CSE 373: Disjoint sets continued

Michael Lee

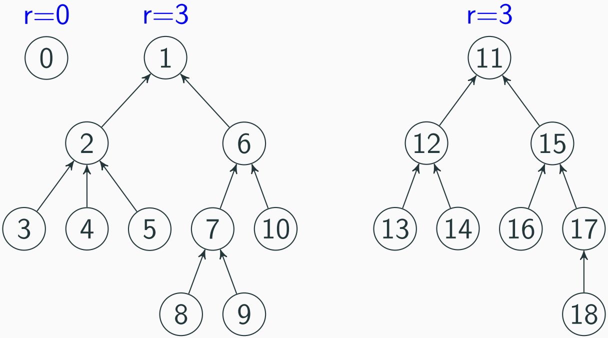
Friday, Mar 2, 2018

Warmup

Consider the following disjoint set.

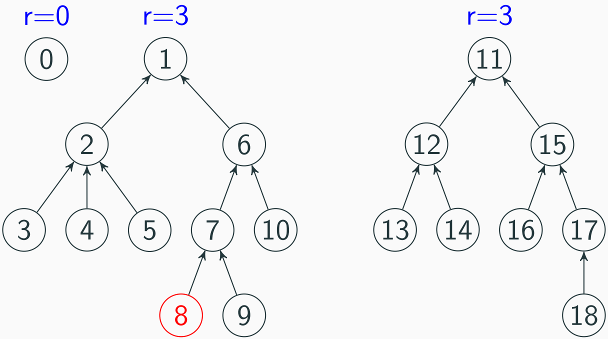
What happens if we run `findSet(8)` then `union(4, 17)`?

Note: the `union(...)` method internally calls `findSet(...)`.



Warmup

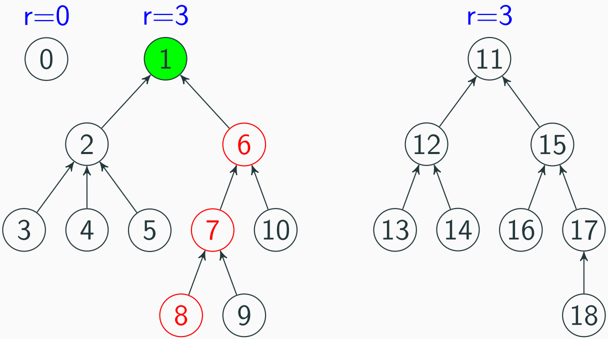
What happens when we run `findSet(8)`?



Step 1: We find the node corresponding to 8 in $\mathcal{O}(1)$ time

Warmup

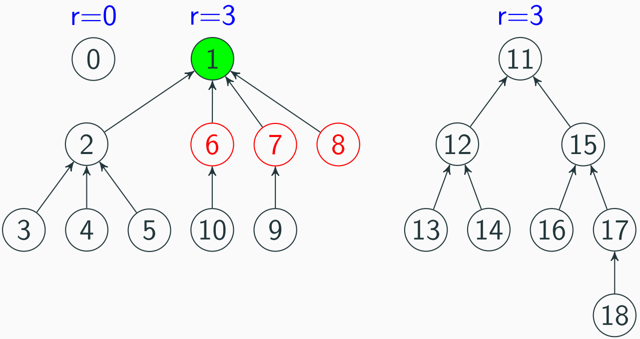
What happens when we run `findSet(8)`?



Step 2: We travel up the tree until we find the root

Warmup

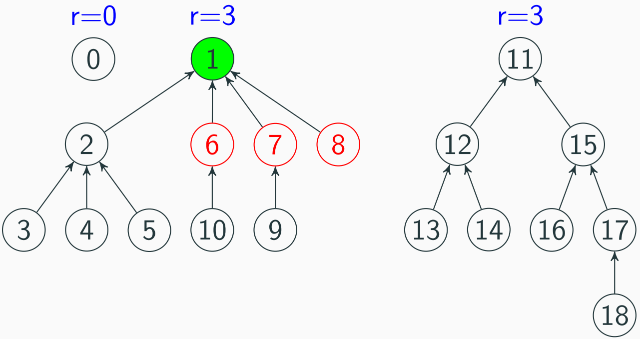
What happens when we run `findSet(8)`?



Step 3: We move each node we passed by (every red node) to point directly at the root.

Warmup

What happens when we run `findSet(8)`?

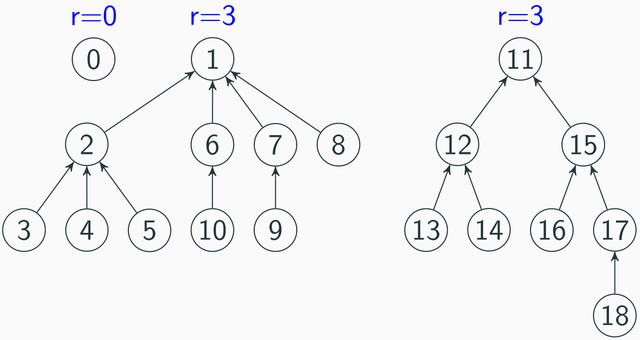


Step 3: We move each node we passed by (every red node) to point directly at the root.

Note: we do not update the rank (too expensive)

Warmup

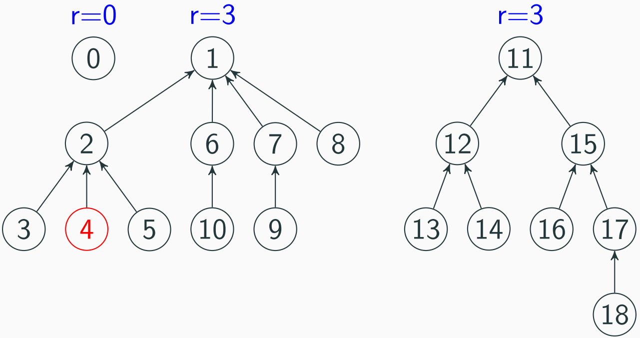
What happens if we run `union(4, 17)`?



link(findSet(4), findSet(17))

Warmup

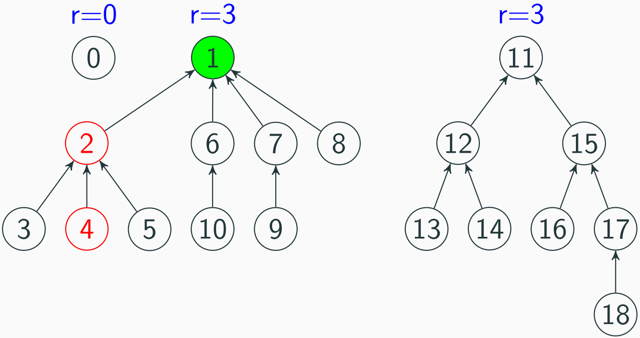
What happens if we run `union(4, 17)`?



Step 1: We first run `findSet(4)`

Warmup

What happens if we run `union(4, 17)`?

