326.041 (2015S) – Practical Software Technology

(Praktische Softwaretechnologie)

Comparing Objects, Simple Data Structures, Backtracking

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Comparable/Comparator interfaces


```
public class Person implements Comparable<Person> {
    private String familyName;
    private String givenName;
    private Date dateOfBirth;
    ...
```

- Sorting objects requires larger/equal/smaller comparisons.
- Collections that depend on sorting require comparisons.
    - Solution 1: Implement the interface **Comparable**.
        - You have to implement the method **compareTo**.
    - Solution 2: Provide a **Comparator** implementation.
        - You have to implement the method **compare**.
- There is a general contract. Given two objects o1 and o2:
    - if o1 is equal to o2 then comparison returns the integer $= 0$.
    - if o1 is larger than o2 then comparison returns an integer > 0.
    - if o1 is smaller than o2 then comparison returns an integer < 0.

Comparable – compareTo

- Provide a **compareTo method**, such that
  - if person is equal to other person then compareTo(other) $= 0$.
  - if person is larger than other person then compareTo(other) > 0.
  - if person is smaller than other person then compareTo(other) < 0.

```
public class Person implements Comparable<Person> {
    private String familyName;
    private String givenName;
    private Date dateOfBirth;
    ...
    public int compareTo(Person other) {
        int cmp = familyName.compareTo(other.familyName);
        if (cmp != 0) return cmp;
        cmp = givenName.compareTo(other.givenName);
        if (cmp != 0) return cmp;
        return dateOfBirth.compareTo(other.dateOfBirth);
    }
```

Comparator – compare

- Write a class which implements the interface Comparator<Person>.
- Provide a **compare method**, such that
  - if person p1 is equal to person p2 then compare(p1, p2) $= 0$.
  - if person p1 is larger than person p2 then compare(p1, p2) > 0.
  - if person p1 is smaller than person p2 then compare(p1, p2) < 0.

```
class PersonComp implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        int cmp = p1.getDateOfBirth().compareTo(
                      p2.getDateOfBirth());
        if(cmp != 0) return cmp;
        return p1.compareTo(p2);
    }
}
```

- Now you can sort a list of persons by their name of by their age:

```
Collections.sort(persons);
Collections.sort(persons, new PersonComp());
```

Anonymous Implementation

- Java allows **anonymous implementation**.
- Provide an anonymous implementation of Comparator, such that
  - if person p1 is equal to person p2 then compare(p1, p2) $= 0$.
  - if person p1 is larger than person p2 then compare(p1, p2) > 0.
  - if person p1 is smaller than person p2 then compare(p1, p2) < 0.

```java
Collections.sort(persons, new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        int cmp = p1.getDateOfBirth().compareTo(
                    p2.getDateOfBirth());
        if(cmp != 0) return cmp;
        return p1.compareTo(p2);
    }
});
```

- To instantiate an anonymous implementation, you have to **implement all the abstract methods**.
- You can also **anonymously override** implemented **methods**.

Membership: equals and compareTo

- Any collection with some sort of **membership** test **uses equals**.
- It is trivial to implement, if you have implemented Comparable:

```
public boolean equals(Object other) {
    if (!(other instanceof Person)) return false;
    return compareTo((Person) other) == 0;
}
```

- Now you can test for membership:

```
List<Person> persons = new ArrayList<>();
...
persons.add(...
persons.add(...
...
Person p = new Person("Touring","Alan", dateOfBirth);
System.out.println(persons.contains(p));
```

java.util.Objects

- Static **utility methods** for operating on objects.
- **null-tolerant** methods for
  - **comparing** two objects.
    - equals(Object a, Object b)
    - deepEquals(Object a, Object b)
    - compare(T a, T b, Comparator<? super T> c)
  - computing the **hash code** of an object,
    - hashCode(Object o)
    - hash(Object... values)
  - returning a **string representation** for an object,
    - toString(Object o)
    - toString(Object o, String nullDefault)

Hashset and Hashmap

- Any collection that depends on hashing requires both equality testing and hash codes.
- If you **implement hashCode,** you must **also implement equals.**

```
public int hashCode() {
    return Objects.hash(familyName, givenName);
}
```

- Hash codes are not unique. (There might be collisions.)
  - The hash function should provide a good distribution.
  - The probability that two different objects have the same hash code should be small.
- If two objects are **equal** then they **must have** the **same hash code**.
  - Person objects which are not equal can have the same hash code.
  - Person objects which are equal must have the same hash code.
- Collections that use hashing show very good runtime complexity if the hash function provides a good distribution (and is reasonably fast).
  - The average case is O(1) for search, insert, delete operations.

