Berner Fachhochschule - Technik und Informatik

Object-Oriented Programming 2

Topic 2: Collections

Prof. Rolf Haenni & Prof. Annett Laube

FS 2016

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

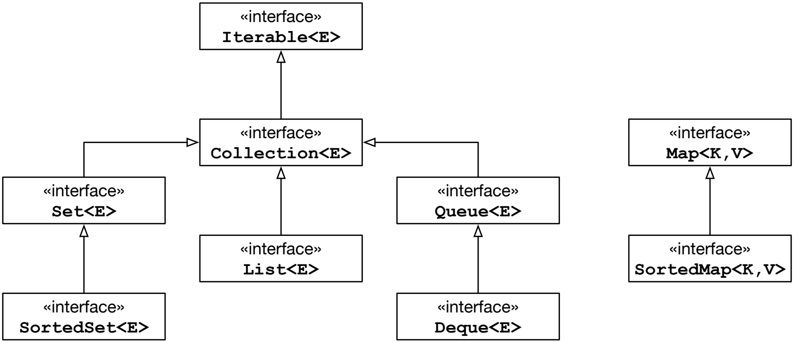
Outline

Outline

The Collection Framework

- A collection represents an iterable group of elements
- There are various types of collections, depending on how the elements are stored and on how manipulations work
  - → Some allow duplicate elements, others do not
  - → Some are ordered, others are unordered
  - → Some have a fixed (or maximal) size, others have a variable size
  - → Some can be modified, others are immutable
- The generic interface `Collection<E>` is the root of the Java Collections Framework (JCF)
- The JCF exists since Java 1.2 (generic types added in Java 5)
- The interfaces `Map<K,V>` and `SortedMap<K,V>` do not inherit from `Collection<E>`, but are still members of the JCF

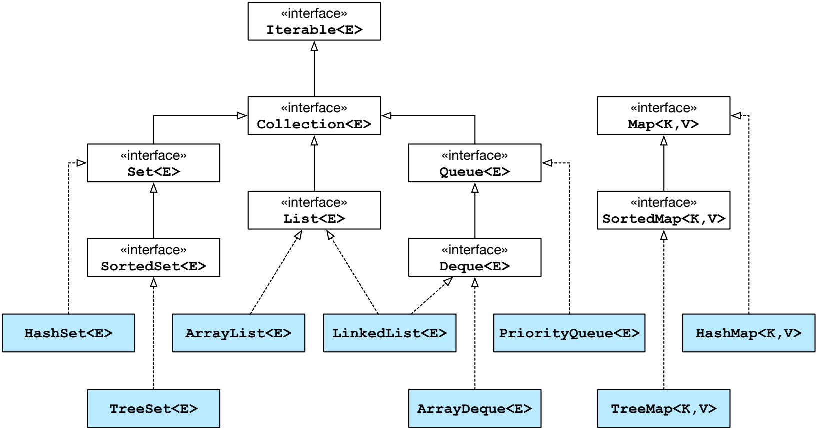
Interfaces of the Collection Framework



The Collection Interface

- The interface `Collection` defines a number of methods that perform basic operations
  - → `boolean isEmpty()`: Returns `true` if the collection is empty
  - → `int size()`: Returns the collection's number of elements
  - → `boolean add(E e)`: Ensures that the collection contains e
  - → `boolean remove(Object e)`: Removes e from the collection
  - → `boolean contains(Object e)`: Returns `true` if the collection contains e
  - → `void clear()`: Removes all elements form the collection
  - → `Object[] toArray()`: Returns an array containing all of the elements from the collection
- An additional method is inherited from `Iterable<E>`
  - → `Iterator<E> iterator()`: Returns an iterator over all elements

Classes of the Collection Framework



Classes of the Collection Framework

- A class that implements `Collection<E>` is supposed to provide at least two constuctors for constructing ...
  - → an empty collection
  - → a collection containing all elements of another collection
- Examples:
  - → `ArrayList()`, `ArrayList(Collection<? extends E> c)`
  - → `LinkedList()`, `LinkedList(Collection<? extends E> c)`
  - → `HashSet()`, `HashSet(Collection<? extends E> c)`
  - → etc.

