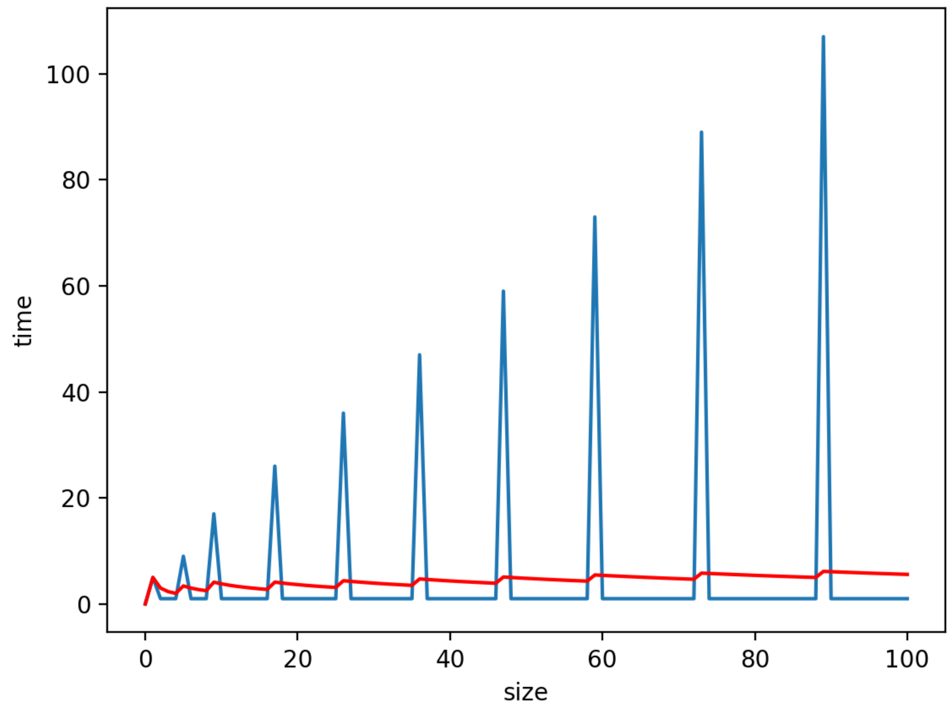
CSE202
Design and Analysis of Algorithms

Week 8 — Amortization

Various Kinds of Complexity Analysis

Worst-case: bound the worst-case scenario.

Amortized: average the worst-case over a sequence of operations.



Average-case: average complexity over random inputs or random executions.

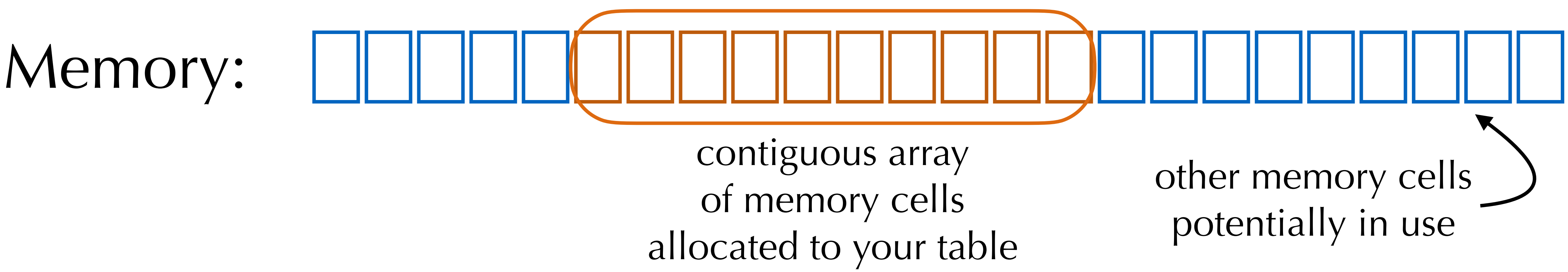
I. Dynamic Tables

Tables in Low-Level Languages

```
A=[]
for i in range(N):
     A.append(1)
```

Tables in Low-Level Languages

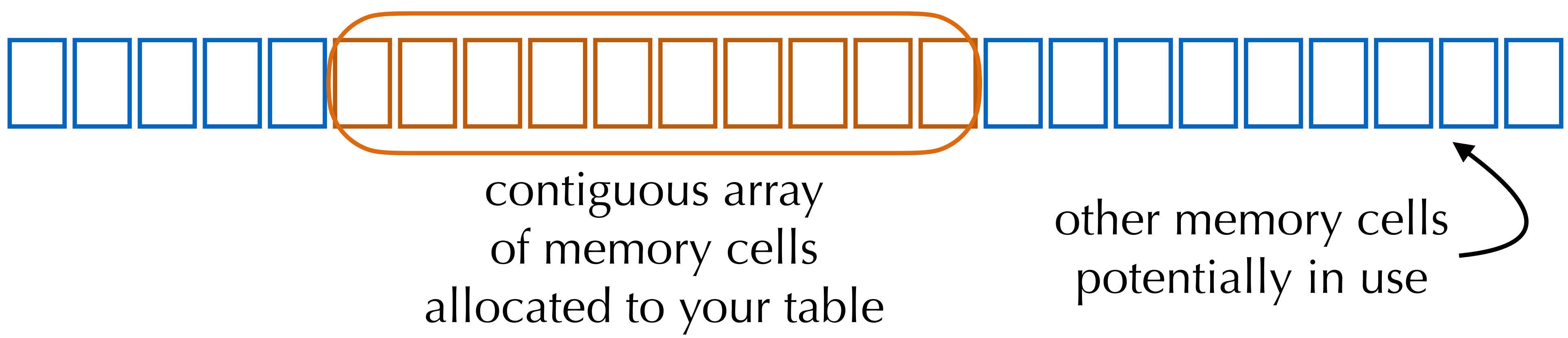
```
A=[]
for i in range(N):
     A.append(1)
```



Tables in Low-Level Languages

```
A=[]
for i in range(N):
     A.append(1)
```

Memory:



Increasing the size of the table requires:

allocating a new array of memory;
copying the old array to the new one.

