PV281: Programování v Rustu

Obsah

1. Generika
2. Utility traity
3. Vektory, iterátory a closures
4. Datové struktury - `std::collections`

Generika

Generika

Umožňuje obecnou definici pro různé typy položek u struktur, výčtů nebo metod.

Generika u struktur

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Generika u struktur (více typů)

```rust
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Generika u výčtu

```
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Generika u metod

```rust
struct Point<T, U> {
    x: T,
    y: U,
}

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point { x: self.x, y: other.y }
    }
}

fn main() {
    let first = Point { x: 5, y: 10.4 };
    let second = Point { x: "Hello", y: 'c' };

    let mixed = first.mixup(second);
    println!("mixed.x = {}, mixed.y = {}", mixed.x, mixed.y);
    // mixed.x = 5, mixed.y = c
}
```

Generika u metod – trait bound

```rust
/// Assume the list parameter is not empty.
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Generika u metod – where

```rust
fn some_function<T, U>(t: &T, u: &U) -> i32
where T: Display + Clone,
      U: Clone + Debug
{
    // ...
}
```

Generika a lifetime

Je možné vynutit lifetime generických typů:

```
fn max<'a, T: PartialOrd>(a: &'a T, b: &'a T) -> &'a T {
    if a < b {
        return b;
    }
    a
}
```

Generika a lifetime

(V tomto případě není nutné brát reference. Pokud `T` implementuje PartialOrd, pak ho implementuje i `&T`. Je pak na uživateli zda předá referenci, nebo vlastněnou hodnotu.)

```
fn max<T: PartialOrd>(a: T, b: T) -> T {
    if a < b {
        return b;
    }
    a
}
```

Utility traity

Utility traity

Důležité traity, které jsou součástí standardní knihovny.

Je vhodné je znát, abychom psali idiomatický kód (tj. takový kód, který je dostateně *Rustic*).

Drop

Rust obecně odvádí dobrou práci při uvolnění zdrojů. Občas ale chceme uvolnění přizpusobit, a k tomu slouží trait `Drop`. Metodu `drop()` Rust volá automaticky při uvolňování paměti.

```
trait Drop {
    fn drop(&mut self);
}
```

Implementace Drop

```rust
struct DataHolder {
    data: String,
}

impl Drop for DataHolder {
    fn drop(&mut self) {
        println!("
            Dropping DataHolder with data `{}`!",
            self.data
        );
    }
}

fn main() {
    let c = DataHolder {
        data: String::from("my stuff"),
    };
    println!("DataHolders created.");
}
```

```
$ cargo run

DataHolders created.
Dropping DataHolder with data `my stuff`!
```

Typ Sized

V Rustu existují typy *sized* a *unsized*. S *unsized* se musí pracovat přes referenci, nemohou být uložené do proměnné.

Příkladem *unsized* je `dyn Trait` (např. `dyn Write`), se kterým jsme se setkali na minulé přednášce.

Typ Sized

Generické typy mají implicitně trait bound `Sized`, takže následující zápisy jsou ekvivalentní:

```
fn generic_function<T>(t: T) { /* ... */ }

fn generic_function<T: Sized>(t: T)  { /* ... */ }
```

Abychom mohli u struktur využít i *unsized* typy, musíme řict, že nemusí jít o typ Sized pomocí `?Sized`. Všimněte si, že musí jít o referenci.

```
fn generic_function<T: ?Sized>(t: &T)  { /* ... */ }
```

Clone

Umožní explicitní vytvoření hluboké kopie.

Lze odvodit pomocí `#[derive(Clone)]`:

```
#[derive(Clone)]
struct MyStruct {
    ...
}
```

Výchozí implementace volá `.clone()` nad všemi položkami struktury (může být drahé časově i paměťově).

Clone - vlastní implementace a `clone_from()`

Definice traitu `Clone`:

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

Vlastní implementace dává kontrolu nad procesem kopírování.
Použití `clone_from()` šetří u collections alokace.

Copy

Umožní implicitní duplikaci hodnoty, a to zkopírováním bitů paměti. Toto chování nelze změnit.

Vyžaduje současně implementovat `Clone`: `#[derive(Copy, Clone)]`.

Zlepšuje sice pohodlí při používání, ale má jistá omezení a cenu.