Example of Using Collections I

```java
public class CollectionTester {

  public static void main(String[] args) {

    Collection<String> c1 = new ArrayList<>();
    c1.add("Hello");
    c1.add("World");
    c1.size(); // returns 2
    System.out.println(c1); // prints "[Hello, World]"

    Collection<String> c2 = new LinkedList<>(c1);
    c2.add("!");
    c2.add("!");
    c2.size(); // returns 4
    c2.remove("World");
    c2.size(); // returns 3
    System.out.println(c2); // prints "[Hello, !, !]"
```

Example of Using Collections II

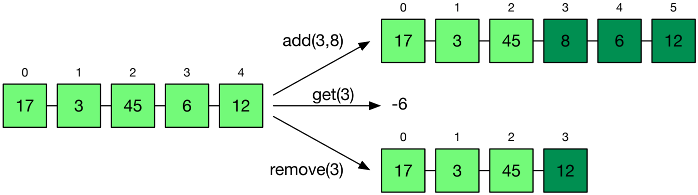
```
    Collection<String> c3 = new HashSet(c2);
    c3.add("World");
    c3.add("World");
    c3.size(); // returns 3
    System.out.println(c3); // prints "[!, Hello, World]"
  }

}
```

Outline

Lists

- The interfaces `List`, `Queue`, and `Deque` extend the interface `Collection` by introducing ...
  - → an ordering of the elements stored in the collection
  - → additional access methods
- A list offers positional access, i.e., manipulations of elements are based on their numerical position (index) in the list



The List Interface

- `E get(int i)`: Returns the element at the specified index `i`
- `E set(int index, E element)`: Replaces the element at the specified index `i` with `e`
- `void add(int i, E e)`: Inserts `e` at the specified index `i`
- `E remove(int i)`: Removes the element at the specified index `i`
- `int indexOf(Object e)`: Returns the index of the first occurrence of `e`
- `int lastIndexOf(Object e)`: Returns the index of the last occurrence of `e`

ArrayList vs. LinkedList

- The main difference between the classes `ArrayList` and `LinkedList` is their internal way of storing the elements
  - → `ArrayList` uses internally an array (which needs to be replaced by a bigger one if it gets full)
  - → `LinkedList` uses internally a doubly linked chain of nodes, which carry the elements
- This has implications on the running times of some methods
  - → `ArrayList` allows for fast random access, but adding or removing elements may require existing elements to be shifted
  - → `LinkedList` allows for constant-time insertions or removals, but only sequential access
- Recommendation: use `ArrayList` except in cases where insertions or removals are the dominant operations

Example of Using Sets I

```
public class ListTester {

  public static int ROUNDS = 500000;

  public static void main(String[] args) {

    List<String> l1 = new ArrayList<>();
    System.out.println("ArrayList: start...");
    for (int i = 1; i <= ROUNDS; i++) {
      l1.add(0, "Hello");
    }
    System.out.println("done");

    List<String> l2 = new LinkedList<>();
    System.out.println("LinkedList: start...");
    for (int i = 1; i <= ROUNDS; i++) {
      l2.add(0, "World");
```

Example of Using Sets II

```
    }
    System.out.println("done");
  }

}
```

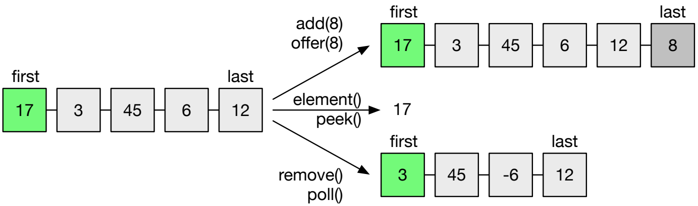
Outline

Queues and Deques (and Stacks)

- A queue offers first-in-first-out access (FIFO) to its elements
  - → Elements are added to the back of the queue
  - → Elements are removed from the front of the queue
- A deque (short form for "double ended queue", pronounced as "deck") offers access to both of its extremities
  - → Elements are added to the front or the back of the deque
  - → Elements are removed from the front or the back of the deque
- A deque also includes typical methods of a stack, which offers last-in-first-out access (LIFO)
  - → Elements are added to the front (push)
  - → Elements are removed from the front (pop)

