Time and Space Tradeoffs

Learning objectives

1. Understand space-time tradeoffs in algorithm design.
2. Two of the paradigms:
   - Input enhancement (sort by counting)
   - Pre-structuring (hashing)

Agenda

1. Overview
2. Sort by Counting
3. Hash Tables
   - Separate Chaining Hashing
   - Open Address Hashing
4. Summary

1. Overview

Time & Space Trade offs

We can gain time by using more space – whole idea behind this lecture.

In this lecture, we discuss two varieties of Time & Space tradeoffs:

1. Input Enhancement
2. Pre-structuring

Time & Space Trade-offs

1. Input Enhancement – pre-process the input to store extra information that will accelerate the solving of the problem.
   - counting sorts
   - prefix sum: calculate sum in ranges
2. Pre-structuring – use extra space to make accessing its elements easier or faster.
   - hashing

2a. Sort by Counting

RMIT UNIVERSITY

Count-based Sorting

When sorting a list with many repeated values, can we do better than the sorts we have seen?

- **Rough idea:** Imagine we have array with three values 1, 2, 3, for example [2,1,2,3,1]
- If we arrange the array with the 1s, then the 2s, then the 3s, then we have sorted the array (1,1,2,2,3).
- Distribution sort does exactly this, in a smart way.

Distribution Sorting

1. Use an auxiliary table to store the frequency of each possible element value (indexed by distinct elements).
2. Compute the cumulative frequency (how many elements in an array have a value <= a particular value).
3. Use cumulative frequency count to copy elements, in sorted order, to a new array. The cumulative counts indicate which position to copy the elements to.

Distribution Sorting

4. We copy the elements in original array going from right to left (in order to have stable sorting).

Consider the array A = {13, 11, 12, 13, 12, 12}

Array Values	11	12	13
Frequencies	1	3	2
Cumulative Frequencies	1	4	6

A = {11, 12, 12, 12, 13, 13}

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
0	0	0	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
0	0	1	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	0	1	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	0	2	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	1	2	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	1	3	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	1	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	2	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
2	2	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
2	3	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
2	3	3	1

Cumulative Freq

4	9	12	15
2	5	8	9

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
2	3	3	1

Cumulative Freq

4	9	12	15
2	5	8	9

1	2	3	4	5	6	7	8	9
4	4	9	9	9	12	12	12	15

Distribution Sorting Analysis

The worst-case analysis for distribution sorting is:

$$
\begin{aligned}
C(n) &= \underbrace{\sum_{j=0}^{n_{\max}} 1}_{\text{initialise freqs}} + \underbrace{\sum_{i=0}^{n-1} 1}_{\text{compute freqs}} + \underbrace{\sum_{j=0}^{n_{\max}} 1}_{\text{compute cumulative freq}} + \underbrace{\sum_{i=0}^{n-1} 1}_{\text{copy values}} \\
&= 2\sum_{i=0}^{n-1} 1 + 2\sum_{j=0}^{n_{\max}} 1 \\
&= 2\mathcal{O}(n) + 2\mathcal{O}(n_{\max}) \\
&\in \mathcal{O}(n), \ \text{if } n > n_{\max}
\end{aligned}
$$

The algorithm also uses an additional $O(n) + O(n_{max})$ space

2b. Radix Sort

Distribution Sorting Issue

The worst-case analysis for distribution sorting is:

$$
\begin{aligned}
C(n) &= \underbrace{\sum_{j=0}^{n_{\max}} 1}_{\text{initialise freqs}} + \underbrace{\sum_{i=0}^{n-1} 1}_{\text{compute freqs}} + \underbrace{\sum_{j=0}^{n_{\max}} 1}_{\text{compute cumulative freq}} + \underbrace{\sum_{i=0}^{n-1} 1}_{\text{copy values}} \\
&= 2\sum_{i=0}^{n-1} 1 + 2\sum_{j=0}^{n_{\max}} 1 \\
&= 2\mathcal{O}(n) + 2\mathcal{O}(n_{\max}) \\
&\in \mathcal{O}(n), \text{ if } n > n_{\max}
\end{aligned}
$$

What if $n_{max} > n$, for example $n_{max} = n^2$? Now, the complexity is $O(n^2)$, which is slow.

Example & Analysis

- The array to be sorted has 100,000 elements, and the value ranges from 0 to 1,000,000,000.
- Even though the maximum value is large, the maximum number of digits to store it is small (10 digits).
- If binary system is used, the maximum number of digits is still small (30 digits).
- Can we use counting sort digit-by-digit?

Radix Sort

- Radix sort can run left to right (most significant digit to least significant digit – MSD radix sort) or right to left (LSD radix sort).
- LSD radix sort is stable.
- Algorithm:
  input: array A[N].
  for i = right_most -> left_most position
      counting_sort(A) based on position i

Radix Sort Complexity

- Counting sort (A) based on a location i-th
  - $O(N + N_{max}) = O(N)$ // assume $N_{max} < N$
  - $N_{max} = 2$ for binary numbers
  - $N_{max} = 10$ for decimal numbers
- Outer loop complexity = O(W), W is the width of the largest number in A (i.e., the number of digits).
- Radix sort complexity: O(W*N)

3. Hash Tables

Set ADT

- Recall the definition of a set, where all the keys are unique. (Note this applies to maps also, where we have key->value pairs).
- What data structures can we use to implement a Set ADT? For example:
  - Linked List
  - Tree (balanced)
  - Array

Set ADT

- What are the worst case complexities of INSERT, DELETE and SEARCH?

