GIUSEPPE TURINI

CS-102 - COMPUTING AND ALGORITHMS 2
LESSON 08 - TABLES AND HASHING

CS-203 - COMPUTING AND ALGORITHMS 3
LESSON 07 - SPACE AND TIME TRADE-OFFS

Kettering UNIVERSITY

DATA ABSTRACTION AND PROBLEM SOLVING WITH JAVA (3RD EDITION), **CHAP 12**, AND **CHAP 13, 643-775**.
INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS (3RD EDITION), **CHAP 7, 253-282**.

HIGHLIGHTS

The ADT Table

- A Sorted Array Implementation, and a Binary Search Tree Implementation
- Tables in the JCF

Hashing

- Hash Functions, Collisions, Efficiency, JCF Hashtable, and JCF TreeMap

Data with Multiple Organizations

- External and Internal Storage

Space and Time Trade-Offs

- Lookup Tables
- Efficiency of Hashing
- B-Trees

STUDY GUIDE

Study Material

- This slides.
- “Data Abstraction and Problem Solving with Java (3rd Ed.)”, chap. 12, pp. 643-775.

Selected Exercises

- Exercises: 12.11-12.13, 12.15, 12.17, 12.19-12.20, 12-22.

Additional Resources

- “Object-Oriented Data Structures Using Java”, chap. 9, pp. 551-582.
- “Object-Oriented Data Structures Using Java”, chap. 8, pp. 516-538.
- visualgo.net/en/hashtable

THE ADT TABLE 1

The ADT **table** (aka **dictionary**) is another value-oriented ADT:

- uses a **search key** to identify its items;
- its items are **records** storing several data fields.

Figure: An ordinary table of cities.

City	Country	Population
Athens	Greece	2,500,000
Barcelona	Spain	1,800,000
Cairo	Egypt	9,500,000
London	England	9,400,000
New York	U.S.A.	7,300,000
Paris	France	2,200,000
Rome	Italy	2,800,000
Toronto	Canada	3,200,000
Venice	Italy	300,000

THE ADT TABLE 2

Operations of the ADT table:

- **create** an empty table;
- determine whether a table **is empty**;
- determine the **number of items** in a table;
- **insert** a new item into a table;
- **delete** the item with a given **search key** from a table;
- **retrieve** the item with a given **search key** from a table;
- **traverse** the items in a table in **sorted search-key order**.

Note: We will assume that all table items have distinct search keys. So, the insertion operation must reject an item whose search key is already in the table.

Note: In many applications, we may expect duplicate search keys. If so, we must redefine some operations to solve the ambiguity arising from duplicate search keys.

THE ADT TABLE 3

Pseudocode for the operations of the ADT table:

```
createTable(); // Creates an empty table.
tableIsEmpty(); // Determines whether a table is empty.
tableLength(); // Determines the number of items in a table.
tableTraverse(); // Traverses a table in sorted search-key order.

// Inserts newItem into a table whose items have distinct search keys that differ
// from newItem search key. Throws TableException if the insertion is not successful.
tableInsert( newItem ) throws TableException;

// Deletes from a table the item whose search key equals searchKey.
// Returns false if no such item exists. Returns true if the deletion was successful.
tableDelete( searchKey );

// Returns the item in a table whose search key equals searchKey.
// Returns null if no such item exists.
tableRetrieve( searchKey );
```

THE ADT TABLE 4

Properties of the ADT table:

- a **search key must remain the same** as long as its item is stored in the table:
  - the **KeyedItem** class stores a search key and only its accessor (read-only) to read its value (preventing any change to the search-key once it is created);
- the **TableInterface** interface defines the table operations.

Note: Both **KeyedItem** and **TableInterface** are generic, and they use:

- **bounded type parameters:** e.g. upper bounded, as **< T1 extends T2 >**), meaning that the generic class accepts only a **T1** class derived from **T2**, and
- **bounded wildcards:** e.g. lower bounded, as **<? super T >**), meaning that the generic class accepts any unknown type parameter that is a super class of **T**.

See: docs.oracle.com/javase/tutorial/java/generics/bounded
See: docs.oracle.com/javase/tutorial/java/generics/wildcards

THE ADT TABLE 5

KEYED ITEM

```
package SearchKeys;
import java.lang.Comparable;

// Class to store a search-key (comparable) providing only an accessor (and no modifier).
public abstract class KeyedItem< KT extends Comparable <? super KT > > {
   private KT searchKey;
   public KeyedItem( KT searchKey ) { this.searchKey = searchKey; }
   public KT getKey() { return searchKey; }
}
```

Note: Classes extending **KeyedItem** will have only the constructor to initialize the search key, so once created, the search key cannot be modified (immutable).

See: docs.oracle.com/javase/tutorial/java/generics/bounded
See: docs.oracle.com/javase/tutorial/java/generics/wildcards

THE ADT TABLE 6

TABLE INTERFACE A

```
package Tables;
import SearchKeys.KeyedItem;

// Interface for the ADT table.
// Note: no two items of the table have the same search key.
// Note: the table items are sorted by search key.
public interface TableInterface< T extends KeyedItem< KT >,
                                 KT extends Comparable <? super KT > > {

   public boolean tableIsEmpty(); // Returns true if the table is empty, false otherwise.
   public int tableLength(); // Returns the number of items in the table.

   // Inserts an item into a table in sorted order according to the item search key.
   // Note: if search key is already stored in the table, a TableException is thrown.
   public void tableInsert( T newItem ) throws TableException;
```

