Data Structures and Algorithms 2

Prof. Ahmed Guessoum
The National Higher School of AI

Chapter 5

Hashing

Motivating Example

We want to store a list whose elements are integers between 1 and 5

We will define an array of size 5, and if the list has element j, then j is stored in A[j-1], otherwise A[j-1] contains 0.

Complexity of find operation is O(1)

- The space for storage is called ``**hash table**,'' H
- Ideal hash table data structure is just an array of some fixed size, ***TableSize***, containing the data items.
- A search for an item is performed on some part (i.e. data member) of the item, called the **key.**
- For example, an item could consist of a string (that serves as the key) and additional data members (for instance, a name that is part of a large employee structure, etc.).
- The common convention is to have the table run from 0 to $TableSize - 1$

Hashing

- Each key is mapped into some number in the range 0 to *TableSize* − 1 and placed in the appropriate cell.
- The mapping is called a **hash function, *h*,** which ideally
  – simple to compute and
  – any two distinct keys should get different cells
- Since there are a finite number of cells and a very large supply of keys, this is clearly impossible,

➔ we seek a hash function that

  ➢ finds an element in constant time "on the average"

  ➢ distributes the keys evenly among the cells

Hash Functions

- Suppose that the hash table has size M
- There is a hash function which maps an element to a value p in 0,…., M-1, and the element is placed in position p in the hash table.
- The function is called *h*; the hash value for key *j* is *h[j]*
- If *h[j]* = *k*, then the element is added to *H[k]*, i.e. at position *k* in *H*.

Example of an ideal hash table

Khalid 25000
Tariq 31250
Aicha 27500
Asma 28200

How to choose a hash function?

How to decide on the table size?

What do we do with collisions?

Choice of a Hash Function

- If the input keys are integers, then *Key mod TableSize* is generally a reasonable strategy, unless *Key* happens to have some undesirable properties.
- One has to be careful in the design of the hash function.
  - E.g., suppose *tableSize* = 10 and that the keys all end in zero, then the standard hash function is clearly a bad choice!
- It is often a good idea to ensure that the table size is prime
- When the input keys are random integers, then the above function is not only very simple to compute but also distributes the keys evenly

Choosing a hash function

- Usually, the keys are strings; in this case, the hash function needs to be chosen carefully.
- One option is to add up the ASCII values of the characters in the string.

Consider **Example 1 of a hash function**

```
int hash( const string & key, int tableSize )
{
        int hashVal = 0;
        for( char ch : key )
                hashVal += ch;
        return hashVal % tableSize;
}
```

- The previous hash function is simple to implement and computes an answer quickly.
- However, if the table size is large, the function does not distribute the keys well (fairly evenly).
- E.g., suppose that *TableSize* = 10,007 (a prime number).
- Suppose all the keys are eight or fewer characters long.
- Since an ASCII character has an integer value <= 127, the values produced by the hash function are between 0 and 1,016, which is $127 * 8$.
- This is clearly not an even distribution over the hash table! (About 90% of the table will never be used!)

Example 2 of a hash function

```
int hash( const string & key, int tableSize )
{
    return ( key[ 0 ] + 27 * key[ 1 ] + 729 * key[ 2 ] ) % tableSize;
}
```

27 is the number of English letters + blank char; 729 is 27^2

- This hash function is easy to compute.
- It examines only the first three characters.
- If characters are random and table size is 10,007, as before, then we would expect a reasonably equitable distribution.
- In fact, looking up a dictionary, there are only 2,851 not 17576 (26^3) combinations. ➔ though no collisions, only 28% of the table is actually hashed to.

Example 3 of a hash function

```
unsigned int hash( const string & key, int tableSize )
{
        unsigned int hashVal = 0;
        for( char ch : key )
                hashVal = 37 * hashVal + ch;
        return hashVal % tableSize;
}
```

- Involves all characters in the key and can generally be expected to distribute well (it computes

