Utility traits

Chapter 12 & 13

Daniël de Kok

Overview

- Equality
- Ordering
- Indexing
- Drop

Equality

Introduction

The equality operators are syntactic sugar for `PartialEq` methods:

- ==: PartialEq::eq

```
a == b; // is sugar for:
a.eq(&b)
```

Introduction

The equality operators are syntactic sugar for `PartialEq` methods:

- `==`: PartialEq::eq

```
a == b; // is sugar for:
a.eq(&b)
```

- `!=`: PartialEq::ne

```
a != b; // is sugar for:
a.ne(&b);
```

The PartialEq trait

```
trait PartialEq<Rhs: ?Sized = Self> {
  fn eq(&self, other: &Rhs) -> bool;

  fn ne(&self, other: &Rhs) -> bool {
    !self.eq(other)
  }
}
```

Example (complex numbers)

```rust
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

Example (complex numbers)

```
impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) -> bool {
        self.re == other.re && self.im == other.im
    }
}
```

Generating the implementation automatically

```rust
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

Generating the implementation automatically

```rust
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

- Compares field-by-field.
- Only implemented for `Complex<T>` where `T` implements `PartialEq`.

The `PartialEq` trait and move types

```
trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;

    // ...
}
```

- `eq` borrows `self` and `other`.
- Implementation of equality for move types, that does not move the values.

Why partial equality?

Why is the trait called `PartialEq`?

- Properties equivalency relation:

Why partial equality?

Why is the trait called `PartialEq`?

- Properties equivalency relation:
  - **Reflexive property:** $a = a$

Why partial equality?

Why is the trait called `PartialEq`?

- Properties equivalency relation:
  - **Reflexive property:** $a = a$
  - **Symmetric property:** if $a = b$, then $b = a$

Why partial equality?

Why is the trait called `PartialEq`?

- Properties equivalency relation:
  - **Reflexive property:** $a = a$
  - **Symmetric property:** if $a = b$, then $b = a$
  - **Transitive property:** if $a = b$ and $b = c$, then $a = c$

Why partial equality?

Why is the trait called `PartialEq`?

- Properties equivalency relation:
  - **Reflexive property:** $a = a$
  - **Symmetric property:** if $a = b$, then $b = a$
  - **Transitive property:** if $a = b$ and $b = c$, then $a = c$
- For floating point numbers, the reflexive property does not hold.

IEEE floating point numbers

- Expressions with no appropriate value evaluate to `NaN`.
  - E.g.: `0.0 / 0.0`

IEEE floating point numbers

- Expressions with no appropriate value evaluate to `NaN`.
  - E.g.: `0.0 / 0.0`
- IEEE 754: `NaN` must be treated as inequal to any value, including `NaN`.

IEEE floating point numbers

- Expressions with no appropriate value evaluate to `NaN`.
  - E.g.: `0.0 / 0.0`
- IEEE 754: `NaN` must be treated as inequal to any value, including `NaN`.
- Consequently:

```
assert!(f64::is_nan(0.0/0.0));
assert_eq!(0.0/0.0 == 0.0/0.0, false);
assert_eq!(0.0/0.0 != 0.0/0.0, true);
```

Why partial equality?

Why is the trait called `PartialEq`?

- Types that only implement `PartialEq` do not have the reflexive property.

Why partial equality?

Why is the trait called `PartialEq`?

- Types that only implement `PartialEq` do not have the reflexive property.
- `PartialEq` not useful when the reflexive property is required (e.g. hash tables).

The Eq trait

```
trait Eq: PartialEq<Self> { }
```

- The Eq trait is used to mark types for which all the equivalency relation properties hold.

  ```
  trait Eq: PartialEq<Self> { }
  ```

The Eq trait

```
trait Eq: PartialEq<Self> { }
```

- The Eq trait is used to mark types for which all the equivalency relation properties hold.

  ```
  trait Eq: PartialEq<Self> { }
  ```

- Trivial implementation:

  ```
  impl<T: Eq> Eq for Complex<T> { }
  ```

Deriving Eq automatically

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    /// Real portion of the complex number
    re: T,

    /// Imaginary portion of the complex number
    im: T,
}
```

Note: Eq requires PartialEq.

Enforcing invariants

- `PartialEq` vs `Eq` is another example where the type system is leveraged to exclude a class of errors.

Enforcing invariants

- `PartialEq` vs `Eq` is another example where the type system is leveraged to exclude a class of errors.
- If you implement an algorithm or data structure that requires reflexity: require types with the `Eq` trait.

Enforcing invariants

- `PartialEq` vs `Eq` is another example where the type system is leveraged to exclude a class of errors.
- If you implement an algorithm or data structure that requires reflexity: require types with the `Eq` trait.
- Not possible to use such an algorithm or data structure with types that **only** implement `PartialEq`.

In-class assignment

Log probabilities

1. Implement a type for log-probabilities ($\log P(x)$).
2. Implement equality for this type, think about whether you can implement `Eq` or only `PartialEq`.

Equality of object identity

Would it be possible to implement equality based on the object identity (as the == operator in Java)?

Ordering

Introduction

The ordered comparison operators <, >, <=, and >= are syntactic sugar for the `PartialOrd` trait:

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
        where Rhs: ?Sized {
    fn partial_cmp(&self, other: &Rhs)
        -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Introduction

The ordered comparison operators <, >, <=, and >= are syntactic sugar for the `PartialOrd` trait:

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
        where Rhs: ?Sized {
    fn partial_cmp(&self, other: &Rhs)
        -> Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Note: `PartialOrd` extends `PartialEq`.

Ordering

```
pub enum Ordering {
    Less,
    Equal,
    Greater,
}
```

PartialOrd

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
        where Rhs: ?Sized {
    fn partial_cmp(&self, other: &Rhs)
        -> Option<Ordering>;

    // ...
}
```

None is returned when the values cannot be compared: NaN in f32 and f64.

PartialOrd

The following properties should hold:

- **Antisymmetry**: if $a < b$ then $a \not> b$
- **Transitivity**: if $a < b$ and $b < c$ then $a < c$

Default implementations: `lt`

```
fn lt(&self, other: &Rhs) -> bool {
    match self.partial_cmp(other) {
        Some(Less) => true,
        _ => false,
    }
}
```

Default implementations: gt

```
fn gt(&self, other: &Rhs) -> bool {
    match self.partial_cmp(other) {
        Some(Greater) => true,
        _ => false,
    }
}
```

Default implementations: `le`

```
fn le(&self, other: &Rhs) -> bool {
    match self.partial_cmp(other) {
        Some(Less) | Some(Equal) => true,
        _ => false,
    }
}
```

Default implementations: ge

```
fn ge(&self, other: &Rhs) -> bool {
    match self.partial_cmp(other) {
        Some(Greater) | Some(Equal) => true,
        _ => false,
    }
}
```

Deriving `PartialOrd` automatically

```
#[derive(Debug, PartialEq, PartialOrd)]
struct WordCount {
    freq: usize,
    word: String,
}
```

- Lexicographic ordering; in
- top-to-bottom declaration order.
- On enums: variants ordered by top-to-bottom order.

In-class assignment

Try to figure out the standard ordering of the variants of `Option` and `Result`.

PartialOrd implementation

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // inclusive
    upper: T // exclusive
}
```

PartialOrd implementation

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // inclusive
    upper: T // exclusive
}
```

If two intervals overlap: ordering not defined.

PartialOrd implementation

```
impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        if self == other { Some(Ordering::Equal) }
        else if self.lower >= other.upper {
            Some(Ordering::Greater)
        }
        else if self.upper <= other.lower {
            Some(Ordering::Less)
        }
        else { None }
    }
}
```

Total order

For a total order the following properties must hold:

- **Antisymmetry:** if $a < b$ then $a \not> b$
- **Transitivity:** if $a < b$ and $b < c$ then $a < c$
- **Connex property**: $a \leq b$ or $b \leq a$
- Types that implement the `Ord` trait have a total order.

Total order

For a total order the following properties must hold:

- **Antisymmetry:** if $a < b$ then $a \not> b$
- **Transitivity:** if $a < b$ and $b < c$ then $a < c$
- **Connex property**: $a \leq b$ or $b \leq a$
- Types that implement the `Ord` trait have a total order.
- Connex property does not hold for floating point types due to `NaN`.

Ord

