GIUSEPPE TURINI

CS-102: COMPUTING AND ALGORITHMS 2
LESSON 07
TABLES, PRIORITY QUEUES, HEAPS, AND HASHING

Kettering
UNIVERSITY

HIGHLIGHTS

Balanced Search Trees

2-3 Tree, 2-3-4 Tree, Red-Black Tree, and AVL Tree

Heap: Definition, Minheap-Maxheap, and Main Operations
Array-Based Implementation, and Heapsort Sorting Algorithm

Table: Definition, Operations, and Implementation Selection
Generic-Based Sorted-Array and BST Implementations, JCF Map and Set

Priority Queue (PQ): Sorted-Linear/BST/Heap Implementations, JCF PriorityQueue

Hashing: Definition and Terminology, Hash Functions, and Collision Resolution
Efficiency, and JCF Hashtable and HashMap

Multi-Data-Organizations: External-Internal Storage, Table Traversal with Hashing

STUDY GUIDE

STUDY MATERIAL

- **This slides.**

SELECTED EXERCISES

- **Set 1:** ex 11-13, ex. 15, ex. 17, ex. 19-20, ex. 22.
- **Set 2:** ex. 14-18, ex. 21-22, ex. 25-26.
- **Set 3:** ex. 1-2, ex. 4-5, ex. 7-8, ex. 10-18, ex. 20.

ADDITIONAL RESOURCES

- **"Object-Oriented Data Structures Using Java (4th Ed.)", chap. 8-9.**
- "Data Abstraction and Problem Solving with Java (3rd Ed.)", chap. 12.
- visualgo.net/en/heap
- visualgo.net/en/hashtable

HEAP - DEFINITION

A **heap** is similar to a BST, with these differences:

- you can view a BST as sorted, but **a heap is sorted in a weaker sense** (and this is enough to preserve its shape);
- a BST may have different shapes, but **a heap is always a complete binary tree**.

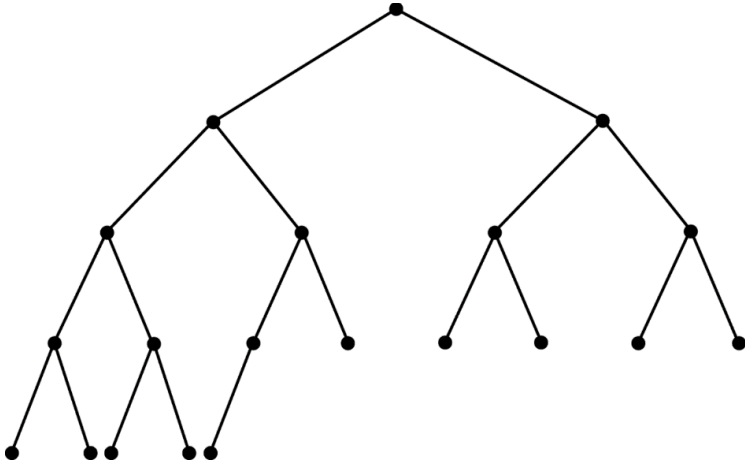


Figure: A complete binary tree.

A heap is a complete binary tree:

- that is empty **OR** whose root stores a key ≥ than the keys in its children, **AND**
- whose root has heaps as its left and right subtrees.

HEAP - MINHEAP, MAXHEAP, AND OPERATIONS

Note: In our definition of a heap, the root contains the item with the max search key. Such heap is also known as a **maxheap**. A **minheap** places the item with the min search key in its root.

Pseudocode for the operations of the ADT heap:

```
createHeap(); // Creates an empty heap.
heapIsEmpty(); // Determines whether a heap is empty.

// Inserts newItem into a heap. Throws HeapException if heap is full.
heapInsert(newItem) throws HeapException;

// Retrieves and then deletes a heap's root item. This item has the largest search key.
heapDelete();
```

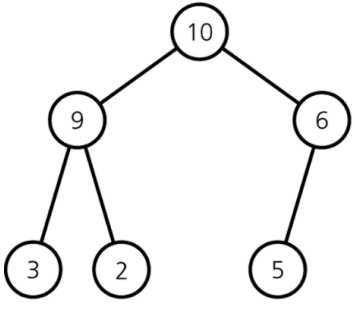
HEAP - ARRAY-BASED DESIGN

Because a heap is a complete binary tree, you can use an **array-based implementation of a binary tree**, if you know the max size of the heap.

Array-based heap:
Let the following data fields represent the heap:

- **items:** an array of heap items;
- **size:** number of items in heap.

Figure: An array-based heap.



0	10
1	9
2	6
3	3
4	2
5	5

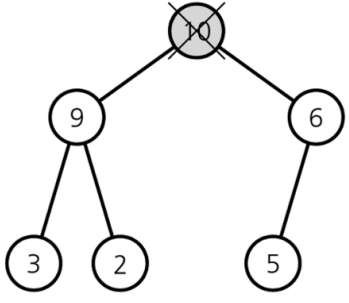
HEAP - ARRAY-BASED DELETE 1

Array-based heap: operation heapDelete

The **heapDelete** operation needs to:

1. locate the max search key, that must be in the root (i.e. array element at index 0);
2. remove the root (leaving the heap with 2 **disjoint heaps**);
3. transform the **remaining nodes back into a heap**.

