted array A'?

Scan array B. For each cell pointing to a non-empty linked list, enumerate all the pairs therein.

Overall time complexity: $O(n + U)$

Dynamic Array vs Linked List

A linked list ensures O(1) insertion cost. A dynamic array guarantees O(1) insertion cost only after amortization.

However, a dynamic array provides constant-time access to any element, which a linked list cannot achieve.

Dynamic Array vs Linked List

Question:

Design a data structure of $O(n)$ space to store a set *S* of *n* integers to satisfy the following requirements:

- An integer can be inserted in $O(1)$ time.
- We can enumerate all integers in $O(n)$ time.

Answer: Linked list.

Dynamic Array vs Linked List

Question:

Design a data structure of $O(n)$ space to store a set *S* of *n* integers to satisfy the following requirements:

- An integer can be inserted in $O(1)$ amortized time.
- We can enumerate all integers in $O(n)$ time.
- For each *i* ∈ $[1, n]$, access *i*-th inserted integer in $O(1)$ time.

Answer: Dynamic array

Space-Update Tradeoff of the Dynamic Array

In the lecture, we expand the array from size n to $2n$ when it is full.

What if we expand the array size to $\lceil 1.5n \rceil$?

Space-Update Tradeoff of the Dynamic Array

- Initially, size 2 (define $s_1 = 2$)
- 1st expansion: size from s_1 to $s_2 = \lceil 1.5 s_1 \rceil = 3$.
- 2nd expansion: from s_2 to $s_3 = \lceil 1.5 s_2 \rceil = 5$.

…

- *i*-th expansion: from s_i to $s_{i+1} = \lceil 1.5 s_i \rceil$.

We can prove: $s_i \le \left(\frac{8}{3}\right) 1.5^i - 2 = O\left(1.5^i\right)$ and $s_i \ge 1.5^i$.

Space-Update Tradeoff of the Dynamic Array

- The total cost of *n* insertions is bounded by:

$$\left(\sum_{i=1}^{n} O(1)\right) + \sum_{i=1}^{h} O(1.5^{i+1}) = O\left(n + 1.5^{h+1}\right)$$

where *h* is the number of expansions.

It must hold that $n > s_h \geq 1.5^h$ (the *h*-th expansion happened because the array of size s_h was full).

Hence, the total cost is *O*(*n*).

Space-Update Tradeoff of the Dynamic Array

- Consider what happens in general. When the array is full, expand its size from n to αn, for some constant $1 < \alpha \leq 2$.

Space-Update Tradeoff of the Dynamic Array

- Initially, size 2 (define $s_1 = 2$)
- 1st expansion: size from s_1 to $s_2 = \lceil \alpha s_1 \rceil$.
- 2nd expansion: from s_2 to $s_3 = \lceil \alpha s_2 \rceil$.

…

- *i*-th expansion: from s_i to $s_{i+1} = \lceil \alpha s_i \rceil$..

We can prove: $s_i = O(\frac{\alpha^i}{\alpha - 1})$ and $s_i \geq \alpha^i$.

Space-Update Tradeoff of the Dynamic Array

The total cost of *n* insertions is bounded by:

$$\left(\sum_{i=1}^{n} O(1)\right) + \sum_{i=1}^{h} O\left(\frac{\alpha^{i+1}}{\alpha - 1}\right) = O\left(n + \frac{\alpha^{h+2}}{(\alpha - 1)^2}\right)$$

where *h* is the number of expansions.

It must hold that $n > s_h \geq \alpha^h$ (the *h*-th expansion happened because the array of size s_h was full).

Hence, the total cost is $O\left(n + \frac{\alpha^2}{(\alpha-1)^2} n\right)$, namely, amortized cost $= O\left(1 + \frac{\alpha^2}{(\alpha-1)^2}\right)$.

Space-Update Tradeoff of the Dynamic Array

Amortized cost $= O\left(1+\frac{\alpha^2}{(\alpha-1)^2}\right)$.

When α decreases, the space consumption goes down, but the insertion cost goes up.