THE ADT TABLE 7

TABLE INTERFACE B

```
   // Deletes an item with a given search key from table.
   // It returns true if item exists, otherwise returns false.
   public boolean tableDelete( KT searchKey );

   // Retrieves an item with a given search key from table, if not found returns null.
   public T tableRetrieve( KT searchKey );

}
```

TABLE EXCEPTION

```
package Tables;
import java.lang.RuntimeException; import java.lang.String;

public class TableException extends RuntimeException {
   public TableException( String s ) { super(s); } }
```

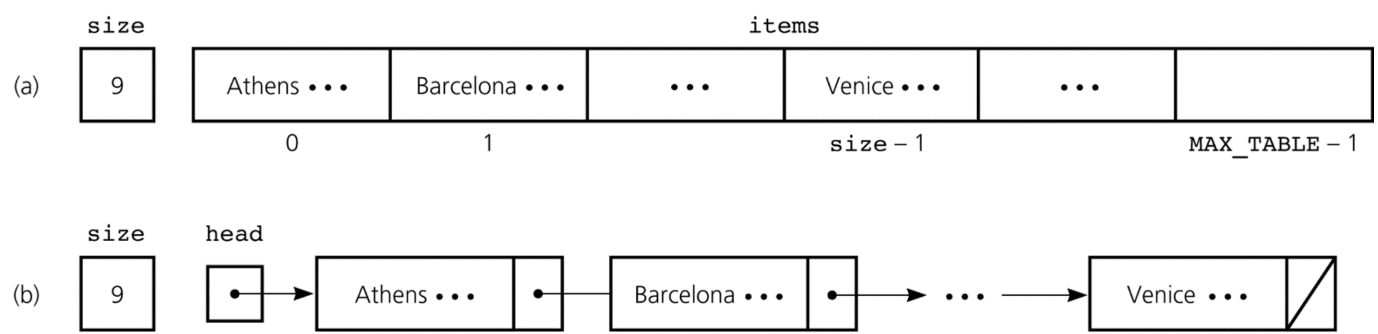
THE ADT TABLE 8

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE A

Categories of **linear implementations** (i.e. array-based or linked-list-based):

- unsorted, array based;
- unsorted, referenced based;
- sorted (by search key), array based;
- sorted (by search key), reference based.

Figure: Array-based **(a)**, and reference-based **(b)** implementations of the ADT table.



THE ADT TABLE 9

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE B

Categories of **non-linear implementations**:

- binary search tree (BST) implementation.

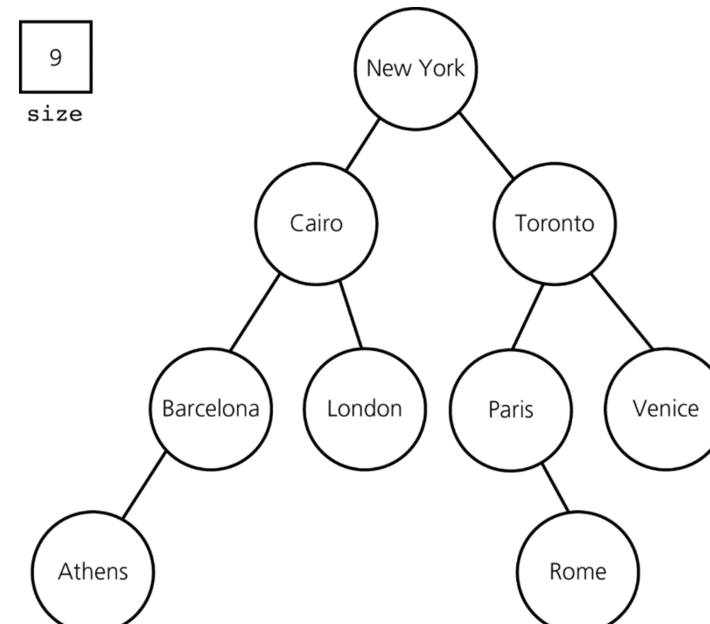


Figure: A BST implementation of the ADT table.

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE C

The BST implementation offers several advantages over linear implementations.

The requirements of a particular application influence the selection of an implementation.

Questions to be considered **to choose an implementation** for the ADT table:

- **What operations are needed in our application?**
- **How frequently is each operation performed in our application?**

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE D

Example (scenario A): Insertion and traversal in no particular order.

An unsorted order is efficient
(both array-based and reference-based **tableInsert** are **O(1)** (constant time)).

Array-based versus reference-based:

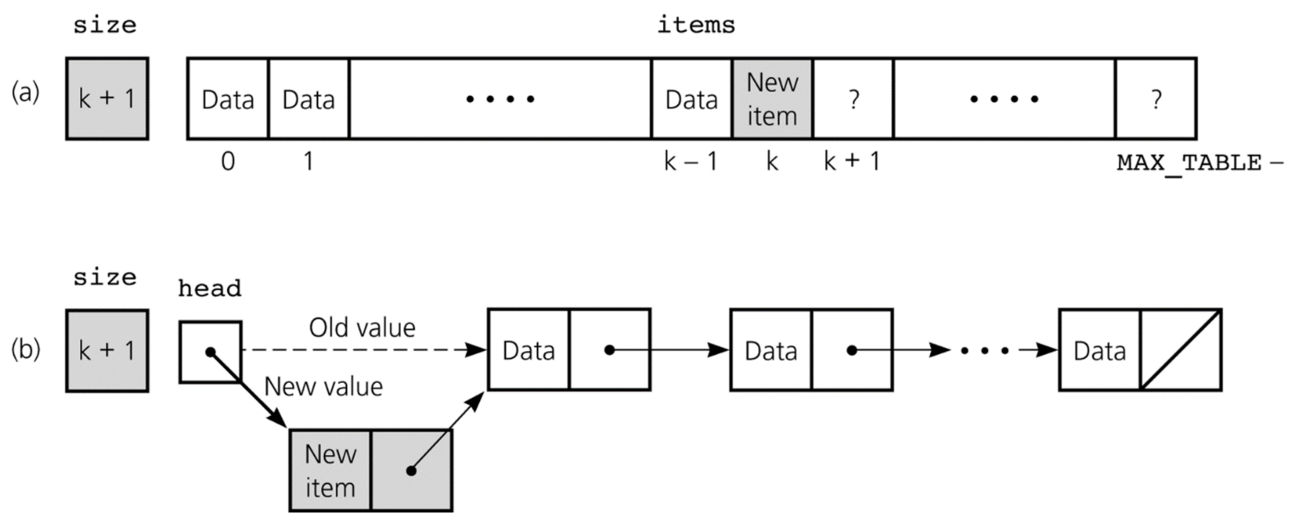
- if a good estimate of the max size of the table is not available,
  - a reference-based implementation is preferred;
- if a good estimate of the max size of the table is available,
  - the choice is mostly a matter of style (e.g. array-based and reference-based implementations offer the similar advantages).

THE ADT TABLE 12

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE E

Example (scenario A, continued): Insertion and traversal in no particular order.

Figure: Insert in unsorted-linear tables: array-based **(a)**, and reference-based **(b)**.



THE ADT TABLE 13

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE F

Example (scenario A, continued): Insertion and traversal in no particular order.

A BST implementation is not appropriate:

- it does more work than the application requires (i.e. it orders the table items);
- the insertion operation is slower (**O(log n)**) in the average case (in respect to **O(1)** for linear implementations).

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE G

Example (scenario B): Retrieval.

Item retrieval via binary search:

- in an array-based table, binary search can be used only if the array is sorted;
- in a reference-based table, binary search is too inefficient to be practical;
- binary search in an array is faster than sequential search in a linked list:
    - **binary search in an array** (in the worst case is $\mathbf{O(\log_2 n)}$);
    - **sequential search in a linked list** (in the worst case is **O(n)**);

In a scenario with frequent retrievals:

- if table max size is known, a sorted array-based table is appropriate;
- if table max size is not known, a BST table is appropriate.

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE H

Example (scenario C): Insert/remove/retrieve/traverse in sorted order.

Steps performed by both insertion and removal in sorted linear tables:

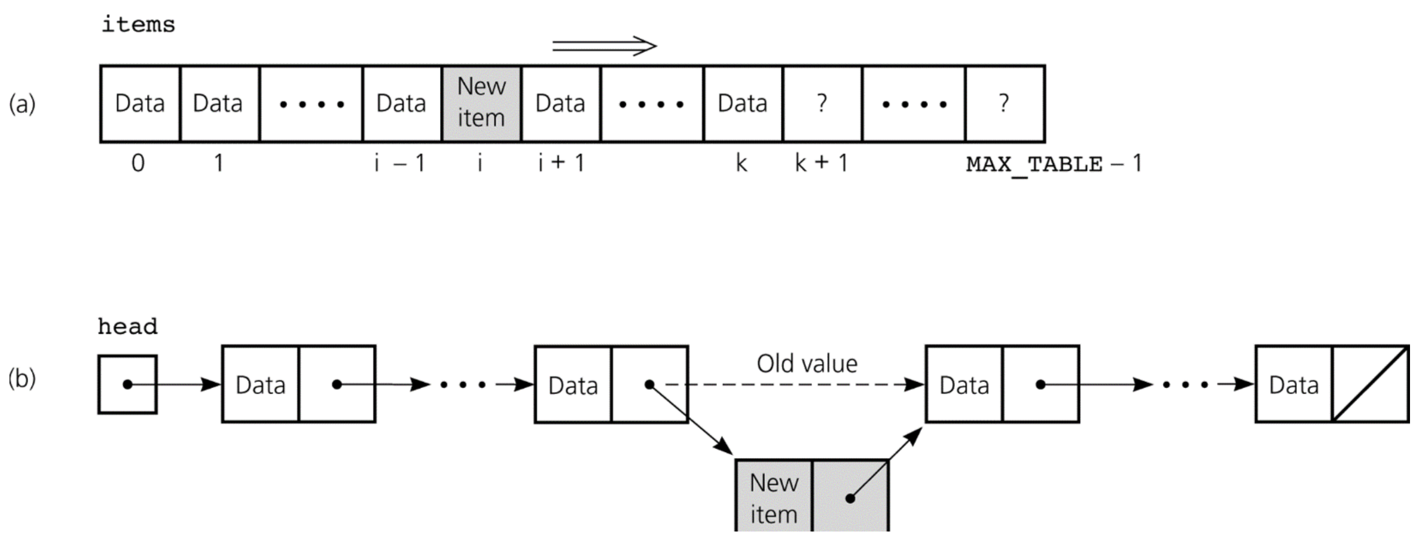
- **step 1:** find the appropriate position in the table;
  - for this step, an array-based table is superior than a reference-based table;
- **step 2:** insert into (or remove from) that position;
  - for this step, a reference-based table is superior than an array-based table (since in sorted array-based tables we need to shift data to insert or remove).

THE ADT TABLE 16

SELECTING AN IMPLEMENTATION FOR THE ADT TABLE I

Example (scenario C, continued): Insert/remove/retrieve/traverse in sorted order.

Figure: Insertion in sorted linear tables: array-based **(a)**, and reference-based **(b)**.