Figure: Two disjoints heaps.



(a)

0	10
1	9
2	6
3	3
4	2
5	5

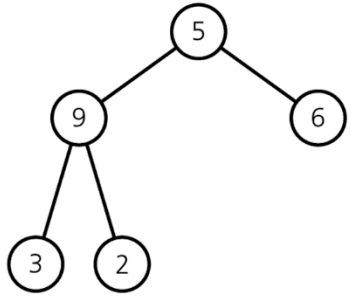
HEAP - ARRAY-BASED DELETE 2

Array-based heap: operation heapDelete (step 3)

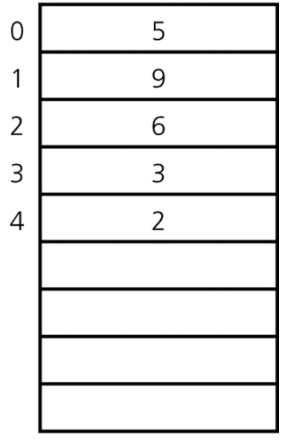
The transformation of 2 disjoint heaps back into a heap is the following:

a. copy the item from the "last" node into the root (resulting in a **semiheap**);
b. move the root (e.g. **5**) down the heap until it reaches a proper location.
   - first, you compare the root (e.g. **5**) against its children (e.g. **9** and **6**);
   - if root (e.g. **5**) is smaller than greater child (e.g. **9**), you **swap** them (e.g. **5-9**);
   - you repeat until the original root (e.g. **5**) is greater than both its children.

Figure: A semiheap.



(b)



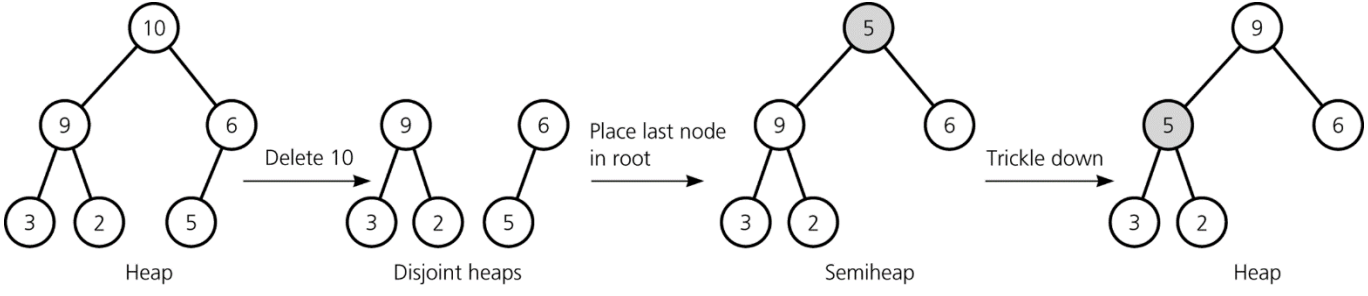
HEAP - ARRAY-BASED DELETE 3

Array-based heap: operation heapDelete (step 3, continued)

The transformation of 2 disjoint heaps back into a heap is the following:

a. copy the item from the "last" node into the root (resulting in a **semiheap**);
b. move the root (e.g. **5**) down the heap until it reaches a proper location.
   - first, you compare the root (e.g. **5**) against its children (e.g. **9** and **6**);
   - if root (e.g. **5**) is smaller than greater child (e.g. **9**), you **swap** them (e.g. **5-9**);
   - you repeat until the original root (e.g. **5**) is greater than both its children.

Figure: The step-by-step deletion of the root from a heap



HEAP - ARRAY-BASED DELETE 4

Array-based heap: operation heapDelete (step 3, continued)

The following **heapRebuild** recursive algorithm converts a semiheap into a heap:

```
protected void heapRebuild( int root ) {
   int leftChild = 2 * root + 1; int greatestChild = leftChild;
   if( leftChild < items.size() ) { // Root is not a leaf, so "leftChild" is valid.
      int rightChild = leftChild + 1;  // Index of root right child, if any.
      // If root has a right child: find the greatest child between left and right.
      if( ( rightChild < items.size() ) &&
          ( compareItems( items.get( rightChild ),items.get( leftChild ) ) ) < 0 ) {
         greatestChild = rightChild; } // Index of greatest child.
      // If the root is smaller than the greatest child, swap their values.
      if( compareItems( items.get( root ), items.get( greatestChild ) ) > 0 ) {
         T temp = items.get( root );
         items.set( root, items.get( greatestChild ) ); items.set( greatestChild, temp );
         heapRebuild( greatestChild ); } } // Transform new subtree into a heap.
   // If root is a leaf, do nothing.
}
```

HEAP - ARRAY-BASED DELETE 5

Array-based heap: operation heapDelete (continued)

