Class 11: Recursive Data Types

6.102 — Software Construction
Spring 2024

Exercise: yellkey.com/**unit**
Nanoquiz: yellkey.com/**cover**

Get started

In `Team.ts`, fill in all `TODO` in immutable class `Team`:

- abstraction function
- rep invariant and `checkRep()`
- rep exposure safety
- missing method at bottom

Nanoquiz

- This quiz is just for you and your own brain:
  - closed-book, closed-notes
  - nothing else on your screen
- Lower your laptop screen when you're done

yellkey.com/**cover**

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

> **bracketize** : Array<Team> → Bracket

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

$\textbf{bracketize} : \text{Array<Team>} \rightarrow \text{Bracket}$

To decide a winner for a tournament, we need to know how good each team is right now:

$\textbf{winner}: \text{Bracket} \times \boxed{???} \rightarrow \text{Team}$ $\boxed{\text{??? = some way to describe who wins vs. who}}$

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> → Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket × ??? → Team   ??? = some way to describe who wins vs. who

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2: and **test** `winner` in BracketTest.ts (complete just the two provided tests).

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> → Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket × ??? → Team   ??? = some way to describe who wins vs. who

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2:   and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.
    Write down the **recursive data type definition** in a comment in Bracket.ts.
TODO-4:   then implement it with new classes at bottom of Bracket.ts

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> → Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket × ??? → Team   ??? = some way to describe who wins vs. who

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2: and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.
Write down the **recursive data type definition** in a comment in Bracket.ts.
TODO-4: then implement it with new classes at bottom of Bracket.ts

TODO-5: Write the **recursive functional definition** of `winner` in a comment,
TODO-6: then implement it with code in the concrete variants.

winner: Bracket × ??? → Team ??? = some way to describe who wins vs. who

Type for ??? argument?

`Map<Team, number>`

winner: Bracket × ??? → Team ??? = some way to describe who wins vs. who

Type for ??? argument?

`Map<Team, number>` 🙀 aaah! `Map<string, number>`

winner: Bracket × ??? → Team ??? = some way to describe who wins vs. who

Type for ??? argument?

`Map<Team, number>` 🙀 aaah! `Map<string, number>`

Implement as a…

A. static function
B. instance method

winner: Bracket × ??? → Team   ??? = some way to describe who wins vs. who

Type for ??? argument?

`Map<Team, number>` 🙀 aaah! `Map<string, number>`

Implement as a…

A. static function
B. instance method ✓

Implementation code in `Bracket`? Y / N

winner: Bracket × ??? → Team ??? = some way to describe who wins vs. who

Type for ??? argument?

`Map<Team, number>` 🙀 aaah! `Map<string, number>`

Implement as a…

A. static function
B. instance method ✓

Implementation code in `Bracket`? Y / No

- declare in interface
- implement in concrete variants

winner: Bracket × Map<string,number> → Team

```
/**
 * @param strength strengths of the teams in this tournament by name
 *
 * @returns winner of this tournament, the team that in every
 *          match of the tournament has higher strength
 */
public winner(strength: Map<string,number>): Team
```

Improve the precondition

winner: Bracket × Map<string,number> → Team

```
/**
 * @param strength strengths of the teams in this tournament by name
 *        requires strength.has(t.name) for every Team t in this
 * @returns winner of this tournament, the team that in every
 *           match of the tournament has higher strength
 */
public winner(strength: Map<string,number>): Team
```

Improve the precondition ✓

Now improve the postcondition

winner: Bracket × Map<string,number> → Team

```
/**
 * @param strength strengths of the teams in this tournament by name
 *         requires strength.has(t.name) for every Team t in this
 * @returns winner of this tournament, ~~the~~ a team that in every
 *           match of the tournament has ~~higher~~ highest strength
 */
public winner(strength: Map<string,number>): Team
```

Improve the precondition ✓

Now improve the postcondition ✓

winner: Bracket × ??? → Team ??? = some way to describe who wins vs. who

Functional approach:

