ZAMAN UNIVERSITY

Data Structures and Algorithms

Chapter 4

Hash Tables

Deletion

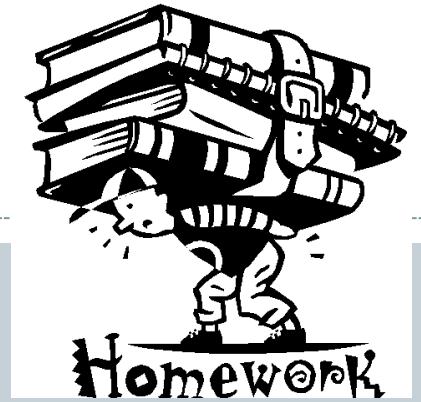
- When deleting records from a hash table, there are two important considerations:
  - Deleting a record must not hinder later searches – thus, the search process must still pass through the newly emptied slot to reach records whose probe sequence passed through this slot. The delete process cannot simply mark the slot as empty, because this will isolate records further down the probe sequence;
  - We do not want to make positions in the hash table unusable because of deletion. The freed slot should be available to a future insertion.
- Both of these problems can be resolved by placing a special mark in place of the deleted record, called a **tombstone**.

Deletion: tombstone

- The **tombstone** indicates that a record once occupied the slot but does so no longer
- If a **tombstone** is encountered when *searching* through a probe sequence, the search procedure is to continue with the search
- When a **tombstone** is encountered during *insertion*, that slot can be used to store the new record
- However, to *avoid inserting duplicate keys*, it will still be necessary for the search procedure to follow the probe sequence until a truly empty position has been found. And the new record would actually be inserted into the slot of the first **tombstone** encountered.

Deletion: Solutions

- Two possible solutions for deletions:
  - Do a local reorganization upon deletion to try to shorten the average path length. For example, after deleting a key, continue to follow the probe sequence of that key and swap records further down the probe sequence into the slot of the recently deleted record
  - Periodically rehash the table by reinserting all records into a new hash table. Not only will this remove the tombstones, but it also provides an opportunity to place the most frequently accessed records into their home positions



Create deletion functions (for linear and quadratic probing) based on idea of Local Reorganization