So the following is the complete **heapDelete** function (efficiency **O(log n)**):

```
// Retrieves and deletes the item in the root (max search key) of a heap.
// Note: if heap is empty, returns null.
public T heapDelete() {
   T rootItem = null; int loc; // Init temp variables.
   // If heap is empty, the removal/swap is impossible and then set() will not work.
   if( !heapIsEmpty() ) {
      rootItem = items.get( 0 ); // Get root item.
      loc = items.size() - 1; // Location of "last" heap item.
      items.set( 0, items.get( loc ) ); // Swap root with "last" item.
      items.remove( loc ); // Remove "last" item.
      heapRebuild( 0 ); } // Convert the semiheap back into a heap.
   return rootItem; // Return the item with the greatest search key.
}
```

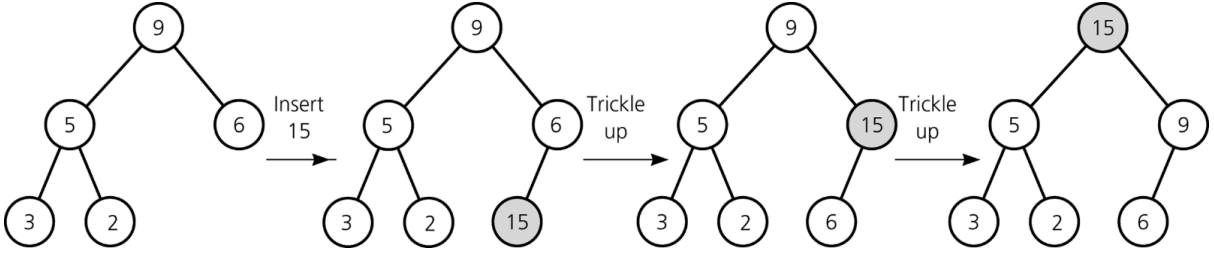
HEAP - ARRAY-BASED INSERT 1

Array-based heap: operation heapInsert

The **heapInsert** algorithm is the opposite of **heapDelete**. A new item is inserted at the "bottom" of the binary tree, and it trickles up to its proper place.

Note: It is easy to trickle up a node, because the parent of the node in **items[i]** (other than the root) is always stored in **items[(i-1)/2]**.

Figure: The step-by-step insertion into a heap



HEAP - ARRAY-BASED INSERT 2

Array-based heap: operation heapInsert (continued)

The following is the complete **heapInsert** function (efficiency **O(log n)**):

```
// Inserts an item into a heap. Note: if heap is full, HeapException is thrown.
public void heapInsert( T newItem ) throws HeapException, ClassCastException {
   if( !items.add( newItem ) ) { throw new HeapException( "Insertion failed!" ); }
   else { // Trickle new item up to its proper position.
      int place = items.size() - 1; int parent = ( place - 1 ) / 2;
      while( ( parent >= 0 ) &&
             ( compareItems( items.get( place ), items.get( parent ) ) ) < 0 ) {
         // Swap items[place] and items[parent].
         T temp = items.get( parent );
         items.set( parent, items.get( place ) );
         items.set( place, temp );
         // Update "place" and "parent" indices.
         place = parent;
         parent = ( place - 1 ) / 2; } }
}
```

HEAP - ARRAY-BASED IMPLEMENTATION 1

HEAP A

```
package Heaps;
import java.util.ArrayList;
import java.util.Comparator;

// Array-based implementation of the ADT heap.
public class Heap<T> {
   private ArrayList<T> items; // Array of heap items.
   private Comparator<? super T > comparator; // Object to compare 2 items.

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

   // Constructor.
   public Heap( Comparator<? super T > comparator ) {
      items = new ArrayList<T>();
      this.comparator = comparator;
   }
```

HEAP - ARRAY-BASED IMPLEMENTATION 2

HEAP B

```
public boolean heapIsEmpty() { return items.size() == 0; }

// Inserts an item into a heap. Note: if heap is full, HeapException is thrown.
public void heapInsert( T newItem ) throws HeapException, ClassCastException {
   if( !items.add( newItem ) ) { throw new HeapException( "Insertion failed!" ); }
   else { // Trickle new item up to its proper position.
      int place = items.size() - 1; int parent = ( place - 1 ) / 2;
      while( ( parent >= 0 ) &&
             ( compareItems( items.get( place ), items.get( parent ) ) ) < 0 ) {
         // Swap items[place] and items[parent].
         T temp = items.get( parent );
         items.set( parent, items.get( place ) );
         items.set( place, temp );
         // Update "place" and "parent" indices.
         place = parent;
         parent = ( place - 1 ) / 2; } }
}
```

HEAP - ARRAY-BASED IMPLEMENTATION 3

HEAP C

```
private int compareItems( T item1, T item2 ) {
   if( comparator == null ) { return ( (Comparable <T>) item1 ).compareTo(item2); }
   else { return comparator.compare(item1, item2); }
}

// Retrieves and deletes the item in the root (max search key) of a heap.
// Note: if heap is empty, returns null.
public T heapDelete() {
   T rootItem = null; int loc; // Init temp variables.
   if( !heapIsEmpty() ) { // If heap empty, swap is impossible so set() will not work.
      rootItem = items.get( 0 ); // Get root item.
      loc = items.size() - 1; // Location of "last" heap item.
      items.set( 0, items.get( loc ) ); // Swap root with "last" item.
      items.remove( loc ); // Remove "last" item.
      heapRebuild( 0 ); } // Convert the semiheap back into a heap.
   return rootItem; // Return the item with the greatest search key.
}
```

