COSC1285/2123: Algorithms & Analysis

Time and Space Tradeoffs

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 7

Overview

Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 7.

Learning outcomes:

- Understand space-time tradeoffs in algorithm design.
- Two of the paradigms:
  - input enhancement (sort by counting)
  - prestructuring (hashing)

Outline

Overview

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

Time & Space Tradeoffs

We can gain time by using more space – whole idea behind this lecture.
In this lecture, we discuss two varieties of Time & Space tradeoffs:

Time & Space Tradeoffs

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** - preprocess the input to store extra information that will accelerate the solving of the problem.
    - counting sorts

Time & Space Tradeoffs

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** - preprocess the input to store extra information that will accelerate the solving of the problem.
   - counting sorts
2. **Prestructuring** - use extra space to make accessing its elements easier or faster.
   - hashing

Overview

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

Count-based Sorting

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

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 unique values, 1, 2, 3
- Imagine if our array consist of 2,1,2,3,1

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 unique values, 1, 2, 3
- Imagine if our array consist of 2,1,2,3,1
- If I arrange the array with the 1s, then the 2s, then the 3s, then I have sorted the array (1,1,2,2,3).

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 unique values, 1, 2, 3
- Imagine if our array consist of 2,1,2,3,1
- If I arrange the array with the 1s, then the 2s, then the 3s, then I 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).

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 array have a $\leq$ value in array).

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 array have a $\leq$ value in array).
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. We copy the elements in original array going from right to left (in order to have stable sorting).

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 array have a $\leq$ value in array).
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. We copy the elements in original array going from right to left (in order to have stable sorting).

Consider the array $A = \{13, 11, 12, 13, 12, 12\}$.

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 array have a $\leq$ value in array).
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. 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

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 array have a $\leq$ value in array).
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. 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

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 array have a $\leq$ value in array).
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. 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

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 array have a $\leq$ value in array).
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. 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

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	5	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	5	8	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	5	8	9

1	2	3	4	5	6	7	8	9

12	4	12	9	12	15	9	4	9

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	5	8	9

1	2	3	4	5	6	7	8	9
				9				

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
2	4	8	9

1	2	3	4	5	6	7	8	9
				9				

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
2	4	8	9

1	2	3	4	5	6	7	8	9
	4			9				

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	4	8	9

1	2	3	4	5	6	7	8	9
	4			9				

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	4	8	9

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	3	8	9

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	3	8	9

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	3	8	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	3	8	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	3	7	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	3	7	8

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

12	4	12	9	12	15	9	4	9

4	9	12	15
1	2	7	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	2	7	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	2	6	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
1	2	6	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
0	2	6	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
0	2	6	8

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

Distribution Sorting

12	4	12	9	12	15	9	4	9

4	9	12	15
0	2	5	8

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

Distribution Sorting

ALGORITHM **DistCountSort** ($A[0 \ldots n-1], u, l$)
// Sort an array by distribution counting
// INPUT : An array $A[0 \ldots n-1]$ of orderable integers between l and u ($l \leq u$), and $n_{\text{max}} = u - l$
// OUTPUT : An array $S[0 \ldots n-1]$ of integers sorted in nondecreasing order.

for $j = 0$ **to** n_{max} **do** ▷ Initialize frequencies
  $\sigma[j] = 0$
end for
for $i = 0$ **to** $n - 1$ **do** ▷ Compute frequencies
  $\sigma[A[i] - l] = \sigma[A[i] - l] + 1$
end for
for $j = 1$ **to** n_{max} **do** ▷ Compute cumulative frequencies
  $\sigma[j] = \sigma[j-1] + \sigma[j]$
end for
for $i = n - 1$ **down to** 0 **do**
  $j = A[i]$
  $S[\sigma[j] - 1] = A[i]$
  $\sigma[j] = \sigma[j] - 1$
end for
return S

Distribution Sorting

The worst-case analysis for distribution sorting is:

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

Distribution Sorting

The worst-case analysis for distribution sorting is:

$$C(n) = \underbrace{\sum_{j=0}^{n_{\max}} 1}_{\text{initialise freqs}} + \underbrace{\sum_{i=0}^{n-1} 1}_{\text{compute freqs}} + \underbrace{\sum_{j=0}^{n_{\max}} 1}_{\text{compute cumulative freq}} + \underbrace{\sum_{i=0}^{n-1} 1}_{\text{copy values}}$$

$$= 2\sum_{i=0}^{n-1} 1 + 2\sum_{j=0}^{n_{\max}} 1$$

$$= 2\mathcal{O}(n) + 2\mathcal{O}(n_{\max})$$

$$\in \mathcal{O}(n), \text{ if } n > n_{\max}$$

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

Overview

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).