```
trait Copy: Clone { }

impl Copy for MyType { }
```

Copy

```
#[derive(Debug)]
struct NonCopyable;

#[derive(Debug, Copy, Clone)]
struct Copyable;

fn main() {
    let x1 = NonCopyable;
    let y1 = x1; // `x1` has moved into `y1` and can no longer be used:
    // println!("{:?}", x1); // error: use of moved value

    let x2 = Copyable;
    let y2 = x2; // `y2` is now a copy of `x2`, which is still valid
    println!("{:?}", x2);
}
```

Trait `Copy` implementují například všechny celočíselné i desetinné typy, `bool` či `char`.

Copy - příklad použití

Může se hodit při použití newtype patternu:

```
#[derive(Copy, Clone)]
struct Id(u64);
```

Pro `y: &Id` jsou pak tyto zápisy ekvivalentní:

```
let x: Id = *y;
```

```
let x: Id = y.clone();
```

Default

Poskytuje výchozí hodnotu.

```
trait Default {
    fn default() -> Self;
}

impl Default for String {
    fn default() -> String {
        String::new()
    }
}
```

Default

Používá se pro metody jako `Option::unwrap_or_default()`.

```
let value_option: Option<String> = ...;
let value: String = value_option.unwrap_or_default();
```

From & Into

Slouží pro konverzi mezi typy.
Při správné implementaci `From` je `Into` automaticky implementováno.

```
trait Into<T>: Sized {
    fn into(self) -> T;
}

trait From<T>: Sized {
    fn from(value: T) -> Self;
}
```

From & Into - příklad

Použití s naším newtype `Id`:

```
impl From<u64> for Id {
    fn from(value: u64) -> Self {
        Self(value)
    }
}

...

let user_id: Id = 42.into();
```

TryFrom & TryInto

Použijeme, pokud konverze užívající `From`, resp. `Into`, může selhat.

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}
pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

```
let smaller: i32 = huge.try_into().unwrap_or(i32::MAX);
```

TryFrom & Into - příklad použití

```
struct Timestamp {...}
```

Konverze ze `String` může selhat, konverze na `String` nikoliv:

```
impl TryFrom<String> for Timestamp {...}
impl Into<String> for Timestamp {...}
```

Vektory, iterátory a closures

Připomenutí vektorů

Souvislý blok paměti, uložený na haldě, lze měnit jeho velikost.

```rust
fn main() {
    let values = vec![1, 2, 3];
    let values = vec![0; 64];

    let mut values = Vec::new();
    values.push(1);
    values.push(2);

    match values.get(2) {
        Some(third) => println!("The third element is {}", third),
        None => println!("There is no third element."),
    }

    for value in &values {
        println!("{}", value);
    }
}
```

Iterátor

Jde o trait, který dává následující položku.
Vrácená položka je typu `Option`. Podle toho poznáme, jestli jsme na konci.

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;

    // ...
}
```

Rychlost iterátoru

Iterátory poskytují stejně rychlý (ne-li rychlejší) kód jako normální `for` cyklus.

Jednoduché srovnání najdete v Rust book: https://doc.rust-lang.org/book/ch13-04-performance.html

Iterátor nad vektorem

Vektor implementuje trait `Iterator`.
Můžeme využít funkce `.iter()`.

```rust
fn main() {
    let values = vec![1, 2, 3];

    let iterator = values.iter();

    // `.iter()` lze využít i ve for cyklu
    for value in values.iter() {
        println!("Got: {}", value);
    }
}
```

Iterovatelné typy

Iterovatelný (*Iterable*) je takový typ, který implementuje `IntoIterator`. Pomocí jeho metody `into_iter()` získáme iterátor.

