DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 30

1. CODE FORMATTING
2. PRIORITY QUEUES
3. POPULAR IMPLEMENTATION: BINARY HEAP

1. CODE FORMATTING

2. PRIORITY QUEUES

3. POPULAR IMPLEMENTATION: BINARY HEAP

DON'T GIVE UP ON CODE FORMATTING!

- Rust doesn't require any specific indentation
- Still a good idea to make your code readable

In [2]:
```
fn h(z:i32)->i32{let mut t=0.max(z.min(1)-0.max(z-1));for y in 1..=2.min(z){t+=h(z-y)}t}
```

DON'T GIVE UP ON CODE FORMATTING!

- Rust doesn't require any specific indentation
- Still a good idea to make your code readable

In [2]:
```
fn h(z:i32)->i32{let mut t=0.max(z.min(1)-0.max(z-1));for y in 1..=2.min(z){t+=h(z-y)}t}
```

In [3]:
```
for i in 0..10 {
    println!("{}:{}",i,h(i));
};
```

```
0:0
1:1
2:1
3:2
4:3
5:5
6:8
7:13
8:21
9:34
```

DIGRESSION: INTERNATIONAL OBFUSCATED C CODE CONTEST (IOCCC)

Digit recognition: `kopczynski.c`, Eryk Kopczyński, 2004:

```
main(O){int I,Q,l=O;if(I=l*4){l=6;if(l>5)l+=Q-8?l-(Q=getchar()-2)%2:l;if(Q*=2)O+="has dirtiest IF"[(I/-Q&12)-l/Q%4];}printf("%d\n",8+O%4);}
```

DIGRESSION: INTERNATIONAL OBFUSCATED C CODE CONTEST (IOCCC)

Digit recognition: `kopczynski.c`, Eryk Kopczyński, 2004:

```
main(O){int I,Q,l=O;if(I=l*4){l=6;if(l>5)l+=Q-8?l-(Q=getchar()-2)%2:l;if(Q*=2)O+="has dirtiest IF"[(I/-Q&12)-l/Q%4];}printf("%d\n",8+O%4);}
```

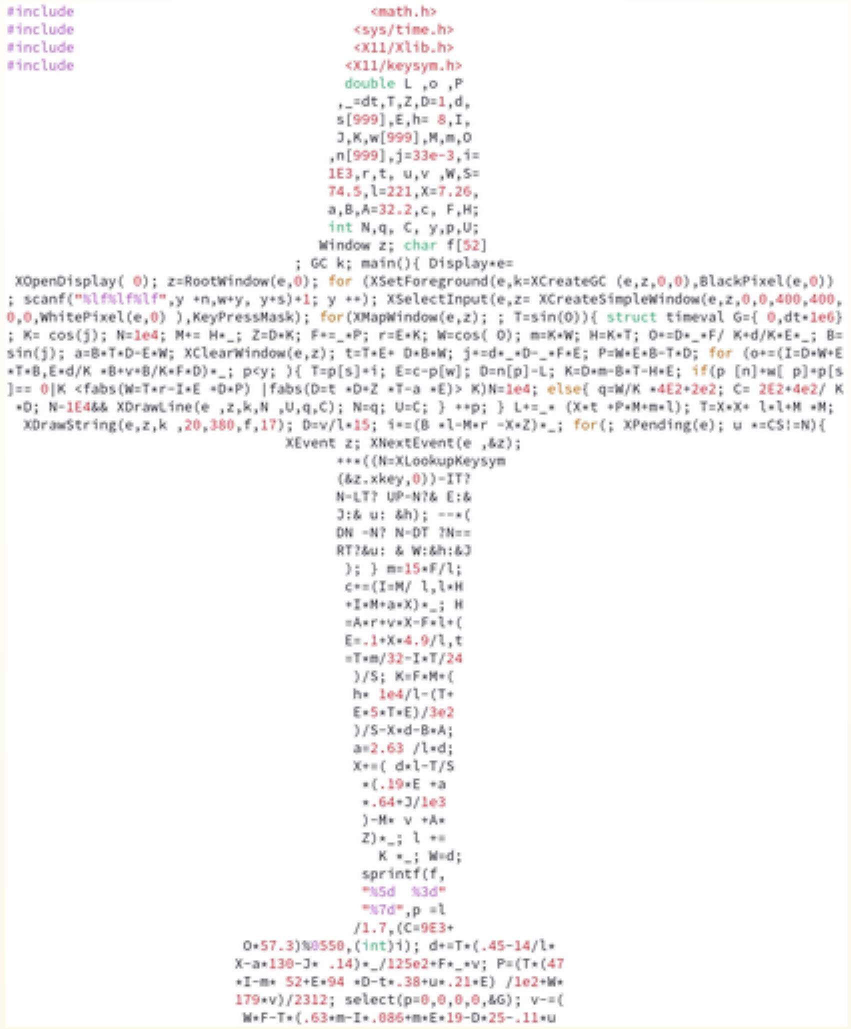
Need a flight simulator?