Team → number

Team × Team → Team

bracketize : Array<Team> → Bracket

```
/**
 * @param teams nonempty array of the unique teams in the tournament
 * @returns tournament of the given teams
 */
export function bracketize(teams: Array<Team>): Bracket
```

Strong enough to implement the two provided `winner` test cases?   Y / N

bracketize : Array<Team> → Bracket

```
/**
 * @param teams nonempty array of the unique teams in the tournament
 * @returns tournament of the given teams
 */
export function bracketize(teams: Array<Team>): Bracket
```

Strong enough to implement the two provided `winner` test cases?   Y / N

Yes, "the unique teams in the tournament" is enough for 1- and 2-team tournaments

bracketize : Array<Team> → Bracket

```
/**
 * @param teams nonempty array of the unique teams in the tournament
 * @returns tournament where teams[i-1] plays teams[i] for odd i less
 *          than teams.size()
 */
export function bracketize(teams: Array<Team>): Bracket
```

Fully determined?   Y / N

bracketize : Array<Team> → Bracket

```
/**
 * @param teams nonempty array of the unique teams in the tournament
 * @returns tournament where teams[i-1] plays teams[i] for odd i less
 *          than teams.size()
 */
export function bracketize(teams: Array<Team>): Bracket
```

Fully determined?   Y / Not even close

bracketize : Array<Team> → Bracket

```
/**
 * @param teams nonempty array of the unique teams in the tournament
 * @returns given 1 team, tournament of only that team; otherwise,
 *          given n > 1 teams, tournament in which the winner from a
 *          tournament among the first ceil(n/2) plays the winner from a
 *          tournament among the last floor(n/2)
 */
export function bracketize(teams: Array<Team>): Bracket
```

Make this spec fully determined

bracketize : Array<Team> → Bracket

```
/**
 * @param teams nonempty array of the unique teams in the tournament
 * @returns given 1 team, tournament of only that team; otherwise,
 *          given n > 1 teams, tournament in which the winner from a
 *          tournament among the first ceil(n/2) plays the winner from a
 *          tournament among the last floor(n/2), where those tournaments
 *          are defined according to the same rule */
export function bracketize(teams: Array<Team>): Bracket
```

Make this spec fully determined ✓

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> $\rightarrow$ Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket $\times$ $\boxed{\text{Map<string,number>}}$ $\rightarrow$ Team

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2: and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.
Write down the **recursive data type definition** in a comment in Bracket.ts.
TODO-4: then implement it with new classes at bottom of Bracket.ts

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> → Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket × Map<string,number> → Team

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2: and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.
Write down the **recursive data type definition** in a comment in Bracket.ts.
TODO-4: then implement it with new classes at bottom of Bracket.ts

Which first? A. choose rep B. write tests C. choose ops

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> → Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket × Map<string,number> → Team

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2: and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.
Write down the **recursive data type definition** in a comment in Bracket.ts.
TODO-4: then implement it with new classes at bottom of Bracket.ts

Which first? A. choose rep B. write tests [✓ C. choose ops]

And which one is "write down the recursive data type definition?"

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> → Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket × Map<string,number> → Team

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2: and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.
Write down the **recursive data type definition** in a comment in Bracket.ts.
TODO-4: then implement it with new classes at bottom of Bracket.ts

Which first? [✓ A. choose rep] B. write tests C. choose ops

And which one is "write down the recursive data type definition?"

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

  bracketize : Array<Team> ⟶ Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

  winner: Bracket × Map<string,number> ⟶ Team

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2:   and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.
  Write down the **recursive data type definition** in a comment in Bracket.ts.
TODO-4:   then implement it with new classes at bottom of Bracket.ts

TODO-5: Write the **recursive functional definition** of `winner` in a comment,
TODO-6:   then implement it with code in the concrete variants.