Poznámka: v Rustu je cyklus `for` syntaktický cukr pro volání `into_iter()`, proto je možné napsat následující kód bez vytvoření iterátoru:

```
let values = vec![1, 2, 3, 4, 5];

for x in values {
    println!("{x}");
}
```

Iterovatelné typy

Obdobně si lze zjednodušit zápis iterování přes reference (není nutné volat `.iter()`):

```
let values = vec![1, 2, 3, 4, 5];

for x in &values {
    println!("{x}");
}
```

Možnosti vzniku iterátoru

`.iter()`: prvky iterátoru budou reference (`&T`)

`.iter_mut()`: prvky iterátoru budou mutable reference (`&mut T`)

`.into_iter()`: iterátor se stane vlastníkem prvků (`T`), původní "kolekce" se zkonzumuje

Klonování prvků v iterátoru

Metoda `cloned()` aplikuje na každý prvek metodu `clone()` z traitu `Clone`.

```
let a = ['1', '2', '3', '∞'];

assert_eq!(a.iter().next(),          Some(&'1'));
assert_eq!(a.iter().cloned().next(), Some('1'));
```

Cycle

Iterátor opakuje hodnoty donekonečna, metoda `next()` nikdy nevrátí `None`.

```
let dirs = ["raz", "dva", "tri"];
let mut spin = dirs.iter().cycle();

assert_eq!(spin.next(), Some(&"raz"));
assert_eq!(spin.next(), Some(&"dva"));
assert_eq!(spin.next(), Some(&"tri"));
assert_eq!(spin.next(), Some(&"raz"));
assert_eq!(spin.next(), Some(&"dva"));
assert_eq!(spin.next(), Some(&"tri"));
```

Closure

Anonymní funkce. Z jiných jazyků znáte jako *lambda funkce*.

```
fn main() {
    let closure_annotated = |i: i32| -> i32 { i + 1 };
    let closure_inferred  = |i     |          i + 1  ;

    let i = 1;

    println!("closure_annotated: {}", closure_annotated(i));
    println!("closure_inferred: {}", closure_inferred(i));

    let closure_parameterless = || 1;
    println!("closure returning one: {}", closure_parameterless());
}
```

Zachycení stavu z vnějšího scope

V rámci closure je zachycený stav z vnějšího scope.

```rust
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

Closure jako vstupní parametr

Argumenty funkcí musí být vždy explicitně otypovány.
Typ closure jakožto parametru musí být jeden z následujících traitů:

`Fn`: closure používá své parametry jako reference (`&T`)
`FnMut`: closure používá své parametry jako mutable reference (`&mut T`)
`FnOnce`: closure se stává vlastníkem svých parametrů (`T`)

Closure jako vstupní parametr

```rust
// Funkce, která bere closure jako parametr a zavolá ji.
// Poznámka: F je typické písmeno generického typu pro otypování closure.
fn apply<F>(f: F)
where
    // Samotná closure nemá žádné vstupní parametry a nic nevrací.
    F: FnOnce(),
{
    f();
}

// Funkce, který bere closure jako parametr a vrací `i32`.
fn apply_to_3<F>(f: F) -> i32
where
    // Samotná closure bere i vrací `i32`.
    F: Fn(i32) -> i32
{
    f(3)
}

// TODO: zkuste si zaměnit `FnOnce()`, `Fn()` a `FnMut()` v kódu výše.
```

Metody pracující s iterátory

1. Metody produkující jiný iterátor
2. Metody konzumující iterátor

Map

Funkcionální přístup k iterování: na každý prvek iterátoru se zavolá closure, výsledkem je nový iterátor s modifikovanými prvky.