Outline

- Hash Tables
- Quadratic Probing
- Separate Chaining
- When to Use What

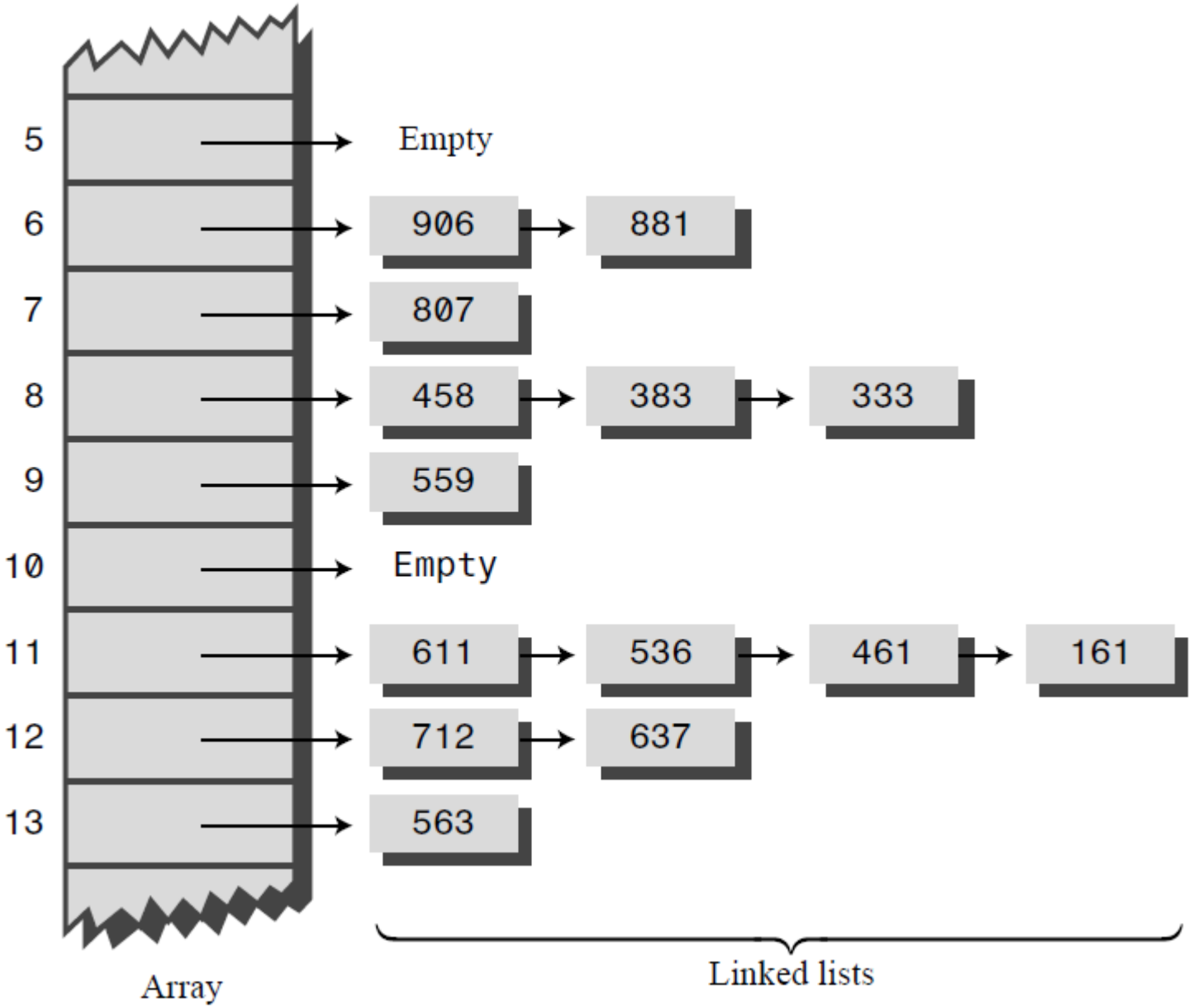
Outline

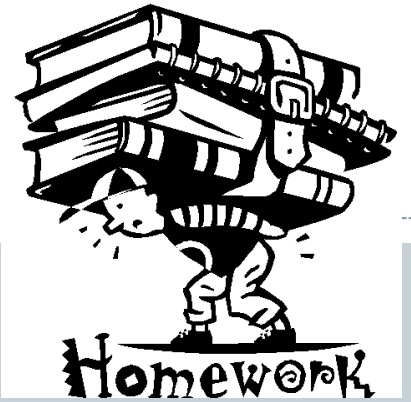
- Hash Tables
- Quadratic Probing
- Separate Chaining
- When to Use What

Separate Chaining

- Collisions are resolved by looking for an open cell in the hash table
- In *separate chaining* a linked list is installed at each index in the hash table;
- A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index;
- Other items that hash to the same index (collisions) are simply inserted in the same linked list;
- there's no need to search for empty cells in the primary array.