SELECTING AN IMPLEMENTATION FOR THE ADT TABLE J

Example (scenario C, continued): Insert/remove/retrieve/traverse in sorted order.

Insertion and removal operations in sorted linear tables:

- both sorted linear implementations (i.e. array-based or reference-based) are comparable, but neither is suitable;
  - in sorted array-based tables, **tableInsert** and **tableDelete** are **O(n)**;
  - in sorted reference-based tables, **tableInsert** and **tableDelete** are **O(n)**;
- a **binary search tree table is suitable** in this scenario, since it combines the best features of the two linear implementations above.

THE ADT TABLE 18

- **Linear implementations:** useful for many applications but with issues.
- **Binary search tree implementations:** better than linear implementations.
- **Balanced binary search tree implementations:** better efficiency of table.

Figure: Average-case efficiency of ADT table operations in different implementations.

	Insertion	Deletion	Retrieval	Traversal
Unsorted array based	O(1)	O(n)	O(n)	O(n)
Unsorted pointer based	O(1)	O(n)	O(n)	O(n)
Sorted array based	O(n)	O(n)	O(log n)	O(n)
Sorted pointer based	O(n)	O(n)	O(n)	O(n)
Binary search tree	O(log n)	O(log n)	O(log n)	O(n)

SORTED ARRAY-BASED TABLE 1

Reasons for studying linear implementations of tables:

- **perspective:** if the size of the problem is small, the difference in efficiency among the different implementation is insignificant;
- **efficiency:** a linear implementation is efficient in some cases (e.g. scenario A);
- **motivation:** analyzing scenarios where linear implementations are not adequate, we are forced to look at other solutions (e.g. binary search trees).

The TableArrayBased class:

- provides an array-based implementation of the ADT table;
- implements **TableInterface**.

The TableBSTBased class:

- represents a **non-linear reference-based** implementation of the ADT table;
- uses a BST to store items in the ADT table, reusing the **BinarySearchTree** class.

SORTED ARRAY-BASED TABLE 2

TABLE ARRAY BASED A

```
package Tables;
import SearchKeys.KeyedItem;
import java.util.ArrayList;

// Sorted array-based implementation of the ADT table.
// Note: table contains at most one item with a given search key at any time.
public class TableArrayBased< T extends KeyedItem< KT >,
                              KT extends Comparable<? super KT > >
                              implements TableInterface< T, KT > {

   final int MAX_TABLE = 100; // Max size of table.
   protected ArrayList<T> items; // Table.

   // Constructor (default).
   public TableArrayBased() { items = new ArrayList<T>( MAX_TABLE ); }
```

SORTED ARRAY-BASED TABLE 3

TABLE ARRAY BASED B

```
public boolean tableIsEmpty() { return tableLength() == 0; }
public int tableLength() { return items.size(); }

public void tableInsert( T newItem ) throws TableException {
   if( tableLength() < MAX_TABLE ) {
      // There is room to insert, locate the position where newItem belongs.
      int spot = position( newItem.getKey() ); // See function "position".
      if( ( spot < tableLength() ) &&
          ( items.get( spot ).getKey() ).compareTo( newItem.getKey() ) == 0 ) {
         // We have found a duplicate key!
         throw new TableException( "Insert failed, duplicate key!" ); }
      else {
         // ArrayList automatically shifts items to make room for the new item.
         items.add( spot, newItem ); } }
   else { throw new TableException( "Table full!" ); }
}
```

SORTED ARRAY-BASED TABLE 4

TABLE ARRAY BASED C

```
public boolean tableDelete( KT searchKey ) {
   int spot = position( searchKey ); // Locate searchKey, see function "position".
   // Is searchKey present in the table?
   boolean success = ( spot <= tableLength() ) &&
                     ( items.get( spot ).getKey().compareTo( searchKey ) == 0 );
   if( success ) { items.remove( spot ); } // ArrayList automatically shifts items.
   return success;
}

public T tableRetrieve( KT searchKey ) {
   int spot = position( searchKey ); // Locate searchKey, see function "position".
   // Is searchKey present in table?
   boolean success = ( spot < tableLength() ) &&
                     ( items.get( spot ).getKey().compareTo( searchKey ) == 0 );
   if( success ) { return items.get( spot ); } // Item present, retrieve it.
   else { return null; }
}
```

SORTED ARRAY-BASED TABLE 5

TABLE ARRAY BASED D

```
    // Finds the position of a table item or its insertion.
    // Note: returns the index [0, size-1] where the search key is stored, otherwise,
    //       if search key not found, returns position [0, size] search key should occupy.
    protected int position( KT searchKey ) {
        int pos = 0;
        while( ( pos < tableLength() ) &&
               ( searchKey.compareTo( items.get( pos ).getKey() ) > 0 ) ) {
            pos++; }
        return pos;
    }

}
```

BST-BASED TABLE 1

TABLE BST BASED A

```
package Tables;

import BinaryTrees.BinarySearchTree;
import BinaryTrees.TreeException;
import SearchKeys.KeyedItem;

// Binary search tree based implementation of the ADT table.
// Note: the table contains at most one item with a given search key at any time.
public class TableBSTBased< T extends KeyedItem< KT >,
                            KT extends Comparable<? super KT > >
                            implements TableInterface< T, KT > {

   protected BinarySearchTree< T, KT > bst; // Binary search tree storing the table.
   protected int size; // Number of items in the table.
```