	List	Balanced Tree	Array
INSERT	O(N)	O(lgN)	O(N)
DELETE	O(N)	O(lgN)	O(N)
SEARCH	O(N)	O(lgN)	O(lgN)

- Is it possible to achieve better efficiency?

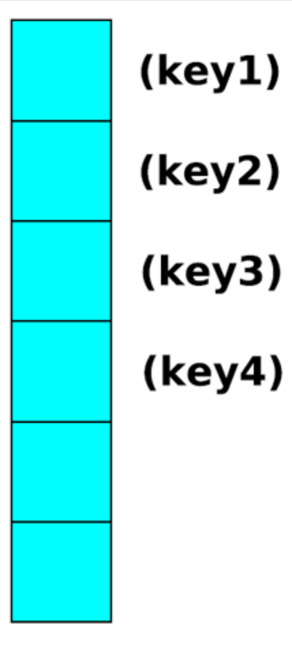
Direct Addressing

- If the universe of keys $|U|$ is small
- Key values range from $U = \{0, 1, \ldots, n\text{-}1\}$
- Use an array A[] of size n, and store each object whose key is K_i at $A[K_i]$ id = 123456
- Insert, Delete, and Search all take O(1). Why?
- What if $|U|$ is large but the number of keys actually stored is small compared to $|U|$?

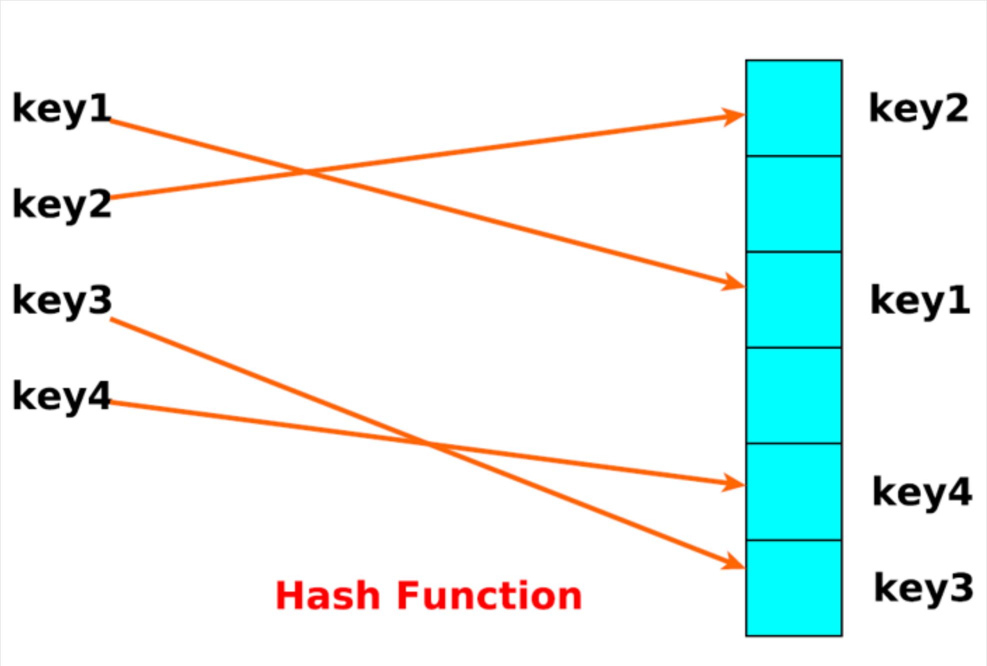
Hash Tables

- Can we "compress" $|U|$ to n?
- Need a method to map a key to a position in this array
- This is the idea behind hash tables
    - Array is called hash table
    - Mapping method called hash function

Hash Tables – Illustration



Direct Addressing



Hash Table

Hash Tables – Definitions

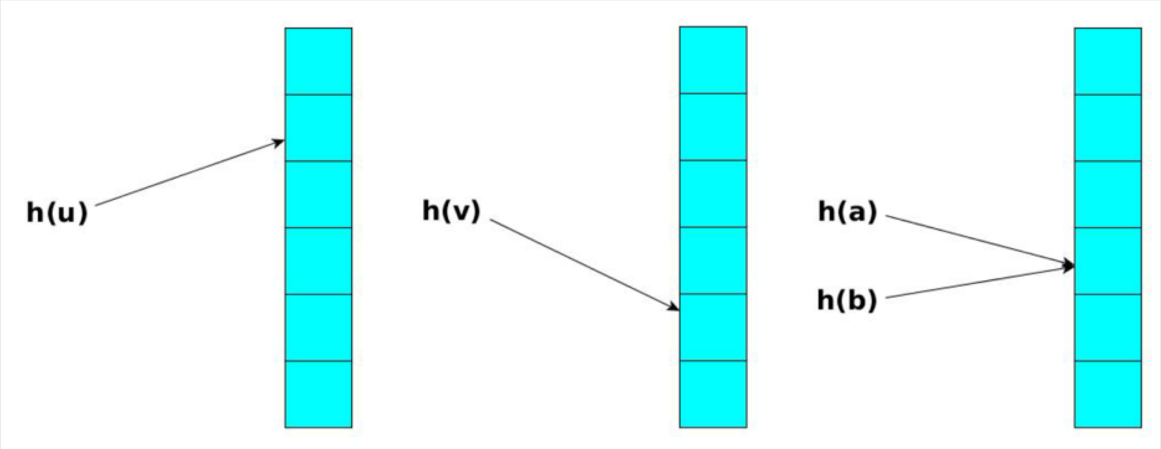
Formally:

- Let H be an array of size n storing the values. H is called a hash table.
- Let the set of possible keys be denoted by the universe $\mathcal{U}$.
- Let h denote a hash function, $h: \mathcal{U} \rightarrow \{0, 1, \ldots, n-1\}$, which maps keys of $\mathcal{U}$ to array positions in H.
- For example, $h(u) = u \,\%\, n$ maps key u to a position in array H.