DIGRESSION: INTERNATIONAL OBFUSCATED C CODE CONTEST (IOCCC)

Digit recognition: `kopczynski.c`, Eryk Kopczyński, 2004:

```
main(O){int I,Q,l=O;if(I=l*4){l=6;if(l>5)l+=Q-8?l-(Q=getchar()-2)%2:l;if(Q*=2)O+="has dirtiest IF"[(I/-Q&12)-l/Q%4];}printf("%d\n",8+O%4);}
```

Need a flight simulator? `banks.c`, Carl Banks, 1998



TOOL FOR FORMATTING RUST CODE: `rustfmt`

- If you have Rust installed, you should already have it.

TOOL FOR FORMATTING RUST CODE: `rustfmt`

- If you have Rust installed, you should already have it.
- `rustfmt [filename]` replaces the file with nicely formatted version
  - use `rustfmt --backup [filename]` to save the original file

TOOL FOR FORMATTING RUST CODE: `rustfmt`

- If you have Rust installed, you should already have it.
- `rustfmt [filename]` replaces the file with nicely formatted version
  - use `rustfmt --backup [filename]` to save the original file

[see demo with comparison via kdiff3]

TOOL FOR FORMATTING RUST CODE: `rustfmt`

- If you have Rust installed, you should already have it.
- `rustfmt [filename]` replaces the file with nicely formatted version
  - use `rustfmt --backup [filename]` to save the original file

[see demo with comparison via kdiff3]

- `rustfmt --help`: see the command line parameters
- `rustfmt --print-config default`: default config that can be adjusted

1. CODE FORMATTING

2. PRIORITY QUEUES

3. POPULAR IMPLEMENTATION: BINARY HEAP

PRIORITY QUEUES

Standard queue:

- things returned in order in which they were inserted

PRIORITY QUEUES

Standard queue:

- things returned in order in which they were inserted

Priority queue:

- items have priorities
- highest priority items returned first

RUST STANDARD LIBRARY IMPLEMENTATION: BinaryHeap<T>

RUST STANDARD LIBRARY IMPLEMENTATION: `BinaryHeap<T>`

- Priorities provided by the ordering of elements of `T` (via trait `Ord`)
- Method `push(T)`:
  push element onto the heap
- Method `pop() -> Option<T>`:
  remove the greatest and return it

RUST STANDARD LIBRARY IMPLEMENTATION: `BinaryHeap<T>`

- Priorities provided by the ordering of elements of `T` (via trait `Ord`)
- Method `push(T)`:
  push element onto the heap
- Method `pop() -> Option<T>`:
  remove the greatest and return it

In [4]:
```
use std::collections::BinaryHeap;

let mut pq = BinaryHeap::new();

pq.push(2);
pq.push(7);
pq.push(3);

println!("{:?}",pq.pop());
println!("{:?}",pq.pop());

pq.push(3);
pq.push(4);

println!("\n{:?}",pq.pop());
println!("{:?}",pq.pop());
println!("{:?}",pq.pop());
println!("{:?}",pq.pop());
```

```
Some(7)
Some(3)

Some(4)
Some(3)
Some(2)
None
```

