Important Dates.

- April 10 --- A4 due (Connect 4, minimax, trees)
- April 15 --- A5 due (Exercises on different topics, to be posted by May 28)
- April 22 --- Prelim 2.
- May 1 --- A6 due (Butterfly, graphs, search).
- May 12 --- Final exam.
Today’s topics

Connect 4.

- Use of **trees** (game-tree) and **recursion** to make a Connect 4 Al.
- **Mini-max.**

Java Collections Framework

- **Generic Data Types**
Game State and Tree

- **Game States:** $s_1, s_2, \ldots, s_5$
  (Also the nodes in the tree)
- **Actions:** edges in the tree
- **Leaf node:** ?
- **Depth of tree at node $s_1$:** ?

X’s turn

```
X O X
O O
X
```

“Game State $s_1$”

O’s turn

```
X O X
O O
X
```

“Game State $s_2$”

```
X O X
O O
X
```

“Game State $s_3$”

```
X O X
O X O
X
```

“Game State $s_4$”

```
X O X
O O O
X
```

“Game State $s_5$”
Games and Mini-Max

- **Minimizing the maximum possible loss.**

- Choose move which results in best state
  - Select highest expected score for you

- Assume opponent is playing optimally too
  - Will choose lowest expected score for you
Game Tree and Mini-Max

What move should \( x \) make?

X’s turn (max)

O’s turn (min)

X’s turn (max)
Properties of Mini-max

- $b$ possible moves and $m$ steps to finish game.
  - Time complexity?
    - $O(b^m)$
  - Space complexity?
    - $O(bm)$ (depth-first exploration)

For tic-tac-toe, $b \leq 9$, and $m \leq 9$.

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games!!
Mini-Max is used in many games!

- Stock Exchange!

Deep Blue vs. Kasparov chess matches

Deep Blue
IBM chess computer

Garry Kasparov
World Chess Champion

Connect Four
The Vertical Four-in-a-Row Checkers Game
Can we have a robot prepare a recipe?

- For example “Avogado”, an Italian dish.

Natural Language → Actions.

What do we need?

- Parsing (to understand natural language)
- Trees: Mini-max to figure out what actions it can do? (some of them lead to success and some of them to disaster)
Robot Programming
Today’s topics

Connect 4.
- Use of trees (game-tree) and recursion to make a Connect 4 AI.
- Mini-max.

Java Collections Framework
- Generic Data Types
Textbook and Homework

- Generics: Appendix B
- Generic types we discussed: Chapters 1-3, 15

- Homework: Use Google to find out about the old Java Vector collection type. Vector has been “deprecated”, meaning that it is no longer recommended and being phased out. What more modern type has taken over Vector’s old roles?
When using a collection (e.g. `LinkedList`, `HashSet`, `HashMap`), we generally have a single type `T` of elements that we store in it (e.g. `Integer`, `String`).

Before Java 5, when extracting an element, had to cast it to `T` before we could invoke `T`'s methods.

Compiler could not check that the cast was correct at compile-time, since it didn't know what `T` was.

Inconvenient and unsafe, could fail at runtime.

- Generics in Java provide a way to communicate `T`, the type of elements in a collection, to the compiler.
  - Compiler can check that you have used the collection consistently.
  - Result: safer and more-efficient code.
Example

```java
/** Return no. of chars in the strings in collection of strings c. */
static int cCount(Collection c) {
    int cnt = 0;
    Iterator i = c.iterator();
    while (i.hasNext()) {
        cnt += ((String)i.next()).length();
    }
    return cnt;
}
```

```java
/** Return no. of chars in c */
static int cCount(Collection<String> c) {
    int cnt = 0;
    Iterator<String> i = c.iterator();
    while (i.hasNext()) {
        cnt += ((String)i.next()).length();
    }
    return cnt;
}
```
/** Return no. of chars in the strings in * collection c of strings. */
static int cCount(Collection c) {
    int cnt = 0;
    Iterator i = c.iterator();
    while (i.hasNext())
        cnt = cnt + ((String)i.next()).length();
    return cnt;
}

/** Return the number of characters in collection c. */
static int cCount(Collection<String> c) {
    int cnt = 0;
    for (String s: c)
        cnt = cnt + s.length();
    return cnt;
}
Using Generic Types

- `<T>` is read, “of T”
  - Example: `Stack<Integer>` is read, “Stack of Integer”. Here the “T” is “Integer”.

