Data Structures and Information Retrieval

Summing up our Knowledge about
- Maps,
- Sorting and
- Searching
Associative Tables, or Maps

- Information entry contains a part which uniquely identifies the entry: the key
- Mapping from key to (rest of) information
- Mapping:
  - left unique relation
  - i.e. unique right value for every left value

- So an Associative Table can be considered a mapping from key to value.
  - usually a unique mapping
- But to retrieve the data, we need a second mapping:
  - from key to storage!
Associative Structures

→ Associative data structure:
   - Retrieval by content
   - Usually a key plus information ("payload")
   - hide physical data structure (storage concept)

• It's all about information retrieval.
• How to implement efficient lookup?
Example: Electronic Patient Records

- Patient Records
- Kept in a database to efficiently find, compare and evaluate patient information
- So a patient record would be a line in a database table
- Unique, identifying **key**: insurance number

<table>
<thead>
<tr>
<th>insuranceNo</th>
<th>dateOfBirth</th>
<th>name</th>
<th>firstname</th>
<th>bloodgroup</th>
<th>inscardread</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEK2233445</td>
<td>1900-02-08</td>
<td>Wassermann</td>
<td>Eugen</td>
<td>0+</td>
<td>1988-05-05</td>
</tr>
<tr>
<td>BKK9953678</td>
<td>1999-07-02</td>
<td>Sandmann</td>
<td>Horst</td>
<td>AB-</td>
<td>2005-01-01</td>
</tr>
<tr>
<td>T KK1702198</td>
<td>1986-06-06</td>
<td>Schnnemann</td>
<td>Jupp</td>
<td>A0</td>
<td>2007-01-06</td>
</tr>
</tbody>
</table>
Information Retrieval by Key

- Either:
  - keep the table SORTED
    (and splayed or balanced, if it is a tree)
- Or:
  - use key hashing to store PatientRecords
Implementation as List of Records

• Type Patient Record
  
  ```java
  public class PatientRecord {
    String insuranceNo;
    Date dateOfBirth;
    String name;
    String firstName;
    BloodGroup bloodGroup;
    Date insuranceCardRead;
  }
  ```

• Table is array or list or tree of Patient Records

  ```java
  PatientRecord[] table;
  or
  LinkedList<PatientRecord> table1;
  or
  SplayTree<PatientRecord> table3;
  ```
Implementing a Hash Table

- either list of records
- handle hash clashes by open addressing

- or list of chainable entries
- handle hash clashes by chaining
Efficiency of Table Insertion

- Immediate insertion in **Linked List**
  - but requires full iteration for identifying the insertion point

- Immediate insertion with **Sorted Tree**
  - requires splaying or rebalancing afterwards for efficiency

- Requires data moving in an **Array** (c.f. Insertionsort below)
  - insertion point efficiently identified (binsearch)
  - successive information moved down before insertion

- Potentially immediate insertion in a **Hash Table**
  - requires open addressing or chaining with hash clashes
Efficiency of Lookup

- \( O(\log n) \) complexity for Sorted Array or Sorted Tree (if balanced)
- \( O(c) \) – constant for Hash Table

- But \( O(n) \) for Linked List
  - that is why Lists offer a function `toArray()`
Index Lookup

- "Find all patients named "Schmiedecke".
- Key sorting is not helpful, because we are looking for a record field!
- For a list (including hash table):
  - create an index (table) for the name field of the PatientRecord
  - and keep it sorted
  - You can create independant indexes for all record fields!

```plaintext
Patient Schmiedecke
Patient Jeffers
Patient Andresen
Patient Schmiedecke

0 1 2 3
0 1 3 2

sort

index for "name"
```

so the only data moved during sorting are small integers
Working with an index:

• find entry point for "Schmiedecke" in nameIndex:

```java
private int nameEntryPoint(String name) {
    for (int i=0; i<nameIndex.length; i++) {
        if (table[nameIndex[i]] == null)
            return i;
        if (table[nameIndex[i]].name.greater(name))
            return i;
    }
    return nameIndex.length;  // watch out, table is full!
}  // rather use binsearch for efficient lookup!!!
```

• **Note:** the "Schmiedecke" record can be entered in any place in table, e.g. at index 238. This number 238 will be stored in `nameIndex[i]`, possibly after moving subsequent information down the `nameIndex` table.
Notice on Key Sorting

- To avoid moving full PatientRecords in order to keep the table key-sorted,
- you can create a sorted index for the key, too!

- Note: This technique is standard in database implementations.
- In relational databases, the user can decide on additional indexes for frequent search criteria.
Improving Data Access

• PatientRecord is a nice application type.

• But accessing data requires time due to indirection:
  - it is a reference type containing references
  - e.g. a name is accessed as table[i].name

• Column operations like
  - initializing a column
  - deleting a column
  - updating a column

require iterating through all PatientRecords.