BST-BASED TABLE 2

TABLE BST BASED B

```java
// Constructor (default).
public TableBSTBased() {
   bst = new BinarySearchTree< T, KT >();
   size = 0;
}

public boolean tableIsEmpty() { return size == 0; }

public int tableLength() { return size; }

public void tableInsert( T newItem ) throws TableException {
   if( bst.retrieve( newItem.getKey() ) == null ) {
      bst.insert( newItem );
      ++size; }
   else { throw new TableException( "Insertion failed, duplicate key item!" ); }
}
```

BST-BASED TABLE 3

TABLE BST BASED C

```
   public T tableRetrieve( KT searchKey ) { return bst.retrieve( searchKey ); }

   public boolean tableDelete( KT searchKey ) {
      try { bst.delete( searchKey ); }
      catch( TreeException e ) { return false; }
      --size;
      return true;
   }

   protected void setSize( int newSize ) { size = newSize; }

}
```

TABLES IN JCF 1

The JCF **Map** interface provides the basis for numerous other implementations of different kinds of maps (classes for key-value objects with unique mappings):

```
// JCF Map interface (partial view).
public interface Map< K, V > {
   void clear(); // Removes all of the mappings from this map (optional operation).
   boolean containsKey( Object key ); // Returns true if map contains the specified key.
   boolean containsValue( Object value ); // Checks if map maps keys to specified value.
   Set< Map.Entry< K, V > > entrySet(); // Returns a Set of mappings stored in this map.
   V get( Object key ); // Returns value to which the specified key is mapped, or null.
   boolean isEmpty(); // Returns true if this map contains no key-value mappings.
   Set<K> keySet(); // Returns a Set view of the keys contained in this map.
   V put( K key, V value ); // Maps input value with input key in this map (optional op).
   V remove( Object key ); // Removes mapping for input key from this map (optional op).
   Collection<V> values(); // Returns a Collection of values stored in this map.
}
```

See: docs.oracle.com/javase/8/docs/api/java/util/map

TABLES IN JCF 2

The JCF **Set** interface is an ordered collection, but only stores single value entries and does not allow duplicates (while a **Collection** does allow duplicates):

```
// JCF Set interface (partial view).
public interface Set<T> {
   boolean add( T o ); // Adds input item to set if not already present (optional op).
   boolean addAll( Collection<? extends T > c ); // Adds collection to set (optional op).
   void clear(); // Removes all of the elements from this set (optional operation).
   boolean contains( Object o ); // Returns true if set contains input item.
   boolean isEmpty(); // Returns true if this set contains no elements.
   Iterator<T> iterator(); // Returns an iterator over the elements in this set.
   boolean remove( Object o ); // Removes input item from set if present (optional op).
   boolean removeAll( Collection<?> c ); // Removes collection from set (optional op).
   boolean retainAll( Collection<?> c ); // Retains only items in set AND collection.
   int size(); // Returns the number of elements in this set (its cardinality).
}
```

See: docs.oracle.com/javase/8/docs/api/java/util/set

HASHING 1

A radically different strategy is necessary to locate (and insert or delete) an item in a table virtually instantaneously.

Imagine an array-table of **n** items (aka **hash table**), with each array slot storing a single item, and a magical function called **"address calculator"** (aka **hash function**).

When a new item has to be inserted into the table, the address calculator (using the item key) determines the index to store it in the table-array.

This technique is called **hashing** (invented in 1950s by IBM), and it provides:

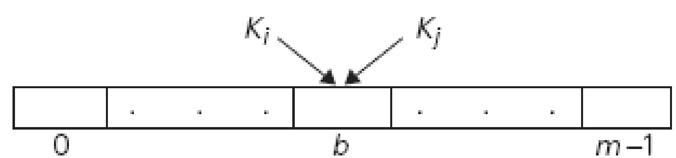
insertions/retrievals/deletes performed in constant time (all in O(1)).

HASHING 2

Hash Function: Maps the search key of a table item into a **location** for the item.

Perfect Hash Function: Maps each search key into a **unique location** (i.e. no collisions) of the hash table. It is only possible if all the search keys are known.

Figure: Collision of 2 different keys $\mathbf{K_i}$ and $\mathbf{K_j}$, mapped by the hash function **h** to the same location $\mathbf{h(K_i) = h(K_j)}$



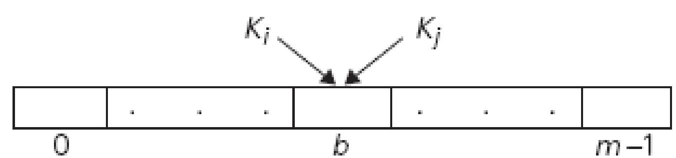
Collision of two keys in hashing: $h(K_i) = h(K_j)$.

HASHING 3

Collision: Occurs when hash function maps **>1 item into the same array location**.

Collision-Resolution Scheme: Assign locations in the hash table to items with different search keys when the items are involved in a collision.

Figure: Collision of 2 different keys $\mathbf{K_i}$ and $\mathbf{K_j}$, mapped by the hash function **h** to the same location $\mathbf{h(K_i) = h(K_j)}$



Collision of two keys in hashing: $h(K_i) = h(K_j)$.

HASHING 4

RESOLVING COLLISIONS A

- **Approach 1 – Closed Hashing / Open Addressing:** a category of collision resolution schemes that probe for an empty (aka open) location in the hash table (the sequence of locations examined is called **probe sequence**).

  - **Linear Probing:** searches hash table sequentially, from the original location specified by the hash function (possible problem: primary clustering).

  - **Quadratic Probing:** searches the hash table starting at the original location specified by the hash function and continues at increments of $\mathbf{1^2}$, $\mathbf{2^2}$, $\mathbf{3^2}$, etc. (possible problem: secondary clustering).