The Queue Interface

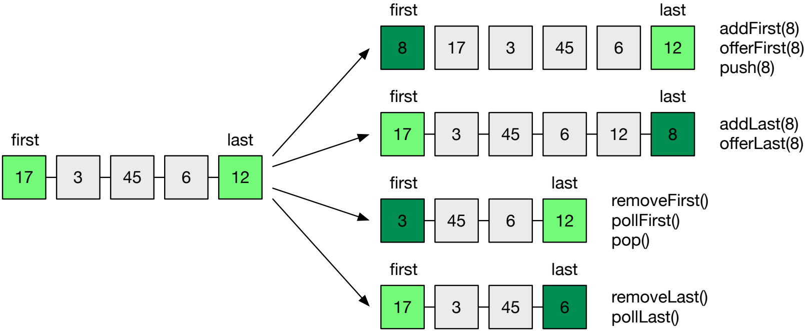
- `E element()`, `E peek()`: Retrieves, but does not remove, the first element of the queue
- `boolean add(E e)`, `boolean offer(E e)`: Inserts e into the queue if it is possible
- `E remove()`, `E poll()`: Retrieves and removes the first element of the queue



The Deque Interface

- `E getFirst()`, `E peekFirst()`: Retrieves, but does not remove, the first element of the deque
- `E getLast()`, `E peekLast()`: Retrieves, but does not remove, the last element of the deque
- `void addFirst(E e)`, `void offerFirst(E e)`, `void push(E e)`: Inserts e at the front of the deque
- `void addLast(E e)`, `void offerLast(E e)`: Inserts e at the end of the deque
- `E removeFirst()`, `E pollFirst()`, `E pop()`: Retrieves and removes the first element of the deque
- `E removeLast()`, `E pollLast()`: Retrieves and removes the last element of the deque

The Deque Interface



ArrayDeque vs. PriorityQueue

- In a `ArrayDeque`, the order of the elements is determined by the sequence of insertion operations
- In a `PriorityQueue`, the order of the elements is determined by their natural order or by specifying a comparator
  - → `PriorityQueue()`
  - → `PriorityQueue(Comparator<? super E> comparator)`
- Another difference is their internal way of storing the elements
  - → `ArrayDeque` uses internally an array (similar to `ArrayList`)
  - → `PriorityQueue` uses internally a heap
- All critical operations are efficient: average $O(1)$ for `ArrayDeque`, worst-case $O(\log n)$ for `PriorityQueue`

Example of Using Queues I

```
public class QueueTester {

  public static void main(String[] args) {

    Queue<String> p1 = new ArrayDeque<>(); // Insertion order
    p1.add("Peter");
    p1.add("John");
    p1.add("Tom");
    p1.add("Andrew");
    while (!p1.isEmpty()) {
      System.out.println(p1.remove());
    } // Loop prints "Peter", "John", "Tom", "Andrew"

    Queue<String> p2 = new PriorityQueue<>(); // Natural order
    p2.add("Peter");
    p2.add("John");
    p2.add("Tom");
```

Example of Using Queues II

```
p2.add("Andrew");
while (!p2.isEmpty()) {
  System.out.println(p2.remove());
} // Loop prints "Andrew", "John", "Peter", "Tom"

// String length comparator
Comparator<String> c = new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    if (s1.length() < s2.length()) {
      return -1;
    }
    if (s1.length() > s2.length()) {
      return 1;
    }
    return 0;
  }
};
```

Example of Using Queues III

```
    Queue<String> p3 = new PriorityQueue<>(c);
    p3.add("Peter");
    p3.add("John");
    p3.add("Tom");
    p3.add("Andrew");
    while (!p3.isEmpty()) {
      System.out.println(p3.remove());
    } // Loop prints "Tom", "John", "Peter", "Andrew"
  }
}
```

Outline

Sets

- A set in Java is a collection that contains no duplicates
- Duplicates are elements `e1` and `e2` such that `e1.equals(e2)`
- Compared to a list, there are two important differences:
  - → A list may contain duplicates
  - → Elements in a list are ordered (positional access)
- Java sets correspond to sets in mathematics, except that Java sets can be modified
  - → `boolean contains(Object x)`: tests if $x \in X$
  - → `boolean containsAll(Collection<?> y)`: tests if $Y \subseteq X$
  - → `boolean addAll(Collection<? extends E> y)`: computes $X \cup Y$
  - → `boolean retainAll(Collection<?> y)`: computes $X \cap Y$
  - → `boolean removeAll(Collection<?> y)`: computes $X \setminus Y$