Example of Separate Chaining



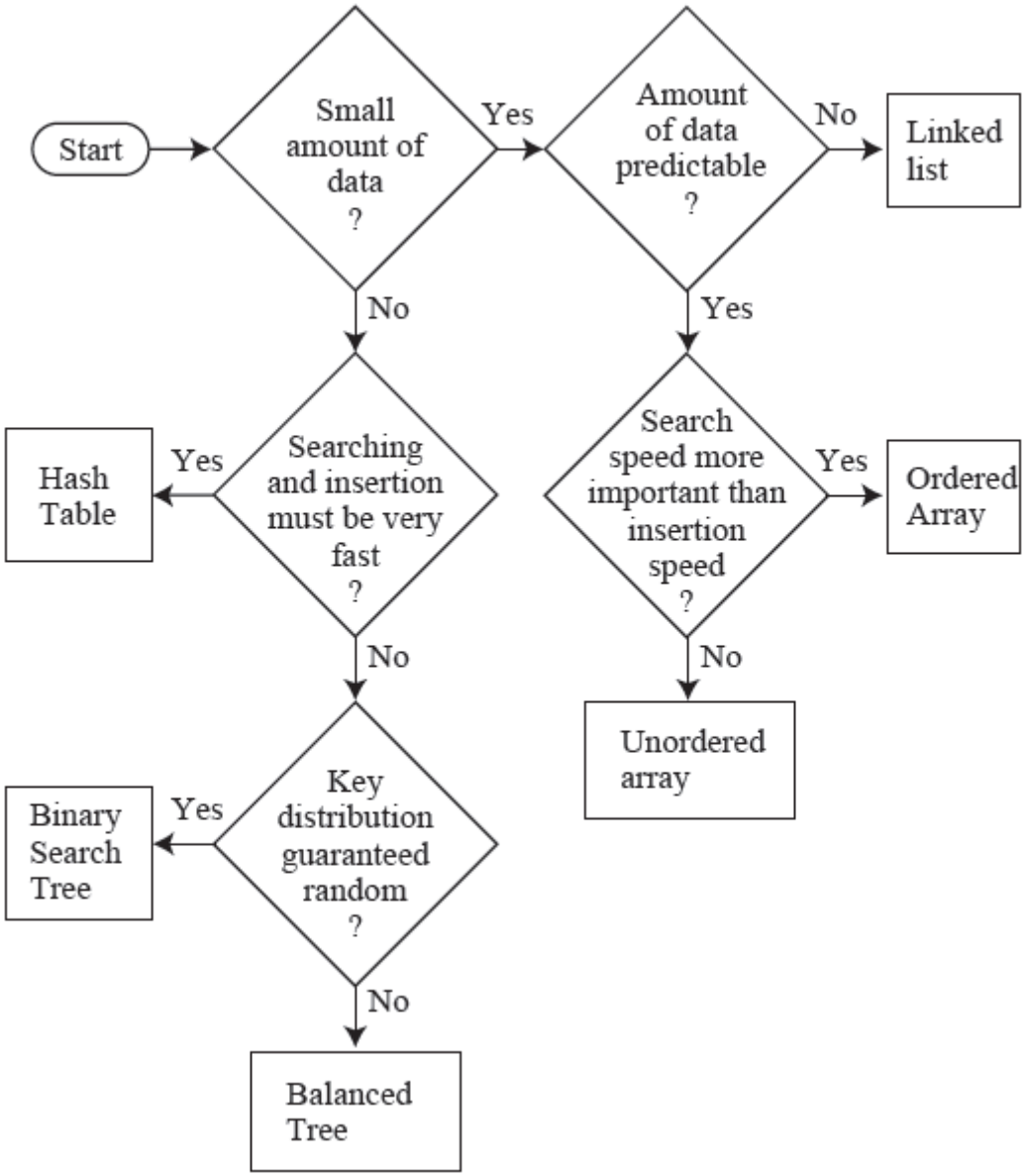


1. Create a Hash Table with size 20
2. Words are converted to number by Multiply Powers;
3. Solve collision problem with **Separate Chaining**

Outline

- Hash Tables
- Quadratic Probing
- Separate Chaining
- When to Use What

General-Purpose Data Structures



Comparing Storage Data Structures

Data Structure	Search	Insertion	Deletion
Array	O(N)	O(N)	O(N)
Ordered Array	O(log N)	O(N)	O(N)
Linked List	O(N)	O(1)	O(N)
Order Linked List	O(N)	O(N)	O(N)
Stack	-	O(1)	O(1)
Queue	-	O(1)	O(1)
Priority Queue	-	O(N)	O(1)
Hash Table	O(1)	O(1)	O(1)

Comparison of Sorting Algorithms

Sort	Average	Worst	Extra Memory
Bubble	$O(N^2)$	$O(N^2)$	No
Insertion	$O(N^2)$	$O(N^2)$	No
Quicksort	O(N*log N)	$O(N^2)$	No
Mergesort	O(N*log N)	O(N*log N)	Yes

End of Chapter IV – Hash Table