```
fn main() {
    let a = [1, 2, 3];

    // typem parametru metody `.map()` je `F: FnMut(Self::Item) -> B`
    let mut iter = a.iter().map(|x| 2 * x);

    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next(), Some(4));
    assert_eq!(iter.next(), Some(6));
    assert_eq!(iter.next(), None);
}
```

Filter

Výsledkem je nový iterátor, jehož prvky tvoří podmnožinu prvků původních.

```
fn main() {
    let a = [1, 4, 2, 3];

    let divisible_by_two = a.iter()
        .cloned() // duplikuje položky
        .inspect(|x| println!("about to filter: {}", x))
        .filter(|x| x % 2 == 0)
        .inspect(|x| println!("made it through filter: {}", x))
}
```

Filter Map

V iterátoru zůstanou jen ty prvky, pro které closure vrací `Some(mapped_value)`.

```rust
fn main() {
    let a = [-1, 1, -10, 10, 0];

    let mut iter = a
        .into_iter()
        .filter_map(|n| if n > 0 { Some(n.to_string()) } else { None });

    assert_eq!(iter.next(), Some("1".to_string()));
    assert_eq!(iter.next(), Some("10".to_string()));
    assert_eq!(iter.next(), None);
}
```

Enumerate

Transformuje iterátor na iterátor dvojic: index a prvek.

```
fn main() {
    let a = ['a', 'b', 'c'];

    let mut iter = a.iter().enumerate();

    assert_eq!(iter.next(), Some((0, &'a')));
    assert_eq!(iter.next(), Some((1, &'b')));
    assert_eq!(iter.next(), Some((2, &'c')));
    assert_eq!(iter.next(), None);
}
```

Poznámka: index je typu `usize`, pro vlastní typ použijte `zip()`.

Skip

Přeskočí prvních *n* prvků.

```
fn main() {
    let a = [1, 2, 3];

    let mut iter = a.iter().skip(2);

    assert_eq!(iter.next(), Some(&3));
    assert_eq!(iter.next(), None);
}
```

Take

Vezme prvních *n* prvků.

```
fn main() {
    let a = [1, 2, 3];

    let mut iter = a.iter().take(2);

    assert_eq!(iter.next(), Some(&1));
    assert_eq!(iter.next(), Some(&2));
    assert_eq!(iter.next(), None);
}
```

Fold

Bere iniciální hodnotu akumulátoru a closure o dvou parametrech. Iterátor je zkonzumován.

```rust
fn main() {
    let a = [1, 4, 2, 3];
    let sum = a.iter().fold(0, |acc, x| acc + x);
    assert_eq!(10, sum);
}
```

Poznámka: `reduce()` používá první prvek iterátoru jako iniciální hodnotu akumulátoru.

Poznámka č. 2: existuje metoda `sum()`.

Zip

```
use std::iter::zip;

fn main() {
    let xs = [1, 2, 3];
    let ys = [4, 5, 6];
    for (x, y) in zip(&xs, &ys) {
        println!("x:{}, y:{}", x, y);
    }

    // Zip můžeme i vnořovat:
    let zs = [7, 8, 9];
    for ((x, y), z) in zip(zip(&xs, &ys), &zs) {
        println!("x:{}, y:{}, z:{}", x, y, z);
    }
}
```

Spojení dvou iterátorů

```rust
fn main() {
    // Varianta 1: enumerate()
    let enumerated: Vec<_> = "foo".chars().enumerate().collect();

    // Varianta 2: zip()
    let zipped: Vec<_> = (0..).zip("foo".chars()).collect();

    assert_eq!((0, 'f'), enumerated[0]);
    assert_eq!((0, 'f'), zipped[0]);

    assert_eq!((1, 'o'), enumerated[1]);
    assert_eq!((1, 'o'), zipped[1]);

    assert_eq!((2, 'o'), enumerated[2]);
    assert_eq!((2, 'o'), zipped[2]);
}
```

Collect

Některé typy umožňují převod na kolekci, často používaný je převod na vektor. K tomu slouží metoda `collect()`.