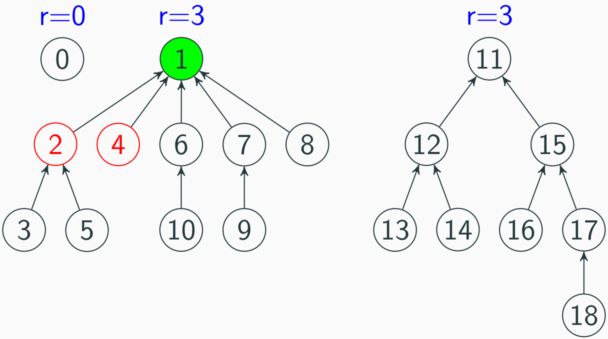


Step 1: We first run `findSet(4)`.

So we need to crawl up and find the parent...

Warmup

What happens if we run `union(4, 17)`?



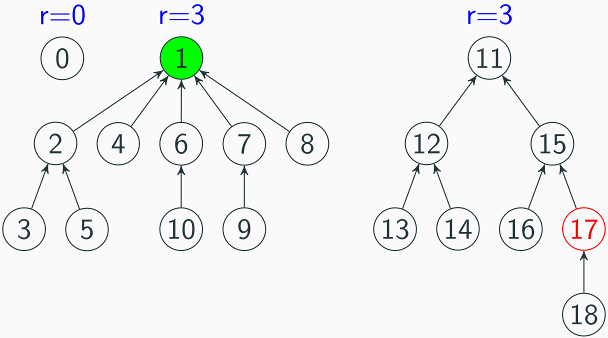
Step 1: We first run `findSet(4)`.

So we need to crawl up and find the parent...

...and make node "4" point directly at the root.

Warmup

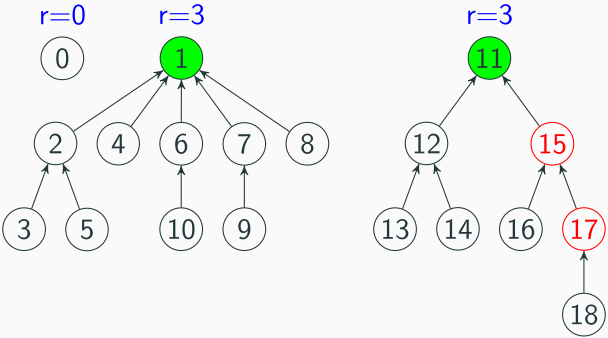
What happens if we run `union(4, 17)`?



Step 2: We next run `findSet(17)` and repeat the process.

Warmup

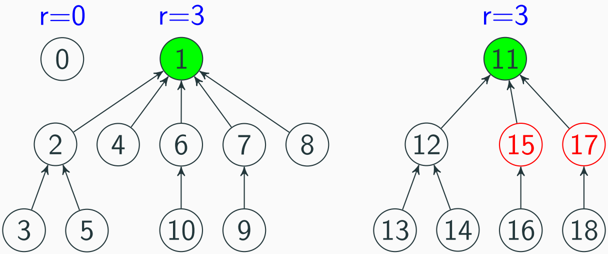
What happens if we run `union(4, 17)`?



Step 2: We next run `findSet(17)` and repeat the process.

Warmup

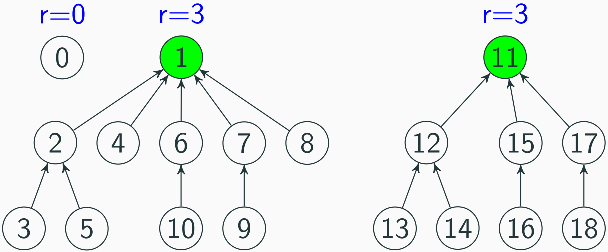
What happens if we run `union(4, 17)`?



Step 2: We next run `findSet(17)` and repeat the process.

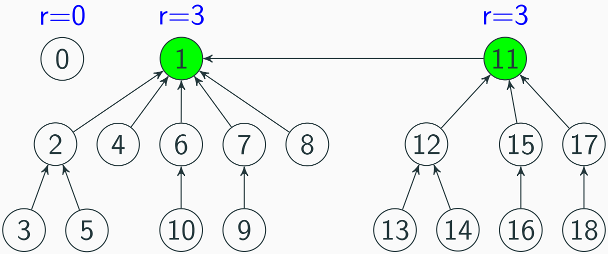
Warmup

We've finished `findSet(4)` and `findSet(17)`, so now we need to finish the rest of `union(4, 17)` by linking the two trees together.



Warmup

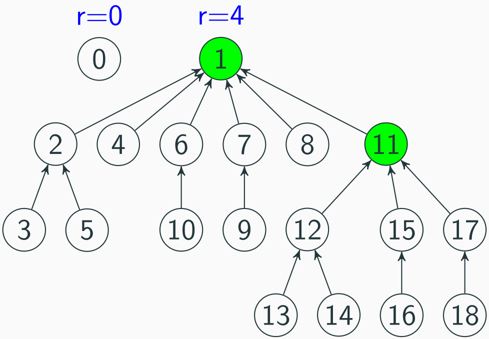
We've finished `findSet(4)` and `findSet(17)`, so now we need to finish the rest of `union(4, 17)` by linking the two trees together.



The ranks are the same, so we arbitrarily make set 1 the root and make set 11 the child.

Warmup

We've finished `findSet(4)` and `findSet(17)`, so now we need to finish the rest of `union(4, 17)` by linking the two trees together.



We then update the rank of set 1 and "forget" the rank of set 11.

Path compression: runtime

Now, what are the worst-case and best-case runtime of the following?

- **`makeSet(x)`**:
- **`findSet(x)`**:
- **`union(x, y)`**:

Path compression: runtime

Now, what are the worst-case and best-case runtime of the following?

- **`makeSet(x)`**:
  $\mathcal{O}(1)$ – still the same
- **`findSet(x)`**:
  In the best case, $\mathcal{O}(1)$, in the worst case $\mathcal{O}(\log(n))$
- **`union(x, y)`**:
  In the best case, $\mathcal{O}(1)$, in the worst case $\mathcal{O}(\log(n))$

Back to Kruskal's

Why are we doing this? To help us implement Kruskal's algorithm:

```
def kruskal():
    for (v : vertices):
        makeMST(v)

    sort edges in ascending order by their weight

    mst = new SomeSet<Edge>()
    for (edge : edges):
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)

    return mst
```

- `makeMST(v)` takes $\mathcal{O}(t_m)$ time
- `findMST(v)`: takes $\mathcal{O}(t_f)$ time
- `union(u, v)`: takes $\mathcal{O}(t_u)$ time

Back to Kruskal's

We concluded that the runtime is:

$$\mathcal{O}\left(\underbrace{|V| \cdot t_m}_{\text{setup}} + \underbrace{|E| \cdot \log(|E|)}_{\text{sorting edges}} + \underbrace{|E| \cdot t_f + |V| \cdot t_u}_{\text{core loop}}\right)$$

