DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 31

1. APPLICATIONS OF PRIORITY QUEUES: SORTING AND SHORTEST PATHS

2. SLICES

1. APPLICATIONS OF PRIORITY QUEUES: SORTING AND SHORTEST PATHS

2. SLICES

LAST TIME: PRIORITY QUEUES

Collection of items:

- `push`: insert an item
- `pop`: remove and return the greatest item in a collection

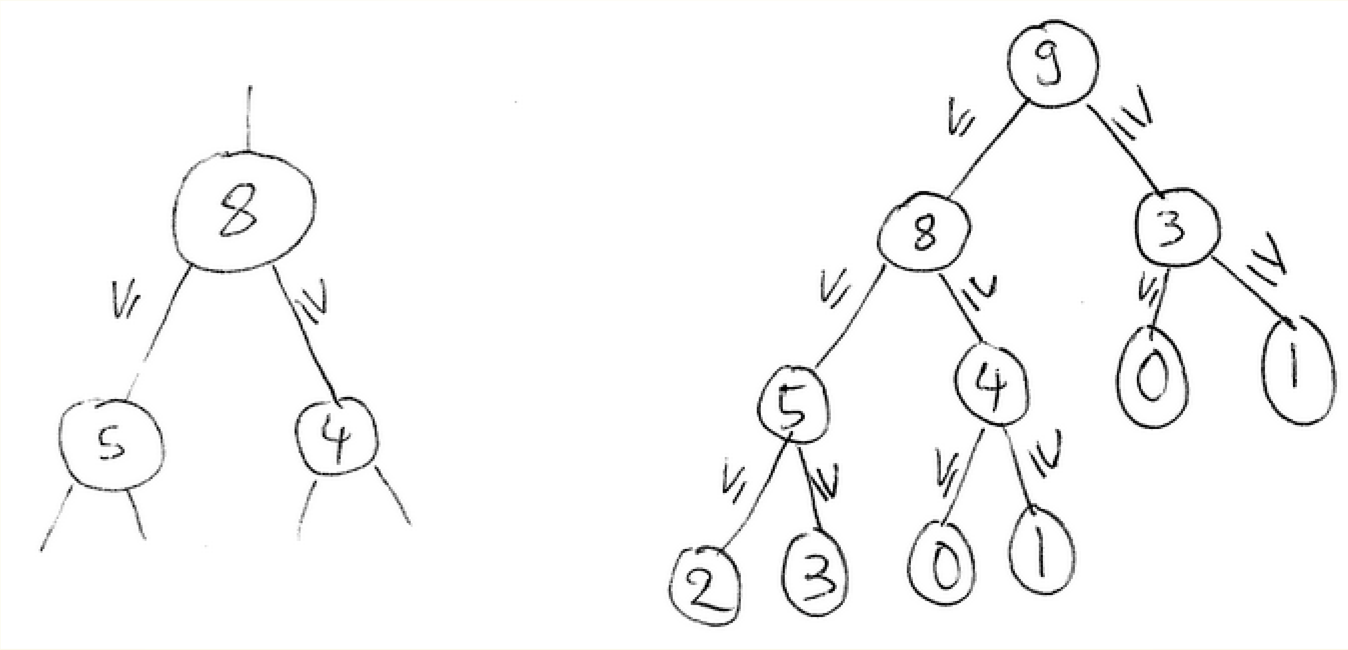
LAST TIME: PRIORITY QUEUES

Collection of items:

- `push`: insert an item
- `pop`: remove and return the greatest item in a collection

Popular implementation: **binary heap**

- push and pop in $O(\log n)$ time
- Rust: `std::collections::BinaryHeap<T>`



APPLICATION 1: SORTING A.K.A. HEAPSORT

APPLICATION 1: SORTING A.K.A. HEAPSORT

- Put everything into a priority queue
- Remove items in order

APPLICATION 1: SORTING A.K.A. HEAPSORT

- Put everything into a priority queue
- Remove items in order

In [2]:
```
use std::collections::BinaryHeap;

fn heap_sort(v:&mut Vec<i32>) {
    let mut pq = BinaryHeap::new();
    for v in v.iter() {
        pq.push(*v);
    }
    for i in (0..v.len()).rev() {
        v[i] = pq.pop().unwrap();
    }
}
```