Hash Tables – Collisions

Collisions:

- If two distinct keys u and v map to the same position/index in the array, i.e., $h(u) = h(v)$, we say that a collision has occurred.



Hash Tables – Choices

When designing the Hash Tables, we consider:

- Hash function
- Size of hash table
- Collision resolution

Hash Tables – Hash Functions

Ideal: Hash function that have no collisions.

- A perfect hash function is one that has no collisions.
- A “good” hash function has to satisfy two requirements which are often in tension:
  1. A hash function needs to distribute keys among positions/cells of the hash table as uniformly as possible. (avoid collisions)
  2. A hash function has to be easy and fast to compute.
- Example: $h(u) = u \ mod \ n$, produces a position index between 0 and $n - 1$.

Perfect Hash Functions (static set)

- If we have a static set, we can achieve perfect hashing and O(1) average (and worst) case timing (given array is big enough).
- One approach to achieve this bound is to generate a perfect hash function for all of the elements a priori.
- **Example 1:** Given $S_1 = \{10; 21; 32; 43; 54; 65; 76; 87\}$, then the function $h_1(x) = x \ mod \ 10$ is perfect.
- **Example 2:** Given $S_2 = \{110; 210; 310; \ldots; 810\}$, then the function $h_2(x) = (x - 10)/100$ is perfect.

Size of Hash Table

If table size (n) < number of keys (p), we are guaranteed to get collisions.

Solutions?

- Choose an initial $n \approx p$
- If dynamic set and p becomes bigger than n, increase size of table (n) and rehash all existing keys.

Collision Resolution

There are two major approaches to handle collisions:

1. Separate Chaining Hashing

2. Open Address Hashing

3a. Separate Chaining Hashing

Separate Chaining Hashing

- What is Separate Chaining Hashing?
  - Hashing that involves chains
  - Separate chains

Separate Chaining Hashing

- Allow more than one key to be stored in a position of the hash table.
- Each position has a linked list, that stores all the keys hashed to that position.
- For completeness, if no key hashed to a position, set linked list pointer to null.

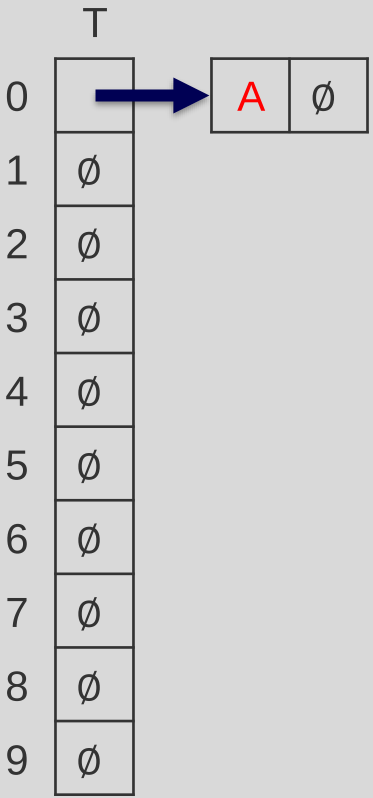
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c = \text{“APJQDHBWM”}$

	T
0	∅
1	∅
2	∅
3	∅
4	∅
5	∅
6	∅
7	∅
8	∅
9	∅

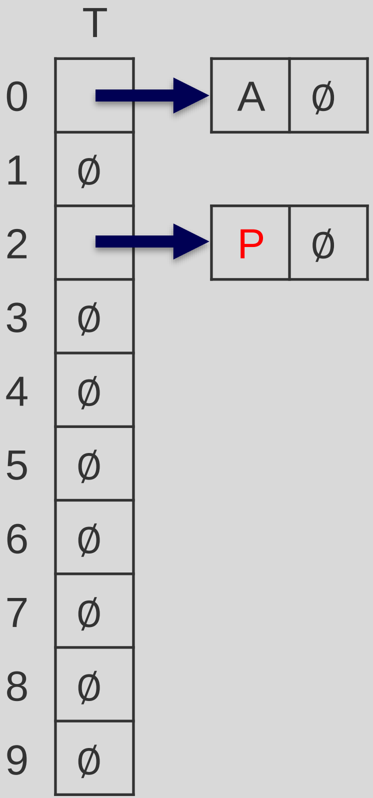
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c =$ "APJQDHBWM"



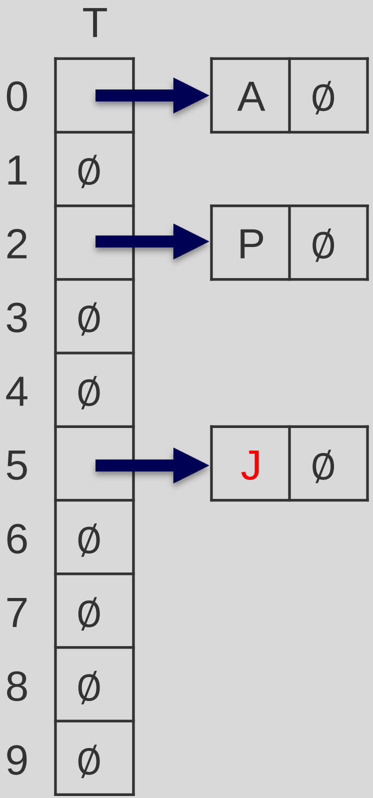
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c = \text{“APJQDHBWM”}$