Back to Kruskal's

We concluded that the runtime is:

$$\mathcal{O}\left(\underbrace{|V| \cdot t_m}_{\text{setup}} + \underbrace{|E| \cdot \log(|E|)}_{\text{sorting edges}} + \underbrace{|E| \cdot t_f + |V| \cdot t_u}_{\text{core loop}}\right)$$

Well, we just said that in the worst case:

- $t_m \in \mathcal{O}(1)$
- $t_f \in \mathcal{O}(\log(|V|))$
- $t_u \in \mathcal{O}(\log(|V|))$

Back to Kruskal's

We concluded that the runtime is:

$$\mathcal{O}\left(\underbrace{|V| \cdot t_m}_{\text{setup}} + \underbrace{|E| \cdot \log(|E|)}_{\text{sorting edges}} + \underbrace{|E| \cdot t_f + |V| \cdot t_u}_{\text{core loop}}\right)$$

Well, we just said that in the worst case:

- $t_m \in \mathcal{O}(1)$
- $t_f \in \mathcal{O}(\log(|V|))$
- $t_u \in \mathcal{O}(\log(|V|))$

So the worst-case overall runtime of Kruskal's is:

$$\mathcal{O}(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|))$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

One minor improvement: since our edge weights are numbers, we can likely use a *linear sort* and improve the runtime to:

$$\mathcal{O}\left(|V| + |E| + (|E| + |V|) \cdot \log(|V|)\right)$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

One minor improvement: since our edge weights are numbers, we can likely use a *linear sort* and improve the runtime to:

$$\mathcal{O}\left(|V| + |E| + (|E| + |V|) \cdot \log(|V|)\right)$$

We can drop the $|V| + |E|$ (they're dominated by the last term):

$$\mathcal{O}\left((|E| + |V|) \cdot \log(|V|)\right)$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}\left(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|)\right)$$

One minor improvement: since our edge weights are numbers, we can likely use a *linear sort* and improve the runtime to:

$$\mathcal{O}\left(|V| + |E| + (|E| + |V|) \cdot \log(|V|)\right)$$

We can drop the $|V| + |E|$ (they're dominated by the last term):

$$\mathcal{O}\left((|E| + |V|) \cdot \log(|V|)\right)$$

...and we're left with something that's basically the same as Prim.

Disjoint-sets, amortized analysis

...or are we?

Disjoint-sets, amortized analysis

...or are we?

Observation: each call to `findSet(x)` improves all future calls.

How much of a difference does that make?

Disjoint-sets, amortized analysis

...or are we?

Observation: each call to `findSet(x)` improves all future calls.
How much of a difference does that make?

Interesting result:

It turns out `union` and `find` are *amortized* $\log^*(n)$.

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

What does this mean?

Interlude: repeated exponentiation

Observation:

- Multiplication is a shorthand for repeated addition*

$$2 \times 5 = 2 + 2 + 2 + 2$$

Interlude: repeated exponentiation

Observation:

- Multiplication is a shorthand for repeated addition*

$$2 \times 5 = 2 + 2 + 2 + 2$$

- Exponentiation is a shorthand for repeated multiplication*

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2$$

Interlude: repeated exponentiation

Observation:

- Multiplication is a shorthand for repeated addition*

$$2 \times 5 = 2 + 2 + 2 + 2$$

- Exponentiation is a shorthand for repeated multiplication*

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2$$

- Is there a way of expressing repeated exponentiation?

$$2 \, ?? \, 5 = 2^{2^{2^{2^2}}}$$

Interlude: repeated exponentiation

Observation:

- Multiplication is a shorthand for repeated addition*

$$2 \times 5 = 2 + 2 + 2 + 2$$

- Exponentiation is a shorthand for repeated multiplication*

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2$$

- Is there a way of expressing repeated exponentiation?

$$2 \,??\, 5 = 2^{2^{2^{2^2}}}$$

- Why stop there – is there a way of expressing repeated whatever-it-is-we-did up above?

$$2 \,??!!???\, 5 = 2 \,??\, 2 \,??\, 2 \,??\, 2 \,??\, 2$$

*assuming we use only integers

Interlude: Knuth's up-arrow notation

Yes – it's called **Knuth's up-arrow notation**

- Repeated addition (multiplication) is still the same:

$$2 \times 5 = 2 + 2 + 2 + 2$$

Interlude: Knuth's up-arrow notation

Yes – it's called **Knuth's up-arrow notation**

- Repeated addition (multiplication) is still the same:

$$2 \times 5 = 2 + 2 + 2 + 2$$

- A single arrow means *repeated multiplication* – exponentiation

$$2 \uparrow 5 = 2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 16$$

Interlude: Knuth's up-arrow notation

Yes – it's called **Knuth's up-arrow notation**

- Repeated addition (multiplication) is still the same:

$$2 \times 5 = 2 + 2 + 2 + 2$$

- A single arrow means *repeated multiplication* – exponentiation

$$2 \uparrow 5 = 2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 16$$

- Two arrows means *repeated exponentiation* – tetration

$$2 \uparrow\uparrow 5 = 2 \uparrow 2 \uparrow 2 \uparrow 2 \uparrow 2 = 2^{2^{2^{2^{2}}}}$$

Interlude: Knuth's up-arrow notation

Yes – it's called **Knuth's up-arrow notation**

- Repeated addition (multiplication) is still the same:

$$2 \times 5 = 2 + 2 + 2 + 2$$

- A single arrow means *repeated multiplication* – exponentiation

$$2 \uparrow 5 = 2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 16$$

- Two arrows means *repeated exponentiation* – tetration

$$2 \uparrow\uparrow 5 = 2 \uparrow 2 \uparrow 2 \uparrow 2 \uparrow 2 = 2^{2^{2^{2^{2}}}}$$

- Three arrows means *repeated tetration*

$$2 \uparrow\uparrow\uparrow 5 = 2 \uparrow\uparrow 2 \uparrow\uparrow 2 \uparrow\uparrow 2 \uparrow\uparrow 2$$

Interlude: Knuth's up-arrow notation

Yes – it's called **Knuth's up-arrow notation**

- Repeated addition (multiplication) is still the same:

$$2 \times 5 = 2 + 2 + 2 + 2$$

- A single arrow means *repeated multiplication* – exponentiation

$$2 \uparrow 5 = 2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 16$$

- Two arrows means *repeated exponentiation* – tetration

$$2 \uparrow\uparrow 5 = 2 \uparrow 2 \uparrow 2 \uparrow 2 \uparrow 2 = 2^{2^{2^{2^{2}}}}$$

- Three arrows means *repeated tetration*

$$2 \uparrow\uparrow\uparrow 5 = 2 \uparrow\uparrow 2 \uparrow\uparrow 2 \uparrow\uparrow 2 \uparrow\uparrow 2$$

- etc...