Sorted Sets

- A sorted set in Java is a set that provides a total order on its elements, which is determined by their natural order or by specifying a comparator
- The interface `SortedSet` provides some additional methods that exploit the existence of an order
  - → `E first()`: Returns the minimal element currently in this set
  - → `E last()`: Returns the maximal element currently in this set
  - → `SortedSet<E> tailSet(E e)`: Returns elements that are greater than or equal to e
  - → `SortedSet<E> headSet(E e)`: Returns elements that are strictly less than e
  - → `SortedSet<E> subSet(E e1, E e2)`: Returns the elements that range from e1 (inclusive) to e2 (exclusive)

HashSet vs. TreeSet

- In a `HashSet`, the order of the elements is unspecified
- In a `TreeSet`, the order is determined by their natural order or by specifying a comparator (similar to `PriorityQueue`)
  - → `TreeSet()`
  - → `TreeSet(Comparator<? super E> comparator)`
- Another difference is their internal way of storing the elements
  - → `HashSet` uses internally a hash table (array)
  - → `TreeSet` uses internally a red-black tree
- All critical operations are efficient: average $O(1)$ for `HashSet`, worst-case $O(\log n)$ for `TreeSet`

Example of Using Sets I

```java
public class SetTester {

  public static void main(String[] args) {

    Set<Integer> s1 = new HashSet<>(Arrays.asList(new Integer
         []{1,2,3,4}));
    Set<Integer> s2 = new HashSet<>(Arrays.asList(new Integer
         []{3,4,5}));
    Set<Integer> s3 = new HashSet<>(Arrays.asList(new Integer
         []{2,5,6}));

    s2.addAll(s3);
    System.out.println(s2); // s2 = s2 cup s3 = {2,3,4,5,6}

    s2.retainAll(s1); // s2 = s2 cap s1 = {2,3,4}
    System.out.println(s2); // s2 = s2 cap s1 = {2,3,4}
```

Example of Using Sets II

```
    s2.removeAll(s3);
    System.out.println(s2); // s2 = s2 minus s3 = {3,4}

    s2.add(3);
    s2.add(5);
    s2.add(5);
    System.out.println(s2); // s2 = s2 cup {3,5,5} = {3,4,5}

    SortedSet<Integer> s = new TreeSet<>(Arrays.asList(new
        Integer[]{1,3,7,2,5,4,7}));
    System.out.println(s.headSet(5)); // {1,2,3,4}
    System.out.println(s.tailSet(5)); // {5,7}
    System.out.println(s.subSet(2,5)); // {2,3,4}
  }
}
```

The Equals Method

- Duplicates in sets and sorted sets are defined differently:
  - → Sets: `e1.equals(e2)`
  - → Sorted sets:
    `e1.compareTo(e2) == 0` or `c.compareTo(e1,e2) == 0`
- The method `equals(Object obj)` as defined in `Object` only performs an identity check `e1 == e2`
- **Important**: If you need a `Set<MyClass>`, you must override `equals(Object obj)` in your class `MyClass`
  - → `e.equals(null) == false`
  - → `e.equals(e) == true` (Reflexivity)
  - → `e1.equals(e2) == e2.equals(e1)` (Symmetry)
  - → `e1.equals(e2) && e2.equals(e3) == e1.equals(e3)` (Transitivity)

Auto-Generated Equals Method I

```
public class MyClass {

  private int x;
  private String y;

  @Override
  public boolean equals(Object obj) {
    if (obj == null) {
      return false;
    }
    if (this == obj) {
      return true;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    MyClass other = (MyClass) obj;
```

Auto-Generated Equals Method II

```
    if (this.x != other.x) {
      return false;
    }
    if (this.y == null) {
      if (other.y != null) {
        return false;
      }
    } else if (!this.y.equals(other.y)) {
      return false;
    }
    return true;
  }
}
```