```
pub trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;

    fn max(self, other: Self) -> Self { ... }
    fn min(self, other: Self) -> Self { ... }
}
```

`Ord` can be derived automatically.

Reverse ordering

An `PartialOrd` and `Ord` types can be wrapped in the `Reverse` tuple struct to reverse the ordering:

```
use std::cmp::Reverse;

let mut v = vec![1, 2, 3, 4, 5, 6];
v.sort_by_key(|&num| (num > 3, Reverse(num)));
assert_eq!(v, vec![3, 2, 1, 6, 5, 4]);
```

Reverse ordering

```
pub struct Reverse<T>(pub T);

impl<T: PartialOrd> PartialOrd for Reverse<T> {
    fn partial_cmp(&self, other: &Reverse<T>)
            -> Option<Ordering> {
        other.0.partial_cmp(&self.0)
    }
}
```

Practical problem: sorting floats

```
let mut v = vec![1f32, 2., 3., 4., 5., 6.];

// The trait `std::cmp::Ord` is not implemented
// for `f32`.
v.sort();
```

- How can you sort floating point numbers?
- Similarly: how would you use floating point numbers in a ordered set/map?

Indexing

Introduction

Indexing (`[]`) is also syntactic sugar for a trait method:

```
a[i]

// Shorthand for:
*a.index(i)

// When used in assignment or mutably:
*a.index_mut(i)
```

Index/IndexMut

```
trait Index<Idx> {
  type Output: ?Sized;

  fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
  fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

Index/IndexMut

```
trait Index<Idx> {
  type Output: ?Sized;

  fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
  fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

Note: flexibility to use your own index types.

Indexing and move types

```
let mut v = vec![String::default(); 10];

// Ok, desugars to:
// *v.index_mut(2) ="hello".to_string()
v[2] = 1.0;

// Ok, desugars to:
// let elem_ref = &*v.index(2)
let elem_ref = &v[2];

// Not ok, desugars to:
// let elem = *v.index(2)
let elem = v[2];

//Not ok, desugars to:
// let mut elem = *v.index_mut(2);
let mut elem = v[2];
```

Example

```
use std::collections::BTreeMap;

let mut m = BTreeMap::new();
m.insert("foo", 10);
assert_eq!(m["foo"], 10);
```

Implementation

```
impl<'a, K: Ord, Q: ?Sized, V> Index<&'a Q> for BTreeMap<K, V>
    where K: Borrow<Q>,
          Q: Ord
{
  type Output = V;