```
let args: Vec<String> = std::env::args().collect();
let args_turbo_fish   = std::env::args().collect::<Vec<String>>();
```

Můžeme převést na jakoukoliv kolekci, která implementuje trait `FromIterator<A>`:

```
trait FromIterator<A>: Sized {
    fn from_iter<T: IntoIterator<Item=A>>(iter: T) -> Self;
}
```

Pro side efekty je vhodnější použít for cyklus…

…pokud se nejedná o poslední volání v dlouhém řetězci metod iterátorů, pak se nabízí metoda `for_each()`.

Pro rozšíření možností iterátorů

využijte crate itertools.

Příklady metod v itertools

`interleave()` - střídavě poskytuje prvky ze dvou iterátorů
`intersperse()` - mezi každý prvek iterátoru vloží hodnotu
`group_by()` - seskupuje po sobě jdoucí prvky se společným klíčem
`merge()` - spojí dva iterátory sléváním
`sorted()` - seřadí iterátor bez potřeby vytvoření vektoru (interně iterátor zkonzumuje, seřadí a vytvoří nový)
`unfold()` - generuje iterátor na základě výchozího stavu a builder funkce

Pro jednoduchou paralelizaci na úrovni iterátoru

využijte crate Rayon.

Instalace Rayon

Do `Cargo.toml` přidáme závislost:

```toml
[dependencies]
rayon = "1.5"
```

Použití Rayon

Metodu `iter()` nahradíme za metodu `par_iter()`.

```
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // <-- just change that!
         .map(|&i| i * i)
         .sum()
}
```

Datové struktury

Modul `std::collections`

Dvousměrný vektor

Využití:

1. Chceme vkládat na začátek.
2. Potřebujeme frontu.
3. Potřebujeme obousměrnou frontu.

Je implementován jako *ring buffer*, tj. nemusí zabírat kontinuální prostor v paměti. Pro transformaci na kontinuální prostor můžeme použít metodu `make_contiguous()` (vhodné třeba pro efektivní sorting).

Ring buffer

Dvousměrný vektor

```
fn main() {
    use std::collections::VecDeque;

    let mut buf = VecDeque::new();
    buf.push_back(3);  // [3]
    buf.push_back(4);  // [3, 4]
    buf.push_back(5);  // [3, 4, 5]
    buf.push_front(2); // [2, 3, 4, 5]

    if let Some(elem) = buf.get_mut(2) {
        *elem = 7;
    }                  // [2, 3, 7, 5]