Interlude: Knuth's up-arrow notation

These functions all also have *inverses*

Interlude: Knuth's up-arrow notation

These functions all also have *inverses*

- Division is the inverse of multiplication:

$$\frac{(2 \times 5)}{2} = 5$$

Interlude: Knuth's up-arrow notation

These functions all also have *inverses*

- Division is the inverse of multiplication:

$$\frac{(2 \times 5)}{2} = 5$$

- $\log(...)$ is the inverse of $\uparrow$ (exponentiation)

$$\log_2(2 \uparrow 5) = \log_2(2^5) = 5$$

Interlude: Knuth's up-arrow notation

These functions all also have *inverses*

- Division is the inverse of multiplication:

$$\frac{(2 \times 5)}{2} = 5$$

- $\log(...)$ is the inverse of $\uparrow$ (exponentiation)

$$\log_2(2 \uparrow 5) = \log_2(2^5) = 5$$

- $\log^*(...)$ is the inverse of $\uparrow\uparrow$ (tetration)

$$\log^*_2(2 \uparrow\uparrow 5) = \log^*_2(2^{2^{2^{2^2}}}) = 5$$

Up-arrows and iterated log

A slightly modified definition:

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

This is equivalent to the inverse of $b \uparrow\uparrow x$.

Up-arrows and iterated log

A slightly modified definition:

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

This is equivalent to the inverse of $b \uparrow\uparrow x$.

What does this look like?

- $\log^*(2 \uparrow\uparrow 1) = \log *(2) = \log(2) = 1$

Up-arrows and iterated log

A slightly modified definition:

Iterated log

The expression $\log^*_b(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

This is equivalent to the inverse of $b \uparrow\uparrow x$.

What does this look like?

- $\log^*(2 \uparrow\uparrow 1) = \log * (2) = \log(2) = 1$
- $\log^*(2 \uparrow\uparrow 2) = \log^*(2^2) = \log(\log(4)) = 2$

Up-arrows and iterated log

A slightly modified definition:

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

This is equivalent to the inverse of $b \uparrow\uparrow x$.

What does this look like?

- $\log^*(2 \uparrow\uparrow 1) = \log*(2) = \log(2) = 1$
- $\log^*(2 \uparrow\uparrow 2) = \log^*(2^2) = \log(\log(4)) = 2$
- $\log^*(2 \uparrow\uparrow 3) = \log^*(2^{2^2}) = \log(\log(\log(8))) = 3$

Up-arrows and iterated log

A slightly modified definition:

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

This is equivalent to the inverse of $b \uparrow\uparrow x$.

What does this look like?

- $\log^*(2 \uparrow\uparrow 1) = \log *(2) = \log(2) = 1$
- $\log^*(2 \uparrow\uparrow 2) = \log^*(2^2) = \log(\log(4)) = 2$
- $\log^*(2 \uparrow\uparrow 3) = \log^*(2^{2^2}) = \log(\log(\log(8))) = 3$
- $\log^*(2 \uparrow\uparrow 4) = \log^*(2^{2^{2^2}}) = \log(\log(\log(\log(65536)))) = 4$

Up-arrows and iterated log

A slightly modified definition:

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

This is equivalent to the inverse of $b \uparrow\uparrow x$.

What does this look like?

- $\log^*(2 \uparrow\uparrow 1) = \log *(2) = \log(2) = 1$
- $\log^*(2 \uparrow\uparrow 2) = \log^*(2^2) = \log(\log(4)) = 2$
- $\log^*(2 \uparrow\uparrow 3) = \log^*(2^{2^2}) = \log(\log(\log(8))) = 3$
- $\log^*(2 \uparrow\uparrow 4) = \log^*(2^{2^{2^2}}) = \log(\log(\log(\log(65536)))) = 4$
- $\log^*(2 \uparrow\uparrow 5) = \log^*(2^{2^{2^{2^2}}}) =$ $\log(\log(\log(\log(\log(2^{65536}))))) = 5$

A big number

And what exactly is 2^{65536}?

A big number

And what exactly is 2^{65536}?

$= 2003529930406846464979072351560255750447825475569751419$
$2650169737108940595563114530895061308809333481010382343429072$
$6318182294938211881266886950636476154702916504187191635158796$
$6347219442930927982084309104855990570159318959639524863372367$
$2030029169695921561087649488892540908059114570376752085002066$
$7156370236612635974714480711177481588091413574272096719015183$
$6282560618091458852699826141425030123391108273603843767876449$
$0432059603791244909057075603140350761625624760318637931264847$
$0374378295497561377098160461441330869211810248595915238019533$
$1030292162800160568670105651646750568038741529463842244845292$
$5373614425336143737290883037946012747249584148649159306472520$
$1515569392262818069165079638106413227530726714399815850881129$
$2628901134237782705567421080070065283963322155077831214288551$

A big number

6755540733451072131124273995629827197691500548839052238043570
4584819795639315785351001899200002414196370681355984046403947
2194016069517690156119726982337890017641517190051133466306898
1402193834814354263873065395529696913880241581618595611006403
6211979610185953480278716720012260464249238511139340046435162
3867567078745259464670903886547743483217897012764455529409092
0219595857516229733335761595523948852975799540284719435299135
4376370598692891375715374000198639433246489005254310662966916
5243419174691389632476560289415199775477703138064781342309596
1909606545913008901888875880847336259560654448885014473357060
5881709016210849971452956834406197969056546981363116205357936
9791403236328496233046421066136200220175787851857409162050489
7117818204001872829399434461862243280098373237649318147898481
1945271300744022076568091037620399920349202390662626449190916

A big number

7985461515778839060397720759279378852241294301017458086862263
3692847258514030396155585643303854506886522131148136384083847
7826379045960718687672850976347127198889068047824323039471865
0525660978150729861141430305816927924971409161059417185352275
8875044775922183011587807019755357222414000195481020056617735
8978149953232520858975346354700778669040642901676380816174055
0405117670093673202804549339027992491867306539931640720492238
4748152806191669009338057321208163507076343516698696250209690
2316285935007187419057916124153689751480826190484794657173660
1005892476655445840838334790544144817684255327207315586349347
6051374197795251903650321980201087647383686825310251833775339
0886142618480037400808223810407646887847164755294532694766170
0424461063311238021134588694532200116564076327023074292426051

A big number

6340696503084422585596703927186946115851379338647569974856867
0079823960604393478850861649260304945061743412365828352144806
7266768418070837548622114082365798029612000274413244384324023
3125740354501935242877643088023285085588608996277445816468085
7875115807014743763867976955049991643998284357290415378143438
8473034842619033888414940313661398542576355771053355802066221
8557706008255128889333222643628198483861323957067619140963853
3832374343758830859233722284644287996245605476932428998432652
6773783731732880632107532112386806046747084280511664887090847
7029120816110491255559832236624486855665140268464120969498259
0565519216188104341226838996283071654868525536914850299539675
5039549383718534059000961874894739928804324963731657538036735
8671017578399481847179849824694806053208199606618343401247609