APPLICATION 1: SORTING A.K.A. HEAPSORT

- Put everything into a priority queue
- Remove items in order

In [2]:
```
use std::collections::BinaryHeap;

fn heap_sort(v:&mut Vec<i32>) {
    let mut pq = BinaryHeap::new();
    for v in v.iter() {
        pq.push(*v);
    }
    for i in (0..v.len()).rev() {
        v[i] = pq.pop().unwrap();
    }
}
```

In [3]:
```
let mut v = vec![23,12,7,37,14,11];
heap_sort(&mut v);
v
```

Out[3]:
```
[7, 11, 12, 14, 23, 37]
```

?

APPLICATION 1: SORTING A.K.A. HEAPSORT

- Put everything into a priority queue
- Remove items in order

In [2]:
```
use std::collections::BinaryHeap;

fn heap_sort(v:&mut Vec<i32>) {
    let mut pq = BinaryHeap::new();
    for v in v.iter() {
        pq.push(*v);
    }
    for i in (0..v.len()).rev() {
        v[i] = pq.pop().unwrap();
    }
}
```

In [3]:
```
let mut v = vec![23,12,7,37,14,11];
heap_sort(&mut v);
v
```

Out[3]:
```
[7, 11, 12, 14, 23, 37]
```

Total running time: $O(n \log n)$ for n numbers

MORE DIRECT, USING RUST OPERATIONS

In [4]:
```
fn heap_sort_2(v:Vec<i32>) -> Vec<i32> {
    BinaryHeap::from(v).into_sorted_vec()
}
```

No extra memory allocated: the initial vector, intermediate binary heap, and final vector all use the same space on the heap

- `BinaryHeap::from(v)` consumes `v`
- `into_sorted_vec()` consumes the intermediate binary heap

MORE DIRECT, USING RUST OPERATIONS

In [4]:
```
fn heap_sort_2(v:Vec<i32>) -> Vec<i32> {
    BinaryHeap::from(v).into_sorted_vec()
}
```

No extra memory allocated: the initial vector, intermediate binary heap, and final vector all use the same space on the heap

- `BinaryHeap::from(v)` consumes `v`
- `into_sorted_vec()` consumes the intermediate binary heap

In [5]:
```
let mut v = vec![7,17,3,1,8,11];
heap_sort_2(v)
```

Out[5]:
```
[1, 3, 7, 8, 11, 17]
```

MORE DIRECT, USING RUST OPERATIONS

In [4]:
```rust
fn heap_sort_2(v:Vec<i32>) -> Vec<i32> {
    BinaryHeap::from(v).into_sorted_vec()
}
```

No extra memory allocated: the initial vector, intermediate binary heap, and final vector all use the same space on the heap

- `BinaryHeap::from(v)` consumes `v`
- `into_sorted_vec()` consumes the intermediate binary heap

In [5]:
```rust
let mut v = vec![7,17,3,1,8,11];
heap_sort_2(v)
```

Out[5]: [1, 3, 7, 8, 11, 17]

Sorting already provided for vectors (currently use other algorithms): `sort` and `sort_unstable`

In [6]:
```rust
let mut v = vec![7,17,3,1,8,11];
v.sort();
v
```

Out[6]: [1, 3, 7, 8, 11, 17]

In [7]:
```rust
let mut v = vec![7,17,3,1,8,11];
v.sort_unstable();
v
```

Out[7]: [1, 3, 7, 8, 11, 17]

APPLICATION 2: SHORTEST WEIGHTED PATHS (DIJKSTRA'S ALGORITHM)

- **Input graph:** edges with *positive* values, directed or undirected
- **Goal:** Compute all distances from a given vertex v

APPLICATION 2: SHORTEST WEIGHTED PATHS (DIJKSTRA'S ALGORITHM)

- **Input graph:** edges with *positive* values, directed or undirected
- **Goal:** Compute all distances from a given vertex v

[see the demo on the board]

APPLICATION 2: SHORTEST WEIGHTED PATHS (DIJKSTRA'S ALGORITHM)

- **Input graph:** edges with *positive* values, directed or undirected
- **Goal:** Compute all distances from a given vertex v