  fn index(&self, key: &Q) -> &V {
    self.get(key).expect("no entry found for key")
  }
}
```

Drop

Introduction

- When a value becomes disowned, Rust drops it.
- Frees other values, heap storage, and system resources (file descriptors, sockets, etc.).

Introduction

- When a value becomes disowned, Rust drops it.
- Frees other values, heap storage, and system resources (file descriptors, sockets, etc.).
- Sometimes, you want to do additional cleanup.

Introduction

- When a value becomes disowned, Rust drops it.
- Frees other values, heap storage, and system resources (file descriptors, sockets, etc.).
- Sometimes, you want to do additional cleanup.
- When a type implements the `Drop` trait, its `drop` method will be called when values of this type are dropped.

Drop

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

Example

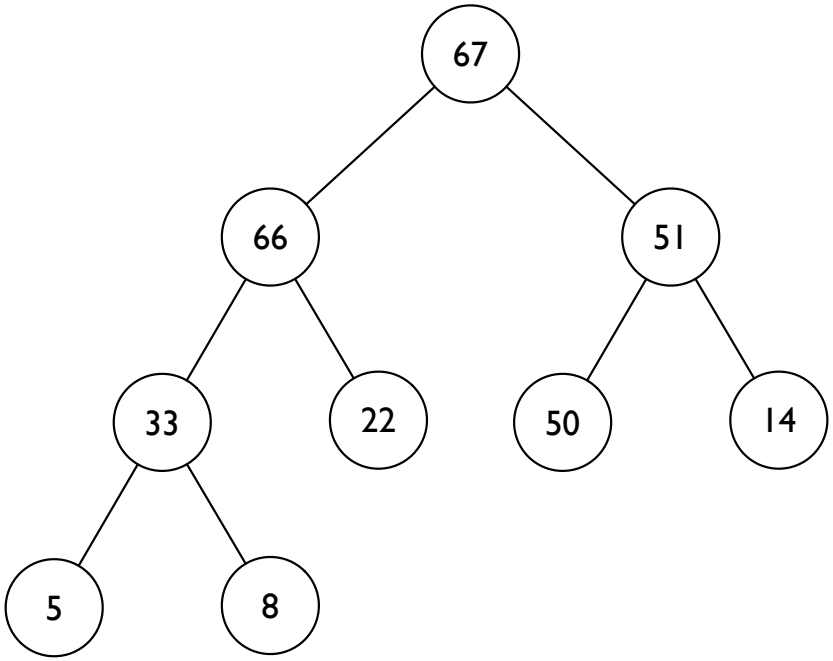
```
impl<W: Write> Drop for BufWriter<W> {
    fn drop(&mut self) {
        if self.inner.is_some() && !self.panicked {
            let _r = self.flush_buf();
        }
    }
}
```

Excursion: `Drop` in Rust's `BinaryHeap`

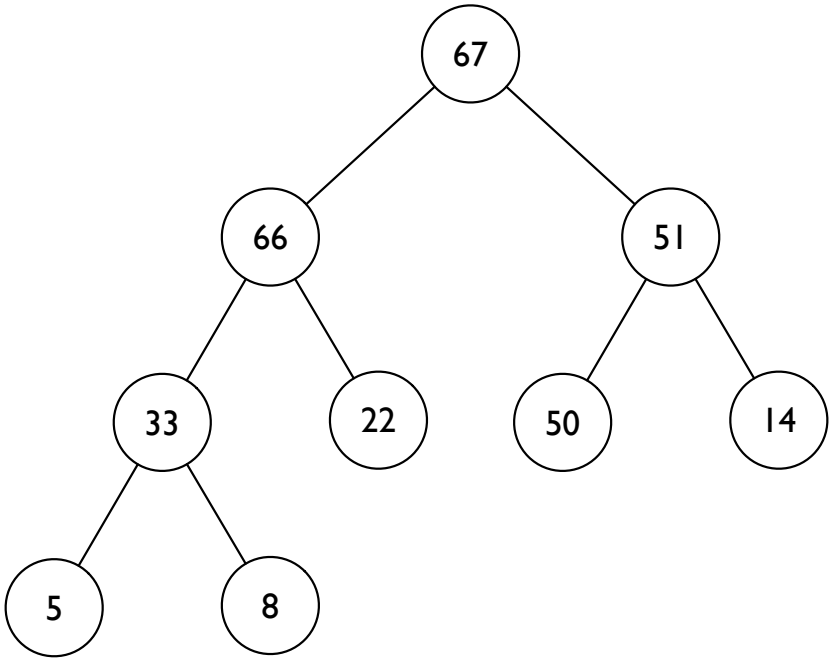
Shows the strengths of:

- Rust traits
- Deterministic destruction (drops)
- Rust's ownership model

Recap max-heap



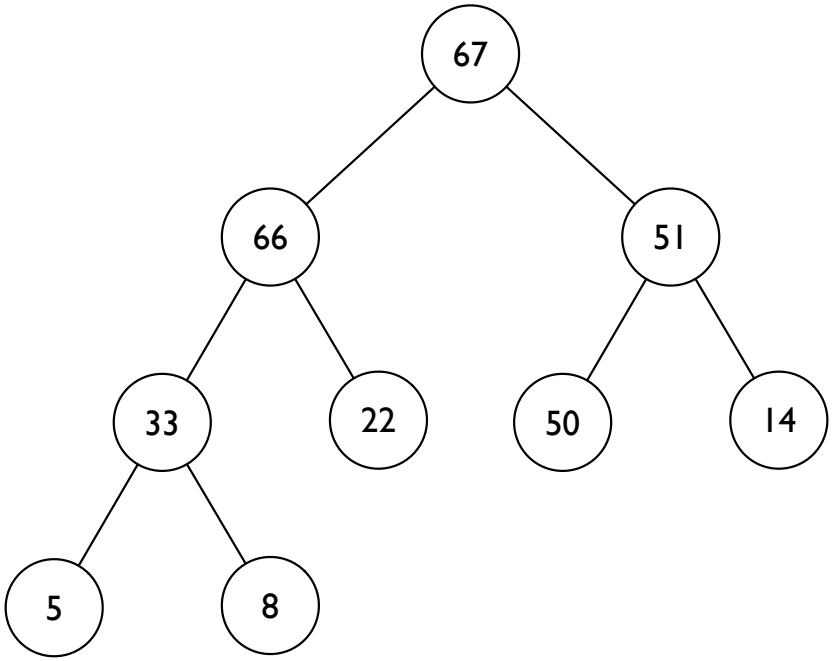
Recap max-heap



Properties:

- A binary heap is a **complete binary tree**.

Recap max-heap



Properties:

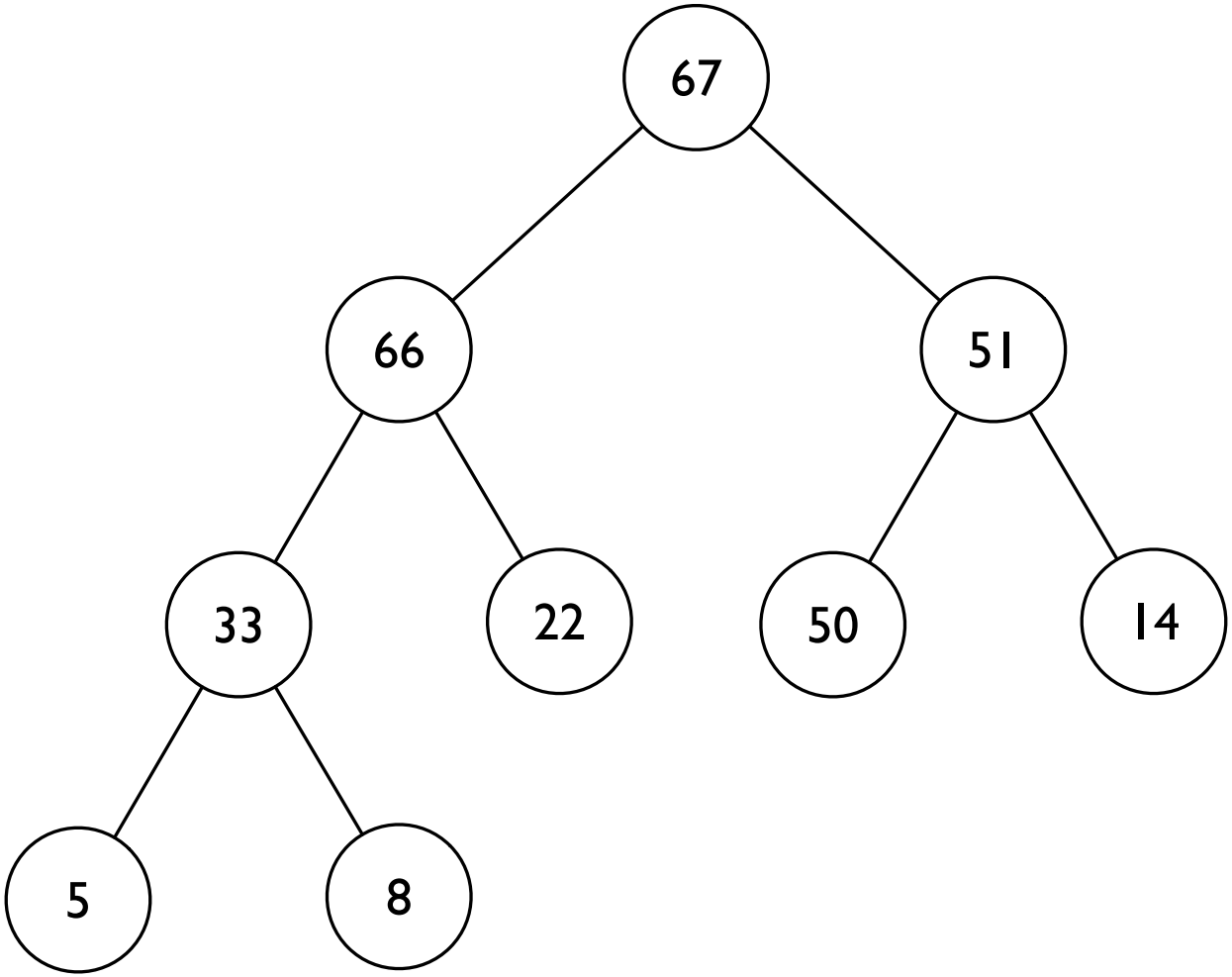
- A binary heap is a **complete binary tree**.