HASHING 5

RESOLVING COLLISIONS B

- **Approach 1 – Closed Hashing / Open Addressing (continued):** a category of collision resolution schemes that probe for an empty (aka open) location in the hash table (the sequence of locations examined is called **probe sequence**).

  - **Double Hashing:** uses **2** hash functions, searches the hash table starting from the location that hash function **A** determines and considers every **n**$^{\mathbf{th}}$ location, where **n** is determined from hash function **B**.

  - **Increasing the size of the hash table:** hash function must be applied to every item in old hash table before an item is placed into new hash table.

HASHING 6

Figure: An example of an hash table using **linear probing** (open addressing). See how the key "SOON" is hashed to an index (11) not-available, triggering the probe sequence that will store the key at a different index (12), also affecting subsequent insertions (for example "PARTED").

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

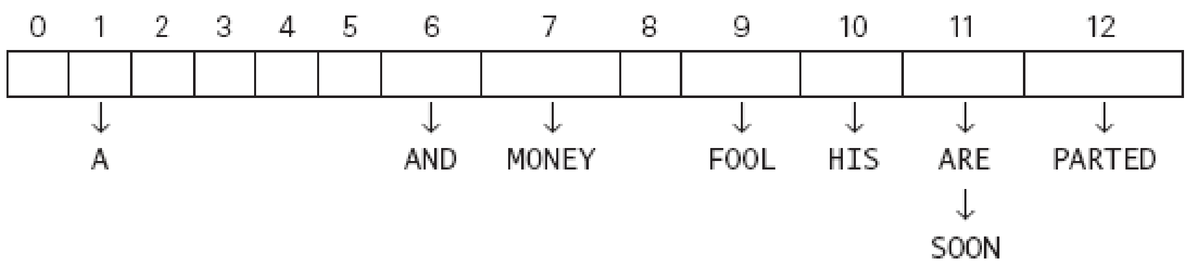
HASHING 7

RESOLVING COLLISIONS C

- **Approach 2 – Open Hashing:** change the table to store >1 item per array cell. Open hashing still works if number of keys is greater than array physical size.

  - **Buckets:** each location in the hash table is itself **an array called a bucket**.
  - **Separate Chaining:** each hash table location is **a linked list**.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12



HASHING 8

WHAT CONSTITUTES A GOOD HASH FUNCTION?

- Work on an **array size comparable to the number of key**.

- **Scatter keys evenly** throughout the hash table.
  - How well does the hash function scatter **random data**?
  - How well does the hash function scatter **non-random data**?

- Be **easy and fast to compute**.
- Calculation should **involve the entire search key**.
- If module arithmetic is used, the **base should be prime**.
- It is sufficient to **operate on integers**.
  - **hash functions on integers:** selecting digits, module arithmetic etc.
  - **converting characters to integers:** char to int before hashing.

HASHING 9

TABLE TRAVERSAL: AN INEFFICIENT OPERATION UNDER HASHING

Hashing as an implementation of the ADT table:

- for many applications, hashing provides the most efficient implementation;
- hashing is **not efficient** for:
  - traversal in sorted order;
  - finding the item with the smallest or largest value in its search key;
  - range queries.

In **external storage** (i.e. reference-based data structure), you can simultaneously use:

- a hashing implementation of the **tableRetrieve** operation;
- a binary search-tree implementation of the ordered operations;

HASHING 10

The JCF **Hashtable** class implements a hash table which maps keys to values:

```
// JCF Hashtable class (partial view).
public class Hashtable< K, V > extends Dictionary< K, V >
                                implements Map< K, V >, Cloneable, Serializable {
   Hashtable(); // Create new empty HT with capacity (11) and load factor 0.75.
   Hashtable( int initialCapacity, float loadFactor ); // ...
   boolean contains( Object value ); // Tests if a key maps into input value in this HT.
   boolean containsKey( Object key ); // Tests if input key is a key in this HT.
   boolean containsValue( Object value ); // Returns true if HT maps a key to this value.
   V get( Object key ); // Returns value to which the specified key is mapped, or null.
   int hashCode(); // Returns the hash code for this HT.
   V put( K key, V value); // Maps input key to input value in this HT.
   protected void rehash(); // Increase HT size and reorganizes it to boost performance.
   V remove( Object key ); // Removes input key (and its value) from this HT.
   boolean replace( K key, V oldVal, V newVal ); // Replaces entry for input key in HT.
}
```

See: docs.oracle.com/javase/8/docs/api/java/util/hashtable

EXTERNAL AND INTERNAL STORAGE 1

In many data structures, complex objects are composed of smaller objects.

These objects are typically stored in one of two ways:

- with **internal storage**, the smaller objects are stored inside the larger object;
- with **external storage**, the smaller objects are allocated in their own location, and the larger object only stores references to them.

See: Wikipedia - Reference (Computer Science)

EXTERNAL AND INTERNAL STORAGE 2

Internal storage is usually more efficient:

- no memory to store references and dynamic allocation metadata;
- no time cost for dereferencing a reference and for memory allocation;
- keeps different parts of the same large object close together in memory.

However, there are situations in which external storage is preferred:

- if **data structure is recursive** (i.e. it may contain itself), it cannot be represented in using internal storage;
- if larger object is stored in an area with limited space, we can prevent running out of storage by storing large component objects in another memory region;
- if smaller objects vary in size, it is often inconvenient to resize the larger object;
- references are often easier to work with and adapt better to new requirements.