HEAP - ARRAY-BASED IMPLEMENTATION 4

HEAP D

```
   // Converts a semiheap rooted at index "root" into a heap.
   protected void heapRebuild( int root ) {
      int leftChild = 2 * root + 1; int greatestChild = leftChild;
      if( leftChild < items.size() ) { // Root is not a leaf, so "leftChild" is valid.
         int rightChild = leftChild + 1;  // Index of root right child, if any.
         // If root has a right child: find the greatest child between left and right.
         if( ( rightChild < items.size() ) &&
             ( compareItems( items.get( rightChild ),items.get( leftChild ) ) ) < 0 ) {
            greatestChild = rightChild; } // Index of greatest child.
         // If the root is smaller than the greatest child, swap their values.
         if( compareItems( items.get( root ), items.get( greatestChild ) ) > 0 ) {
            T temp = items.get( root );
            items.set(root, items.get(greatestChild)); items.set(greatestChild, temp);
            heapRebuild( greatestChild ); } } // Transform new subtree into a heap.
      // If root is a leaf, do nothing.
   }
}
```

HEAP - HEAPSORT SORTING ALGORITHM

To be completed...

TABLE - DEFINITION

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.

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

Figure: An ordinary table of cities.

TABLE - OPERATIONS 1

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.

TABLE - OPERATIONS 2

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 );
```

TABLE - IMPLEMENTATION SELECTION 1

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.

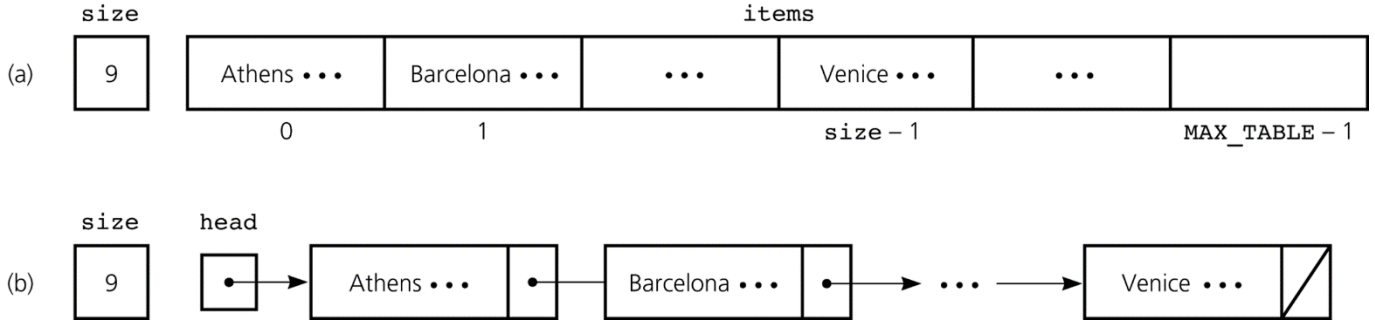


TABLE - IMPLEMENTATION SELECTION 2

Categories of **non-linear implementations**:

- binary search tree (BST) implementation.

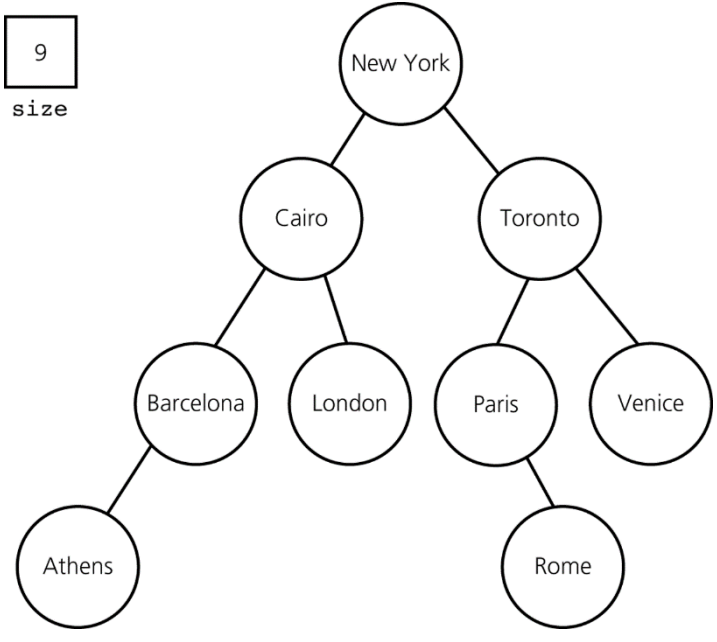


Figure: A BST implementation of the ADT table.

TABLE - IMPLEMENTATION SELECTION 3

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?**

TABLE - IMPLEMENTATION SELECTION 4

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).

TABLE - IMPLEMENTATION SELECTION 5

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

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

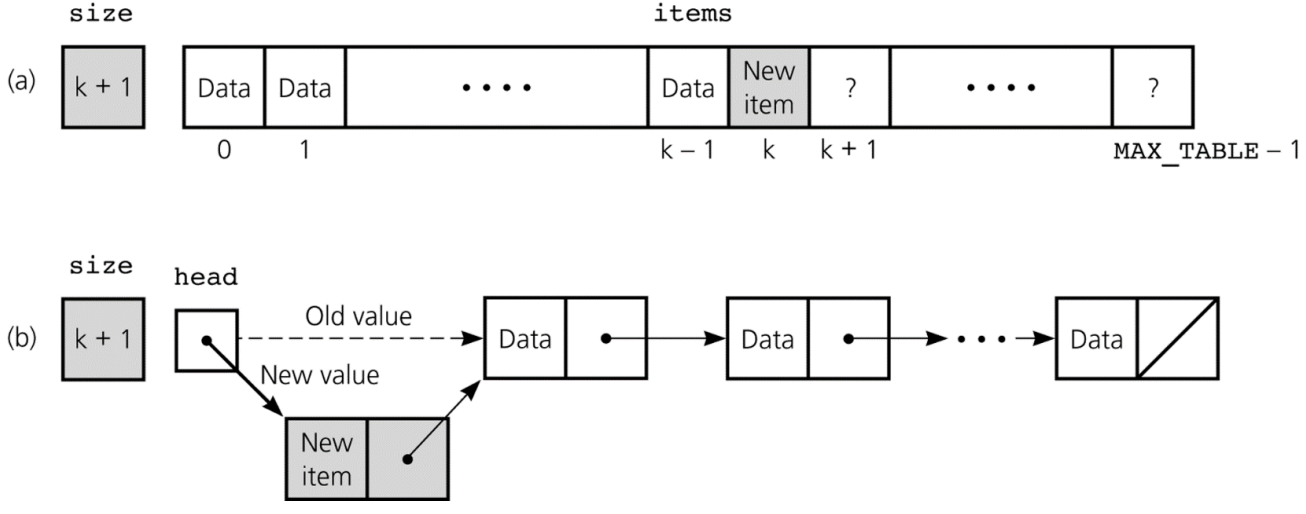


TABLE - IMPLEMENTATION SELECTION 6

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).

TABLE - IMPLEMENTATION SELECTION 7

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 **$O(\log_2 n)$**);
    - **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.

TABLE - IMPLEMENTATION SELECTION 8

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).

TABLE - IMPLEMENTATION SELECTION 9

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

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

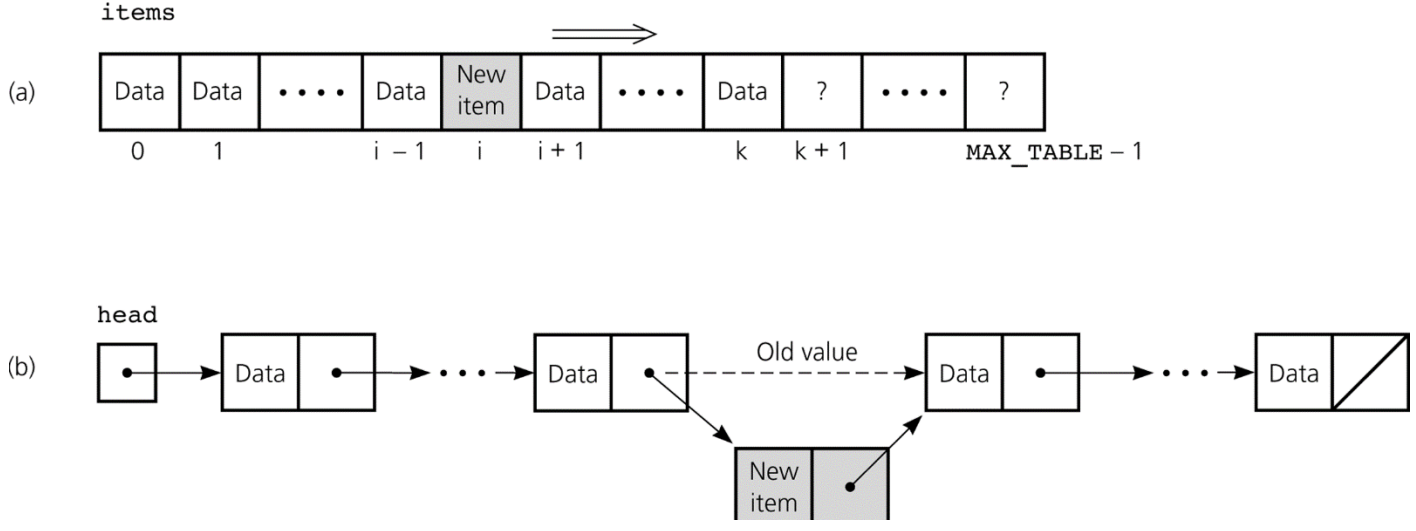


TABLE - IMPLEMENTATION SELECTION 10

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.

TABLE - IMPLEMENTATION SELECTION 11

- **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)

TABLE - GENERIC IMPLEMENTATIONS 1

A sorted-array implementation (TableArrayBased class):

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

A BST implementation (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.

TABLE - GENERIC IMPLEMENTATIONS 2

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

TABLE - GENERIC IMPLEMENTATIONS 3

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

TABLE - GENERIC IMPLEMENTATIONS 4

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;
```