    assert_eq!(d.pop_front(), Some(2));
    assert_eq!(buf[1], 7);
}
```

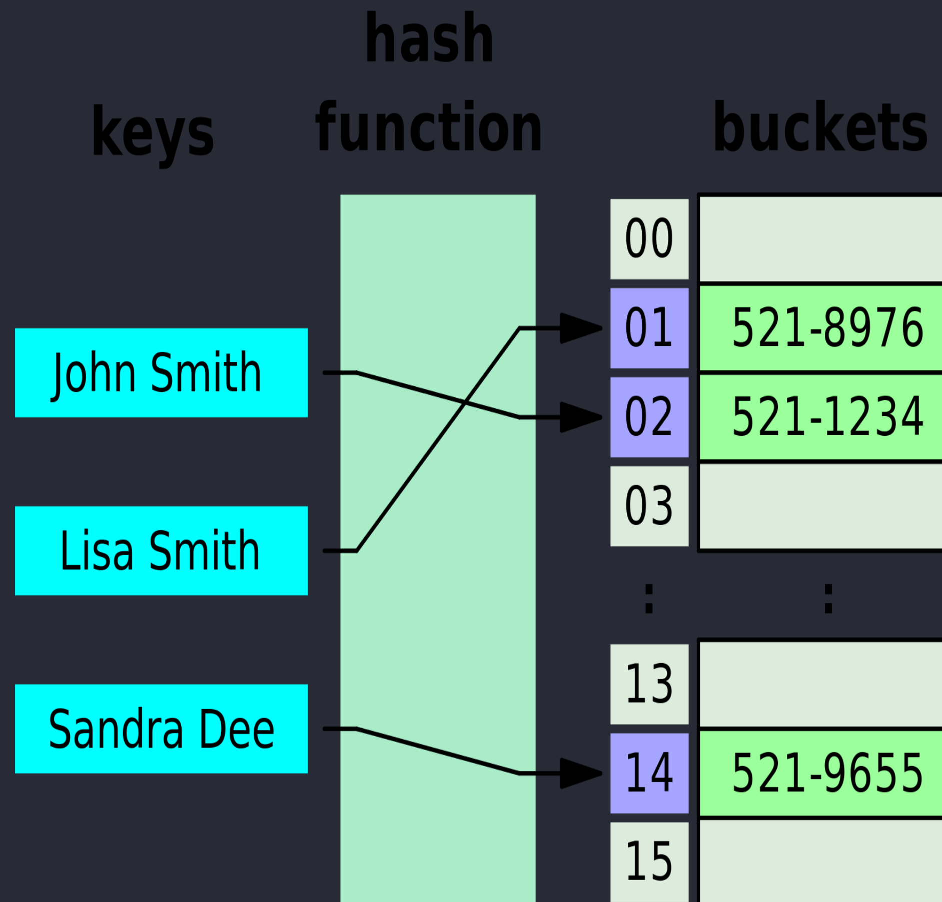
Hašovací tabulka

Využití:

1. Potřebujeme slovník.
2. Potřebujeme cache.

Implementovaná podle *Google SwissTable*, jako hashovací funkci používá *SipHash 1-3*. Ta je vhodná pro středně velké slovníky a je odolná proti HashDoS útokům.
Pro malé a velké hashovací tabulky je vhodnější použít jinou hashovací funkci.

Hašovací tabulka



Hašovací tabulka

```
fn main() {
    use std::collections::HashMap;

    let mut scores: HashMap<String, i32> = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    // Inserts if the key does not exist.
    scores.entry(String::from("Yellow")).or_insert(50);
    scores.entry(String::from("Blue")).or_insert(50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name);
}
```

Hašovací tabulka – ukázka 2

```
fn main() {
    let mut book_reviews = HashMap::new();

    // Review some books.
    book_reviews.insert("Adventures of Huckleberry Finn".to_string(), "My favorite book. 10/10.".to_string());

    if !book_reviews.contains_key("Les Misérables") {
        println!("We've got {} reviews, but Les Misérables ain't one.", book_reviews.len());
    }

    // Oops, this review has a lot of spelling mistakes, let's delete it.
    book_reviews.remove("The Adventures of Sherlock Holmes");

    // Look up the values associated with some keys.
    let to_find = ["Pride and Prejudice", "Alice's Adventure in Wonderland"];
    for &book in &to_find {
        match book_reviews.get(book) {
            Some(review) => println!("{}: {}", book, review),
            None => println!("{} is unreviewed.", book)
        }
    }
}
```

Množina

Využití:

1. Chceme zaznamenávat prošlé prvky.
2. Chceme mít hodnotu uloženou pouze jednou.

Nejrychlejší implementace je `HashSet`.
To platí ale jen do chvíle, než potřebujeme mít položky seřazené.
Potom už použijeme `BTreeSet`.

Množina

```
fn main() {
    use std::collections::HashSet;

    // Type inference lets us omit an explicit type signature (which would be `HashSet<String>` in this example).
    let mut books = HashSet::new();

    // Add some books.
    books.insert("A Dance With Dragons".to_string());
    books.insert("To Kill a Mockingbird".to_string());
    books.insert("The Odyssey".to_string());
    books.insert("The Great Gatsby".to_string());

    // Check for a specific book.
    if !books.contains("The Winds of Winter") {
        println!("We have {} books, but The Winds of Winter ain't one.",
                 books.len());
    }