GETTING THE SMALLEST ELEMENT OUT FIRST

`Reverse<T>`: wrapper that reverses the ordering of elements of a type

In [5]:
```
3 < 4
```
Out[5]: true

In [6]:
```
use std::cmp::Reverse;
Reverse(3) < Reverse(4)
```
Out[6]: false

GETTING THE SMALLEST ELEMENT OUT FIRST

`Reverse<T>`: wrapper that reverses the ordering of elements of a type

In [5]:
```
3 < 4
```
Out[5]: true

In [6]:
```
use std::cmp::Reverse;
Reverse(3) < Reverse(4)
```
Out[6]: false

In [7]:
```
5 < 3
```
Out[7]: false

In [8]:
```
Reverse(5) < Reverse(3)
```
Out[8]: true

GETTING THE SMALLEST ELEMENT OUT FIRST

`Reverse<T>`: wrapper that reverses the ordering of elements of a type

In [5]:
```
3 < 4
```
Out[5]: true

In [6]:
```
use std::cmp::Reverse;
Reverse(3) < Reverse(4)
```
Out[6]: false

In [7]:
```
5 < 3
```
Out[7]: false

In [8]:
```
Reverse(5) < Reverse(3)
```
Out[8]: true

In [9]:
```
let mut pq = BinaryHeap::new();

pq.push(Reverse(3));
pq.push(Reverse(1));
pq.push(Reverse(7));

println!("{:?}",pq.pop());
println!("{:?}",pq.pop());

pq.push(Reverse(0));

println!("\n{:?}",pq.pop());
```

```
Some(Reverse(1))
Some(Reverse(3))

Some(Reverse(0))
```

DEFAULT LEXICOGRAPHIC ORDERING ON TUPLES AND STRUCTS

Lexicographic ordering:

- Compare first elements
- If equal, compare second elements
- If equal, compare third elements...

DEFAULT LEXICOGRAPHIC ORDERING ON TUPLES AND STRUCTS

Lexicographic ordering:

- Compare first elements
- If equal, compare second elements
- If equal, compare third elements...

TUPLES

```
In [10]: (3,4) < (2,7)

Out[10]: false
```

```
In [11]: (11,2,7) < (11,3,4)

Out[11]: true
```

DEFAULT LEXICOGRAPHIC ORDERING ON TUPLES AND STRUCTS

Lexicographic ordering:

- Compare first elements
- If equal, compare second elements
- If equal, compare third elements...

TUPLES

In [10]:
```
(3,4) < (2,7)
```
Out[10]: false

In [11]:
```
(11,2,7) < (11,3,4)
```
Out[11]: true

STRUCT (DERIVE `Ord`)

In [12]:
```
#[derive(PartialEq,Eq,PartialOrd,Ord,Debug)]
struct Point {
    x: i32,
    y: i32,
}
```

In [13]:
```
let p = Point{x:3,y:4};
let q = Point{x:2,y:7};
println!("{}", p < q);
println!("{}", p > q);
```
```
false
true
```

ANOTHER OPTION: IMPLEMENT YOUR OWN COMPARISON

- More complicated, won't cover today
- See the documentation for `Ord` or examples online

HOW TO IMPLEMENT A PRIORITY QUEUE?

Assumptions:

- At most n elements
- Comparison takes $O(1)$ time

HOW TO IMPLEMENT A PRIORITY QUEUE?

Assumptions:

- At most n elements
- Comparison takes $O(1)$ time

STRAIGHFORWARD

Representation: a vector of elements

HOW TO IMPLEMENT A PRIORITY QUEUE?

Assumptions:

- At most n elements
- Comparison takes $O(1)$ time

STRAIGHFORWARD

Representation: a vector of elements

Push:

- add to the end of the vector
- Time complexity: $O(1)$ (amortized) time

HOW TO IMPLEMENT A PRIORITY QUEUE?

Assumptions:

- At most n elements
- Comparison takes $O(1)$ time

STRAIGHFORWARD

Representation: a vector of elements

Push:

- add to the end of the vector
- Time complexity: $O(1)$ (amortized) time

Pop:

- go over all elements, select the greatest
- Time complexity: $O(n)$

1. CODE FORMATTING

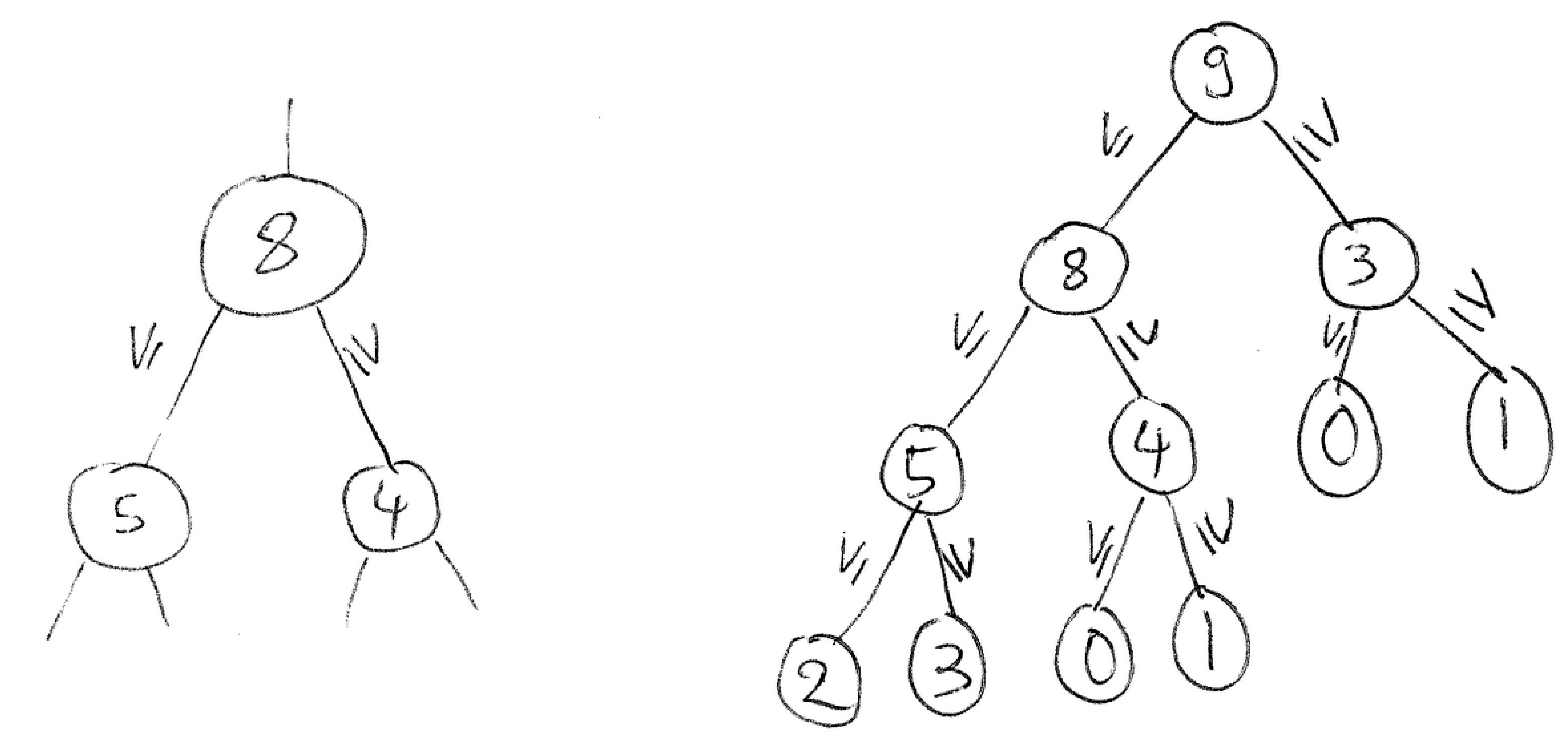
2. PRIORITY QUEUES

3. POPULAR IMPLEMENTATION: BINARY HEAP

BINARY HEAPS

- Data organized into a binary tree
- Every internal node not smaller than its children