- A binary heap satisfies the **heap property**: the value in every node is greater than or equal to its children.

Bottom-up reheapify (swim)

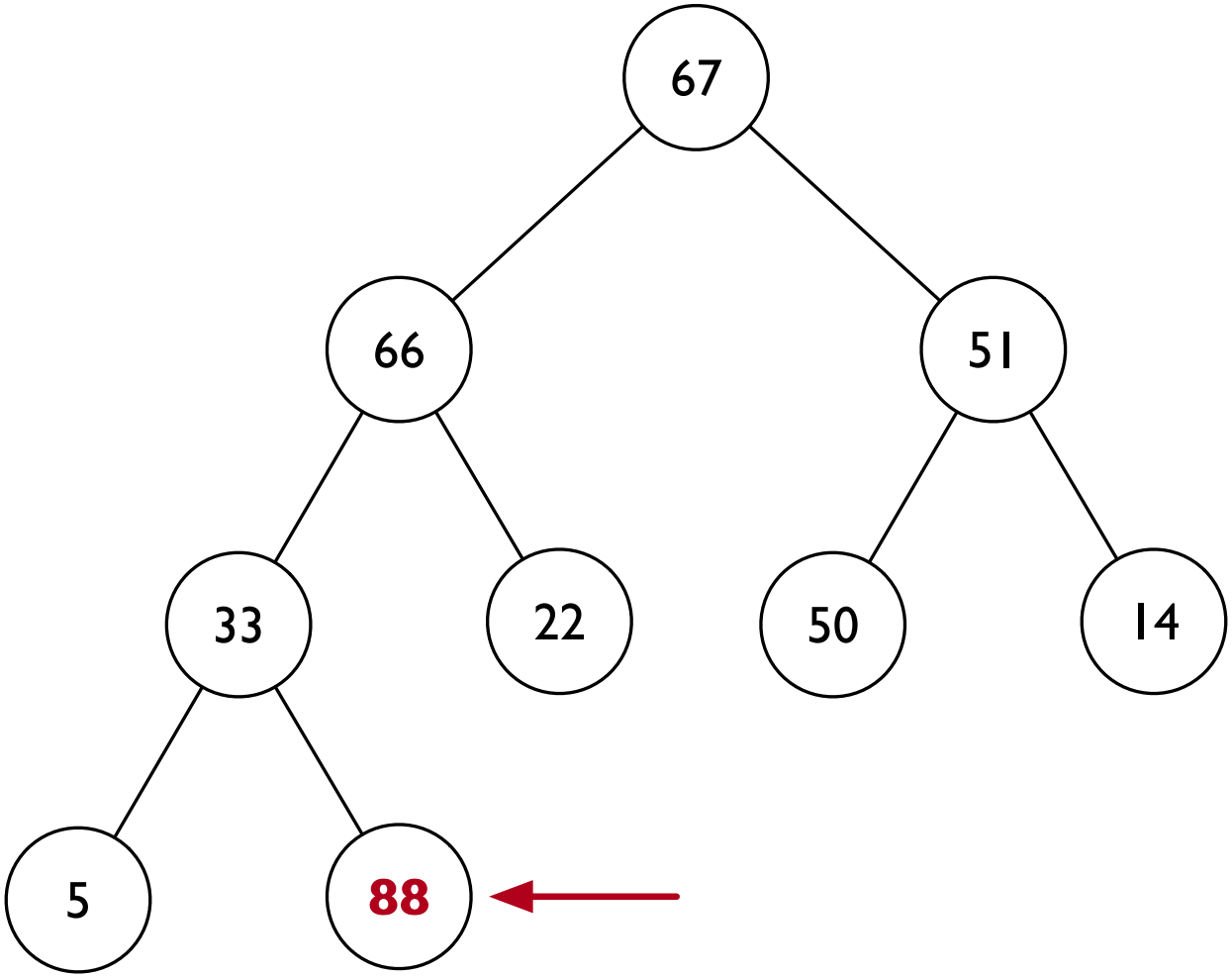
If the heap property is violated by modifying the value in a node, such that its value is **larger** than the parent, the heap property can be restored using **bottom-up reheapify** (**swim**):

1. Exchange the values of the node and its parent.
2. If the value is larger than its new parent, goto (1).

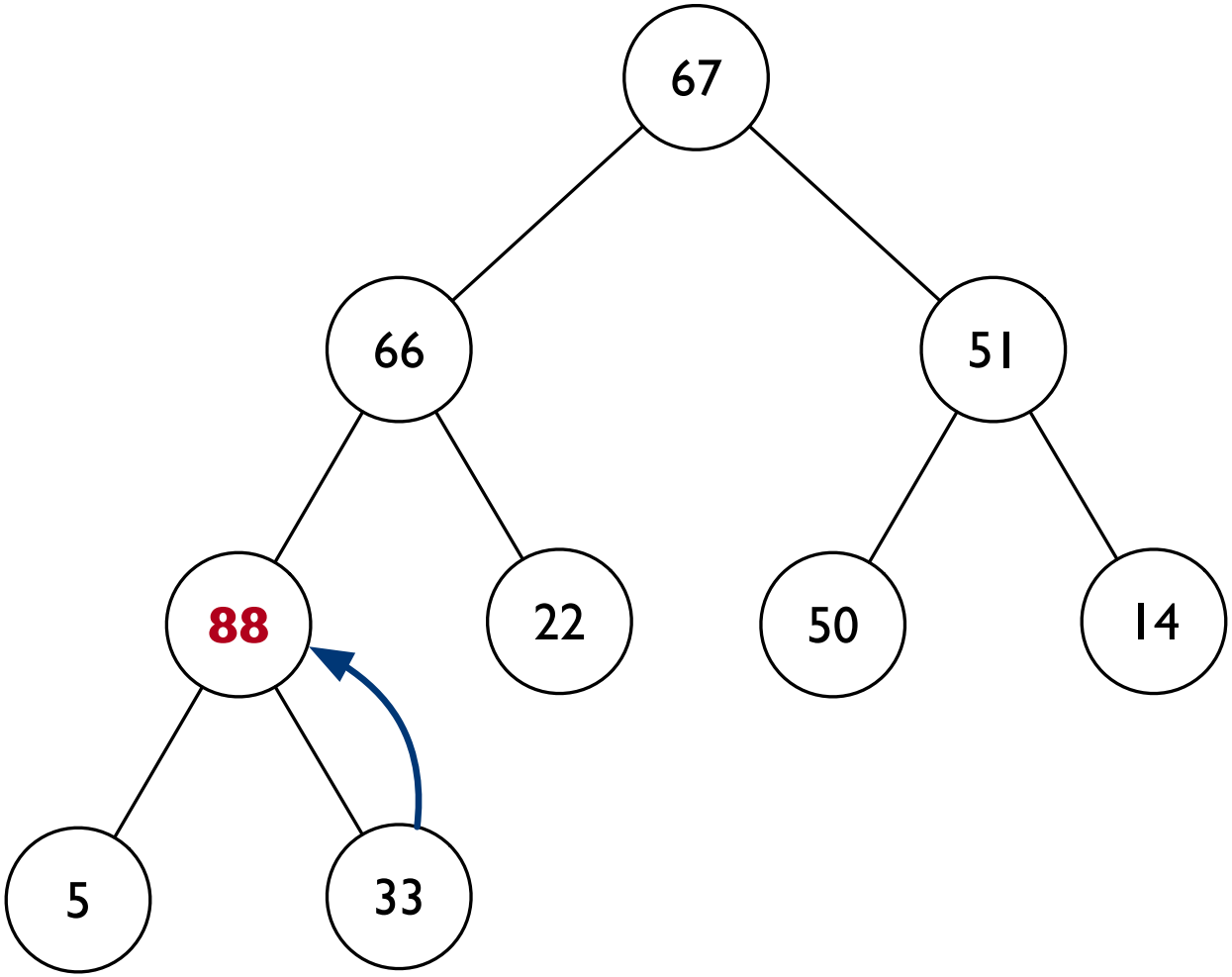
Swim (example)



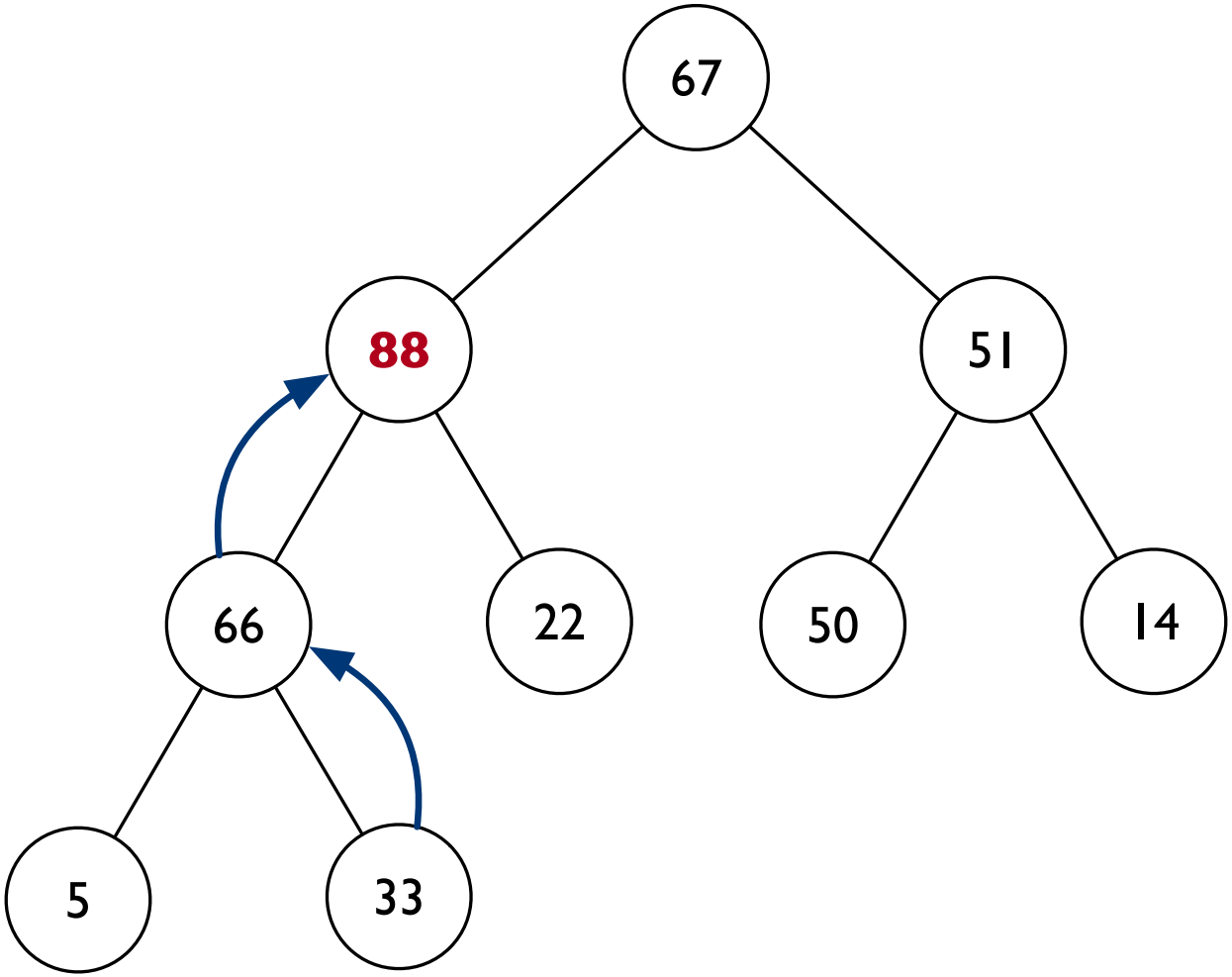
Swim (example)



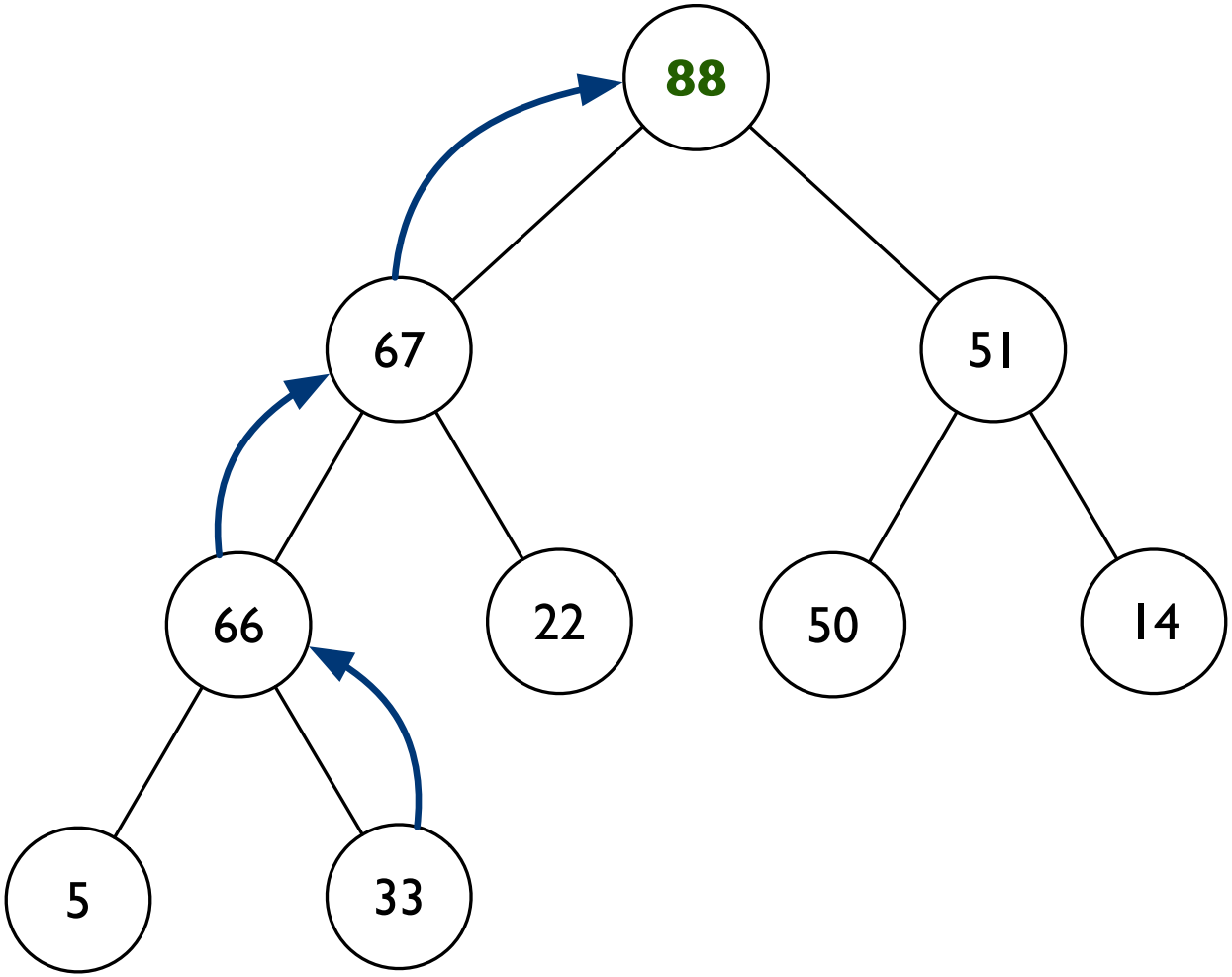
Swim (example)



Swim (example)



Swim (example)

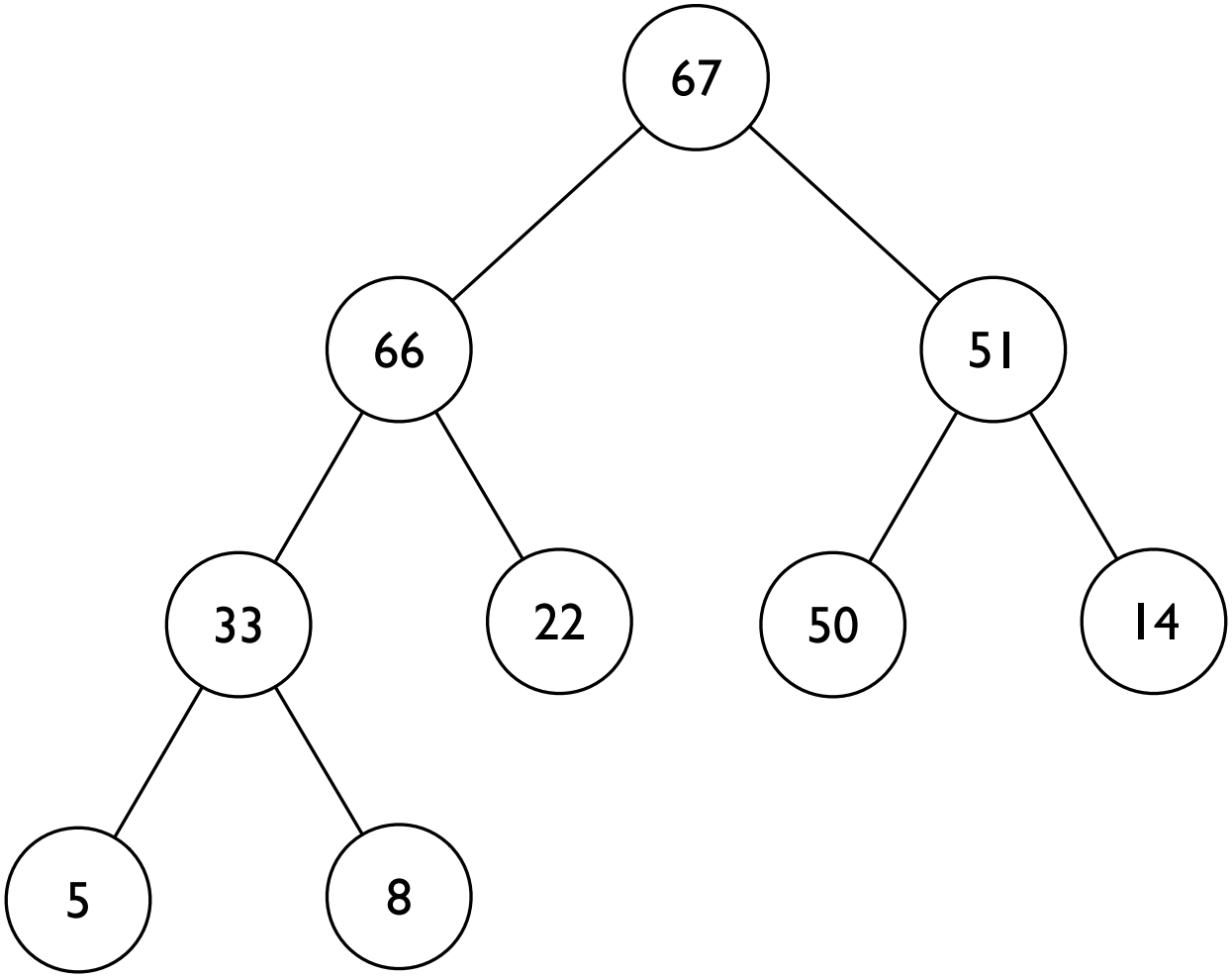