    // Remove a book.
    books.remove("The Odyssey");
}
```

B-strom

Využití:

1. Chceme mapu seřazenou podle klíčů.
2. Chceme získávat položky v nějakém rozsahu.
3. Chceme rychle získávat nejmenší nebo největší položku.
4. Chceme najít klíče, které jsou větší nebo menší než jiný klíč.

B-strom

```
fn main() {
    use std::collections::BTreeMap;

    // Type inference lets us omit an explicit type signature (which would be `BTreeMap<&str, u8>` in this example).
    let mut player_stats = BTreeMap::new();

    // Insert a key only if it doesn't already exist.
    player_stats.entry("health").or_insert(100);

    // Insert a key using a function that provides a new value only if it doesn't already exist.
    player_stats.entry("defence").or_insert_with(|| 42);

    // Update a key, guarding against the key possibly not being set.
    let stat = player_stats.entry("attack").or_insert(100);
    *stat += 13;
}
```

B-strom – ukázka 2

```rust
fn main() {
    use std::collections::BTreeMap;

    // Type inference lets us omit an explicit type signature (which would be `BTreeMap<&str, &str>` in this example).
    let mut movie_reviews = BTreeMap::new();

    // Review some movies.
    movie_reviews.insert("Office Space",       "Deals with real issues in the workplace.");
    movie_reviews.insert("Pulp Fiction",       "Masterpiece.");
    movie_reviews.insert("The Godfather",      "Very enjoyable.");
    movie_reviews.insert("The Blues Brothers", "Eye lyked it a lot.");

    // Check for a specific one.
    if !movie_reviews.contains_key("Les Misérables") {
        println!("We've got {} reviews, but Les Misérables ain't one.", movie_reviews.len());
    }

    // Continued on the next slide...
}
```

B-strom – ukázka 2

```rust
fn main() {
    // Continued from previous slide...

    // Oops, this review has a lot of spelling mistakes, let's delete it.
    movie_reviews.remove("The Blues Brothers");

    // Look up the values associated with some keys.
    let to_find = ["Up!", "Office Space"];
    for movie in &to_find {
        match movie_reviews.get(movie) {
            Some(review) => println!("{}: {}", movie, review),
            None => println!("{} is unreviewed.", movie)
        }
    }

    // Look up the value for a key (will panic if the key is not found).
    println!("Movie review: {}", movie_reviews["Office Space"]);

    // Iterate over everything.
    for (movie, review) in &movie_reviews {
        println!("{}: \"{}\"", movie, review);
    }
}
```

Halda

Využití:

1. Potřebujeme prioritní frontu.
2. Chceme pracovat s největší/nejdůležitější položkou.

Halda

```rust
fn main() {
    use std::collections::BinaryHeap;

    // Type inference lets us omit an explicit type signature (which would be `BinaryHeap<i32>` in this example).
    let mut heap = BinaryHeap::new();

    // We can use peek to look at the next item in the heap. In this case, there's no items in there yet so we get None.
    assert_eq!(heap.peek(), None);

    // Let's add some scores...
    heap.push(1);
    heap.push(5);
    heap.push(2);

    // Continued on the next slide...
}
```

Halda

```rust
fn main() {
    // Continued from previous slide...

    // Now peek shows the most important item in the heap.
    assert_eq!(heap.peek(), Some(&5));

    // We can check the length of a heap.
    assert_eq!(heap.len(), 3);

    // We can iterate over the items in the heap, although they are returned in a random order.
    for x in &heap {
        println!("{}", x);
    }

    // If we instead pop these scores, they should come back in order.
    assert_eq!(heap.pop(), Some(5));
    assert_eq!(heap.pop(), Some(2));
    assert_eq!(heap.pop(), Some(1));
    assert_eq!(heap.pop(), None);

    // We can clear the heap of any remaining items.
    heap.clear();

    // The heap should now be empty.
    assert!(heap.is_empty())
}
```

Dotazy?

Děkuji za pozornost