The HashCode Method

- A set implemented based on hash tables (e.g. `HashSet`) requires a hash function that is compatible with `equals`
- The method `hashCode()` as defined in `Object` only transforms the memory address of the object into an integer
- **Important**: If you need a `HashSet<MyClass>`, you must override `hashCode()` in your class `MyClass`
  - → If `e1.equals(e2)`, then `e1.hashCode() == e2.hashCode()`
  - → Otherwise, `e1.hashCode() !=e2.hashCode()` with high probability
- General recommendation: Always override `hashCode()` when you override `equals(Object obj)`

Auto-Generated HashCode Method

- Let $h_1, \ldots, h_n$ be the hash codes of the n fields of an object x, then a good practice is to define $h_0 = 1$ and compute
$$hashCode(x) = \sum_{i=0}^{n} h_i \cdot 31^{n-i}$$
- Example: Let $n = 2$, then $hashCode(x) = 31^2 + 31h_1 + h_2$

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + x;
  result = prime * result + ((y == null) ? 0 : y.hashCode());
  return result;
}
```

Outline

Maps and Sorted Maps

- A map in Java is an object that maps keys to values (similar to a function $f : \{k_1, \ldots, k_n\} \rightarrow V$ in mathematics)

$$
\begin{gathered}
k_1 \mapsto v_1 = f(k_1) \\
\vdots \\
k_n \mapsto v_n = f(k_n)
\end{gathered}
$$

- A map cannot contain duplicate keys (each key maps to at most one value), but different keys can map to the same value
- The goal of a map is to provide a key-based access to the values stored in the map
- A sorted map is a map that provides a total ordering on its keys (similar to `SortedSet`)

The Map Interface

- Since `Map<K,V>` does not inherit from `Collection<E>`, it defines its own methods `isEmpty()`, `size()`, and `clear()`
  - → `boolean containsKey(Object key))`: Returns `true` if the map contains a mapping with the given value
  - → `boolean containsValue(Object key))`: Returns `true` if the map contains a mapping for the given key
  - → `V get(Object key)`: Returns the value of the mapping for the given key (or `null` if the key does not exist)
  - → `V put(K key, V value)`: Adds a new mapping or replaces the value of an existing mapping
  - → `V remove(Object key)`: Removes the mapping if the given key exists
  - → `V replace(K key, V value)`: Replaces the value in the mapping if the given key exists
  - → `Collection<V> values()`: Returns a collection of all values
  - → `Set<K> keySet()`: Returns a set of all keys

HashMap vs. TreeMap

- In a `HashMap`, the order of the keys is unspecified
- In a `TreeMap`, the order is determined by their natural order or by specifying a comparator (similar to `TreeSet`)
  - → `TreeMap()`
  - → `TreeMap(Comparator<? super E> comparator)`
- Another difference is their internal way of storing the elements
  - → `HashMap` uses internally a hash table (array)
  - → `TreeMap` uses internally a red-black tree
- All critical operations are efficient: average $O(1)$ for `HashMap`, worst-case $O(\log n)$ for `TreeMap`

Example of Using Maps I

```
public class MapTester {

  public static void main(String[] args) {

    Map<Integer,String> map = new HashMap<>();
    map.put(1, "Hello");
    map.put(5, "World");
    map.put(7, "Hello");
    map.put(10, "Hello World");
    map.put(10, "Hello World!"); // replaces previous value

    System.out.println(map.keySet()); // prints [1, 5, 7, 10]
    System.out.println(map.values()); // prints [Hello, World,
        Hello, Hello World!]

    String str1 = map.get(1);
    if (str1 != null) {
```

Example of Using Maps II

```
      System.out.println(str1); // prints "Hello"
    }
    String str2 = map.get(2);
    if (str2 != null) {
      System.out.println(str2); // nothing is printed
    }

    // the following code is less efficient
    int key = 1;
    if (map.containsKey(key)) {
      System.out.println(map.get(key)); // prints "Hello"
    }
  }
}
```

Outline

Iterators

- An iterator in Java is an object that helps iterating through all elements in a collection
- The interface `Iterator<E>` defines the following methods:
  - → `boolean hasNext()`: Returns `true` if the iteration has more elements
  - → `E next()`: Returns the next element in the iteration
- Note that an iterator only allows a single iteration
- The following is a typical example of usage:

```
Iterator<String> iterator = ...;
while(iterator.hasNext()) {
  System.out.println(iterator.next());
}
```

Array Iterator I