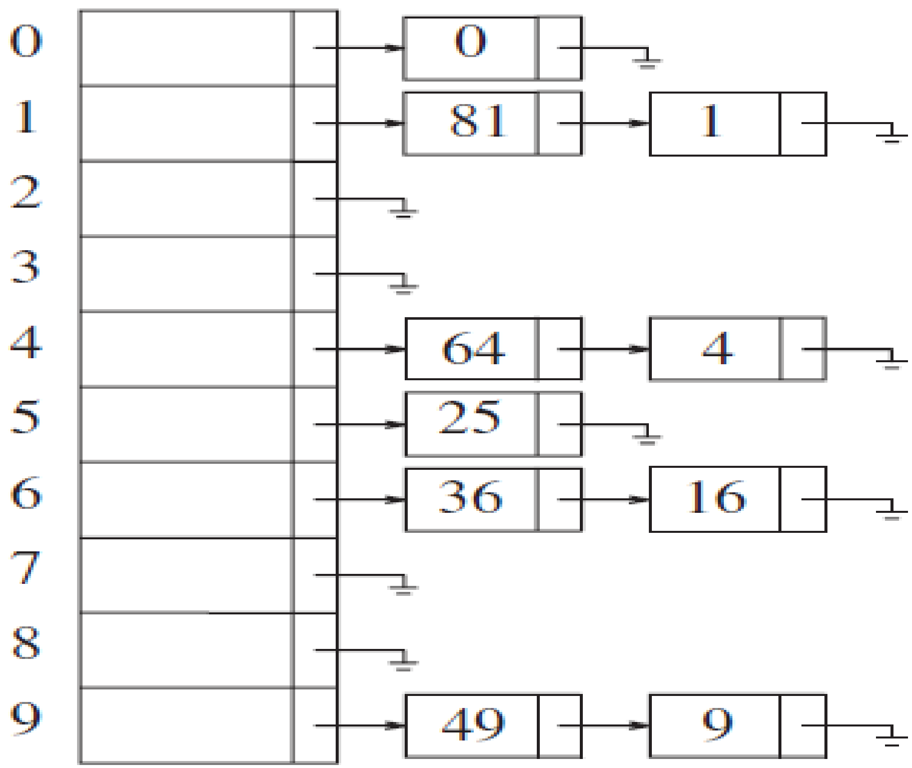
$$\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^i$$

- The code computes a polynomial function (of 37).

- The hash function takes advantage of the fact that overflow is allowed and uses unsigned int to avoid introducing a negative number.
- This hash function has a reasonable table distribution, not necessarily the best.
- It does have the merit of extreme simplicity and is reasonably fast.
- If the keys are very long, the hash function will take too long to compute.
  - A common practice in this case is not to use all the characters.
  - E.g. a street address key: Use a couple of characters from the street address, a couple from city, and from zip code.

Handling Collisions: Separate Chaining

- **Separate chaining approach:** keep a linked list of all the elements that hash to the same value.
- Suppose that the keys are the first 10 perfect squares and hashing function is hash(x) = x mod 10



Operations using Hashing

- **Search in Hash Table:** use the hash function to determine which list to traverse. Then search the appropriate list.
- **Insertion in Hash Table:** check the appropriate list to see if element is found (if duplicates are expected, an extra counter data member is incremented). Otherwise, insert it at front of the list: convenience and likelihood of frequent access.
- **Deletion of an element:** do the hashing, then delete from the linked list.
- <u>Note</u>: the hash tables in this chapter work only for objects that provide a hash function and equality operators (operator== an/or operator!=). Comparables?

Hash function implementation

- Use of function object template (C++11)

```
template <typename Key>
class hash
{
   public:
         size_t operator() ( const Key & k ) const;
};
```

- The type *size_t* is an unsigned integral type that represents the size of an object; ➔ it is guaranteed to be able to store an array index
- On a 32-bit system size_t will take 32 bits, on a 64-bit one 64 bits

Default implementations of hash function template using standard type string:

```
template <>
class hash<string>
{
  public:
      size_t operator()( const string & key )
      {
            size_t   hashVal = 0;
            for( char ch : key )
                  hashVal = 37 * hashVal + ch;
            return hashVal;
      }
};
```

Alternatives to Linked Lists?

- Any scheme could be used besides linked lists to resolve the collisions.
- A binary search tree or even another hash table would work.
- If the table is large and the hash function is good, all the lists should be short ➔ so basic separate chaining makes no attempt to try anything complicated.

Load factor of a hash table

- **Load factor** of a hash table:

  λ = # of elements in the hash table / table size
- In previous example, $\lambda = 1.0$.
- Usually, a threshold is set on λ to do the **rehashing**: i.e. expanding the table and re-calculating the hash code of already stored entries
- Time required to perform a search = constant time to evaluate the hash function + time to traverse the list.