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?

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? E.g.,

- List
- Tree (balanced)
- Array

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? E.g.,

- List
- Tree (balanced)
- Array

What is the average case time complexities of INSERT, DELETE , MEMBER?

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? E.g.,

- List
- Tree (balanced)
- Array

What is the average case time complexities of INSERT, DELETE , MEMBER?

	List	Bal Tree	Array
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
DELETE	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
MEMBER	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

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? E.g.,

- List
- Tree (balanced)
- Array

What is the average case time complexities of INSERT, DELETE , MEMBER?

	List	Bal Tree	Array
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log\ n)$	$\mathcal{O}(n)$
DELETE	$\mathcal{O}(n)$	$\mathcal{O}(\log\ n)$	$\mathcal{O}(n)$
MEMBER	$\mathcal{O}(n)$	$\mathcal{O}(\log\ n)$	$\mathcal{O}(\log\ n)$

Is it possible to achieve better efficiency?

Hash Tables

If we have an array holding keys, and $\mathcal{O}(1)$ method to map a key to a position in this array, we can improve on previous average case complexities.

Hash Tables

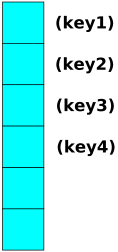
If we have an array holding keys, and $\mathcal{O}(1)$ method to map a key to a position in this array, we can improve on previous average case complexities.

This is the idea behind hash tables. Array is called hash table, mapping method called hash function.

Hash Tables

If we have an array holding keys, and $\mathcal{O}(1)$ method to map a key to a position in this array, we can improve on previous average case complexities.

This is the idea behind hash tables. Array is called hash table, mapping method called hash function.

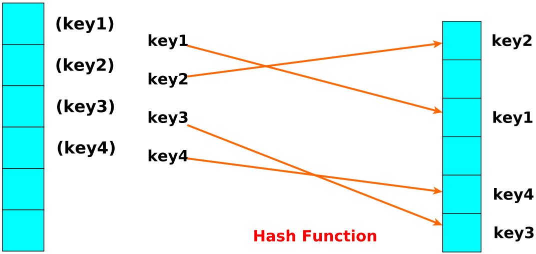


(e) Array

Hash Tables

If we have an array holding keys, and $\mathcal{O}(1)$ method to map a key to a position in this array, we can improve on previous average case complexities.

This is the idea behind hash tables. Array is called hash table, mapping method called hash function.



(g) Array (h) Hash table

Hash Tables

Formally:

- Let H be an array of size n storing the values. H is called the hash table.

Hash Tables

Formally:

- Let H be an array of size n storing the values. H is called the hash table.
- Let the set of possible keys be denoted by the universe $\mathcal{U}$.

Hash Tables

Formally:

- Let H be an array of size n storing the values. H is called the hash table.
- Let the set of possible keys be denoted by the universe $\mathcal{U}$.
- Let the 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.

Hash Tables

Formally:

- Let H be an array of size n storing the values. H is called the hash table.
- Let the set of possible keys be denoted by the universe $\mathcal{U}$.
- Let the 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.
- E.g., $h(u)$ maps key u to a position in array H.

Hash Tables

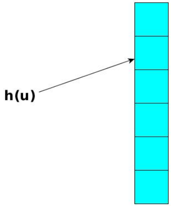
Formally:

- Let H be an array of size n storing the values. H is called the hash table.
- Let the set of possible keys be denoted by the universe $\mathcal{U}$.
- Let the 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.
- E.g., $h(u)$ maps key u to a position in array H.

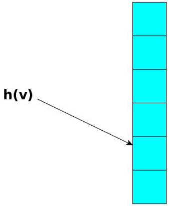
Collisons:

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

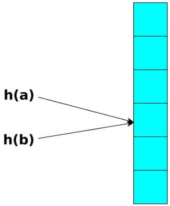
Hash Tables



(i) h(u)



(j) h(v)



(k) Collision

Hash Table Choices

- Hash function.
- Size of hash table.
- Collison resolution.

Hash Functions

Ideal: Hash function that have no collisons.

- A perfect hash function is one that has no collisions.

Hash Functions

Ideal: Hash function that have no collisons.

- 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.

Hash Functions

Ideal: Hash function that have no collisons.

- 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 $\mathcal{O}(1)$ average (and worst) case timing (given array is big enough).

Perfect Hash Functions (static set)

- If we have a static set, we can achieve perfect hashing and $\mathcal{O}(1)$ 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*.