```
public class ArrayIterator<E> implements Iterator<E> {

  private E[] array;
  private int currentIndex;

  public ArrayIterator(E[] array) {
    this.array = array;
    this.currentIndex = 0;
  }

  @Override
  public boolean hasNext() {
    return this.currentIndex < this.array.length;
  }

  @Override
  public E next() {
```

Array Iterator II

```
    if (!this.hasNext()){
      throw new NoSuchElementException();
    }
    return this.array[this.currentIndex++];
  }

}
```

Reverse Array Iterator I

```
public class ReverseArrayIterator<E> implements Iterator<E> {

  private E[] array;
  private int currentIndex;

  public ReverseArrayIterator(E[] array) {
    this.array = array;
    this.currentIndex = array.length - 1;
  }

  @Override
  public boolean hasNext() {
    return this.currentIndex >= 0;
  }

  @Override
  public E next() {
```

Reverse Array Iterator II

```
    if (!this.hasNext()) {
      throw new NoSuchElementException();
    }
    return this.array[this.currentIndex--];
  }

}
```

Iterator Tester I

```java
public class IteratorTester {

  public static void main(String[] args) {

    Integer[] values = new Integer[] { 1, 3, 3, 5, 6, 9 };

    Iterator<Integer> i1 = new ArrayIterator<>(values);
    while (i1.hasNext()) {
      System.out.print(i1.next() + " ");
    }
    System.out.println();
    // prints 1 3 3 5 6 9

    Iterator<Integer> i2 = new ReverseArrayIterator<>(values);
    while (i2.hasNext()) {
      System.out.print(i2.next() + " ");
    }
```

Iterator Tester II

```
    System.out.println();
    // prints 9 6 5 3 3 1
  }

}
```

List Iterators

- The `ListIterator<E>` interface extends `Iterator<E>` with methods for traversing a list backwards
  - → `boolean hasPrevious()`: Returns `true` if a backward iteration has more elements
  - → `E previous()`: Returns the previous element in a backward iteration
- It also provides methods for adding, removing, or replacing element in the list
  - → `void add(E e)`: Inserts an element into the list
  - → `void remove()`: Removes the current element from the list
  - → `void set(E e)`: Replaces the current element in the list

Obtaining Iterators from Collections

- An iterator can be obtained from a collection using a method inherited from `Iterable<E>`
  - → `Iterator<E> iterator()`: Returns an iterator over all elements in the collection (the order is unspecified)
- Similarly, a list iterator can be obtained from a list
  - → `ListIterator<E> listIterator()`: Returns a list iterator over all elements in the list
- Note that the Java for-each loop work for any class that implements the `Iterable<E>` interface