- The type annotation `<T>` indicates that all extractions from this collection should be automatically cast to T

- Specify type in declaration, can be checked at compile time
  - Can eliminate explicit casts

- In effect, T is a parameter, but it does not appear where method parameters appear
Advantage of Generics

- Declaring `Collection<String> c` tells us something about variable `c` (i.e. `c` holds only Strings)
  - This is true wherever `c` is used
  - The compiler won’t compile code that violates this

- Without use of generic types, explicit casting must be used
  - A cast tells us something the programmer thinks is true at a single point in the code
  - The Java virtual machine checks whether the programmer is right only at runtime
Subtypes: Example

Stack<Integer> not a subtype of Stack<Object>

But Stack<Integer> is a subtype of Stack (for backward compatibility with previous Java versions)

```java
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
// Following gives compiler error
Stack<Object> t = s; ...
```

```java
Stack<Integer> s = new Stack<Integer>();
s.push(new Integer(7));
// Compiler allows this
Stack t = s;
```
To use interface `List<E>`, supply a type argument, e.g. `List<Integer>`

All occurrences of the type parameter (`E` in this case) are replaced by the type argument (`Integer` in this case)
public class Queue<T> extends AbstractBag<T> {
    private java.util.LinkedList<T> queue = new java.util.LinkedList<T>();

    public void insert(T item) { queue.add(item); }

    public T extract() throws java.util.NoSuchElementException {
        return queue.remove(); }

    public void clear() { queue.clear() }

    public int size() { return queue.size(); }
}
public class InsertionSort<Comparable<T>> {
    /** Sort x */
    public void sort(T[] x) {
        for (int i = 1; i < x.length; i++) {
            // invariant is: x[0..i-1] is sorted
            // Put x[i] in its rightful position
            T tmp = x[i];
            int j;
            for (j = i; j > 0 && x[j-1].compareTo(tmp) > 0; j = j-1)
                x[j] = x[j-1];
            x[j] = tmp;
        }
    }
}
Java **Collections** Framework

- **Collections**: holders that let you store and organize objects in useful ways for efficient access

- **Package** `java.util` includes interfaces and classes for a general collection framework

- **Goal**: conciseness
  - A few concepts that are broadly useful
  - Not an exhaustive set of useful concepts

- The collections framework provides
  - Interfaces (i.e., ADTs)
  - Implementations
JCF Interfaces and Classes

- **Interfaces**
  - Collection
  - Set (no duplicates)
  - SortedSet
  - List (duplicates OK)
  - Map (i.e., Dictionary)
  - SortedMap
  - Iterator
  - Iterable
  - ListIterator

- **Classes**
  - HashSet
  - TreeSet
  - ArrayList
  - LinkedList
  - HashMap
  - TreeMap
interface java.util.Collection<E>

- public int size(); Return number of elements
- public boolean isEmpty(); Return true iff collection is empty
- public boolean add(E x);
  - Make sure collection includes x; return true if it has changed (some collections allow duplicates, some don’t)
- public boolean contains(Object x);
  - Return true iff collection contains x (uses method equals)
- public boolean remove(Object x);
  - Remove one instance of x from the collection; return true if collection has changed
- public Iterator<E> iterator();
  - Return an Iterator that enumerates elements of collection
Iterators: How “foreach” works

The notation for(Something var: collection) { … } is syntactic sugar. It compiles into this “old code”:

```java
Iterator<E> _i =
    collection.iterator();
while (_i.hasNext()) {
    E var = _i.Next();
    . . . Your code . . .
}
```

The two ways of doing this are identical but the foreach loop is nicer looking.

You can create your own iterable collections
java.util.Iterator<E> (an interface)

- public boolean hasNext();
  - Return true if the enumeration has more elements

- public E next();
  - Return the next element of the enumeration
  - Throws NoSuchElementException if no next element

- public void remove();
  - Remove most recently returned element by next() from the underlying collection
  - Throws IllegalStateException if next() not yet called or if remove() already called since last next()
  - Throw UnsupportedOperationException if remove() not supported
Additional Methods of Collection\(<E>\)

```java
public Object[] toArray()
```
- Return a new array containing all elements of collection

```java
public <T> T[] toArray(T[] dest)
```
- Return an array containing all elements of this collection; uses dest as that array if it can

- **Bulk Operations:**
  ```java
  public boolean containsAll(Collection<?> c);
  public boolean addAll(Collection<? extends E> c);
  public boolean removeAll(Collection<?> c);
  public boolean retainAll(Collection<?> c);
  public void clear();
  ```
java.util.Set<E> (an interface)