Stack – Push Top

- **Last in** first out (LIFO).

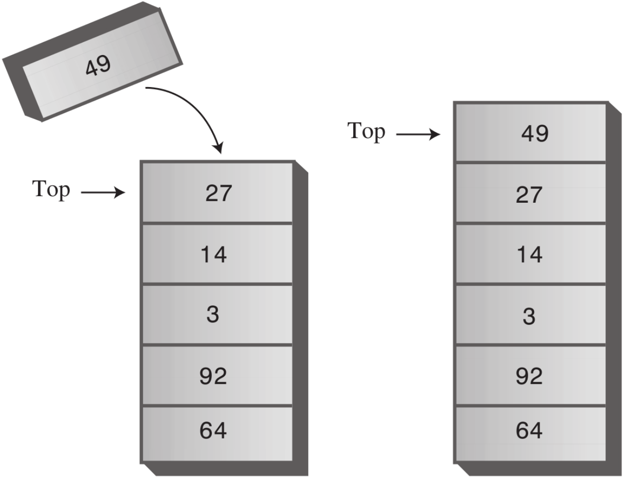


Figure: New item pushed on stack

Stack – Pop Top

- Last in **first out** (LIFO).

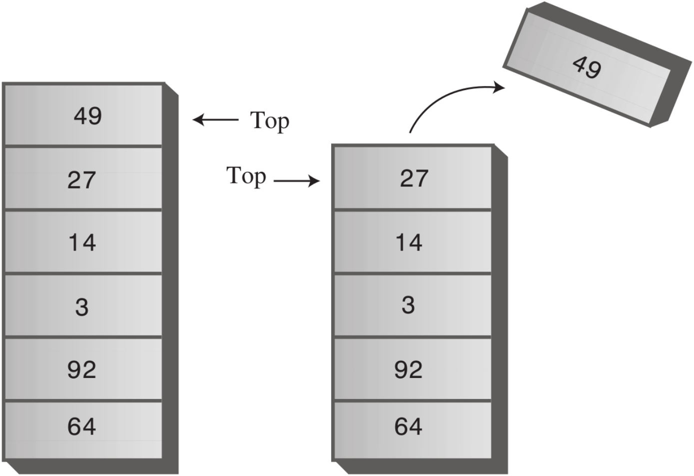


Figure: Top item popped from stack

Stack – Implementation in Java

- A generic stack can be implemented as a recursive data structure:

```
public class Stack<T> {
    private T top;
    private Stack<T> tail;
    ...
    public void push(T elem) {
        tail = new Stack<T>(top, tail);
        top = elem;
    }
    public T pop() {
        T ret = top;
        top = tail.top;
        tail = tail.tail;
        return ret;
    }
    public boolean isEmpty() {
        return tail == null;
    }
}
```

Queue – FIFO

Figure: A queue of some people

Queue – Add Rear

- First in first out (FIFO).

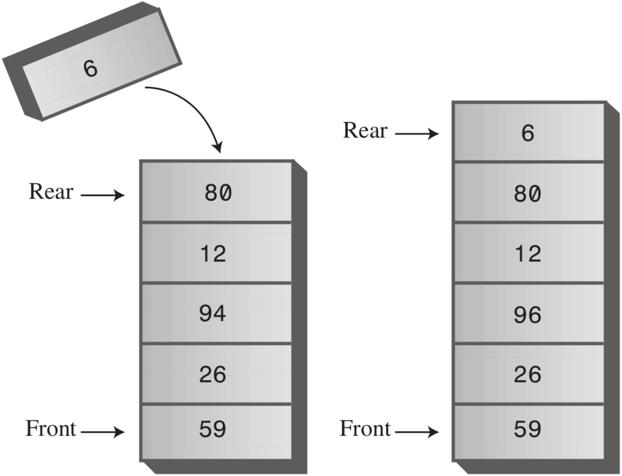


Figure: New item inserted at rear of queue

Queue – Remove Front

- First in first out (FIFO).