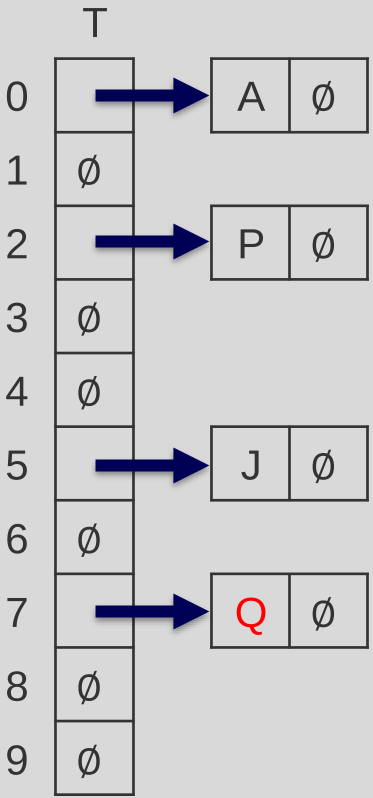
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c =$ “APJQDHBWM”



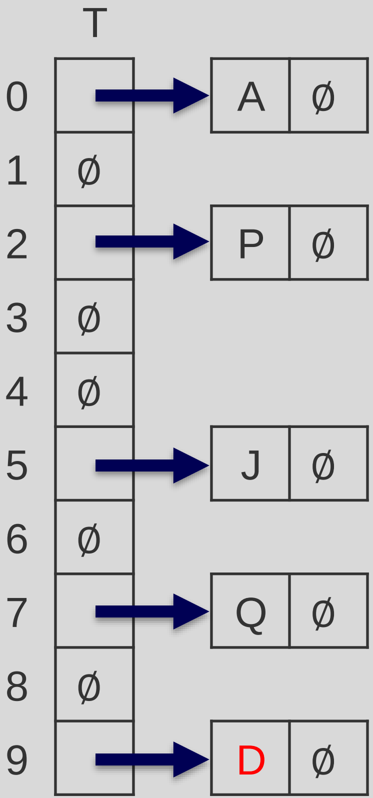
Separate Chaining Hashing – Example

A chained hash table for the sequence T_c = "APJQDHBWM"



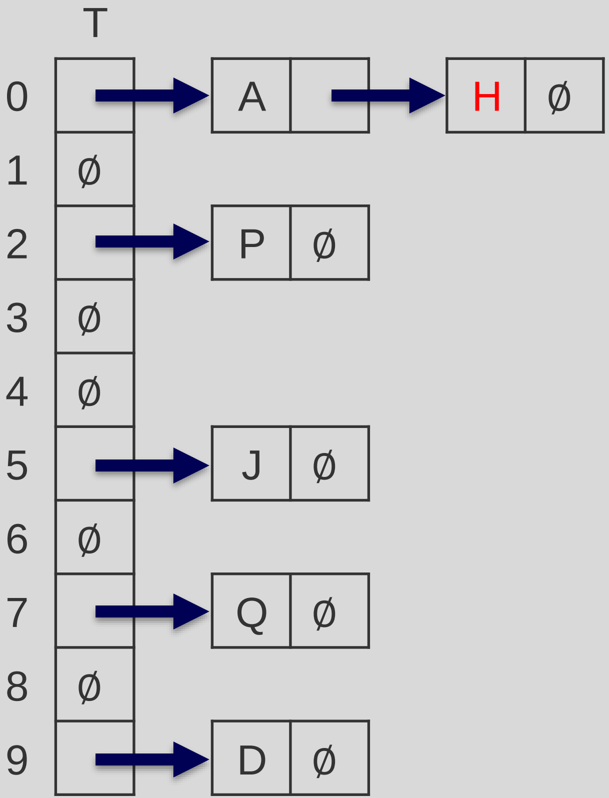
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c =$ "APJQDHBWM"



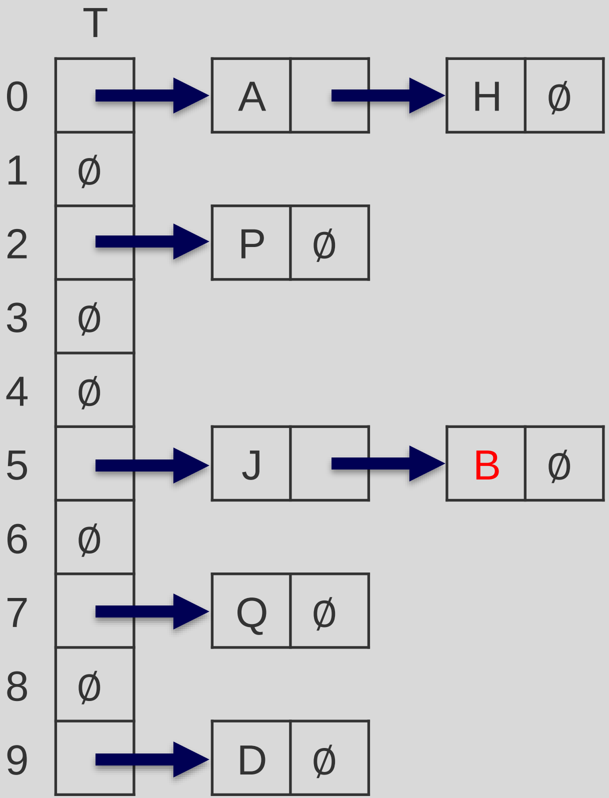
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c =$ "APJQDHBWM"