- Set extends Collection
  - Set inherits all its methods from Collection

- A Set contains no duplicates
  - If you attempt to add() an element twice then the second add() will return false (i.e. the Set has not changed)

- Write a method that checks if a given word is within a Set of words

- Write a method that removes all words longer than 5 letters from a Set

- Write methods for the union and intersection of two Sets
Set Implementations

`java.util.HashSet<E>` *(a hashtable)*

- **Constructors**
  - `public HashSet();`
  - `public HashSet(Collection<? extends E> c);`
  - `public HashSet(int initialCapacity);`
  - `public HashSet(int initialCapacity, float loadFactor);`

`java.util.TreeSet<E>` *(a balanced BST [red-black tree]*)

- **Constructors**
  - `public TreeSet();`
  - `public TreeSet(Collection<? extends E> c);`
  - `...`
SortedSet extends Set
For a SortedSet, the iterator() returns elements in sorted order

Methods (in addition to those inherited from Set):
- public E first();
  - Return first (lowest) object in this set
- public E last();
  - Return last (highest) object in this set
- public Comparator<? super E> comparator();
  - Return the Comparator being used by this sorted set if there is one; returns null if the natural order is being used
- …
java.langComparable<T> (an interface)

- `public int compareTo(T x);`
  - Return a value (< 0), (= 0), or (> 0)
  - (< 0) implies this is before x
  - (= 0) implies this.equals(x)
  - (> 0) implies this is after x

- Many classes implement Comparable
  - String, Double, Integer, Char, java.util.Date, ...
  - If a class implements Comparable then that is considered to be the class’s natural ordering
java.util.Comparator<T> (an interface)

- public int compare(T x1, T x2);
  - Return a value (< 0), (= 0), or (> 0)
    - (< 0) implies x1 is before x2
    - (= 0) implies x1.equals(x2)
    - (> 0) implies x1 is after x2

- Can often use a Comparator when a class’s natural order is not the one you want
  - String.CASE_INSENSITIVE_ORDER is a predefined Comparator
  - java.util.Collections.reverseOrder() returns a Comparator that reverses the natural order
SortedSet Implementations

- java.util.TreeSet<E>
  constructors:
  - public TreeSet();
  - public TreeSet(Collection<? extends E> c);
  - public TreeSet(Comparator<? super E> comparator);
  - ...

- Write a method that prints out a SortedSet of words in order
- Write a method that prints out a Set of words in order
java.util.List<E> (an interface)

- List extends Collection items accessed via their index
- Method add() puts its parameter at the end of the list
- The iterator() returns the elements in list-order
- Methods (in addition to those inherited from Collection):
  - public E get(int i); Return the item at position i
  - public E set(int i, E x); Place x at position i, replacing previous item; return the previous item value
  - public void add(int i, E x);
    - Place x at position index, shifting items to make room
  - public E remove(int index); Remove item at position i, shifting items to fill the space; Return the removed item
  - public int indexOf(Object x);
    - Return index of the first item in the list that equals x (x.equals())
  - ...
  - ...
List Implementations. Each includes methods specific to its class that the other lacks

- java.util.ArrayList\(<E>\) (an array; doubles the length each time room is needed)
  
  Constructors
  - public ArrayList();
  - public ArrayList(int initialCapacity);
  - public ArrayList(Collection<? extends E> c);

- java.util.LinkedList\(<E>\) (a doubly-linked list)
  
  Constructors
  - public LinkedList();
  - public LinkedList(Collection<? extends E> c);
Efficiency Depends on Implementation

- Object \( x = \text{list.get}(k) \):
  - \( O(1) \) time for \( \text{ArrayList} \)
  - \( O(k) \) time for \( \text{LinkedList} \)

- \( \text{list.remove}(0) \):
  - \( O(n) \) time for \( \text{ArrayList} \)
  - \( O(1) \) time for \( \text{LinkedList} \)

- if (set.contains(x)) ...
  - \( O(1) \) expected time for \( \text{HashSet} \)
  - \( O(\log n) \) for \( \text{TreeSet} \)
What if you need $O(1)$ for both?

- Database systems have this issue

- They often build “secondary index” structures
  - For example, perhaps the data is in an ArrayList
  - But they might build a HashMap as a quick way to find desired items

- The $O(n)$ lookup becomes an $O(1)$ operation!