[see the demo on the board]

How it works:

- Greedily take the closest unprocessed vertex
  - Its distance must be correct
- Keep updating distances of unprocessed vertices

AUXILIARY GRAPH DEFINITIONS

In [8]:

```
type Vertex = usize;
type Distance = usize;
type Edge = (Vertex, Vertex, Distance);

#[derive(Debug,Copy,Clone)]
struct Outedge {
    vertex: Vertex,
    length: Distance,
}

type AdjacencyList = Vec<Outedge>;

#[derive(Debug)]
struct Graph {
    n: usize,
    outedges: Vec<AdjacencyList>,
}

impl Graph {
    fn create_directed(n:usize,edges:&Vec<Edge>) -> Graph {
        let mut outedges = vec![vec![];n];
        for (u, v, length) in edges {
            outedges[*u].push(Outedge{vertex: *v, length: *length});
        }
        Graph{n,outedges}
    }
}
```

LOAD OUR GRAPH

In [9]:
```
let n = 6;
let edges: Vec<Edge> = vec![(0,1,5),(0,2,2),(2,1,1),(2,4,1),(1,3,5),(4,3,1),(1,5,11),(3,5,5),(4,5,8)];
let graph = Graph::create_directed(n, &edges);
graph
```

Out[9]:
```
Graph { n: 6, outedges: [[Outedge { vertex: 1, length: 5 }, Outedge { vertex: 2, length: 2 }], [Outedge { vertex: 3, length: 5
}, Outedge { vertex: 5, length: 11 }], [Outedge { vertex: 1, length: 1 }, Outedge { vertex: 4, length: 1 }], [Outedge { verte
x: 5, length: 5 }], [Outedge { vertex: 3, length: 1 }, Outedge { vertex: 5, length: 8 }], []] }
```

OUR IMPLEMENTATION

In [10]:
```
let start: Vertex = 0;

let mut distances: Vec<Option<Distance> > = vec![None; graph.n];
distances[start] = Some(0);
```

In [11]:
```
use core::cmp::Reverse;

let mut pq = BinaryHeap::<Reverse<(Distance,Vertex)>>::new();
pq.push(Reverse((0,start)));
```

OUR IMPLEMENTATION

In [10]:
```
let start: Vertex = 0;

let mut distances: Vec<Option<Distance> > = vec![None; graph.n];
distances[start] = Some(0);
```

In [11]:
```
use core::cmp::Reverse;

let mut pq = BinaryHeap::<Reverse<(Distance,Vertex)>>::new();
pq.push(Reverse((0,start)));
```

In [12]:
```
while let Some(Reverse((dist,v))) = pq.pop() {
    if distances[v].unwrap() == dist {
        for Outedge{vertex,length} in graph.outedges[v].iter() {
            let new_dist = dist + *length;
            let update = match distances[*vertex] {
                None => {true} |
                Some(d) => {new_dist < d}
            };
            if update {
                distances[*vertex] = Some(new_dist);
                pq.push(Reverse((new_dist,*vertex)));
            }
        }
    }
};
```

OUR IMPLEMENTATION

In [10]:
```
let start: Vertex = 0;

let mut distances: Vec<Option<Distance> > = vec![None; graph.n];
distances[start] = Some(0);
```

In [11]:
```
use core::cmp::Reverse;

let mut pq = BinaryHeap::<Reverse<(Distance,Vertex)>>::new();
pq.push(Reverse((0,start)));
```

In [12]:
```
while let Some(Reverse((dist,v))) = pq.pop() {
    if distances[v].unwrap() == dist {
        for Outedge{vertex,length} in graph.outedges[v].iter() {
            let new_dist = dist + *length;
            let update = match distances[*vertex] {
                None => {true} |
                Some(d) => {new_dist < d}
            };
            if update {
                distances[*vertex] = Some(new_dist);
                pq.push(Reverse((new_dist,*vertex)));
            }
        }
    }
};
```

In [13]:
```
distances
```

Out[13]:
```
[Some(0), Some(3), Some(2), Some(4), Some(3), Some(9)]
```

1. APPLICATIONS OF PRIORITY QUEUES: SORTING AND SHORTEST PATHS

2. SLICES

SLICES

Slice = reference to subsection of the data

SLICES

Slice = reference to subsection of the data

Slices of an array:

- array of type `[T, _]`
- slice of type `&[T]` or `&mut [T]`

```
In [14]: {
             // immutable slice of an array
             let arr: [i32; 5] = [0,1,2,3,4];
             let slice: &[i32] = &arr[1..3];
             println!("{:?}",slice);
             println!("{}", slice[0]);
         };
```

```
[1, 2]
1
```

SLICES

Slice = reference to subsection of the data

Slices of an array:

- array of type `[T, _]`
- slice of type `&[T]` or `&mut [T]`

In [14]:

```
{
    // immutable slice of an array
    let arr: [i32; 5] = [0,1,2,3,4];
    let slice: &[i32] = &arr[1..3];
    println!("{:?}",slice);
    println!("{}", slice[0]);
};
```

```
[1, 2]
1
```

In [15]:

```
{
    // mutable slice of an array
    let mut arr = [0,1,2,3,4];
    let mut slice = &mut arr[2..4];
    println!("{:?}",slice);
    slice[0] = slice[0] * slice[0];
    println!("{}", slice[0]);
    println!("{:?}",arr);
};
```

```
[2, 3]
4
[0, 1, 4, 3, 4]
```

SLICES

Work for vectors too!

In [16]:
```
let mut v = vec![0,1,2,3,4];
{
    let slice = &v[1..3];
    println!("{:?}",slice);
};
```

```
[1, 2]
```

SLICES

Work for vectors too!

In [16]:
```
let mut v = vec![0,1,2,3,4];
{
    let slice = &v[1..3];
    println!("{:?}",slice);
};
```

```
[1, 2]
```

In [17]:
```
{
    let mut slice = &mut v[1..3];

    // iterating over slices works as well
    for x in slice.iter_mut() {
        *x *= 1000;
    }
};
v
```

Out[17]:
```
[0, 1000, 2000, 3, 4]
```

SLICES ARE REFERENCES: ALL BORROWING RULES STILL APPLY!

- At most one mutable reference at a time
- No immutable references allowed with a mutable reference
- Many immutable references allowed simultaneously

SLICES ARE REFERENCES: ALL BORROWING RULES STILL APPLY!

- At most one mutable reference at a time
- No immutable references allowed with a mutable reference
- Many immutable references allowed simultaneously

In [18]:

```
// this won't work!
let mut v = vec![1,2,3,4,5,6,7];
{
    let ref_1 = &mut v[2..5];
    let ref_2 = &v[1..3];
    ref_1[0] = 7;
    println!("{}",ref_2[1]);
}
```

```
    let ref_2 = &v[1..3];
                 ^ immutable borrow occurs here
    let ref_1 = &mut v[2..5];
                     ^ mutable borrow occurs here
    ref_1[0] = 7;
    ^^^^^^^^ mutable borrow later used here
cannot borrow `v` as immutable because it is also borro
wed as mutable
```

SLICES ARE REFERENCES: ALL BORROWING RULES STILL APPLY!

- At most one mutable reference at a time
- No immutable references allowed with a mutable reference
- Many immutable references allowed simultaneously

In [18]:

```
// this won't work!
let mut v = vec![1,2,3,4,5,6,7];
{
    let ref_1 = &mut v[2..5];
    let ref_2 = &v[1..3];
    ref_1[0] = 7;
    println!("{}",ref_2[1]);
}
```

```
    let ref_2 = &v[1..3];
                ^ immutable borrow occurs here
    let ref_1 = &mut v[2..5];
                    ^ mutable borrow occurs here
    ref_1[0] = 7;
    ^^^^^^^^ mutable borrow later used here
cannot borrow `v` as immutable because it is also borro
wed as mutable
```

In [19]:

```
// and this reordering will
let mut v = vec![1,2,3,4,5,6,7];
{
    let ref_1 = &mut v[2..5];
    ref_1[0] = 7;
    let ref_2 = &v[1..3];
    println!("{}",ref_2[1]);
};
```

```
7
```

MEMORY REPRESENTATION OF SLICES

- Pointer (to heap or stack)
- Length

Compared to vector: no capacity (cannot be extended)