TABLE - GENERIC IMPLEMENTATIONS 5

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); } }
```

TABLE - SORTED ARRAY IMPLEMENTATION 1

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 ); }
```

TABLE - SORTED ARRAY IMPLEMENTATION 2

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!" ); }
}
```

TABLE - SORTED ARRAY IMPLEMENTATION 3

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; }
}
```

TABLE - SORTED ARRAY IMPLEMENTATION 4

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;
   }

}
```

TABLE - BST IMPLEMENTATION 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.
```

TABLE - BST IMPLEMENTATION 2

TABLE BST BASED B

```
// 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!" ); }
}
```

TABLE - BST IMPLEMENTATION 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; }

}
```

TABLE - JCF MAP AND SET CLASSES 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

TABLE - JCF MAP AND SET CLASSES 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

PQ - DEFINITION AND OPERATIONS

The ADT priority queue (PQ) is a variation of the ADT table:

- a priority queue orders its items by a priority value, and
- the first item removed is the one having the highest priority value.

Operations of the ADT priority queue:

- **create** an empty priority queue;
- determine whether a priority queue **is empty**;
- **insert** a new item into a priority queue;
- **retrieve** and then **delete** the item with the highest priority value.

Pseudocode for the operations of the ADT priority queue:

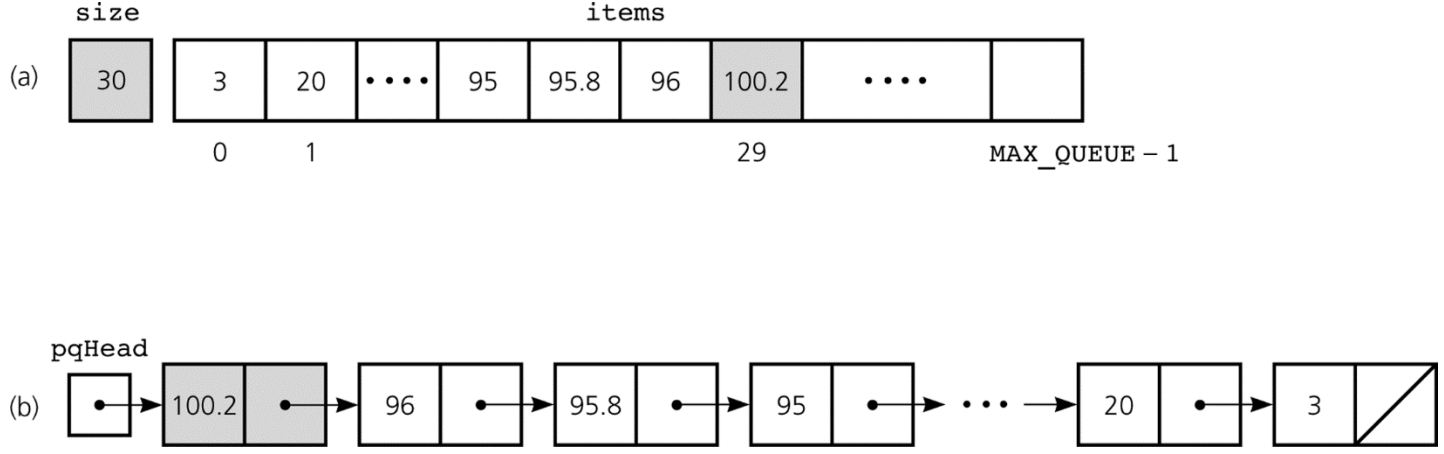
```
createPQ (); // Create an empty PQ.
pqIsEmpty(); // Determine if a PQ is empty.
pqInsert( newItem ) throws PQException; // Insert new item (exception if PQ is full).
pqDelete(); // Retrieve and then delete item with max priority from PQ.
```

PQ - SORTED LINEAR IMPLEMENTATIONS

Sorted linear implementations of the ADT priority queue:

- appropriate if the number of items in the priority queue is small;
- array-based priority queues keep items sorted in ascending order of priority;
- reference-based priority queues keep items sorted in descending priority.

Figure: Array-based priority queue **(a)**, and reference-based priority queue **(b)**.

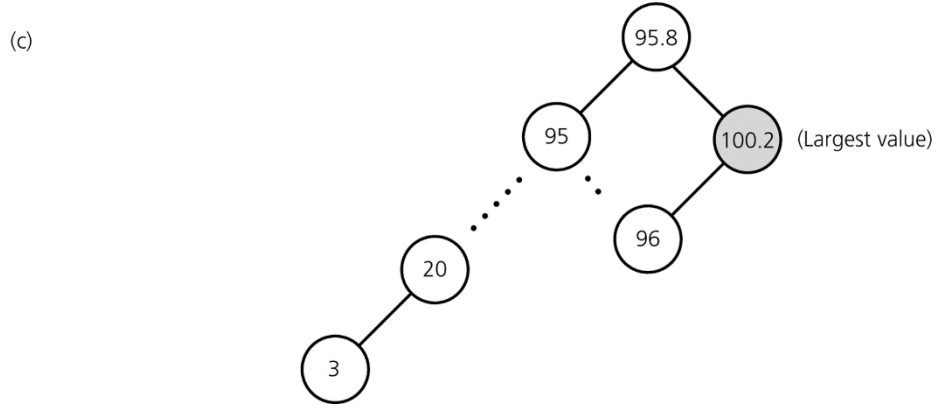


PQ - BINARY SEARCH TREE IMPLEMENTATION

Binary search tree implementation of the ADT priority queue:

- appropriate for any priority queue;
- the **pqDelete** operation has to locate the item with max priority (i.e. BST rightmost node), and then it has to remove it (i.e. a node with at most 1 child).

Figure: A binary search tree implementation of a priority queue **(c)**.



PQ - HEAP IMPLEMENTATION 1

If you know the max size of a PQ, a better choice than a BST is an array-based heap.

A heap is often the most appropriate implementation for a PQ (but it is not at all appropriate to implement a table).

Heap implementation of a PQ:
If a heap implementation is available, the PQ implementation is straightforward:

- the priority queue **operations are exactly analogous** to heap operations;
- the **priority** value in PQ items is the **search key** of an item in the heap;
- so the class **PriorityQueue** will include an **Heap** object as data field.

PQ - HEAP IMPLEMENTATION 2

PRIORITY QUEUE A