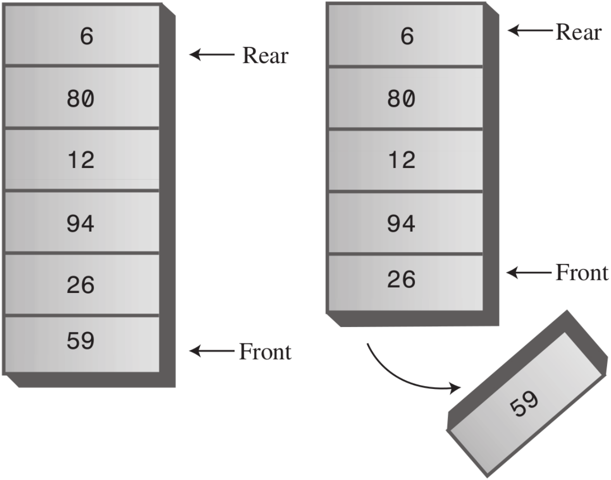


Figure: Item removed from front of queue

Circular Queue

- Bounded queues can be implemented as rings.

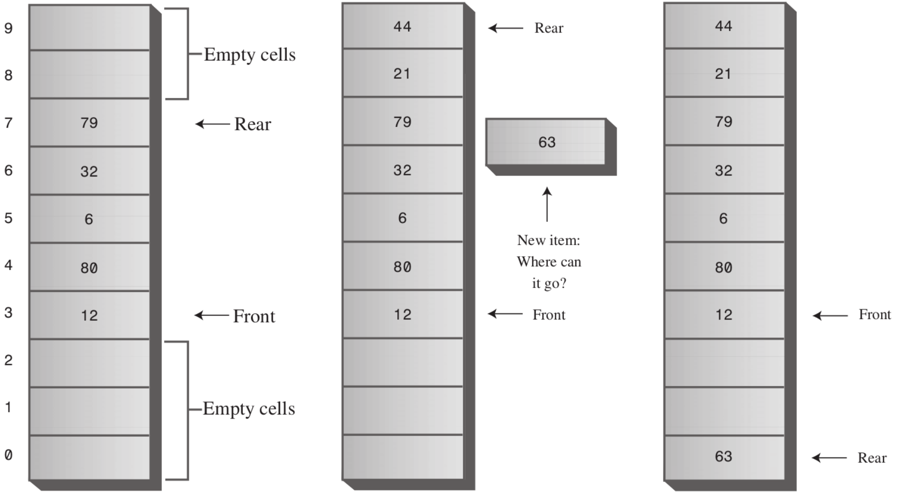


Figure: Rear and front pointer modulo length

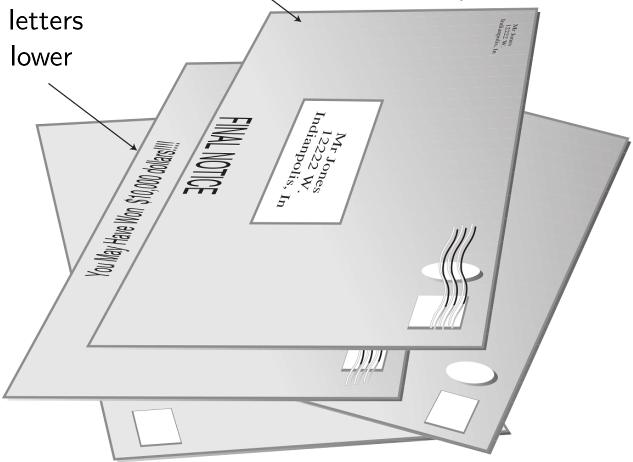
```
public class Queue<T> {
    private Object[] q;
    private int front = -1;
    private int rear = -1;

    public Queue(int maxSize) {
        q = new Object[maxSize];
    }
    public void insert(T elem) {
        q[(++rear) % q.length] = elem;
    }
    public T remove() {
        return (T) q[(++front) % q.length];
    }
    public int size() {
        return rear - front;
    }
}
```

Priority Queue

Letter on top is always processed first

More urgent letters are inserted higher

Less urgent letters are inserted lower



Priority Queue – Insert

- Most important first out.

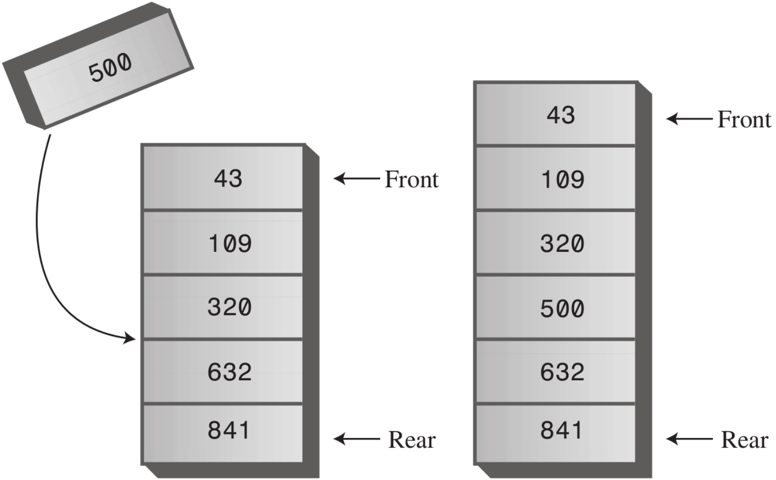


Figure: New item inserted in priority queue

Priority Queue – Remove

- Most important first out.