Perfect Hash Functions (static set)

- If we have a static set, we can achieve perfect hashing and $\mathcal{O}(1)$ 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 $S_2 = \{110, 210, 310, \dots, 810\}$, 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.

Size of Hash Table

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

Solutions?

- Choose an initial $n \approx p$ and a prime number (reduce collison chance, but not waste space).
- 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 collisons:

1. Separate Chaining Hashing.
2. Open Address Hashing.

Separate Chaining Hashing – Overview

- Allow more than one key to be stored in a position of the hash table.

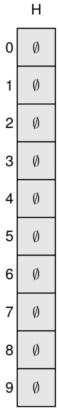
Separate Chaining Hashing – Overview

- 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.

Separate Chaining Hashing – Overview

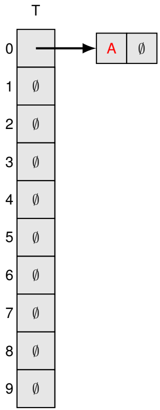
- Allow more than one key to be stored in a position of the hash table.
- Each position has a linked list, that stores all the keys hashed to that position.
- For completeness, if no key hashed to a position, set linked list pointer to null.

Separate Chaining Hashing – Example



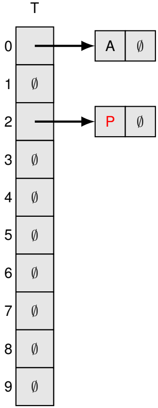
A chained hash table for the sequence $\mathcal{T}_c =$ “APJQDHBWM”.

Separate Chaining Hashing – Example



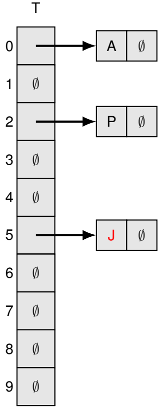
A chained hash table for the sequence $\mathcal{T}_c =$ "APJQDHBWM".

Separate Chaining Hashing – Example



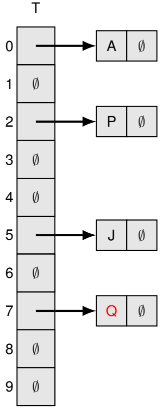
A chained hash table for the sequence $\mathcal{T}_c =$ "APJQDHBWM".

Separate Chaining Hashing – Example



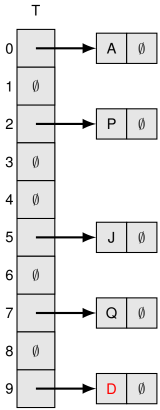
A chained hash table for the sequence $\mathcal{T}_c =$ "APJQDHBWM".

Separate Chaining Hashing – Example



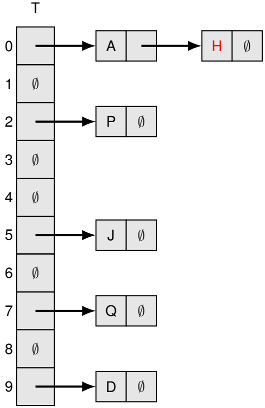
A chained hash table for the sequence $\mathcal{T}_c =$ "APJQDHBWM".

Separate Chaining Hashing – Example



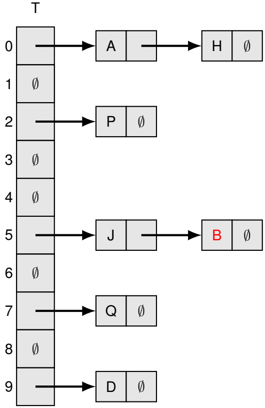
A chained hash table for the sequence $\mathcal{T}_c =$ “APJQDHBWM”.

Separate Chaining Hashing – Example



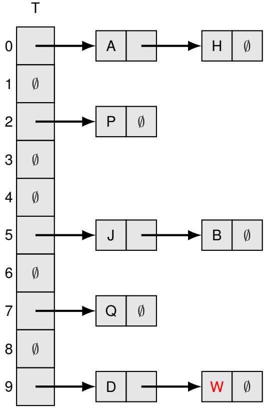
A chained hash table for the sequence $\mathcal{T}_c =$ "APJQDHBWM".

Separate Chaining Hashing – Example



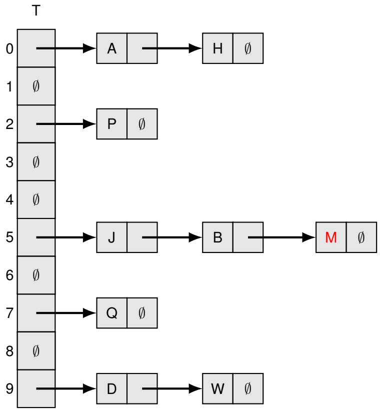
A chained hash table for the sequence $\mathcal{T}_c =$ “APJQDHBWM”.