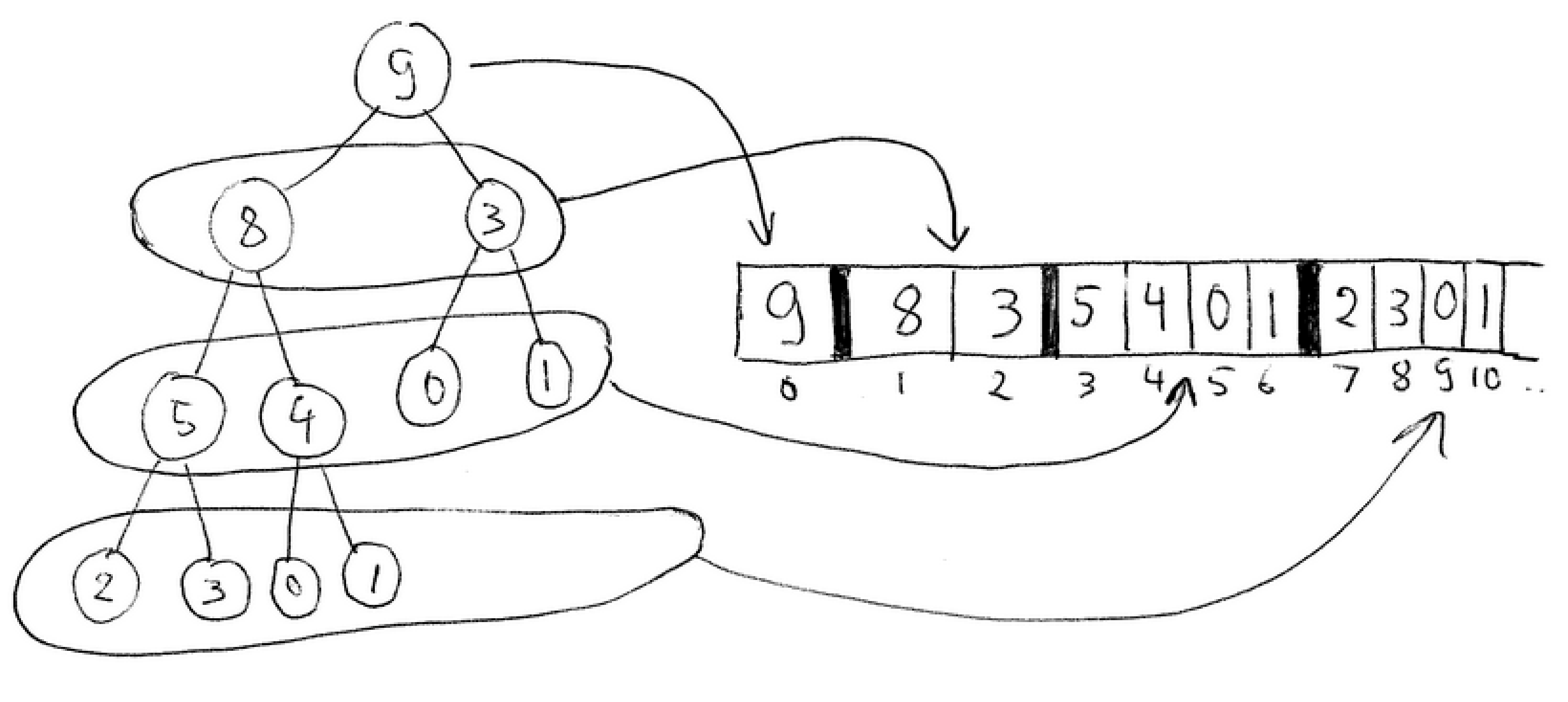
Basic property: The root has the current maximum, i.e., the answer to next `pop`



BINARY HEAPS

Efficient storage:

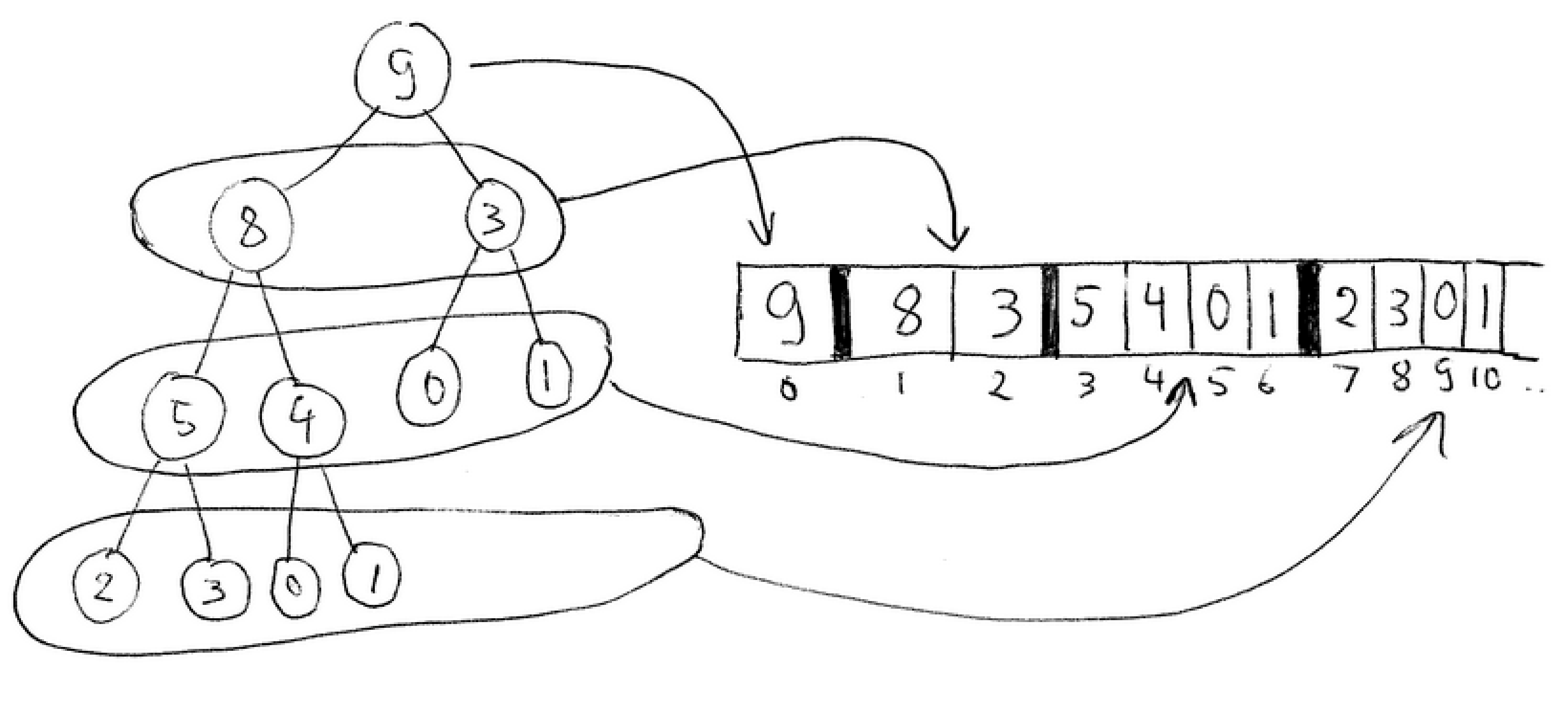
- Tree levels filled from left to right
- Can be mapped to a vector