Top-down reheapify (sink)

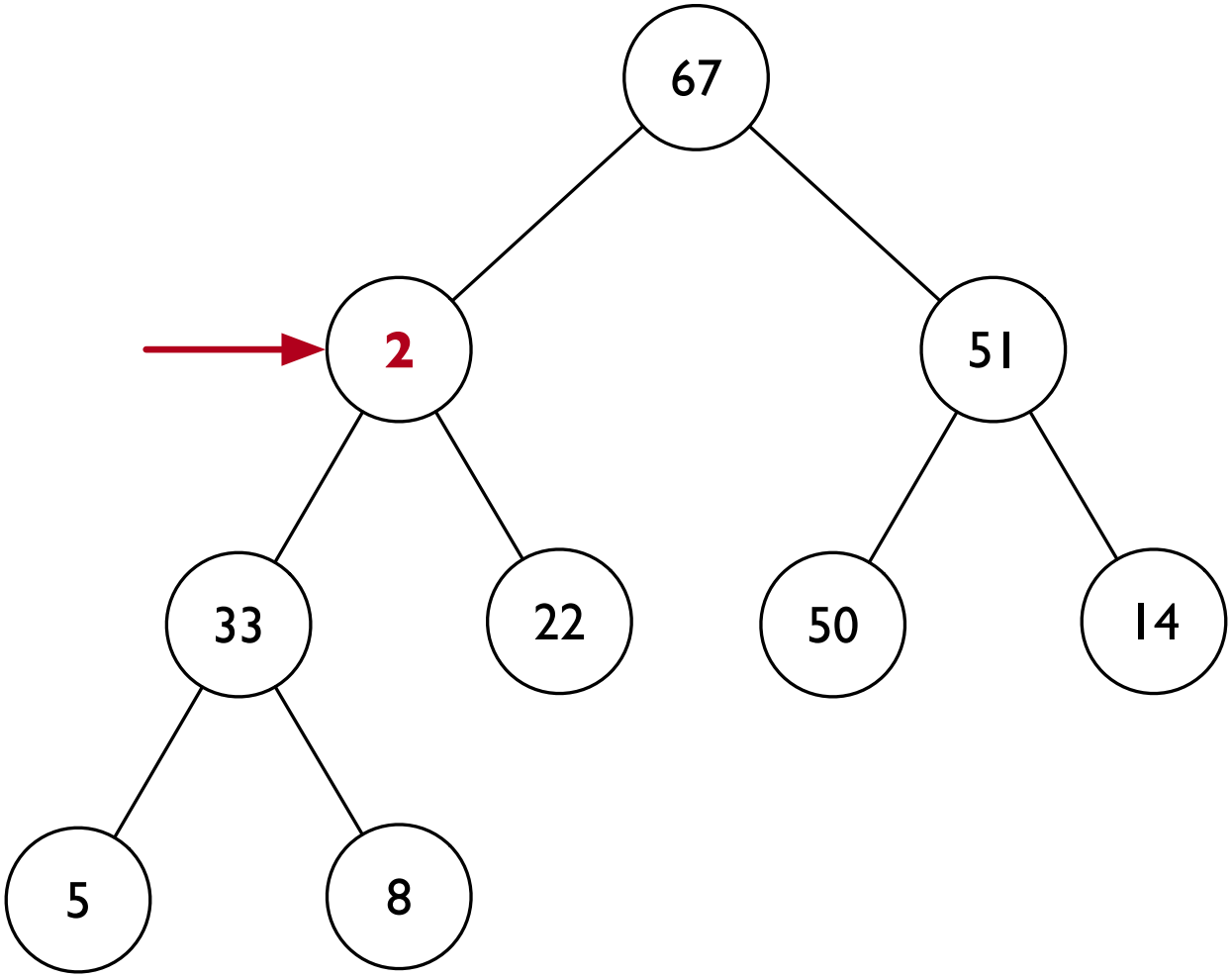
If the heap property is violated by changing a node, such that its value is **smaller** than its children, the heap property can be restored using **top-down reheapify** (**sink**):

1. Exchange the values of the and node its largest child.
2. If the value is smaller than its new children, goto (1).

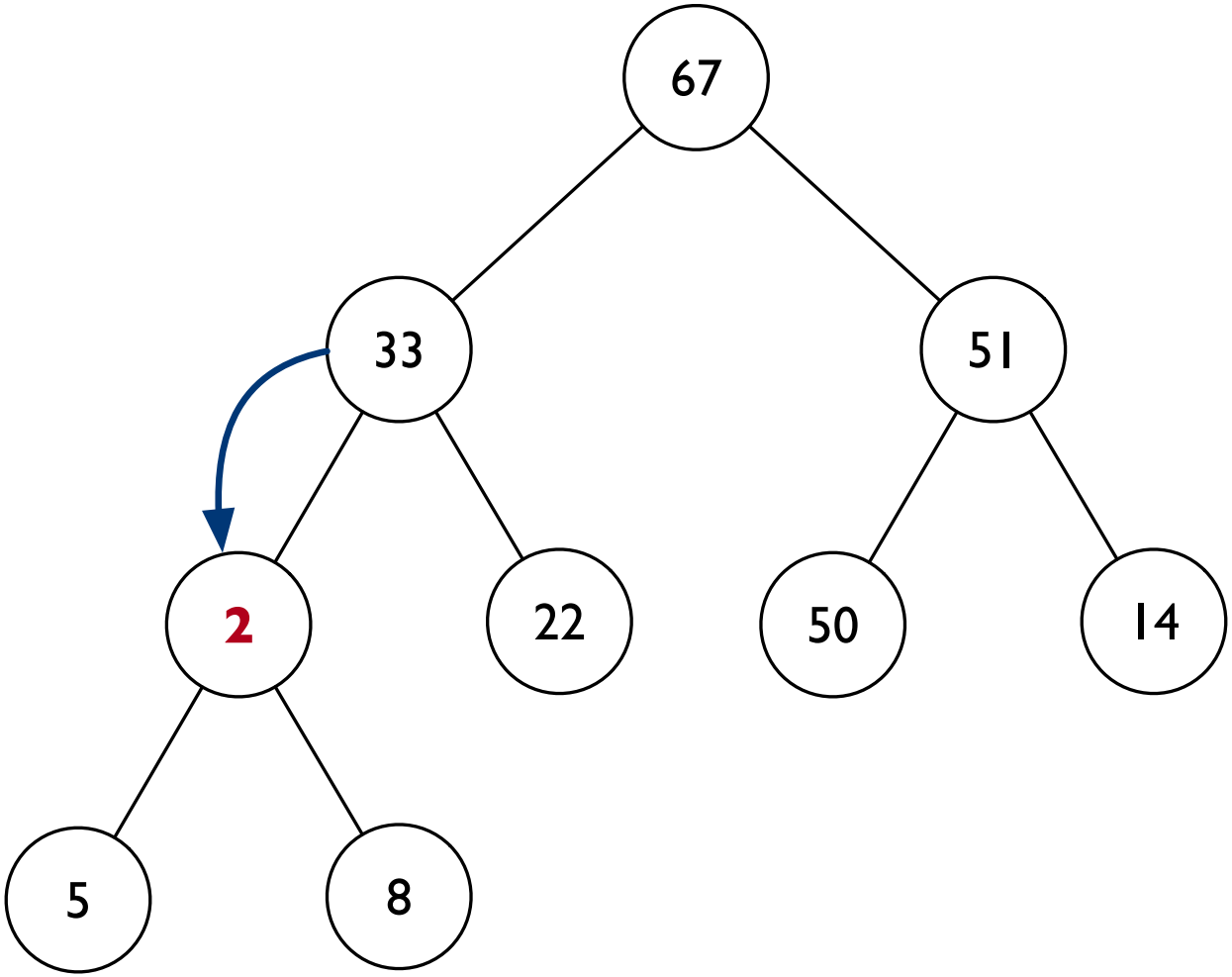
Sink (example)



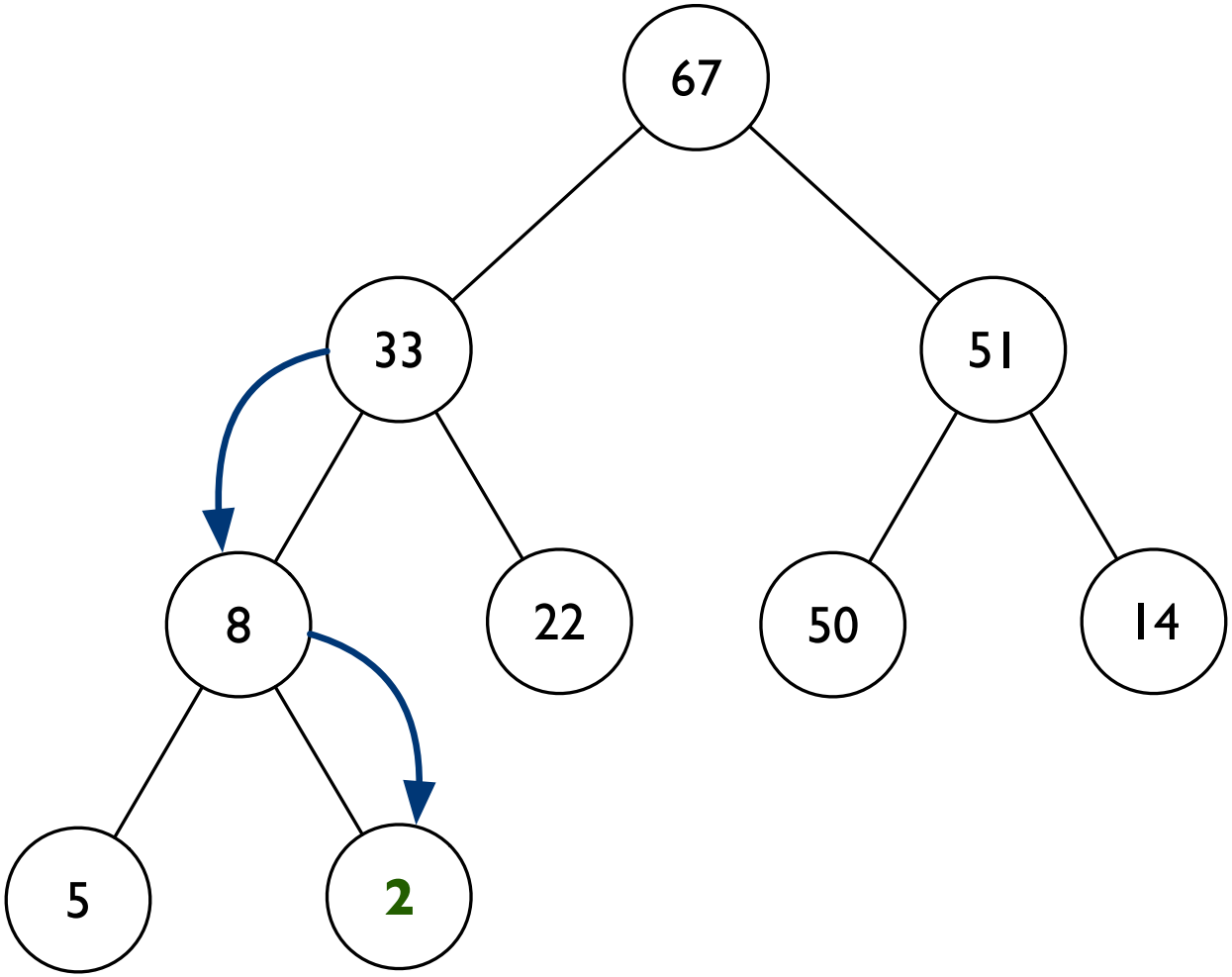
Sink (example)



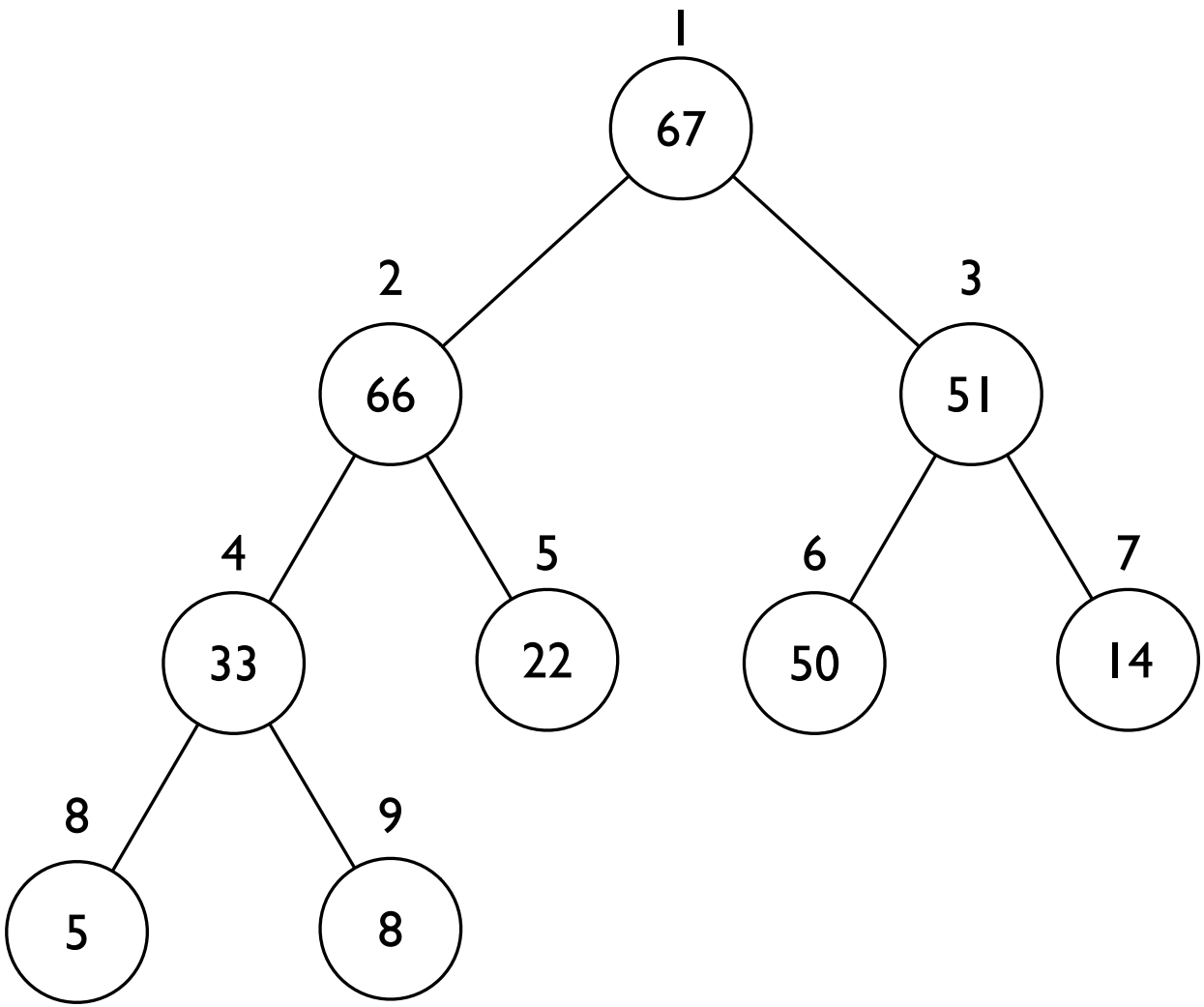
Sink (example)



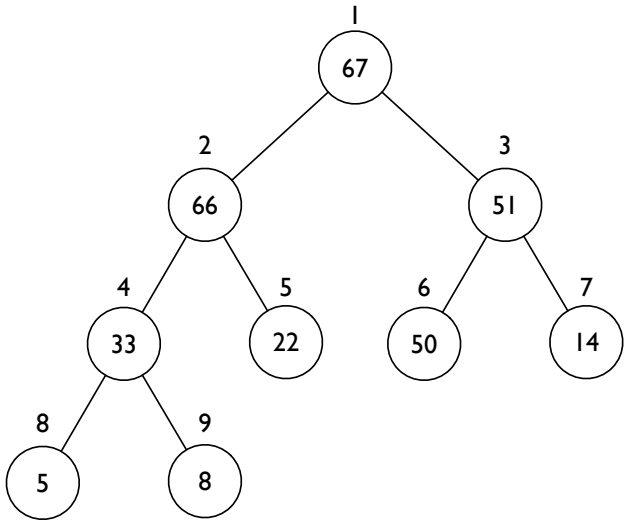
Sink (example)



Binary heap as an array

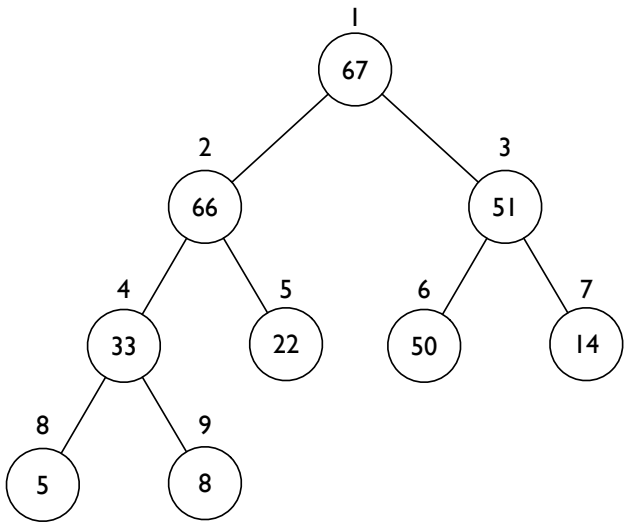


Binary heap as an array



	67	66	51	33	22	50	14	5	8