A big number

6639519778021441199752546704080608499344178256285092726523709
8986515394621930046073645079262129759176982938923670151709920
9153156781443979124847570623780460000991829332130688057004659
1458387208088016887445835557926258465124763087148566313528934
1661174906175266714926721761283308452739364692445828925713888
7783905630048248379983969202922221548614590237347822268252163
9957440801727144146179559226175083889020074169926238300282286
1721445314257494401506613946316919762918150657974552623619122
4848063890033669074365989226349564114665503062965960199720636
2026035219177767406687774635493753188995878662821254697971020
6574723272137291814466665942187200347450894283091153518927111
4287108376159222380276605327823351661555149369375778466670145
7179719012271178127804502400263847587883393968179629506907988

A big number

1712169068692953824852983002347606845411417813911064856023654
9754227497231007615131870024053910510913817843721791422528587
4320985249578780346837033378184214440171386881242499844186181
2927119853331538256732187042153063119774853521467095533462633
6610864667332292409879849256691109516143618601548909740241913
5096230436121961281659505186660220307156136847323646608689050
1426391390651506390819937885231836505989729912540447944342516
6774299659811849233151555272883274028352688442408752811283289
9806259126736995462473415433335001472314306127503903073971352
5206933817384332295070104906186753943313078479801565513038475
8155685236218010419650255596181934986315913233036096461905990
2361126811960234418433633345949276319461017166529138237171823
9429921627253846177606569454229787707138319881703696458868981

A big number

1863210976900355735884624464835706291453052757101278872027965
3688753719872913083173803391101612854741537737771595172808411
1627597186384924222802373441925469991983672192131287035585307
9669427134163910338827543186136434901009431974090473310144762
9986172542442335561223743571582593338280498624389249822278071
5951762757847109475119033482241412025182688713728193104253478
1961284401764795315050571107229743145699152234516431218486575
7578652819756484350895838472292353455946452121583165775147129
8708225909292655638836651120681943836904116252668710044560243
7042006637090019411855571604720446436969328500600469281405071
1906926139399390273553454556747031490388602202463994826050176
2431969305640666366626090207048887438898907498152865444381862
9173829010518208699363826618683039152732645812867828066013375

A big number

0009659336462514609172318031293034787742123467911845479131110
9897794648216922505629399956793483801699157439700537542134485
8745868560472867510654233418938390991105864655951136460610551
5683854121745980180713316361257307961116834386376766730735458
3494789788316330129240800836356825939157113130978030516441716
6825183465736759341980849589479409832925000863897785634946932
1247342610306271374507728615692259662857385790553324064184901
8451328284632709269753830867308409142247659474439973348130810
7012560584236558953969030647496558414798131099715754204325639
5776070485100881578291408250777738559790129129407309462785944
5058594122731948127532251523248015034665190482289614066468903
0510251091623777044848623022948896671138055560795662073244937
3374027836767300203011615227008921843515652121379215748206859

A big number

3569207902145022771330999877294595969528170445821819560809658
1170279806266989120506156074232568684227130629500986442185347
0810407128917646906550836129916694778023822502789667843489199
4096573617045867862425540069425166939792926247145249454088584
2272615375526007190433632919637577750217600519580069384763578
9586878489536872122898557806826518192703632099480155874455575
1753127364714212955364940843855866152080121150790750685533444
8925869328385965301327204697069457154695935365857178889486233
3292465202735853188533370948455403336565356988172582528918056
6354883637437933484118455801683318276768346462919956055134700
3914787680864032262961664156066750815371064672310846196424753
7490553744805318226002710216400980584497526023035640038083472
0531499411729657367850664214008426964971032419191821212132069

A big number

3976914392336837470922826773870813223668008692470349158684099
1153098315412063566123187504305467536983230827966457417620806
5835125982108076910196105222926387974504901925431190062056190
6577452416191913187533984049343976823310298465893318373015809
5925228292068208622303325852801192664963144413164427730032377
9227471233069641714994553226103547514563129066885434542686978
8447742981777493710117614651624183616680254815296335308490849
9430067636548061029400946937506098455885580439704859144495844
4507997849704558355068540874516331646411808312307970438984919
0506587586425810738422420591191941674182490452700288263983057
9500573417114870311871428341844991534567029152801044851451760
5530697144176136858238410278765932466268997841831962031226242
1177391477208004883578333569204533935953254564897028558589735

A big number

5057512351295365405028420810227852487766035742463666731486802
7948605244578267362623085297826505711462484659591421027812278
8941448163994973881884622768244851622051817076722169863265701
6543169197426512300417573299044735376725368457927543654128265
5358185804684006936771860502007054724754840080553042495185449
5267247261347318174742180078574693465447136036975884118029408
0396167469462885406791721386012254195038197045384172680063988
2065632879283958270851091995883944829777564715202613287108952
6163417707151642899487953564854553553148754978134009964854498
4620203201336835038542536031363676357521260470742531120923340
2837482949453104727418969287275572027615272268283376741393425
6526532830684699975970977500055608899326850250492128840682741
3988163154045649035077587168007405568572402175868543905322813

A big number

3770707415830756269628316955687424060527726485853050611356384
8519659189686495963355682169754376214307786659347304501648224
3296489127070989807667662567151726906205881554966638257382927
4182082278960684488222983394816670984039024283514306813767253
4601260072692629694686727507943461904399966189796119287505194
4235640264430327173734159128149605616835398818856948404534231
1424613559925272330064881627466723523751234311893442118885085
0793581638489944875447563316892138696755743027379537852625423
2902488104718193903722066689470220425883689584093999845356094
8869946833852579675161882159410981624918741813364726965123980
6775619479125579574464714278686240537505761042042671493660849
8023827468057598259133100691994190465190653117190892607794911
9217946407355129633864523035673345588033313197080365457184791

A big number

5504326548995597058628882868666066180218822486021449999731221
6413817065348017551043840662441282280361664890425737764095632
6482825258407669045608439490325290526337532316509087681336614
8008839555494223709673484007264270570116508907519615537018626
4797456381187856175457113400473810762763014953309735174180655
4791126609380343113785325328835333520249343659791293412848549
7094682632907583019307266533778255931433111096384805394085928
3988907796210479847919686876539987477095912788727475874439806
7798249682782722009264499445593804146087706419418104407582698
0568803894965461658798390466058764534181028990719429302177451
9976104495043196841503455514044820928933378657363052830619990
0777487269229986082790531716918765788609089418170579934048902
1844155979109267686279659758395248392673488363474565168701616

A big number

6240642424241228961118010615682342539392180052483454723779219
9112285959141918774917938233400100781283265067102817813960291
2091472010094787875255126337288422235386949006792766451163475
8101193875319657242121476038284774774571704578610417385747911
3019085838778901523343430130052827970385803598151829296003056
8261209195094373732545417105638388704752895056396102984364136
0935641632589408137981511693338619797339821670761004607980096
0160248230969430438069566201232136501405495862506152825880330
2290838581247846931572032323360189946943764772672187937682643
1828382603564520699468630216048874528424363593558622333506235
7749020024955273873458595640516083058305377073253397155262044
4705429573538361113677523169972740292941674204423248113875075
6313190782721888640533746942138421699288629404796353051505607