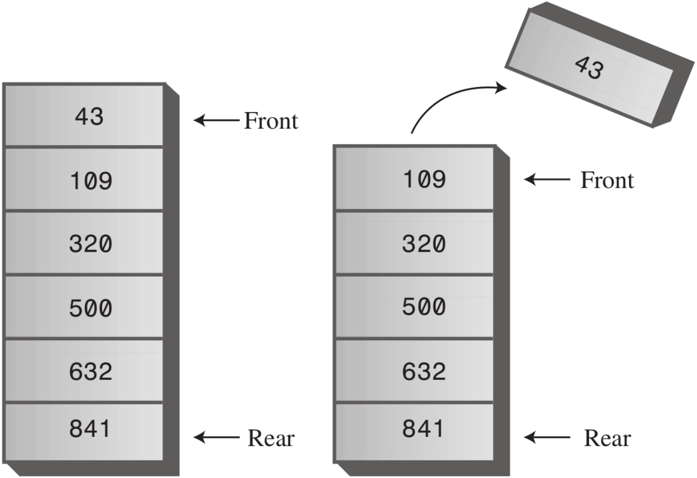


Figure: Most important items removed from front of priority queue

Priority Queue – Implementation

```
public class PriorityQueue {
    private int maxSize;
    private int[] q;
    private int size;

    public PriorityQueue(int maxSize) {
        this.maxSize = maxSize;
        this.q = new int[maxSize];
    }
    public void insert(int item) {
        int j = size++;
        while (--j >= 0 && item > q[j])
            q[j + 1] = q[j]; // shift item up
        q[j + 1] = item;
    }
    public int remove() {
        return q[--size];
    }
    public int size() { return size; }
}
```

Backtracking – Introduction

- **Given:** A problem which has a set of **constraints**.
- **Find:** A solution that fulfills all the constraints.
- We can represent the search space by a **tree**:
  - The root of the tree represents 0 choices.
  - Nodes at depth 1 represent first choice.
  - Nodes at depth 2 represent the second choice, etc.
  - A path from the root to a leaf represents a candidate solution.

Example – Sum of Subsets

- **Given:** n positive integers $w_1, \ldots, w_n$ and a positive integer S.
- **Find:** All subsets of $w_1, \ldots, w_n$ that sum to S.
- It is a problem which has a set of **constraints:**
  - Iterate the subsets of $w_1, \ldots, w_n$.
  - The constraint is that the subset has to sum up to S.
- Example: $n = 3, w_1 = 2, w_2 = 4, w_3 = 6$, and $S = 6$.
  - Subsets: $\{\}, \{2\}, \{4\}, \{6\}, \{2,4\}, \{2,6\}, \{4,6\}, \{2,4,6\}$.
  - Two solutions $\{6\}$ and $\{2,4\}$ fulfill the constraint $S = 6$.

Example – Tree of Sums of Subsets

- We draw a binary tree.
  - Nodes: Represent the sum.
  - Edges: Left for include w_i and right for exclude w_i.
  - Leafs: Are the possible combinations.

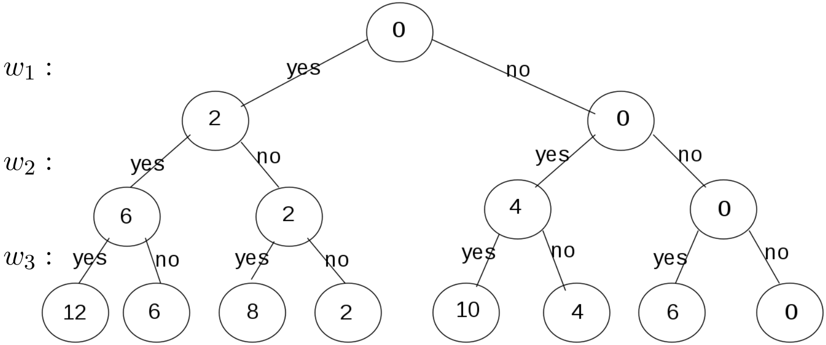


Figure: Tree of sums of subsets

Example – Depth First Search

- Problem can be solved using depth first search of the tree.
- If a node is a leaf, check if the solution satisfies the constraints.
- Backtracking:
  - If a node can not lead to a solution, then go back to the parent.
  - Follow one of the edges and after going back try the other one.
- Backtracking can be implemented by recursion.