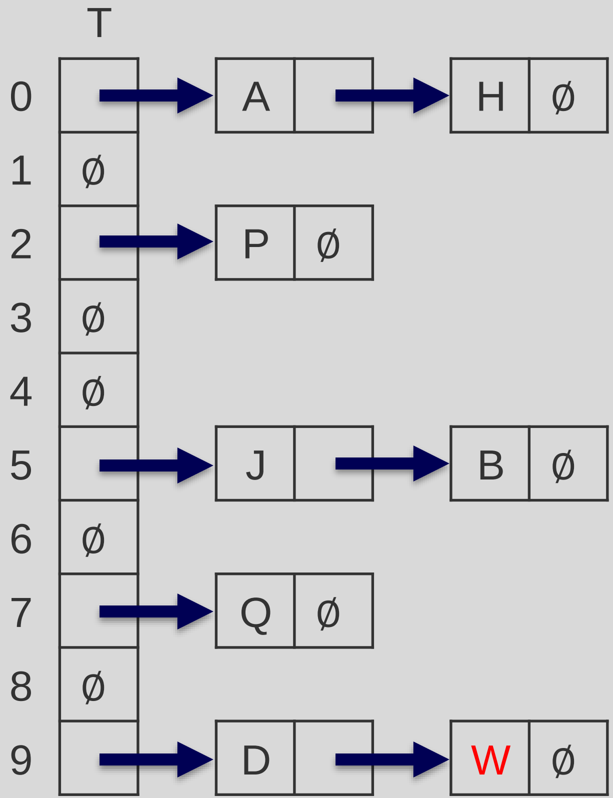
Separate Chaining Hashing – Example

A chained hash table for the sequence
$T_c =$ "APJQDHBWM"



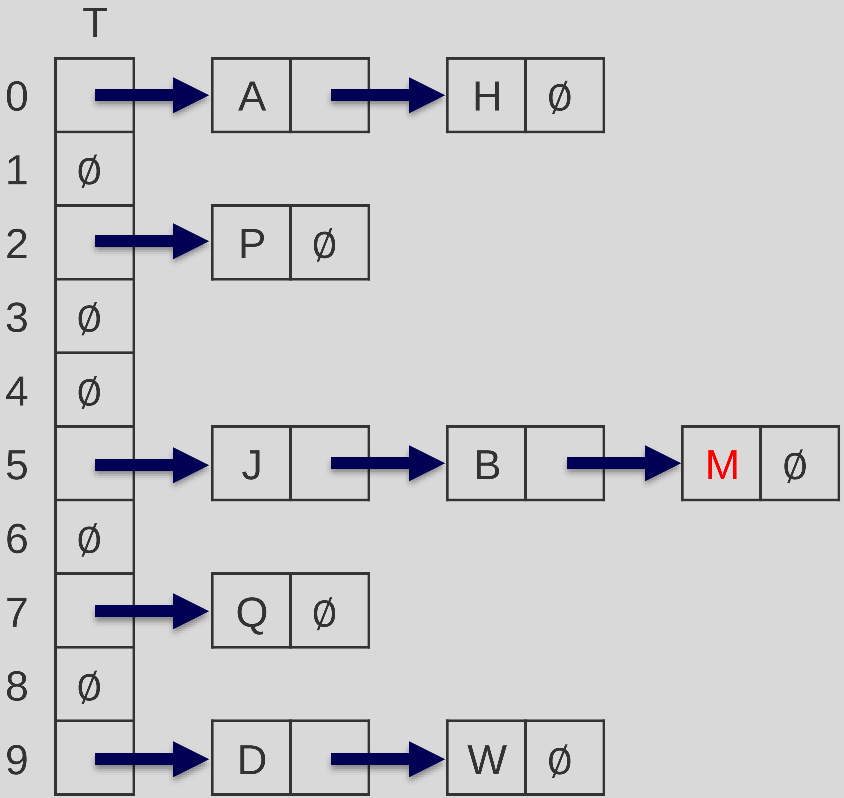
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c = $ “APJQDHBWM”