A big number

8812636620649723125757901959887304119562622734372890051656111
1094111745277965482790471250581999077498063821559376885546498
8229389854082913251290764783863224947810167534916934892881042
0301561028338614382737816094634133538357834076531432141715065
5877547820252454780657301342277470616744241968952613164274104
6954746214837562882997718041867850845469656191509086958742511
8443583730659095146098045124740941137389992782249298336779601
1015387096129749705566301637307202750734759922943792393824427
4211861582361613178863925530951171884212985083072382597291441
4225157940388301135908333165185823496722125962181250705811375
9495525022747274674369887131926670769299199084467161228738858
4575846227265733307537355728239516169641751986750126817454293
2373829414382481437713986190671665757294580780482055951188168

A big number

71880752129718326364421553367877512747669407901170575098195755
0845635652173895441798750745238544552001335720333323798950743
9390531291821225525983379090946363020218535384885482506289771
5616963860712382771725621313460549401770413581731931763370136
9150304916533946476371776643912079834749462739782217150209067
0190302469762151278521956142070806461631373236517853976292092
0255002889620129701413796400380557349492690735351459612086747
9654773369295877362863566014376796403843079686413856344780132
8261284589184898528048048844180821639423974014362903481665458
1144543664600324906187630395023564020445307482102413668951966
4422133920075747912868380517515063466256939193774028351207566
6260829890491877287833852178522792045771846965855278790447562
1926639920084093020756739253637356283908298175779021532021064

A big number

0961737328359849406665214119818381088451545977289516457213189
7797907491941013148368544639616904607030107596818933741217575
9881651270007612627891695104063158576375347874200702220510708
9125761236165802680681585849985263146587808661680073326467683
0206391697203064894405628195406190685242003053463156621891327
3090696873531816410945142880366059952202482488867115544291047
2192913424834643870536850864874909917881267056566538719104972
1820042371492740164460943459845392536706132210616533085662021
1889682340057526754861014769936887382095845522115719234796868
8816085363161586288015039594941852948922707441082820716930338
7818084936204018255222271010985653444817207470756019245915599
5356302418122547326609330271039796809106493927272268303541046
7632591355279683837705019855234621222858410557119921731717969

A big number

8043393177077507556270560478317798444476375602546370333692471
1422081551997369137197516324130274871219986340454824852457011
8553342675264715978310731245663429805221455494156252724028915
3333543493412178620370072603152798707718724912344944771479095
2073476138542548531155277330103034247683586549609372232400715
4518129732692081058424090557725645803681462234493189708138897
1432998313476177996797124537823107037391514738786921191875667
0031932128189680332269659445928621060743882741691946516226763
2540665070881071030394178860564893769816734159025925194611823
6429456526693722031555047002135988462927580125277154220166299
5486313032491231102962792372389976641680349714122652793190763
6326136814145516376656559839788489381733082668779901962886932
2965973799519316211872154552873941702436698855938887933167445

A big number

3336311954151840408828381519342123412282003095031334105070476
0159987985472529190665222479319715440331794836837373220821885
7733416238564413807005419135302459439135025545318864547962522
6025176292837433046510236105758351455073944333961021622967546
1415781127197001738611494279501411253280621254775810512972088
1296773075919738297344144525668877085532457088895832099382343
2102718224114763732791357568615421252849657903335093152776925
5058456440105521926445053120737562877449981636463328358161403
3017581396735942732769044892036188038675495575180689005853292
7201493923500525845146706982628548257883267398735220457228239
2902071448222198855871028969919358730742778151597576207640239
5124386020203259659625021257834995771008562638611823381331850
9014686577064010676278617583772772895892746039403930337271873

A big number

85053691295712671506689668849388088514294360996201296675907922
25082275313812849851526902931700263136328942095797577959327632
55311620667534886513173238724387480635133145126448899675898281
81292548007642518658649024111112730135719718138160258317850692
32244007998656635371544088454866393181708395735780799059730833
90948818040609359591909074739609044101505163217496814121007654
71917748376735575100073361692238653742907945780320004233745280
07566153042929014495780629634138383551783599764708851349004851
69736979652386958459945955920907090589568914511414126845054621
11794502661175016692826025095077077821195043261738322356243760
01776799362796099368975191394965033358507155418436456852616671
42436889203710374953284259271316105378349807407391586338179670
10116137124237614267225417320559592027821293257259471464172245

A big number

9773213163818453265552796042705418714962365852524586489332541
4506264233788565146467060429856478196846159366328895429978072
2542264790400616019751975007460545150060291806638271497016110
9879513366337713784344161940531214452918551801365755586676150
1937302969193207612000925506508158327550849934076879725236998
7023567931026804136745718956641431852679054717169962990363015
5456450900448027890557019683283136307189976991531666792089587
6857229060091547291963638167359667395997571032601557192023734
8580521128117458610065152598883843114511894880552129145775699
1465775300413847171245779650481758563950728953375397558220877
7750607233944558789590571915673 6

A big number

If we count, $2 \uparrow\uparrow 5$ has **19729 digits**!

A big number

If we count, $2 \uparrow\uparrow 5$ has **19729 digits**!

And yet, $\log^*(2 \uparrow\uparrow 5)$ equals just 5!

A big number

If we count, $2 \uparrow\uparrow 5$ has **19729 digits**!

And yet, $\log^*(2 \uparrow\uparrow 5)$ equals just 5!

Punchline? $\log^*(n) \leq 5$, for basically any reasonable value of n.

A big number

If we count, $2 \uparrow\uparrow 5$ has **19729 digits**!

And yet, $\log^*(2 \uparrow\uparrow 5)$ equals just 5!

Punchline? $\log^*(n) \leq 5$, for basically any reasonable value of n.

Runtime of Kruskal?

$$\mathcal{O}\left((|E| + |V|) \log^*(|V|)\right) \leq \mathcal{O}\left((|E| + |V|)5\right) \approx \mathcal{O}\left(|E| + |V|\right)$$

Inverse of the Ackermann function

But wait!

Somebody then came along and proved an even tighter bound. It turns out `findSet(...)` and `union(...)` are amortized $\mathcal{O}\left(\alpha(n)\right)$ – the inverse of the Ackermann function.

The Ackermann function

The Ackermann function is a recursive function designed to grow extremely quickly:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

The Ackermann function

The Ackermann function is a recursive function designed to grow extremely quickly:

$$A(m, n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

This function grows even more quickly then $m \uparrow\uparrow n$ – this means the inverse Ackermann function $\alpha(...)$ grows even more slowly then $\log^*(...)$!