Separate Chaining Hashing – Example



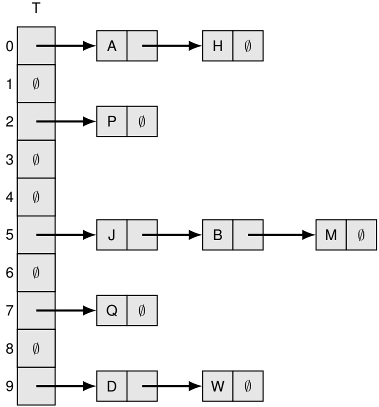
A chained hash table for the sequence $\mathcal{T}_c =$ "APJQDHBWM".

Separate Chaining Hashing – Example



A chained hash table for the sequence $\mathcal{T}_c =$ “APJQDHBWM”.

Separate Chaining Hashing – Example



A chained hash table for the sequence $\mathcal{T}_c =$ "APJQDHBWM".

Separate Chaining Hashing – Cost

- INSERT in $\mathcal{O}(1)$ worst-case by inserting a new element at the front or end of the list depending on the implementation.

Separate Chaining Hashing – Cost

- INSERT in $\mathcal{O}(1)$ worst-case by inserting a new element at the front or end of the list depending on the implementation.
- DELETE proportional to length of list.

Separate Chaining Hashing – Cost

- INSERT in $\mathcal{O}(1)$ worst-case by inserting a new element at the front or end of the list depending on the implementation.
- DELETE proportional to length of list.
- MEMBER proportional to length of list.

Separate Chaining Hashing – Cost

- INSERT in $\mathcal{O}(1)$ worst-case by inserting a new element at the front or end of the list depending on the implementation.
- DELETE proportional to length of list.
- MEMBER proportional to length of list.
- Average case time is $\mathcal{O}(1)$ for all operations, assuming simple uniform hashing (distribute keys uniformally).

Separate Chaining Hashing – Analysis

- It is not unusual for $p > n$ in separate chaining hash tables in practice (p = number of keys, n = size of array).

Separate Chaining Hashing – Analysis

- It is not unusual for $p > n$ in separate chaining hash tables in practice (p = number of keys, n = size of array).
- If the hash function distributes keys uniformly (simple uniform hashing), the average length of any linked list will be $\alpha = p/n$. This ratio is called the load factor.

Separate Chaining Hashing – Analysis

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

Separate Chaining Hashing – Analysis

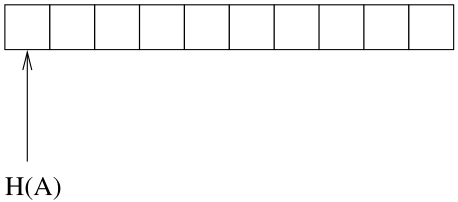
- It is not unusual for $p > n$ in separate chaining hash tables in practice (p = number of keys, n = size of array).
- If the hash function distributes keys uniformly (simple uniform hashing), the average length of any linked list will be $\alpha = p/n$. This ratio is called the load factor.
- The expected number of probes for a successful **MEMBER** is $\approx 1 + \alpha/2$.
- The expected number of probes for an unsuccessful **MEMBER** is α.
- The load α is typically kept small (and ideally it is 1).

Overview

Open Address Hashing – Overview

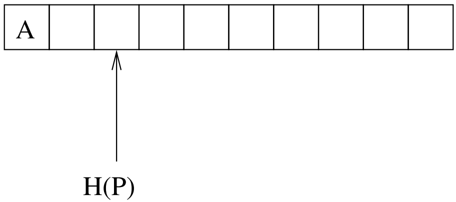
- Open address hashing is an alternative method to handle collisions.
- Each cell in the base array can store exactly one item.
- Linear probing - store the item in the next free cell.
- Double hashing - use a second hash function to compute the increment.

Open Address Hashing: Linear probing



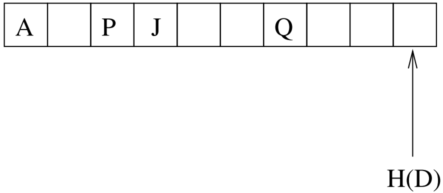
For the sequence $\mathcal{T}_c =$ "APJQDHBWM", construct an open addressing hash table.