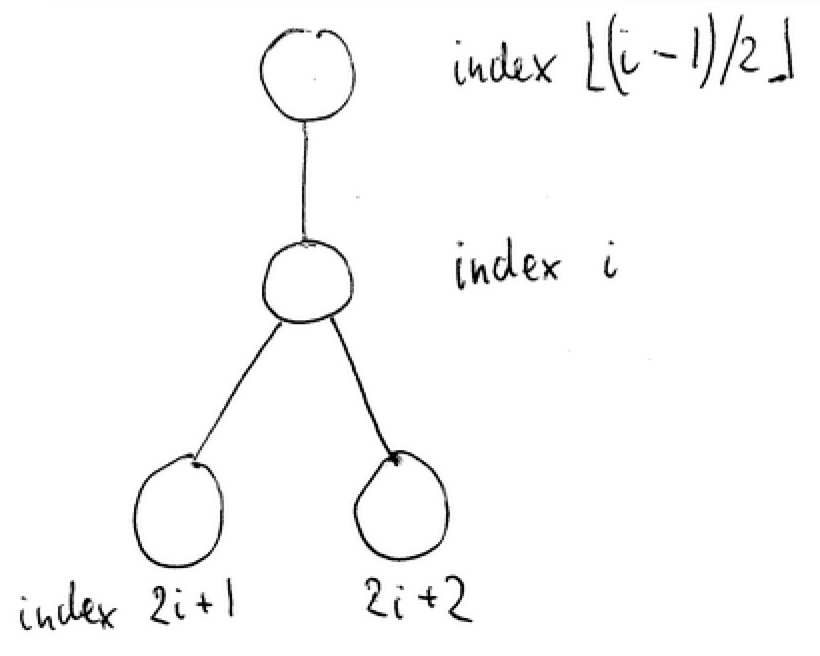
BINARY HEAPS

Efficient storage:

- Tree levels filled from left to right
- Can be mapped to a vector



- Easy to move to the parent or children using vector indices



HOW ARE OPERATIONS IMPLEMENTED?

HOW ARE OPERATIONS IMPLEMENTED?

PUSH

- add at the end the array
- fix the ordering by pushing the element up

HOW ARE OPERATIONS IMPLEMENTED?

PUSH

- add at the end the array
- fix the ordering by pushing the element up

POP

- remove and return the root
- replace with the last element
- fix the ordering, pushing the element down

HOW ARE OPERATIONS IMPLEMENTED?

PUSH

- add at the end the array
- fix the ordering by pushing the element up

POP

- remove and return the root
- replace with the last element
- fix the ordering, pushing the element down

COMPLEXITY OF PUSH AND POP

- Proportional to the number of levels
- So $O(\log n)$