HTs without LLs: probing hash tables

- Hashing with separate chaining has the disadvantage of using linked lists ➔ can slow the algorithm down
- **Alternative approach** (to resolving collisions with linked lists) is to try alternative cells until an empty cell is found.
- More formally, cells $h_0(x)$, $h_1(x)$, $h_2(x)$, . . . are tried in succession, where

$$h_i(x) = (hash(x) + f(i)) \text{ mod } TableSize, \text{ with } f(0) = 0.$$

- f is the collision resolution strategy.
- All the data go inside the table ➔ a bigger table is needed in this approach
- Generally, the load factor should be below $\lambda = 0.5$

Linear Probing

- **Linear probing**: f is a linear function of i, typically $f(i) = i$

➔trying cells sequentially (with wraparound) in search of an empty cell

e.g. with hash(x) = x mod 10 and linear probing, insert 89, 18, 49, 58, 69

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Linear Probing (cont.)

- As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect, known as **primary clustering,** ➔ key that hashes into the cluster will require several attempts to resolve the collision, and then it will be added to the cluster.
- It can be shown that the expected number of probes using linear probing is roughly $1/2 \ (1 + 1/(1 - \lambda^2))$ for insertions and unsuccessful searches, and $1/2 \ (1 + 1/(1 - \lambda))$ for successful searches.

Quadratic probing

- Quadratic probing: a collision resolution method that eliminates the primary clustering problem of linear probing.
- Collision function is quadratic. Popular choice is $f(i) = i^2$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Probing properties

- For linear probing, it is a bad idea to let the hash table get nearly full, because performance degrades.
- For quadratic probing, the situation is even more drastic: There is no guarantee of finding an empty cell once
  – the table gets more than half full, or
  – even before the table gets half full if the table size is not prime.
- This is because at most half of the table can be used as alternative locations to resolve collisions.
- **Theorem 5.1:** If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Code for hash tables using probing strategies

Double Hashing

- For double hashing, one popular choice is $f(i) = i \cdot hash2(x)$.
- This formula says that we apply a second hash function to x and probe at a distance $hash2(x)$, $2hash2(x)$, . . . , and so on.
- A poor choice of $hash2(x)$ would be disastrous.
- For instance, the obvious choice $hash2(x) = x \bmod 9$ would not help if 99 were inserted into the input in the previous examples.
- Thus, the function must never evaluate to zero.
- It is also important to make sure all cells can be probed

- A function as $hash2(x) = R - (x \bmod R)$, with R a prime smaller than $TableSize$, will work well.
- Below: same previous example with $R = 7$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

- Important reminder: Size of table should be prime!
- Size = 10 in example was for convenience of mod 10.

Rehashing

If the hash table is close to full, then running time for the operations will start taking too long, and insertions might fail if separate chaining with quadratic probing

➔ a hash table of bigger size (~ twice as big) is used with a new hash function

➔ compute the new hash value for each element of the original table and insert it in the new table.

The old hash table is subsequently deleted.

This operation is called **Rehashing**.

It Should be done infrequently.

Rehashing example

Insert 13, 15, 24, 6 in hash table of size 7 with h(x) = x mod 7 (with linear probing)

6
15
23
24
13

If 23 is inserted (linear probing), then the table > 70% full

➔New table created; 17 is next prime number about twice as large as 7

➔New hash function h(x) = x mod 17

Exercise: You can easily check the new hash table with these data elements.

When to rehash?

- Rehashing can be implemented in several ways with quadratic probing
- Rehash as soon as the table is half full.
- The other extreme is to rehash only when an insertion fails (even with probing).
- A third, middle-of-the-road strategy is to rehash when the table reaches a certain load factor.
- Since performance does degrade as the load factor increases, the third strategy, implemented with a good cutoff, could be best.

Implementation of rehashing for quadratic probing

See in textbook rehashing for separate chaining hash table.