→ Organize table as array or list of columns rather than a list of records
Table as List of Columns

<table>
<thead>
<tr>
<th>BEK775534</th>
<th>12/12/1944</th>
<th>Wassermann</th>
<th>Horst</th>
<th>01/06/2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>TKK170286</td>
<td>17/02/1986</td>
<td>Sandmann</td>
<td>Sepp</td>
<td>01/05/2007</td>
</tr>
<tr>
<td>BKK887506</td>
<td>01/06/1999</td>
<td>Schneemann</td>
<td>Jutta</td>
<td>12/07/2005</td>
</tr>
</tbody>
</table>

- Table implemented as **list of columns**.
- Each column has its **data type**.
- Each column may have a **name**.
- A **PatientRecord** is a set of column values with identical index.
  
  ➔ If you have looked up a column entry, e.g. "Schneemann", its **index** gives access to the entire **PatientRecord**.
Implementation

public class PatientTable {

    Vector<String> insuranceNo = new Vector<String>();
    Vector<Date> dateOfBirth = new Vector<Date>();
    Vector<String> name = new Vector<String>();
    Vector<String> firstName = new Vector<String>();
    Vector<BloodGroup> bloodGroup = new Vector<BloodGroup>();
    Vector<Date> insuranceCardRead = new Vector<Date>();

    Vector[] table = {insuranceNo, dateOfBirth, name, firstName, bloodGroup, insuranceCardRead};
Implementation

```java
public PatientRecord getPatient(String insuranceNumber) {
    int index = lookupKey(insuranceNumber); // using keyIndex
    if (index<0) return null; // not found
    return new PatientRecord(insuranceNo.get(index),
                              dateOfBirth.get(index),
                              name.get(index),
                              firstName.get(index),
                              bloodGroup.get(index),
                              insuranceCardRead.get(index));
}

private int lookupKey(String key){
    // returns -1 as code for "not found"
    for (int i=0; i<insuranceNo.size(); i++) {
        if (insuranceNo.get(keyIndex[i]).equals(key)) return i;
        if (insuranceNo.get(keyIndex[i]).greater(key)) return -1;
    }
}

// lookupName, lookupBloodGroup, etc. similar
// but use Binsearch instead of linear search!!
```
Summary Associative Structures

- Associative structures, or "maps", allow to retrieve information using part of it, the "key".
- Information retrieval requires searching for the key.

- To allow for efficient lookup, we use either
  - key sorting, or
  - key hashing

- Key hashing is applied when there is no order for keys, or if order does not matter.
- Index tables are used to avoid moving large amounts of data during sorting.
- Multiple index tables allow efficient lookup by different criteria.

**Question:**
*Can you use multiple index tables if you use the list-of-records implementation (slide 6) rather than the list-of-columns implementation of the table?*
Sorting and Searching

• Now that we know how important sorting and searching is
• let's compile our knowledge about it:

• **Searching:**
  • 3 techniques:
    - "blindfold" immediate retrieval → hashing
      O(k)
    - systematic retrieval (Binsearch) → requires sorted structures
      O(log n)
    - full linear search
      O(n)

→ So to do well, sorting is important!
Sorting Algorithms

- BubbleSort
- InsertionSort
- QuickSort
- MergeSort
- TreeSort
- HeapSort
- ShellSort
- ShakerSort

http://cg.scs.carleton.ca/~morin/misc/sortalg/
public class BubbleSorter {
  private int[] a;
  private int n;
  public void sort(int[] a) {
    this.a=a;
    n=a.length;
    bubblesort();
  }
  private void bubblesort() {
    for (int i=n; i>1; i--)
      for (int j=1; j<i; j++)
        if (a[j-1]>a[j]) exchange(j-1, j);
  }
  private void exchange(int i, int j) {
    int t=a[i]; a[i]=a[j]; a[j]=t;
  }
} // end class BubbleSorter
InsertionSort

Idea:

Regard a growing section of the list as sorted and insert the next "unsorted" element.

```
5 7 0 3 4 2 6 1
0 5 7 3 4 2 6 1
0 3 5 7 4 2 6 1
0 3 4 5 7 2 6 1
0 2 3 4 5 7 6 1
0 2 3 4 5 6 7 1
0 1 2 3 4 5 6 7
```
public class InsertionSorter {
    private int[] a;
    private int n;
    public void sort(int[] a) {
        this.a=a;
        n=a.length;
        insertionsort();
    }
    private void insertionsort() {
        for (int i=1; i<n; i++) {
            int j=i;
            int temp=a[j];
            while (j>0 && a[j-1]>temp) { a[j]=a[j-1]; j--; }
            a[j]=temp;
        }
    }
} // end class InsertionSorter

O(n²)
Insertionsort Discussion

- Time complexity $O(n^2)$!
- But no preparation or cleaning up operations.
- → good for inserting data into sorted lists.
Quicksort

Divide:
- choose middle element x

Conquer:
- find first element from left which is bigger than x
- find first element from right which is smaller than x
- exchange if in wrong order
- find further exchange pairs until indeces meet

Recurse on both partitions.