```
Iterable<Integer> values =Arrays.asList(new Integer
    []{1,3,3,5,7,9});
int sum = 0;
for (Integer value: values) {
  sum = sum + value;
}
```

Outline

The Minimax-Algorithm

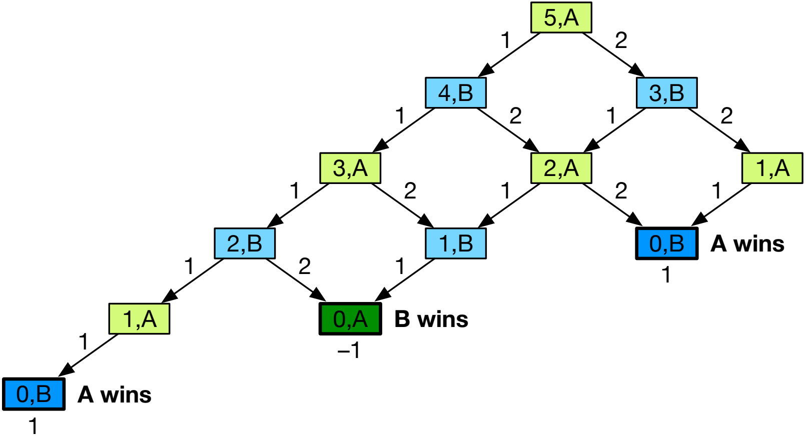
- The minimax algorithm is an optimal game playing algorithm for two player games such as tic-tac-toe, chess, go, uril, ...
- Let A and B be the two players and S the finite set of possible game states, where ...
  - → $s_0 \in S$ is the initial state
  - → $S^* \subseteq S$ are final states, in which the game ends
- For all final states $s^* \in S^*$, the winner of the game is defined by a function $E : S^* \rightarrow [-1, 1]$, where

$$E(s^*) = \begin{cases} 1, & \text{if } A \text{ wins} \\ 0, & \text{if the game ends as draw} \\ -1, & \text{if } B \text{ wins} \end{cases}$$

Example: Subtraction Game

- In a (k, n)-subtraction game, A and B take turns in removing up to k objects from a pile with initially n objects
  - → A begins
  - → Whoever removes the last object from the pile wins
- Example: $(2, 5)$-subtraction game
  - → States: $S = \{(5, A), (3, A), \ldots, (0, A), (4, B), \ldots, (0, B)\}$
  - → Initial state: $s_0 = (5, A)$
  - → Final states: $S^* = \{(0, A), (0, B)\}$
  - → $E(s^*) = \begin{cases} 1, & \text{for } s^* = (0, B) \\ -1, & \text{for } s^* = (0, A) \end{cases}$

Example: Subtraction Game



Minimax Algorithm: General Idea

- Let $next(s) \subseteq S$ be the reachable states from $s \in S$
- Example: $(2,5)$-subtraction game
  - → $next((5,A)) = \{(4,B),(3,B)\}$
  - → $next((4,B)) = \{(3,A),(2,A)\}$

    $\vdots$
  - → $next((0,A)) = next(0,B) = \{\}$
- The minimax algorithm extends E from $E : S^* \to [-1,1]$ to $F : S \to [-1,1]$ by computing $F(s)$ recursively by for all $s \in S$

$$F(s) = \begin{cases} E(s), & \text{if } s \in S^* \\ \max\{F(s') : s' \in next(s)\}, & \text{if it is } A\text{'s turn} \\ \min\{F(s') : s' \in next(s)\}, & \text{if it is } B\text{'s turn} \end{cases}$$

Minimax Algorithm: Pseudocode

Algorithm: Minimax(s, d)

if $s \in S^*$ **then** // game ends
  return $E(s)$

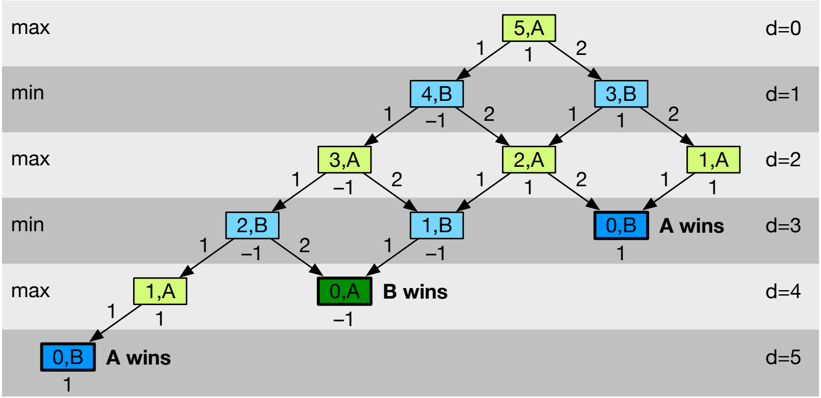
if $d \bmod 2 = 0$ **then** // A's turn
  $m \leftarrow -1$
  for $s' \in next(s)$ **do**
    $m \leftarrow \max(m, \text{Minimax}(s', d+1))$

else // B's turn
  $m \leftarrow 1$
  for $s' \in next(s)$ **do**
    $m \leftarrow \min(m, \text{Minimax}(s', d+1))$

return m

Example: Subtraction Game

Initial call: $\text{Minimax}((5, A), 0)$
Return value: $F((5, A)) = 1$, i.e. A wins by removing 2 objects



Minimax Algorithm: Performance

- The branching factor b of a game is the average number legal moves (children in the game tree)
- Examples:
  - → (k, n)-subtraction game: $b \leq k$, $h \leq n$
  - → Chess: $b \approx 35$, $h \approx 80$
  - → Go: $b \approx 250$
- The minimax algorithms runs in $O(b^h)$ time, where h denotes the height (maximal depth) of the game tree
- In other words, exploring the full game tree is impossible for most non-trivial games