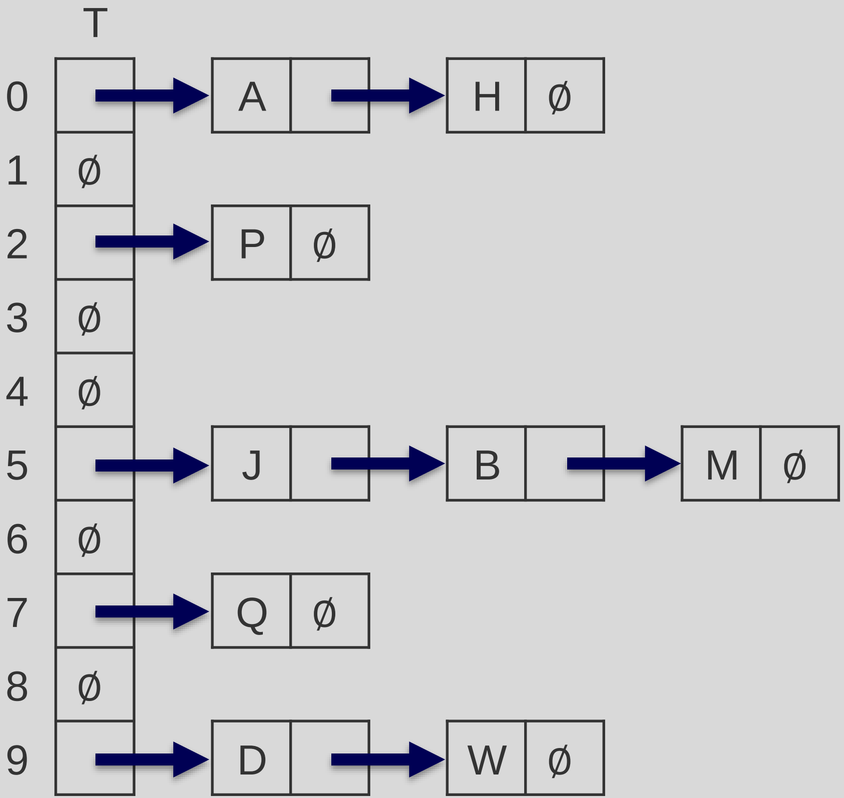
Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c =$ “APJQDHBWM”



Separate Chaining Hashing – Example

A chained hash table for the sequence $T_c = \text{“APJQDHBWM”}$



Separate Chaining Hashing – Cost

- INSERT in O(1) best-case by inserting a new element at the front. It is proportional to length of list if there are collisions.
- DELETE proportional to length of list.
- SEARCH proportional to length of list.
- Average case time is O(1) for all operations, assuming simple uniform hashing (distribute keys uniformly).

Separate Chaining Hashing – Analysis

- It is not unusual for $p > n$ in practice (p = number of keys, n = size of array)
- If the hash function distributes keys uniformly, the average length of any linked list will be $\alpha = p / n$. This ratio is called the load factor
- The number of probes for a successful SEARCH is $1 + \alpha / 2$
- The number of probes for an unsuccessful SEARCH is 1 or $1 + \alpha$

3b. Open Address Hashing

Open Address Hashing – Overview

- Open address hashing is an alternative method to handle collisions.
- Each cell in the base array can store exactly one item.
- Linear probing - store the item in the next free cell.
- Double hashing - use a second hash function to compute the increment.

OAH – Linear Probing

For the sequence T_c = “APJQDHBWM”, construct an open addressing hash table.

OAH – Linear Probing

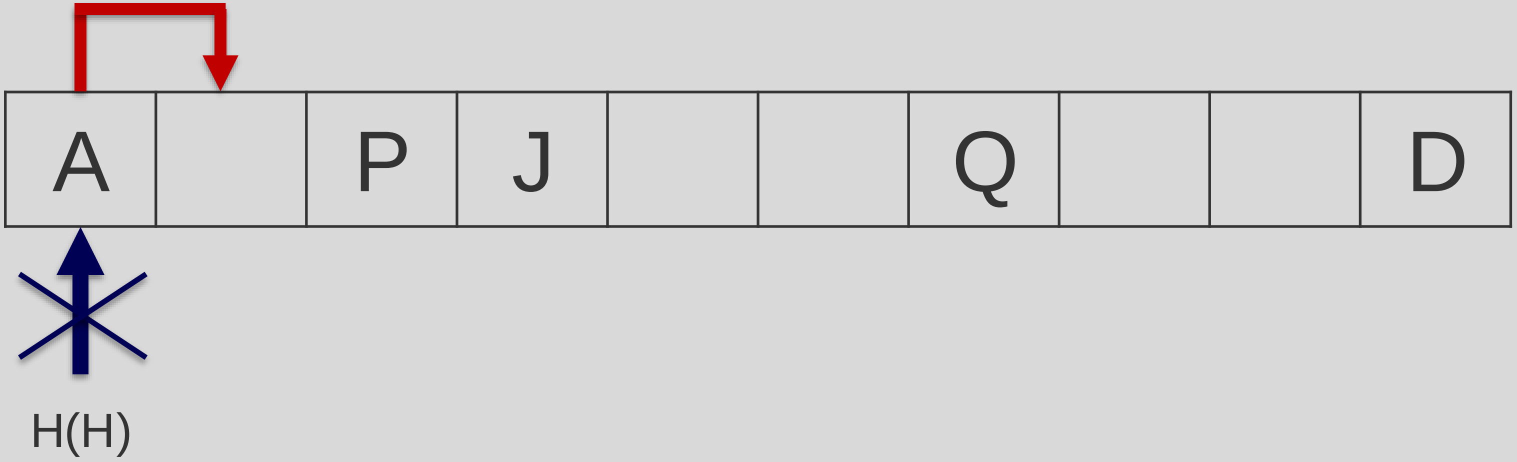
For the sequence $T_c = \text{“APJQDHBWM”}$, construct an open addressing hash table.

OAH – Linear Probing

For the sequence T_c = "APJQDHBWM", construct an open addressing hash table.