Single-elimination tournaments

Bracket = OneTeam(t: Team) + Game(p1, p2: Team)

Single-elimination tournaments

Bracket = OneTeam(t: Team) + Game(p1, p2: Team)

Bracket = Empty + Game(b1,b2: Bracket)

Single-elimination tournaments

Bracket = OneTeam(t: Team) + Game(p1, p2: Team)

Bracket = Empty + Game(b1,b2: Bracket)

Bracket = *undefined* + Game(b1,b2: Bracket)

Single-elimination tournaments

Bracket = OneTeam(t: Team) + Game(p1, p2: Team)

Bracket = Empty + Game(b1,b2: Bracket)

Bracket = *undefined* + Game(b1,b2: Bracket)

Bracket = Single(t: Team) + Game(b1,b2: Bracket)

Single-elimination tournaments

Bracket = OneTeam(t: Team) + Game(p1, p2: Team)

Bracket = Empty + Game(b1,b2: Bracket)

Bracket = *undefined* + Game(b1,b2: Bracket)

Bracket = Single(t: Team) + Game(b1,b2: Bracket)

Bracket = Team(home,name: string) + Game(b1,b2: Bracket)

Single-elimination tournaments

We have one way to make a tournament right now, **bracketize** : Array<Team> → Bracket
Clients have asked for more ways to make tournaments:

```
/**
 * @param team the only team in the tournament
 * @returns tournament with only the given team
 */
single(team: Team): Bracket
```

What kind of operation is this?

Single-elimination tournaments

We have one way to make a tournament right now, **bracketize** : Array<Team> → Bracket
Clients have asked for more ways to make tournaments:

```
/**
 * @param team the only team in the tournament
 * @returns tournament with only the given team
 */
single(team: Team): Bracket
```

What kind of operation is this? creator

```
/**
 * TODO
 */
match(???): Bracket
```

What should the args be?

Single-elimination tournaments

We have one way to make a tournament right now, **bracketize** : Array<Team> → Bracket
Clients have asked for more ways to make tournaments:

```
/**
 * @param team the only team in the tournament
 * @returns tournament with only the given team
 */
single(team: Team): Bracket
```

What kind of operation is this? creator

```
/**
 * TODO
 */
match(???): Bracket
```

What should the args be? b1: Bracket, b2: Bracket (spec: winner of b1 plays winner of b2)
And what kind of operation is that?

Single-elimination tournaments

We have one way to make a tournament right now, **bracketize** : Array<Team> → Bracket
Clients have asked for more ways to make tournaments:

```
/**
 * @param team the only team in the tournament
 * @returns tournament with only the given team
 */
single(team: Team): Bracket
```

What kind of operation is this? creator

```
/**
 * TODO
 */
match(???): Bracket
```

What should the args be? b1: Bracket, b2: Bracket (spec: winner of b1 plays winner of b2)
And what kind of operation is that? producer
Wait, does this create rep exposure, or break rep independence? Y / N

Single-elimination tournaments

We have one way to make a tournament right now, **bracketize** : Array<Team> → Bracket
Clients have asked for more ways to make tournaments:

```
/**
 * @param team the only team in the tournament
 * @returns tournament with only the given team
 */
single(team: Team): Bracket
```

What kind of operation is this? creator

```
/**
 * TODO
 */
match(???): Bracket
```

What should the args be? b1: Bracket, b2: Bracket (spec: winner of b1 plays winner of b2)
And what kind of operation is that? producer
Wait, does this create rep exposure, or break rep independence?   Y / Nope

yellkey.com/**unit**

Single-elimination tournaments

We want to construct (recursive) single-elimination tournaments:

bracketize : Array<Team> → Bracket

To decide a winner for a tournament, we need to know how good each team is right now:

winner: Bracket × Map<string,number> → Team

TODO-1: Write **specs** for `bracketize` and `winner` in Bracket.ts,
TODO-2: and **test** `winner` in BracketTest.ts (complete just the two provided tests).

TODO-3: Design an immutable, recursive Bracket ADT.

```
Bracket = Single(t: Team) + Game(b1,b2: Bracket)
```

TODO-4: then implement it with new classes at bottom of Bracket.ts

TODO-5: Write the **recursive functional definition** of `winner` in a comment,
TODO-6: then implement it with code in the concrete variants.

Single-elimination tournaments

Equality