0	1	2	3	4	5	6	7	8	9

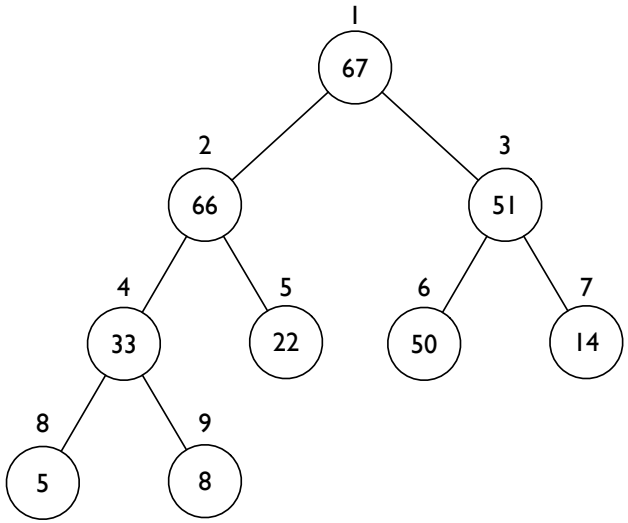
Binary heap as an array



	67	66	51	33	22	50	14	5	8
0	1	2	3	4	5	6	7	8	9

- Array representation: heap written out layer by layer

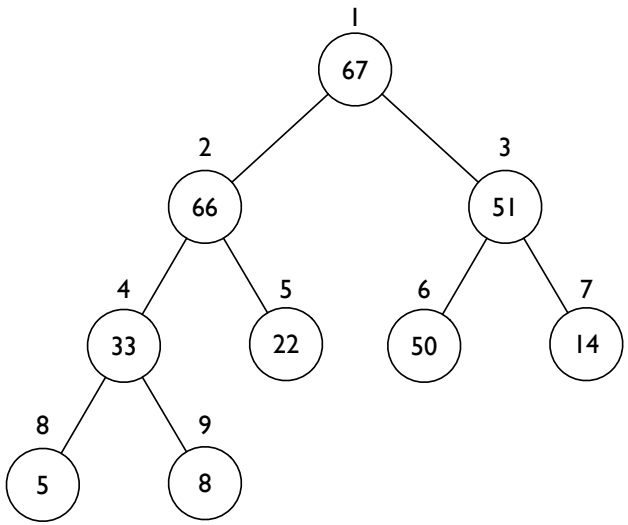
Binary heap as an array



	67	66	51	33	22	50	14	5	8
0	1	2	3	4	5	6	7	8	9

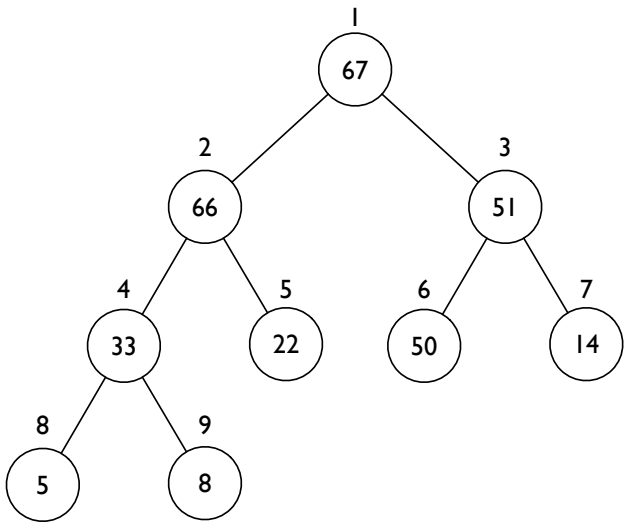
- Array representation: heap written out layer by layer
- 1-indexing not necessary, but makes the parent/child computations simpler

Binary heap as an array



	67	66	51	33	22	50	14	5	8
0	1	2	3	4	5	6	7	8	9

Binary heap as an array



	67	66	51	33	22	50	14	5	8
0	1	2	3	4	5	6	7	8	9

For a node a at index i:

- **Parent:** $i/2$
- **Left child:** $2i$
- **Right Child:** $2i + 1$

Back to Rust's `BinaryHeap`