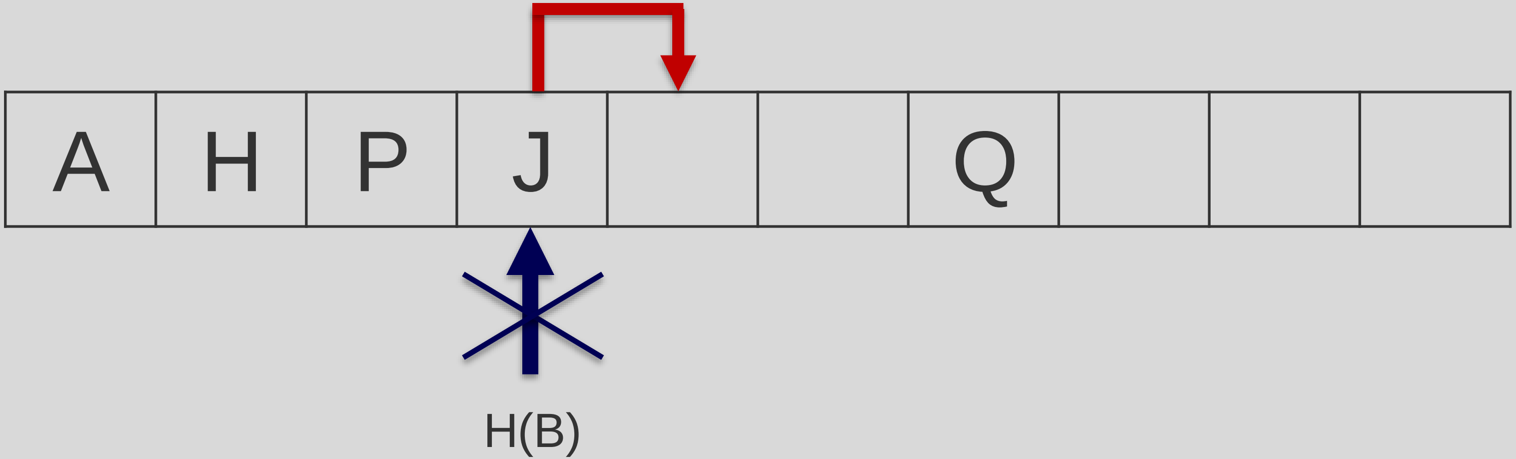
OAH – Linear Probing

For the sequence $T_c = $ “APJQDHBWM”, construct an open addressing hash table.



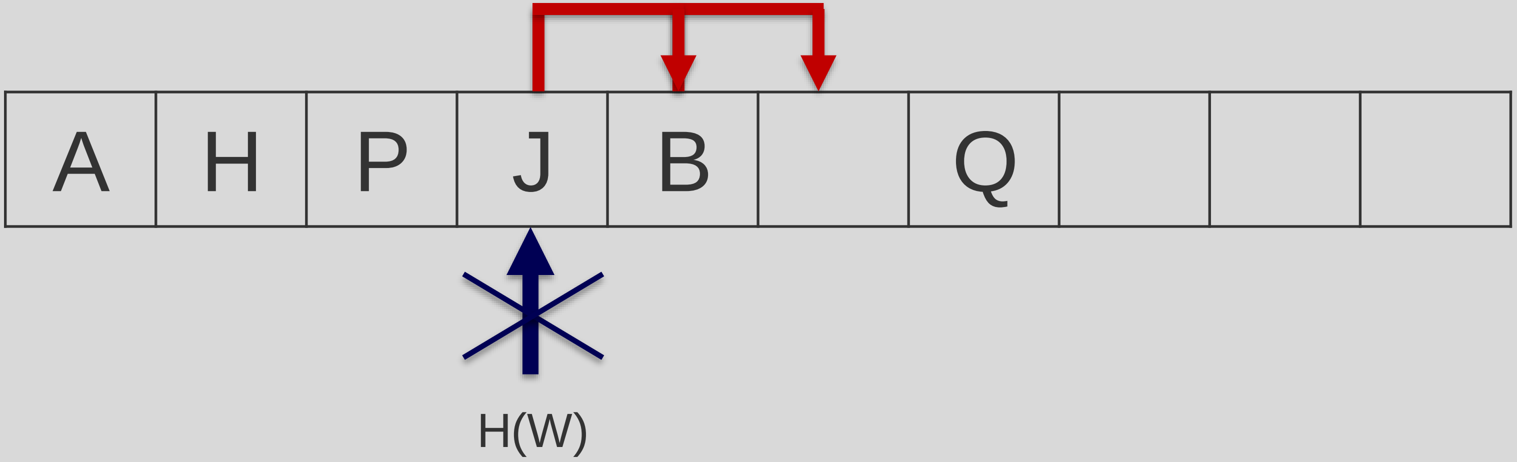
OAH – Linear Probing

For the sequence $T_c =$ "APJQDHBWM", construct an open addressing hash table.