Open Address Hashing: Linear probing



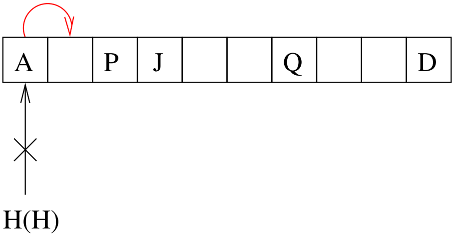
For the sequence $\mathcal{T}_c =$ “APJQDHBWM”, construct an open addressing hash table.

Open Address Hashing: Linear probing



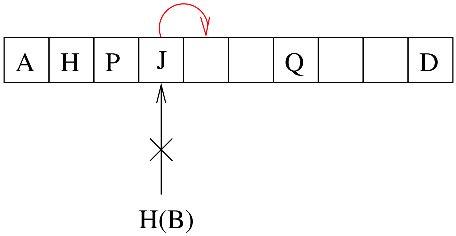
For the sequence $\mathcal{T}_c =$ "APJQDHBWM", construct an open addressing hash table.

Open Address Hashing: Linear probing



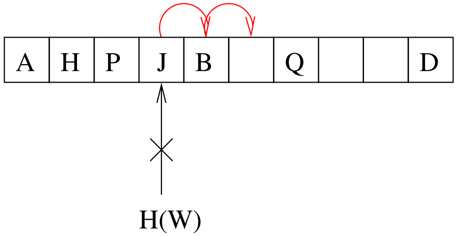
For the sequence $\mathcal{T}_c =$ "APJQDHBWM", construct an open addressing hash table.

Open Address Hashing: Linear probing



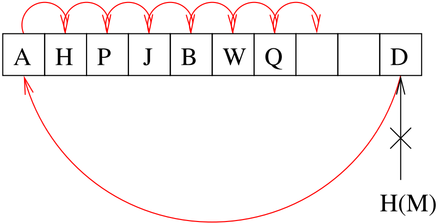
For the sequence $\mathcal{T}_c =$ “APJQDHBWM”, construct an open addressing hash table.

Open Address Hashing: Linear probing



For the sequence $\mathcal{T}_c =$ “APJQDHBWM”, construct an open addressing hash table.

Open Address Hashing: Linear probing



For the sequence $\mathcal{T}_c =$ “APJQDHBWM”, construct an open addressing hash table.

Open Address Hashing: Linear probing

A	H	P	J	B	W	Q	M		D

For the sequence $\mathcal{T}_c =$ "APJQDHBWM", construct an open addressing hash table.

Linear probing – Analysis

- Does not work if $p > n$.

Linear probing – Analysis

- Does not work if $p > n$. (If p becomes bigger than n, we can resize and rehash.)

Linear probing – Analysis

- Does not work if $p > n$. (If p becomes bigger than n, we can resize and rehash.)
- Number of probes for the three key set operations depend on the load factor $\alpha = p/n$.

Linear probing – Analysis

- Does not work if $p > n$. (If p becomes bigger than n, we can resize and rehash.)
- Number of probes for the three key set operations depend on the load factor $\alpha = p/n$.
- For linear probing, a successful search is $\approx \frac{1}{2}(1 + 1/(1-\alpha))$ probes and an unsuccessful search is $\approx \frac{1}{2}(1 + 1/(1-\alpha)^2)$ probes.
- It is much more difficult to derive these bounds than in the separate chaining hash table.

Linear probing – Analysis

As the table fills ($\alpha \to 1$), the number of probes necessary increases dramatically.

α	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

Open Address Hashing – 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).

Open Address Hashing – 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).

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

- We first do: $h_1(u) \mod n$

Open Address Hashing – 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).

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

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

Open Address Hashing – 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).

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

Open Address Hashing – 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).

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

E.g. $h_1(a) = 4$, but H[4] is occupied. Next position to check is not 5. Let $h_2(a) = 3$, then next positon to check is $h_1(u) + h_2(u) = 7$.

Open Address Hashing – 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).

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

E.g. $h_1(a) = 4$, but H[4] is occupied. Next position to check is not 5. Let $h_2(a) = 3$, then next positon to check is $h_1(u) + h_2(u) = 7$.

If h_2 is chosen well, we can avoid clustering effects, which can lead to faster collison resolution.

Double hashing – Comments

- Difficult to analyse the complexity of successful and unsuccessful searches.
- Empirically shown double hashing performs better than linear probing, especially when table is more full.

Other Applications of Hashing

Checksums and signatures:

`https://www.youtube.com/watch?v=b4b8ktEV4Bg`

Locality sensitive hashing:

`https://www.youtube.com/watch?v=Ha7_Vf2eZvQ`

Overview

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
- **prestructuring**
  - construct data structures (space) that have faster or more flexible access to data.
  - e.g., hashing