```
pub fn peek_mut(&mut self) -> Option<PeekMut<T>>
```

Returns a mutable reference to the greatest item in the binary heap, or `None` if it is empty.

How does `BinaryHeap` guarantee the heap property is maintained, given that the caller gets control over the value in the root node?

peek_mut

```
pub fn peek_mut(&mut self) -> Option<PeekMut<T>> {
    if self.is_empty() {
        None
    } else {
        Some(PeekMut {
            heap: self,
            sift: true,
        })
    }
}
```

PeekMut

```rust
pub struct PeekMut<'a, T: 'a + Ord> {
    heap: &'a mut BinaryHeap<T>,
    sift: bool,
}
```

The magic!

```
impl<'a, T: Ord> Drop for PeekMut<'a, T> {
    fn drop(&mut self) {
        if self.sift {
            self.heap.sift_down(0);
        }
    }
}
```

The magic!

```
impl<'a, T: Ord> Drop for PeekMut<'a, T> {
    fn drop(&mut self) {
        if self.sift {
            self.heap.sift_down(0);
        }
    }
}
```

- `sift_down` implements the sink operation.

The magic!

```
impl<'a, T: Ord> Drop for PeekMut<'a, T> {
    fn drop(&mut self) {
        if self.sift {
            self.heap.sift_down(0);
        }
    }
}
```

- `sift_down` implements the sink operation.
- The heap property is thus restored by `drop`.

Loose end

- However: isn't the heap inconsistent between the mutation of the root and the drop of the `PeekMut` value?

Loose end

- However: isn't the heap inconsistent between the mutation of the root and the drop of the `PeekMut` value?

```
pub fn peek_mut(&mut self) -> Option<PeekMut<T>>
```

Returns a mutable reference to the greatest item in the binary heap, or `None` if it is empty.

- `peek_mut` borrows `self` mutably.

Loose end

- However: isn't the heap inconsistent between the mutation of the root and the drop of the `PeekMut` value?

```
pub fn peek_mut(&mut self) -> Option<PeekMut<T>>
```

Returns a mutable reference to the greatest item in the binary heap, or `None` if it is empty.

- `peek_mut` borrows `self` mutably.
- Rust ownership rules: **single mutable reference xor multiple immutable references**.

Loose end

- However: isn't the heap inconsistent between the mutation of the root and the drop of the `PeekMut` value?

```
pub fn peek_mut(&mut self) -> Option<PeekMut<T>>
```

Returns a mutable reference to the greatest item in the binary heap, or `None` if it is empty.

- `peek_mut` borrows `self` mutably.
- Rust ownership rules: **single mutable reference xor multiple immutable references**.
- Technically, the heap is temporarily in a consistent state, but there is no one to witness it ;).

The end