See: Wikipedia - Reference (Computer Science)

EFFICIENCY OF HASHING 1

An analysis of the **average-case efficiency** of hashing involves the **load factor**:

- **Load factor α:** ratio between current number of items **n** in the table and max size **$size_{max}$** of the array table; **α should not exceed 2/3.**

$$\alpha = \frac{n}{size_{max}}$$

Hashing efficiency for a particular search also depends on **search success/failure**:

- **Unsuccessful searches usually require more time than successful searches.**

EFFICIENCY OF HASHING 2

- **Linear Probing:**
  - ins/del/search successful: $\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
  - ins/del/search unsuccessful: $\approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

The average number of accesses to the hash table is very small even for dense tables.

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

EFFICIENCY OF HASHING 3

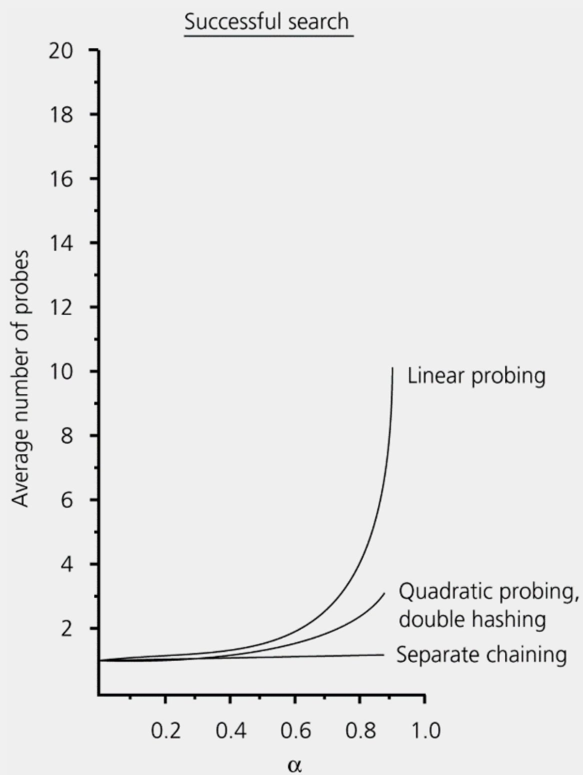
- **Quadratic Probing and Double Hashing:**
  - insertion/retrieval/delete successful (search): $\approx \dfrac{-\log_e(1-\alpha)}{\alpha}$
  - insertion/retrieval/delete unsuccessful (search): $\approx \dfrac{1}{(1-\alpha)}$

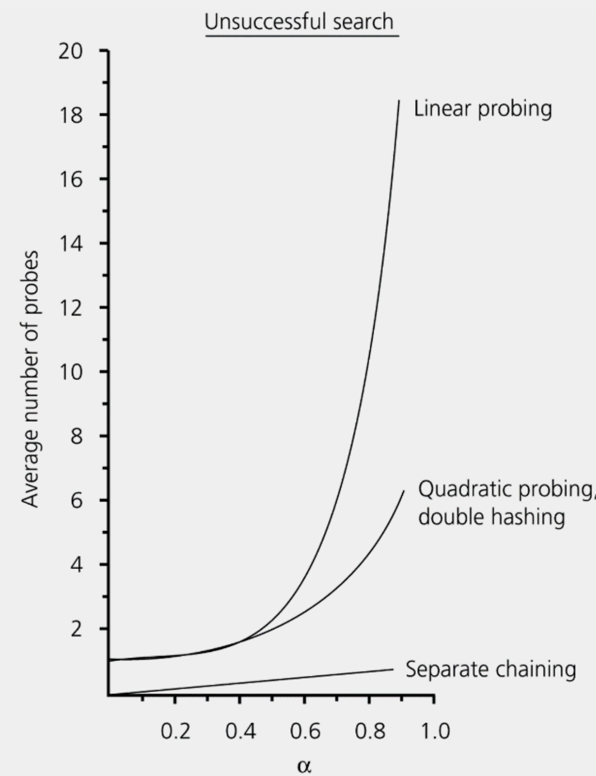
EFFICIENCY OF HASHING 4

- **Separate Chaining:**
  - load factor **α** represents average length of each linked list.
  - insertion: $\in O(1)$
  - retrieval/delete successful (search): $\approx 1 + \frac{\alpha}{2}$
  - retrieval/delete unsuccessful (search): $= \alpha$

EFFICIENCY OF HASHING 5

The efficiency of the main collision-resolution schemes





B-TREES 1

B-trees extend 2-3 trees by permitting more than 1 key in a tree node.

In a B-tree, all data records are stored at leaf nodes, in increasing order of their keys.
In a B-tree, parental nodes are only used for indexing, and store only keys (not data).
In a B-tree, each parental node stores n-1 keys, and n pointers to subtrees.

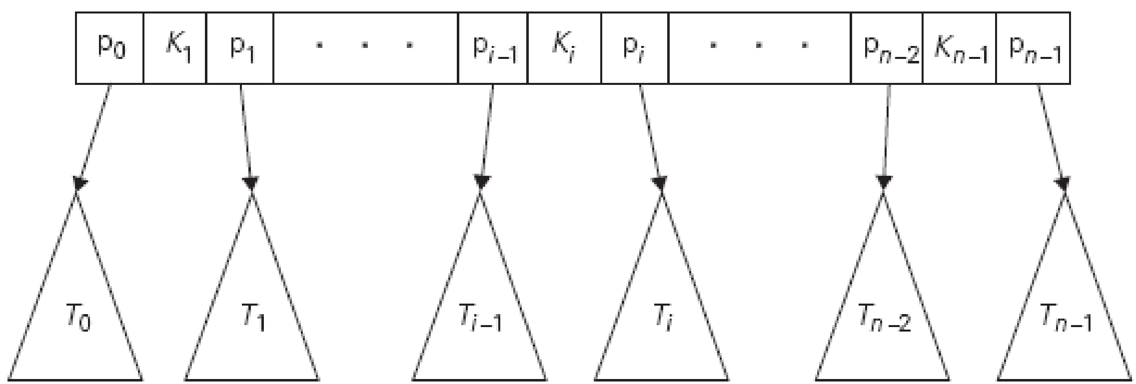


FIGURE 7.7 Parental node of a B-tree.