The Ackermann function

The Ackermann function is a recursive function designed to grow extremely quickly:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

This function grows even more quickly then $m \uparrow\uparrow n$ – this means the inverse Ackermann function $\alpha(...)$ grows even more slowly then $\log^*(...)$!

So, the runtime of Kruskal's is even better! It's

$$\mathcal{O}\left((|E| + |V|)\alpha(|V|)\right) \le \mathcal{O}\left((|E| + |V|)4\right)$$

...for any practical size of $|V|$.

Are we done yet?

But wait, there's more!

Recap

To recap, we found that the runtimes of `findSet(...)` and `union(...)` were...

- Originally $\mathcal{O}(n)$

Recap

To recap, we found that the runtimes of `findSet(...)` and `union(...)` were...

- Originally $\mathcal{O}(n)$
- After applying **union-by-rank**, $\mathcal{O}(\log(n))$

Recap

To recap, we found that the runtimes of `findSet(...)` and `union(...)` were...

- Originally $\mathcal{O}(n)$
- After applying **union-by-rank**, $\mathcal{O}(\log(n))$
- After applying **path compression**, $\mathcal{O}(\alpha(n)) \approx \mathcal{O}(1)$

Recap

To recap, we found that the runtimes of `findSet(...)` and `union(...)` were...

- Originally $\mathcal{O}(n)$
- After applying **union-by-rank**, $\mathcal{O}(\log(n))$
- After applying **path compression**, $\mathcal{O}(\alpha(n)) \approx \mathcal{O}(1)$
- One final optimization: **array representation**.

Recap

To recap, we found that the runtimes of `findSet(...)` and `union(...)` were...

- Originally $\mathcal{O}(n)$
- After applying **union-by-rank**, $\mathcal{O}(\log(n))$
- After applying **path compression**, $\mathcal{O}(\alpha(n)) \approx \mathcal{O}(1)$
- One final optimization: **array representation**.
  It doesn't lead to an asymptotic improvement, but it does lead to a constant factor speedup (and simplifies implementation).

Array representation

So far, we've been thinking about disjoint sets in terms of nodes and pointers.

Array representation

So far, we've been thinking about disjoint sets in terms of nodes and pointers.

For example:

```
private static class Node {
    private int vertexNumber;
    private Node parent;
}
```

Array representation

So far, we've been thinking about disjoint sets in terms of nodes and pointers.

For example:

```
private static class Node {
    private int vertexNumber;
    private Node parent;
}
```

Observation: It seems wasteful to have allocate an entire object just to store two fields

Array representation

Java is technically allowed to store and represent its objects however it wants, but in a modern 64-bit JDK, this object will probably be 32 bytes:

Array representation

Java is technically allowed to store and represent its objects however it wants, but in a modern 64-bit JDK, this object will probably be 32 bytes:

- The int field takes up 4 bytes

Array representation

Java is technically allowed to store and represent its objects however it wants, but in a modern 64-bit JDK, this object will probably be 32 bytes:

- The int field takes up 4 bytes
- The pointer to the parent takes up 8 bytes (assuming 64-bit)

Array representation

Java is technically allowed to store and represent its objects however it wants, but in a modern 64-bit JDK, this object will probably be 32 bytes:

- The int field takes up 4 bytes
- The pointer to the parent takes up 8 bytes (assuming 64-bit)
- The object itself also uses up an additional 16 bytes

Array representation

Java is technically allowed to store and represent its objects however it wants, but in a modern 64-bit JDK, this object will probably be 32 bytes:

- The int field takes up 4 bytes
- The pointer to the parent takes up 8 bytes (assuming 64-bit)
- The object itself also uses up an additional 16 bytes
- This adds up to 28, but in a 64 bit computer, we always "pad" or round up to the nearest multiple of 8. So, this object will use up 32 bytes of memory.

Array representation

Idea: Just use an array of ints instead!

Array representation

Idea: Just use an array of ints instead!

Core idea:

- Make the index of the array be the vertex number

Array representation

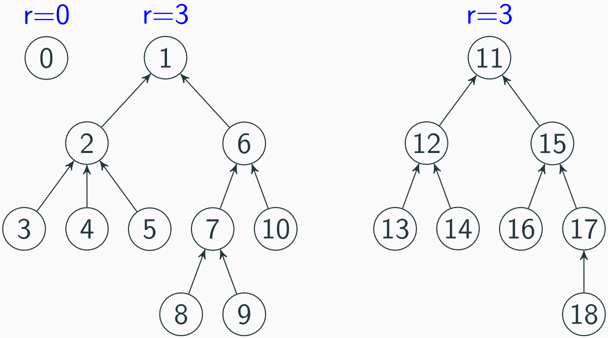
Idea: Just use an array of ints instead!

Core idea:

- Make the index of the array be the vertex number
- Make the element in the array be the index of the parent

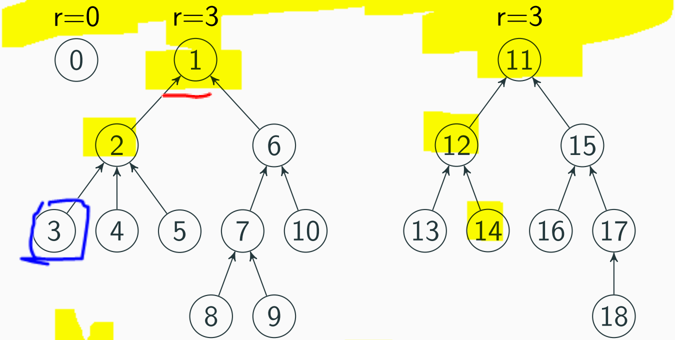
Array representation

Example:



Array representation

Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-	-	1	2	2	2	1	6	7	7	7	-	11	12	12	11	15	15	17

Array representation

So, rather then using 32 bytes per element, we use just 4!

Array representation

So, rather then using 32 bytes per element, we use just 4!

Question: Where do we store the ranks?

Array representation

So, rather then using 32 bytes per element, we use just 4!

Question: Where do we store the ranks?

Observation: Hey, each root has some unused space...

Array representation

So, rather then using 32 bytes per element, we use just 4!

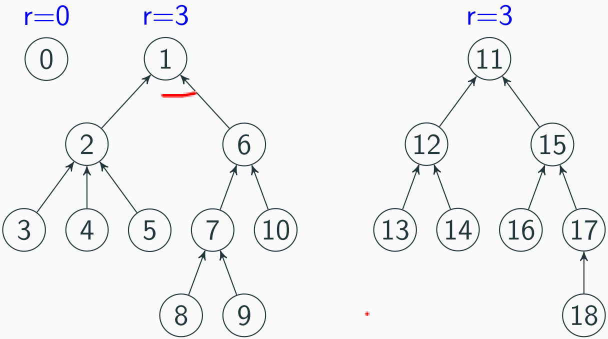
Question: Where do we store the ranks?

Observation: Hey, each root has some unused space...

Idea 1: Rather then leaving the root cells empty, just stick the ranks there.