Hash tables with worst-case O(1) Access

- Hash tables examined so far all have the property that with reasonable load factors, and appropriate hash functions, we can expect *O* (1) cost on average for insertions, removes, and searching
- In fact, expected worst case for a search assuming a reasonably well-behaved hash function:
  - for $\lambda = 1$, classical **balls and bins problem**: Given *N* balls placed randomly (uniformly) in *N* bins, what is the expected number of balls in the most occupied bin?
  - **Answer:** Θ(logN/ log logN), i.e. on average, we expect some queries to take ~logarithmic time.
- BUT, we would like to obtain *O*(1) worst-case cost

Perfect Hashing

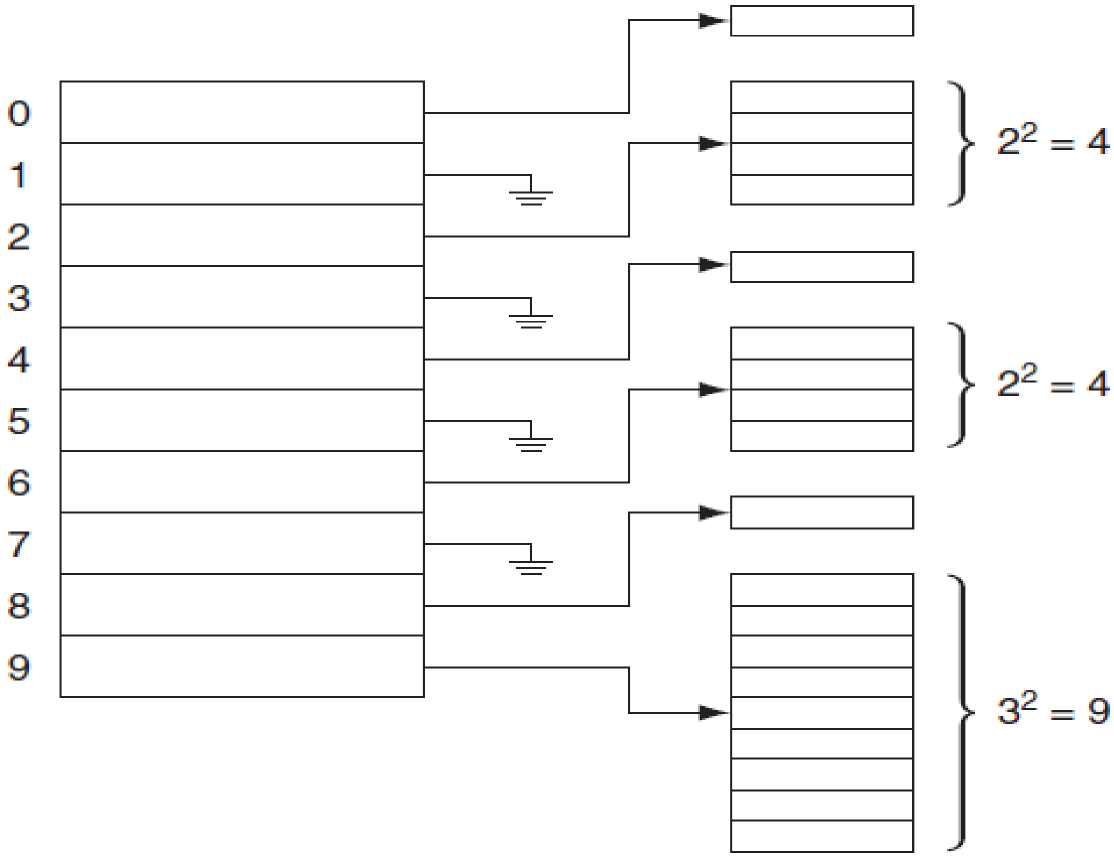
- If a separate chaining implementation guarantees each list has at most a constant number of items, we would be done.
- We know: more lists ➔ the lists will on average be shorter

➔ theoretically if we have enough lists, then with a reasonably high probability we might expect to have no collisions at all

- Suppose number of lists (*TableSize*) *M* is sufficiently large to guarantee that with probability at least 0.5 , there will be no collisions.
- If collision, then clear out table & try another hash function; if collision, try a third function; etc.
- Expected number of trials will be at most 2 (since the success probability is 0.5)

Perfect Hashing

- Idea: Use only N bins, but resolve the collisions in each bin by using hash tables instead of linked lists
- Because the bins are expected to have only a few items each, the hash table that is used for each bin can be quadratic in the bin size
- Each secondary hash table will be constructed using a different hash function until it is collision free.
- The primary hash table can also be constructed several times if the number of produced collisions is higher than required.
- It is proved that the total size of the secondary hash tables is linear (at most $2N$)



Primary hash table has 10 bins. Bins 1, 3, 5, and 7 are all empty.
Bins 0, 4, and 8 have one item ➔ resolved by a secondary hash table with one position.
Bins 2 and 6 have two items, so they will be resolved into a secondary hash table with four (2^2) positions.
And bin 9 has three items, so it is resolved into a secondary hash table with nine (3^2) positions.