```
package PriorityQueues;
import Heaps.Heap; import Heaps.HeapException; import java.util.Comparator;

// Heap implementation of the ADT priority queue.
public class PriorityQueue<T>{
   private Heap<T> h; // ...

   public PriorityQueue() { h = new Heap<T>(); }
   public PriorityQueue( Comparator<? super T > comp) { h = new Heap<T>( comp ); }
   public boolean pqIsEmpty() { return h.heapIsEmpty(); }
   public T pqDelete() { return h.heapDelete(); }

   public void pqInsert( T newItem ) throws PriorityQueueException {
      try { h.heapInsert( newItem ); }
      catch( HeapException e ) { throw new PriorityQueueException( "Insert failed!" ); }
   }
}
```

PQ - HEAP IMPLEMENTATION 3

Question: How does a heap compare to a BST to implement a priority queue?
Answer: If you know the max number of items in the priority queue, the heap is the better implementation.
Because a heap is a complete binary tree, it is balanced, which is a major advantage.
If the BST is balanced, both will have the same average performance: **O(log n)**.
However, the BST height can increase depending on insertions/removals, so its shape can reach an unbalanced structure, degrading its performance to **O(n)** in worst case.
The heap implementation preserve the balance, avoiding this performance decrease.

Pros and cons of a heap implementation of the ADT priority queue:

- **Cons:** requires the knowledge of the max size of the priority queue.
- **Pros:** always balanced, able to handle finite priority values (heap of queues).

PQ - JCF PRIORITYQUEUE CLASS

The JCF **PriorityQueue** class stores a single data-type parameter in ascending order (relying on the **Comparable** interface or a **Comparator** object):