Sum of Subsets – Recursive Solution

```
private int[] numbers;
private boolean[] include;
...
public void findSubset(int sum) {
    findSubset(0, 0, sum);
}
private void findSubset(int lvl, int nodeSum, int sum) {
    if (lvl == numbers.length) {
        if (nodeSum == sum) solutionFound();
    } else if (nodeSum <= sum) {
        findSubset(lvl + 1, nodeSum, sum);
        include[lvl] = true;
        findSubset(lvl + 1, nodeSum + numbers[lvl], sum);
        include[lvl] = false;
    }
}
```

- The variable lvl is the current depth.
- The boolean array $include$ is the current path of decisions.
  - true stands for yes and false for no.

Backtracking – 8 Queens Problem

- **Problem:** How to place 8 queens on chess-board, such that they do not capture each other.
- **Solution:** Use backtracking.
- **Approach:** Two queens at the **same row** cannot be a solution.

Chess Board

Q_0			Q_j				

Backtracking – 8 Queens Problem

- **Problem:** How to place 8 queens on chess-board, such that they do not capture each other.
- **Solution:** Use backtracking.
- **Approach:** Two queens at the **same row** cannot be a solution.

Chess Board

Q_0							
			Q_1				

q[ ]
0
3

~~Same row~~

- It suffices to use an array q[ ] with the position of a queen per row.

Backtracking – 8 Queens Problem

- **Problem:** How to place 8 queens on chess-board, such that they do not capture each other.
- **Solution:** Use backtracking.
- **Approach:** Two queens at the **same column** cannot be a solution.

Chess Board

Q_0							
Q_1							

q[ ]

0
0

~~Same row~~

q[i] $\neq$ q[j]

- It suffices to use an array q[ ] with the position of a queen per row.

Backtracking – 8 Queens Problem

- **Problem:** How to place 8 queens on chess-board, such that they do not capture each other.
- **Solution:** Use backtracking.
- **Approach:** Two queens at the **same diagonal** cannot be a solution.

Chess Board

Q_0							
	.						
		.				Q_2	
			.		.		
				Q_4			

q[ ]
0
6
4

~~Same row~~

q[i] $\neq$ q[j]

|q[j] $-$ q[i]|$\neq j - i$

- It suffices to use an array q[ ] with the position of a queen per row.

8 Queens – Recursive Solution

- We solve a more general problem, the n queens problem:

```java
private int[] q = new int[n];  // n = 8 for 8 queens

private boolean isConsistent(int n) {
    for (int i = 0; i < n; i++) {
        if (q[i] == q[n]) return false;
        if (Math.abs(q[i] - q[n]) == n - i) return false;
    }
    return true;
}
public void solveBoard() { solveBoard(0); }
private void solveBoard(int n) {
    if (n == q.length) solutionFound();
    else
        for (int i = 0; i < q.length; i++) {
            q[n] = i;
            if (isConsistent(n)) solveBoard(n + 1);
        }
}
```

Exercise

The priority queue from the lecture features fast removal of the high-priority item O(1) but slow insertion of new items O(n).

Modify the priority queue from the lecture:

- Modify the runtime behavior such that the priority queue guarantees O(1) insertion time but slower removal of the high-priority item O(n).
- Make the priority queue generic (like the circular queue from the lecture).
  - Therefore, you should only allow types which implement the Comparable interface.
  - The priority is determined by the method compareTo.
- Override the method toString from java.lang.Object such that it returns the string representation of the contents of the priority queue.

See the guidance for this exercise on the Moodle page.

- Find a way through a maze.
- Create a class Maze which reads a 2D maze from a text file.
  - Provide a public constructor which has java.io.File as its argument.
  - Use a recursive backtracking approach like in the lecture.

```
#S##########   #S##########
# #        #   #.#........#
#   ###### #   #...######.#
# ###      #   # ###......#
# # # ######   # # #.######
#   #    # #   #   #....# #
### #### # #   ### ####.# #
#   #  #   #   #   #  #...#
# # ## ### #   # # ## ###.#
#      #   #   #      #...#
######## ###   ########.###
```

The letter S denotes the start position.

Use the dot to draw the way out of the maze.

- Provide a documentation which describes the algorithm.