Cuckoo Hashing

- Two tables are maintained, each more than half empty, along with two independent hash functions that can assign each item to a position in each table
- Each key is hashed by the 2 hash functions; the obtained cell in first table is used
- If an item already exists in that cell in Table 1, the existing item is displaced into Table 2 using its second hash value
- Example: A dataset of 6 items; 2 hash tables of size 5 each (just to illustrate)

Insert A (0,2), then B (0,0), then C (1,4), then D (1,0), then E (3,2), then F (3,4)

Table 1		Table 2	
0	A	0	
1		1	
2		2	
3		3	
4		4	

Table 1		Table 2	
0	B	0	
1		1	
2		2	A
3		3	
4		4	

Table 1		Table 2	
0	B	0	
1	C	1	
2		2	A
3		3	
4		4	

Table 1		Table 2	
0	B	0	
1	D	1	
2		2	A
3	E	3	
4		4	C

Table 1		Table 2	
0	B	0	
1	D	1	
2		2	A
3	F	3	
4		4	C

Table 1		Table 2	
0	B	0	
1	D	1	
2		2	E
3	F	3	
4		4	C

Table 1		Table 2	
0	A	0	
1	D	1	
2		2	E
3	F	3	
4		4	C

Table 1		Table 2	
0	A	0	B
1	D	1	
2		2	E
3	F	3	
4		4	C

Insert G (1,2) leads to a cycle of displacements D, B, A, E, F, C, G

Cuckoo Hashing

- Probability that a single insertion would require a new set of hash functions can be made to be $O(1/N^2)$
- The new hash functions themselves generate N more insertions to rebuild the table: this means the rebuilding cost is minimal.
- However, if the table load factor is at 0.5 or higher, then probability of a cycle becomes drastically higher, and this scheme is unlikely to work well

Extensions of Cuckoo Hashing

- Instead of two tables, 3 or 4 (or more) tables can be used
  - increases the cost of a lookup, and drastically increases the theoretical space utilization.
- In some applications lookups through separate hash functions can be done in parallel and thus cost little to no additional time.
- Allow each table to store multiple keys; this can increase space utilization and make it easier to do insertions
- Some variations attempt to place an item in the second hash table immediately if there is an available spot, rather than starting a sequence of displacements

Hopscotch Hashing

- An algorithm that tries to improve on the classic linear probing algorithm
- <u>Idea</u>: bound the maximal length of the probe sequence by a predetermined constant (*MAX_DIST)* that is optimized to the underlying architecture of the computer.

➔ item x must be found somewhere in the *MAX_DIST* positions listed in $hash(x)$, $hash(x) + 1, . . . , hash(x) + (MAX\_DIST - 1)$.

- <u>Goal</u>: give constant-time lookups in the worst case (which could be parallelized to simultaneously check the bounded set of possible locations)

- If an insertion would place a new item too far from its hash location, then efficiently go backward toward the hash location, evicting potential items.
- With proper care, the evictions can be done quickly and guarantee that those evicted are not placed too far from their hash locations.
- The algorithm is deterministic: given a hash function, either the items can be evicted or they can’t.
- The latter case implies that the table is likely too crowded, and a rehash is in order.
- Rehash would happen only at very high load factors, > 0.9
- For a table with a load factor of <= 0.5, the failure probability is almost zero.

Hopscotch hashing table

- A column of hop bit arrays is added to the hash table. It indicates where conflicting items have been repositioned. (e.g. *MAX_DIST=4)*
- Maintain information that tells for each position *x*, whether the item in the alternate position is occupied by an element that hashes to position *x*

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11	G	1000
12	F	1000
13		0000
14		0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11

- If Hop[*pos*] contains all 1s for some *pos*, then an attempt to insert an item whose hash value is *pos* will fail
- Insert item H with hash value 9
- Normal linear probing would try to place it in position 13, but that is too far from the hash value of 9.
- Instead, we look to evict an item and relocate it to position 13.
- The only candidates to go into position 13 would be items with hash value of 10, 11, 12, or 13.
- If we examine Hop[10], we see that there are no candidates with hash value 10.
- But Hop[11] produces a candidate, *G*, with value 11 that can be placed into position 13. Since position 11 is now close enough to the hash value of *H*, we can now insert *H*.