B-TREES 2

A B-tree of order m≥2 must satisfy these properties:

- The root node is either a leaf node or has between **2** and **m** child nodes.
- Each node, except for the root node and the leaf nodes, stores between **[m/2]** and **m** pointers to child nodes (and between **[m/2]-1** and **m-1** keys).
- The tree is **always perfectly balanced** (all its leaf nodes are at the same level).

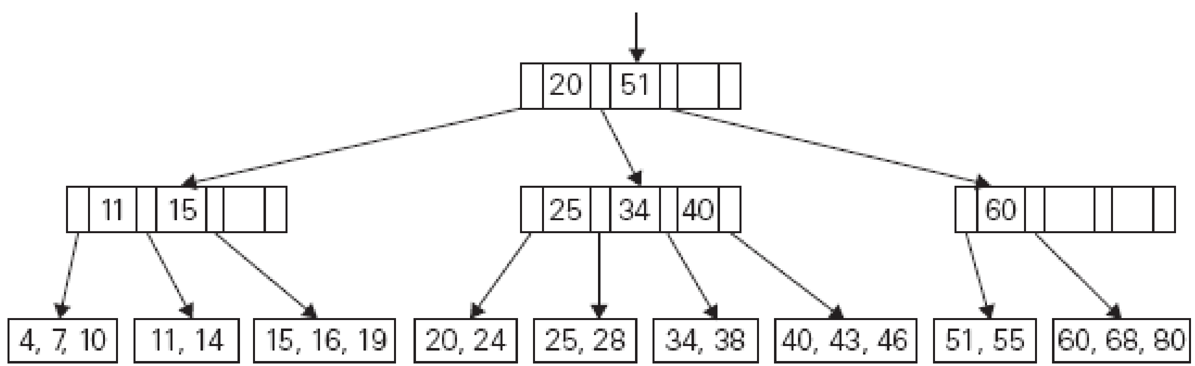


FIGURE 7.8 Example of a B-tree of order 4.

B-TREES 3

Figure: Insertion of item **65** in a B-tree of order 4, with the constraint that leaf nodes can store max 3 items.

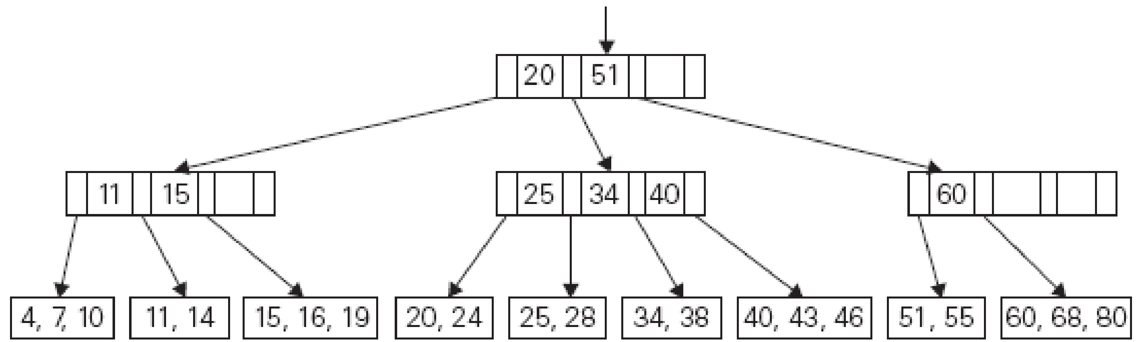


FIGURE 7.8 Example of a B-tree of order 4.

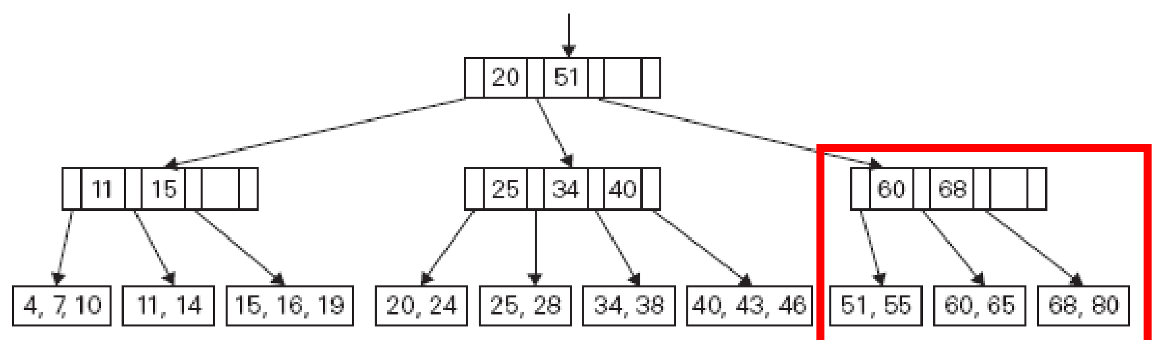


FIGURE 7.9 B-tree obtained after inserting 65 into the B-tree in Figure 7.8.

Kettering
UNIVERSITY