```
public Class PriorityQueue<T> { // JCF Set interface (partial view).
   PriorityQueue( int initialCapacity ); // Creates a PQ with the input initial capacity.
   PriorityQueue( int initialCapacity, Comparator<? super T > comparator ); // See above.
   boolean add( T o ); // Inserts the input item into the PQ.
   void clear(); // Removes all of the items from the PQ.
   Comparator<? super T > comparator(); // Returns the comparator used in the PQ.
   boolean contains( Object o ); // Returns true if the PQ contains the input item.
   Iterator<T> iterator(); // Returns an iterator over the items in the PQ.
   boolean offer( T o ); // Inserts the input item into the PQ.
   T peek(); // Retrieves, but does not remove, the head of the PQ.
   T poll(); // Retrieves and removes the head of the PQ.
   boolean remove( Object o ); // Removes a single instance of input item from the PQ.
}
```

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

HASHING - DEFINITION

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 (which each array slot storing a single table item) and a seemingly magical box called an **"address calculator"** (aka **hash function**).

Whenever you have a new item that you want to insert into the table, the address calculator will tell you where you should place it in the array. The same address calculator will tell you where is an item previously stored in the table.

This technique is called **hashing**, a table organized in this way is called **hash table** and it provides insertions/retrievals/ removals performed in constant time: all in **O(1)**.

HASHING - TERMINOLOGY

Hashing: Method to a table item in **constant time** independent of the table size.

Hash Table: An **array** that contains the table items, as assigned by a hash function.

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.

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.

HASHING - HASH FUNCTION 1

REQUIREMENTS FOR A HASH FUNCTION

- be **easy and fast to compute**;
- **place items evenly** throughout the hash table;
- it is sufficient for hash functions to **operate on integers**;
    - **simple hash functions that operate on positive integers:** selecting digits; folding, module arithmetic;
    - **converting a character string to an integer:** if the search key is a character string, it can be converted into an integer before applying the hash function.

HASHING - HASH FUNCTION 2

Issues to consider on how evenly a hash function scatters the keys:

- How well does the hash function scatter **random data**?
- How well does the hash function scatter **non-random data**?

General requirements of a hash function:

- The calculation of the hash function should **involve the entire search key**.
- If a hash function uses **module arithmetic, the base should be prime**.

HASHING - COLLISION RESOLUTION 1

RESOLVING COLLISIONS A

Two approaches exist for collision resolution:

- **Approach 1 - 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 - COLLISION RESOLUTION 2

RESOLVING COLLISIONS B

Two approaches exist for collision resolution:

- **Approach 1 - 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^{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 - COLLISION RESOLUTION 3

RESOLVING COLLISIONS C

Two approaches exist for collision resolution:

- **Approach 2 - Restructuring the Hash Table:** changes the structure of the hash table so that it can accommodate more than one item in the same location.
    - **Buckets:** each location in the hash table is itself **an array called a bucket**.
    - **Separate Chaining:** each hash table location is a linked list.

HASHING - EFFICIENCY 1

EFFICIENCY OF HASHING A

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 $\mathbf{size_{max}}$ of the array table; **α should not exceed 2/3.**

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

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

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

HASHING - EFFICIENCY 2

EFFICIENCY OF HASHING B

- **Linear Probing:** insertions, retrievals, and deletions:
  - successful search: **$\frac{1}{2} [1 + 1 (1 - \alpha)]$**
  - unsuccessful search: **$\frac{1}{2} [1 + 1 (1 - \alpha)^2]$**

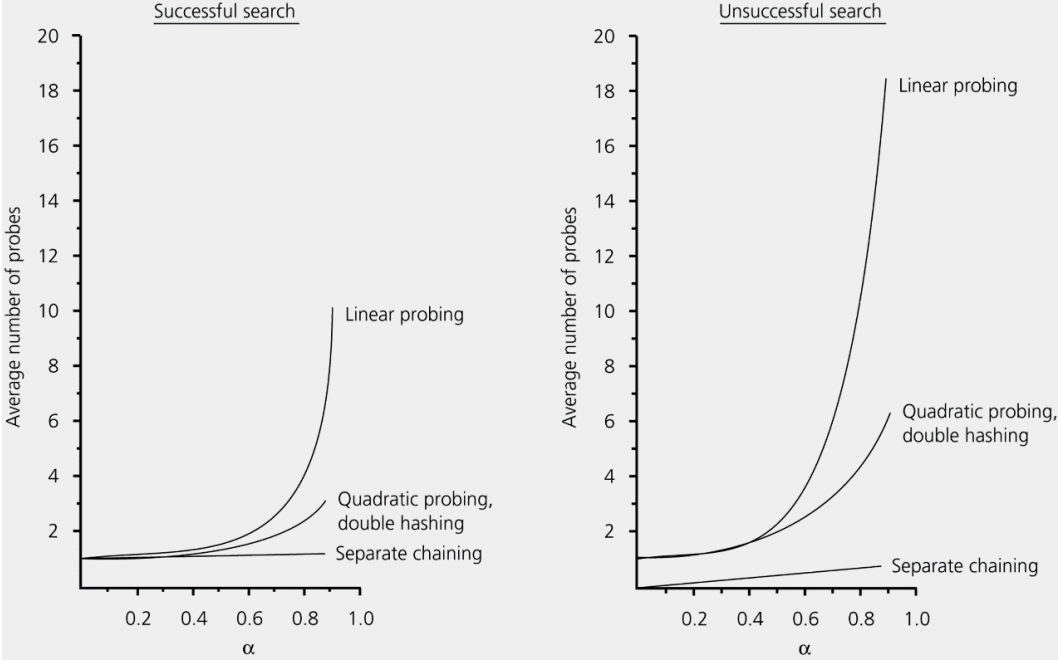
- **Quadratic Probing and Double Hashing:** insertions, retrievals, and deletions:
  - successful search: **$\{ -1 [\log_e (1 - \alpha)] \} / \alpha$**
  - unsuccessful search: **$1 / (1 - \alpha)$**

- **Separate Chaining:** insertions are **O(1)**, whereas retrievals and deletions:
  - successful search: **$1 + (\alpha / 2)$**
  - unsuccessful search: **α**

HASHING - EFFICIENCY 3

EFFICIENCY OF HASHING C

Figure: The relative efficiency of 4 collision-resolution methods.



HASHING - JCF HASHTABLE CLASS

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** (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)

DATA WITH MULTIPLE ORGANIZATIONS

EXAMPLE: TABLE TRAVERSAL IS INEFFICIENT USING HASHING

Hashing as an implementation of the ADT table:

- In many applications, hashing is the most efficient implementation of a table.
- However, 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;