Insert item H with hash value 9

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11	G	1000
12	F	1000
13		0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1000
10	E	0000
11		**0010**
12	F	1000
13	G	0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	**1010**
10	E	0000
11	**H**	0010
12	F	1000
13	G	0000
14		0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9

Insert item I with hash value 6. Linear probing suggests position 14; too far away! Rather check 11, 12, 13

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0010
12	F	1000
13	G	0000
14		0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	**0001**
12	F	1000
13		0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12		**0100**
13	**F**	0000
14	G	0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9
I: 6

Hop[11] tells evict G (moved down to 14). Go backward. 13 freed ➔ see Hop[10]: all zeros; no eviction. we examine Hop[11] : all zeros in the first two positions. Try Hop|12]. First position is 1 ➔ move F down

Otherwise we could have had to rehash!

The remaining eviction is from position 9: B goes down to 12; and subsequent placement of *I*.

	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9	B	1010
10	E	0000
11	H	0001
12		0100
13	F	0000
14	G	0000
...		

→

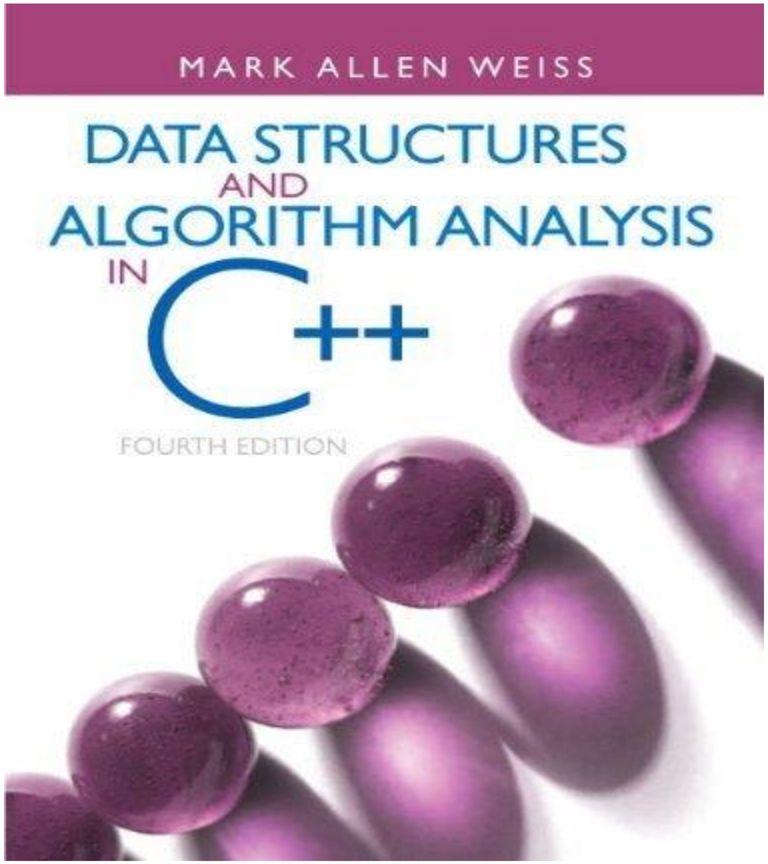
	Item	Hop
...		
6	C	1000
7	A	1100
8	D	0010
9		**0011**
10	E	0000
11	H	0001
12	**B**	0100
13	F	0000
14	G	0000
...		

→

	Item	Hop
...		
6	C	**1001**
7	A	1100
8	D	0010
9	**I**	0011
10	E	0000
11	H	0001
12	B	0100
13	F	0000
14	G	0000
...		

A: 7
B: 9
C: 6
D: 7
E: 8
F: 12
G: 11
H: 9
I: 6

Slides based on the textbook



Mark Allen Weiss, (2014) Data Structures and Algorithm Analysis in C++, 4th edition, Pearson.

Acknowledgement: This **course PowerPoints** make substantial (non-exclusive) use of the PPT chapters prepared by Prof. Saswati Sarkar from the University of Pennsylvania, USA, themselves developed on the basis of the course textbook. Other references, if any, will be mentioned wherever applicable.