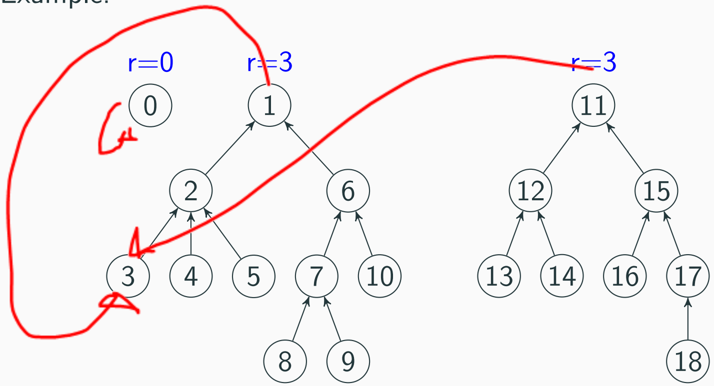
Array representation

Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	3	1	2	2	2	1	6	7	7	7	3	11	12	12	11	15	15	17

Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	3	1	2	2	2	1	6	7	7	7	3	11	12	12	11	15	15	17

What's wrong with this idea?

Array representation

Problem: How do we tell whether a number is supposed to be a rank or an index to the parent?

Array representation

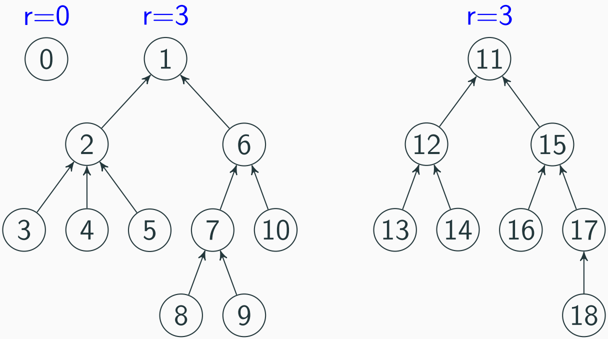
Problem: How do we tell whether a number is supposed to be a rank or an index to the parent?

A trick: Rather then storing just the rank, let's store the negative of the rank!

So, if a number is positive, it's an index. If the number is negative, it's a rank (and that node is a root).

Array representation

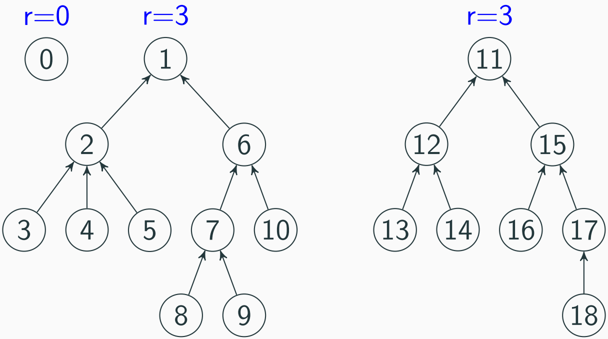
Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-0	-3	1	2	2	2	1	6	7	7	7	-3	11	12	12	11	15	15	17

Array representation

Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-0	-3	1	2	2	2	1	6	7	7	7	-3	11	12	12	11	15	15	17

What's wrong with this idea?

Problem: What's the difference between 0 and -0?

Array representation

Problem: What's the difference between 0 and -0?

Solution: Instead of just storing $-\text{rank}$, store $-\text{rank} - 1$.

Array representation

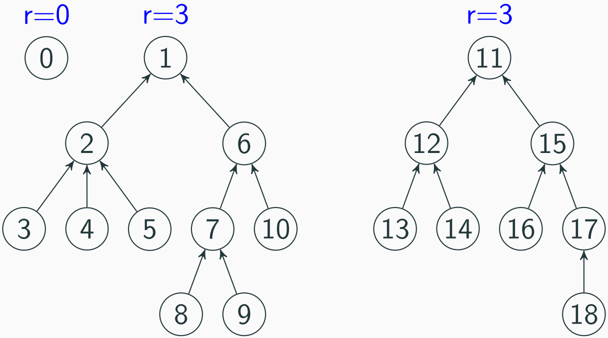
Problem: What's the difference between 0 and -0?

Solution: Instead of just storing $-\text{rank}$, store $-\text{rank} - 1$.

(Alternatively, redefine the rank to be the upper bound of the number of *levels* in the tree, rather then the *height*.)

Array representation

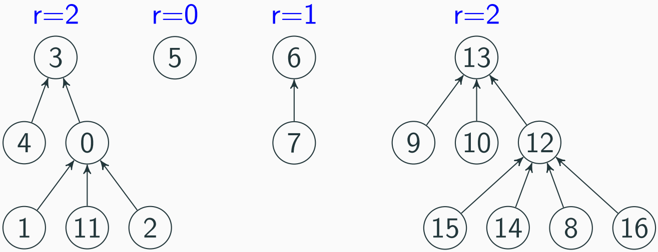
Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-1	-4	1	2	2	2	1	6	7	7	7	-4	11	12	12	11	15	15	17

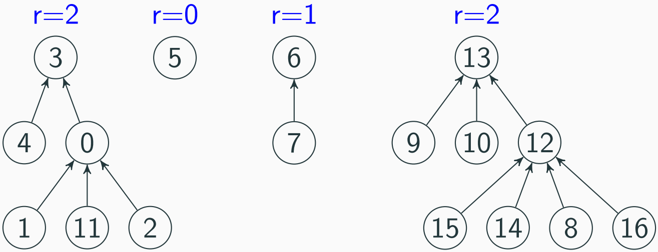
Array representation

Now you try – what is the array representation of this disjoint set?



Array representation

Now you try – what is the array representation of this disjoint set?



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	0	0	-3	3	-1	-2	6	12	13	13	0	13	-3	12	12	12

Recap

And that's it for graphs. Topics covered:

- Graph definitions, graph representations

Recap

And that's it for graphs. Topics covered:

- Graph definitions, graph representations
- Graph traversal: BFS and DFS

Recap

And that's it for graphs. Topics covered:

- Graph definitions, graph representations
- Graph traversal: BFS and DFS
- Finding the shortest path: Dijkstra's algorithm

Recap

And that's it for graphs. Topics covered:

- Graph definitions, graph representations
- Graph traversal: BFS and DFS
- Finding the shortest path: Dijkstra's algorithm
- Topological sort

Recap

And that's it for graphs. Topics covered:

- Graph definitions, graph representations
- Graph traversal: BFS and DFS
- Finding the shortest path: Dijkstra's algorithm
- Topological sort
- Minimum spanning trees: Prim's and Kruskal's

Recap

And that's it for graphs. Topics covered:

- Graph definitions, graph representations
- Graph traversal: BFS and DFS
- Finding the shortest path: Dijkstra's algorithm
- Topological sort
- Minimum spanning trees: Prim's and Kruskal's
- Disjoint sets

Recap

Next time: What does it mean for a problem to be "hard"?