Pruned Minimax Algorithm

- To apply the minimax algorithm to non-trivial games, the game tree exploration must be pruned
- The simplest pruning method is to stop the recursion when a maximal depth $d_{\max}$ is reached
- When the recursion stops at state $s \in S$, then a evaluation function $\tilde{E} : S \to [-1, 1]$ is applied to s
  - → For $s \in S^*$, let $\tilde{E}(s) = E(s)$
  - → Otherwise, let $\tilde{E}(s)$ be an estimate of the advantage of state s relative to A and B, such that $\tilde{E}(s) = 1$ means maximal advantage for A and $\tilde{E}(s) = -1$ maximal advantage for B
- The quality of the estimate $\tilde{E}(s)$ and $d_{\max}$ determine the quality and accuracy of the final minimax return value

Pruned Minimax Algorithm: Pseudocode

Algorithm: Minimax$(s, d, d_{\max})$

if $s \in S^*$ **or** $d = d_{\max}$ **then** // game ends
  return $\tilde{E}(s)$

if $d \bmod 2 = 0$ **then** // A's turn
  $m \leftarrow -1$
  for $s' \in next(s)$ **do**
    $m \leftarrow \max(m, \text{Minimax}(s', d+1, d_{\max}))$

else // B's turn
  $m \leftarrow 1$
  for $s' \in next(s)$ **do**
    $m \leftarrow \min(m, \text{Minimax}(s', d+1, , d_{\max}))$

return m

Defining the Evaluation Function

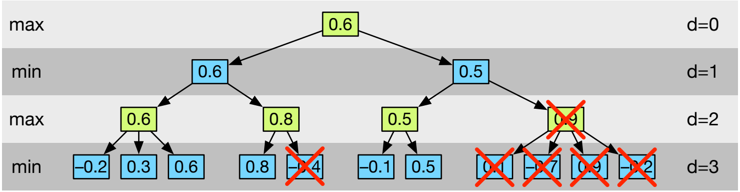
- One popular strategy for constructing an evaluation function $\tilde{E} : S \rightarrow [-1, 1]$ is as a weighted sum

$$\tilde{E}(s) = \frac{1}{W} \sum_{i=1}^{k} w_i \cdot \tilde{E}_i(s)$$

  of k individual evaluation criteria
- The value $w_i \in [0, 1]$ denotes the weight of criterion i and $W = \sum_{i=1}^{k} w_i$ denotes the total weight of all criteria
- $\tilde{E}_i : S \rightarrow [-1, 1]$ defines the evaluation function of criterion i
- Example from chess:
  $\tilde{E}(s) = 9 \cdot (Q - Q') + 5 \cdot \frac{R-R'}{2} + 3 \cdot \frac{B-B'}{2} + 3 \cdot \frac{N-N'}{2} + \frac{P-P'}{8} + \ldots$

Alpha-Beta Pruning

- There are many ways of optimizing the minimax algorithm
- The general idea is to prune branches of the game tree that will not influence the final minimax return value
- The simplest optimization is known as alpha-beta pruning



- The following version of the minimax algorithm is initially called with $Minimax(s_0, 0, d_{\max}, -1, 1)$

Alpha-Beta Pruning: Pseudocode I

Algorithm: Minimax$(s, d, d_{\max}, \alpha, \beta)$

if $s \in S^*$ **or** $d = d_{\max}$ **then** // game ends
    return $\tilde{E}(s)$

if $d \bmod 2 = 0$ **then** // A's turn
    $m \leftarrow -1$
    for $s' \in next(s)$ **do**
        $m \leftarrow \max(m, \text{Minimax}(s', d+1, d_{\max}, \alpha, \beta))$
        $\alpha \leftarrow \max(\alpha, m)$
        if $\alpha \geq \beta$ **then**
            return m // β cutoff

Alpha-Beta Pruning: Pseudocode II

```
else                                              // B's turn
    m ← 1
    for s' ∈ next(s) do
        m ← min(m, Minimax(s', d + 1, , d_max, α, β))
        β ← min(β, m)
        if α ≥ β then
            return m                              // α cutoff
return m
```