Illustration using an array of ones (grey) and zeros (white),
taken from Lang, Algorithmen, a.a.O
public class QuickSorter {
    private int[] a;
    public void sort(int[] a) {
        this.a=a;
        quicksort(0, a.length-1);
    }
    private void quicksort (int lo, int hi) {
        int i=lo, j=hi; int mid=a[(lo+hi)/2]; // divide
        while (i<=j) { // conquer
            while (a[i]<mid) i++;
            while (a[j]>mid) j--;
            if (i<=j) {
                exchange(i, j);
                i++; j--;
            }
        }
        if (lo<j) quicksort(lo, j);
        if (i<hi) quicksort(i, hi); // Recursion
    }
    private void exchange(int i, int j) {
        int temp=a[i]; a[i]=a[j]; a[j]=temp;
    }
} // end class QuickSorter
Quicksort Discussion

• Worst Case Complexity $O(n^2)$ if middle element is smallest or biggest
• Average Complexity $O(n \cdot \log(n)) \rightarrow$ optimum

• In most tests, best known algorithm
• Improvement by choosing the median as middle element - can be done in linear time
Heapsort

Idea:
A heap is a complete bintree with all successors smaller than their predecessors.
A semi-heap is a heap with a possibly wrong root.
A semi-heap is transformed into a heap by successive rotations.

Build heap:
Leaves are heaps, so start with lowest bintrees and rotate them into heaps.
Move one step up and recurse.

Collect elements:
Remove root (biggest element) and replace by smallest leaf.
Build heap, recurse.
Heapsort Data Structure

- Remember that a heap is stored in an array as follows:
- a[n] is the root of the two subtrees a[2n+1] and a[2n+2], if they exist.
Heapsort Implementation

```java
public class HeapSorter {
    private int[] a, n;

    public void sort(int[] a) {
        this.a = a; n = a.length; heapsort();
    }

    private void heapsort() {
        buildheap();
        while (n > 1) { n--; exchange(0, n); downheap(0); }
    }

    private void buildheap() {
        for (int vertex = n / 2 - 1; vertex >= 0; vertex--) downheap(vertex);
    }

    private void downheap(int v) {
        int w = 2 * v + 1; // first descendant of v
        while (w < n) { // is there a second descendant?
            if (w + 1 < n) // w is the maximum descendant of v
                if (a[w + 1] > a[w]) w++; // v has heap property
            if (a[v] >= a[w]) return;
            exchange(v, w); // exchange labels of v and w
            v = w; // continue on subtree
            w = 2 * v + 1;
        }
    }

    private void exchange(int i, int j) {
        int temp = a[i]; a[i] = a[j]; a[j] = temp;
    }
} // end class HeapSorter
```
Heapsort Discussion

- Heapsort has a Worst Case time complexity of $O(n \cdot \log(n))$
- $\rightarrow$ optimal
- No extra space requirement.

- In tests, Quicksort does better on average than Heapsort!
MergeSort

Idea:
• Split list in two,
• sort sublists separately,
• merge sublists
public class MergeSorter {
    private int[] a, b; // auxiliary array b
    public void sort(int[] a) {
        this.a = a;
        int n = a.length;
        b = new int[(n+1)/2]; // auxiliary array only half a's size
        mergesort(0, n-1);
    }

    private void mergesort(int lo, int hi) {
        if (lo<hi) {
            int mid = (lo+hi)/2;
            mergesort(lo, mid);
            mergesort(mid+1, hi);
            merge(lo, m, hi);
        }
    }
}
private static void merge(int[] data, int temp[],
        int low, int middle, int high) {
    // a little tricky, saving memory in recursive calls:
    // assume that the lower half of temp and the upper half of data
    // are sorted lists;
    // merge them into data, starting at the free bottom part.
    int ri = low; // result index
    int ti = low; // temp index
    int di = middle; // destination index
    // merge small ends of lists until empty:
    while (ti<middle && di <=high) {
        if (data[di] < temp[ti]  // smallest is in high data
            { data[ri] = data[di]; ri++; di++; }
        else { // smallest is in temp
            data[ri] = temp[ti]; ri++; ti++; }
    }
    // copy remainder from temp
    while (ti<middle)
        { data[ri] = temp[ti]; ri++; ti++; }
}
MergeSort Discussion

- MergeSort has a time Complexity of $2n \cdot \log(n)$, which is $O(n \cdot \log(n))$
- $\rightarrow$ optimal
- requires $n/2$ extra space

- Statistically, Quicksort and Heapsort do better.
Sorting Summary

• Sorting is a **prerequisite for efficient searching**.
• It is **most important for information retrieval**!

• Sorting algorithms cannot do better than $O(n \log(n))$.
• Practically, there are **noticable differences** between algorithms of the same time complexity.
• Some algorithms need **extra space** (Mergesort), **preparation** (Heapsort), or **finalizing** (Mergesort), which influence the performance.
• Choice depends of the **initial state** of the list – semi-ordered or complete chaos?

• For many sorting algorithms, **recursion** is the most natural way of programming 😊
Seen enough loops and recursions? 😊

Next week we will be climbing on graphs...