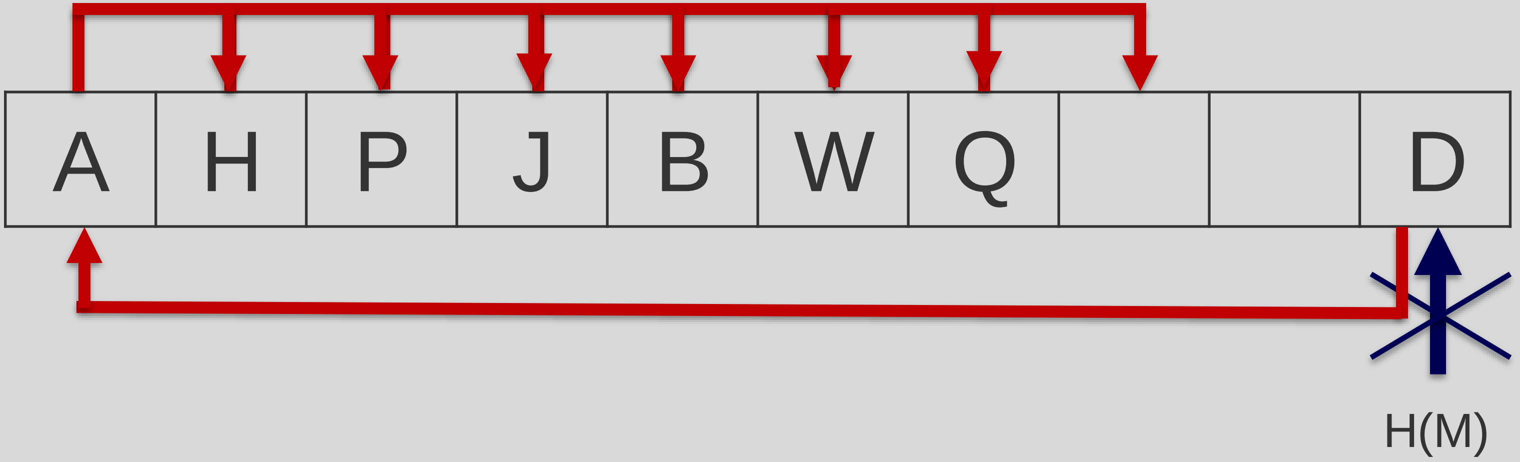
OAH – Linear Probing

For the sequence $T_c =$ "APJQDHBWM", construct an open addressing hash table.



OAH – Linear Probing

For the sequence T_c = "APJQDHBWM", construct an open addressing hash table.



OAH – Linear Probing

For the sequence T_c = "APJQDHBWM", construct an open addressing hash table.

A	H	P	J	B	W	Q	M		D

Linear Probing - Search

- Problem: given a key *K*, return the matching element in the hash table T
- Calculate the position of K: $p = h(K)$
  - If T[p] == null: K does not exist in T
  - If T[p].key == K: return the element at position p
  - Try the next position $(p + 1)$ until either
    - The element at it is null: K does not exist in T
    - The element at it has the same key as K: return that element

Linear Probing - Delete

- Problem: delete (remove) the element whose key is K from a hash table T
- Calculate the position of the element, assume it is *p*
- But simply setting *T[p]* to null may make the search fails. Why?
- Idea: set *T[p]* to a special value (DELETED)
- Search: continue the search process if DELETED is found
- Insert: both empty (null) and DELETED slots can be used to store new elements

OAH – Double Hashing

Double hashing uses two hash functions:

- one is to determine the initial position (same as linear probing)
- the other to determine the size of interval to step (linear probing always has interval of 1)

OAH – Double Hashing

Given two (usually independent universal) hashing functions h_1 and h_2:

- We first do: $h_1(u) \; mod \; n$
- If clash then do: $h_1(u) + 1 \cdot h_2(u) \; mod \; n$
- If clash again then do: $h_1(u) + 2 \cdot h_2(u) \; mod \; n$
- etc

OAH – Double Hashing

- For example: $h_1(a) = 4$
  - But $\mathrm{H}[4]$ is occupied. Next position to check is not 5.
- Let $h_2(a) = 3$
  - then next position to check is $h_1(u) + h_2(u) = 7$
- If 7th is occupied, then check the 10th position (i.e., 4 + 2*3)
- If h_2 is chosen well, we can avoid clustering effects, which can lead to faster collision resolution.

OAH – Requirements

- For every key K, call this a probe sequence: $h(K, 0), h(K, 1), \ldots, h(K, n-1)$
  - $h(K, 0)$: the position returned after the first hash
  - $h(K, 1)$: the next hash position if h(K, 0) has a collision, and so on
- For every K, the probe sequence must be a permutation of the set $\{0, 1, \ldots, N-1\}$ => no position is skipped when the table is filled up
- The probe sequence of Linear Probing satisfies the above requirement

OAH – Requirements

- For Double Hashing, it is not always satisfied
- $\mathrm{h(K, step) = (h_1(K) + step^*h_2(K))\ \%\ n}$
- Example: $\mathrm{h(K) = (K + step^*(3K))\ \%\ n}$
  - $\mathrm{h_1 = 1}$
  - $\mathrm{h_2 = 3}$
  - If hash tale size = 9
  - Not all positions are probed. Why?
- $\mathrm{GCD(h_2(K), N)}$ must be 1 to probe all positions (GCD is the greatest common divisor)
- N is usually selected as a prime number

Double Hashing – Comments

- Difficult to analyse the complexity of successful and unsuccessful searches (depends on load factor)
- Empirically shown double hashing performs better than linear probing, especially when table is more full.
  - The cost for second hashing is O(1) and we can reduce the chance of collision

Java Collections Classes

- If you use the Set ADT, Java Collections Framework provides a Set interface
- The Set interface has some implementations
    - HashSet: use a hash table as the underlying data structure, maintain no order between elements
    - TreeSet: use a tree (a Red-Black tree) as the underlying data structure, maintain a natural order based on keys
    - LinkedHashSet: use a hash table and a doubly linked list, maintain an insertion order

Wrapping things up

RMIT UNIVERSITY

Summary

The two types of space-time tradeoffs discussed:

- **input enhancement**
  - preprocess input and store relevant information that speeds up solving the problem.
  - e.g., distribution sorting
- **Pre-structuring**
  - construct data structures (space) that have faster or more flexible access to data.
  - e.g., hashing

RMIT UNIVERSITY