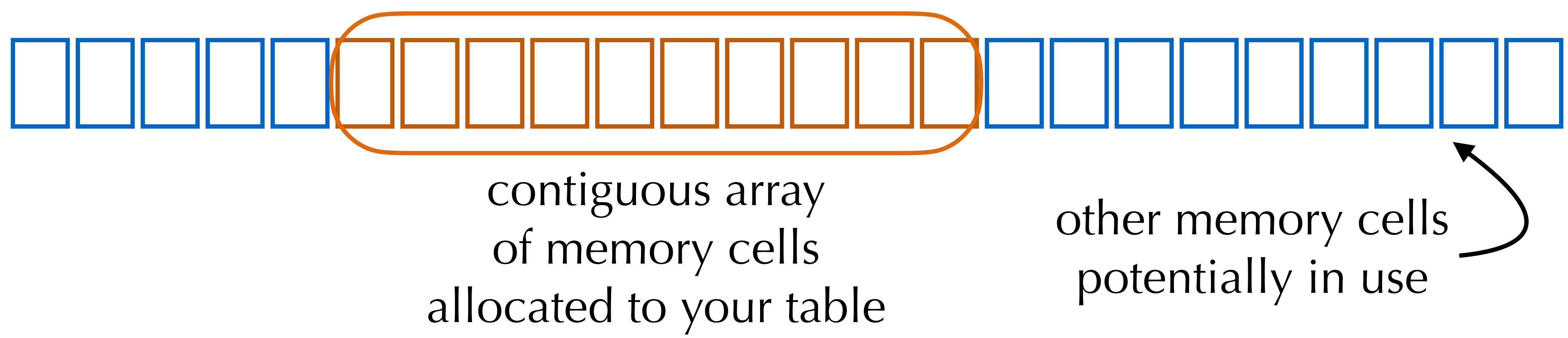
Complexity linear in the size of the array

Tables in Low-Level Languages

```
A=[]
for i in range(N):
    A.append(1)
```

would have quadratic complexity with a naive implementation.

Memory:



Increasing the size of the table requires:

allocating a new array of memory;
copying the old array to the new one.

Complexity linear in the size of the array

Dynamic Tables

Use three fields:
size, capacity, pointer to the array.

This is how lists are implemented in Python

```
def __init__(self):
    self.size = 0
    self.capacity = 0
    self.table = []

def __getitem__(self,i):
    if i>=self.size: raise IndexError
    return self.table[i]

def __setitem__(self,i,v):
    if i>=self.size: raise IndexError
    self.table[i] = v

def append(self,v):
    n = self.size
    self.resize(n+1)
    self.table[n] = v

def resize(self,newsize):
    if newsize>self.capacity:
        self.realloc((int)(α*newsize))
    self.size=newsize
```

Simplified & Pythonized C-code

Dynamic Tables

Use three fields:
size, capacity, pointer to the array.

This is how lists are implemented in Python

```
def __init__(self):
    self.size = 0
    self.capacity = 0
    self.table = []

def __getitem__(self,i):
    if i>=self.size: raise IndexError
    return self.table[i]

def __setitem__(self,i,v):
    if i>=self.size: raise IndexError
    self.table[i] = v

def append(self,v):
    n = self.size
    self.resize(n+1)
    self.table[n] = v

def resize(self,newsize):
    if newsize>self.capacity:
        self.realloc((int)(α*newsize))
    self.size=newsize
```

Simplified & Pythonized C-code

Capacity is increased faster than size

Choice of $\alpha > 1$:
after the analysis

In Python
$\alpha \approx 9/8$

Dynamic Tables

Use three fields:
size, capacity, pointer to the array.

This is how lists are implemented in Python

```
def __init__(self):
    self.size = 0
    self.capacity = 0
    self.table = []

def __getitem__(self,i):
    if i>=self.size: raise IndexError
    return self.table[i]

def __setitem__(self,i,v):
    if i>=self.size: raise IndexError
    self.table[i] = v

def append(self,v):
    n = self.size
    self.resize(n+1)
    self.table[n] = v

def resize(self,newsize):
    if newsize>self.capacity:
        self.realloc((int)(α∗newsize))
    self.size=newsize
```

Simplified & Pythonized C-code

Capacity is increased faster than size

Choice of $\alpha > 1$:
after the analysis

In Python
$\alpha \approx 9/8$

Worst-Case cost of append:
$O(\text{size})$

Amortized Cost of a Sequence of Append

```
A=[]
for i in range(N):
    A.append(1)
```

Sequence of capacities:

$$t_{k+1} = \lfloor \alpha(t_k + 1) \rfloor, \quad t_0 = 0.$$

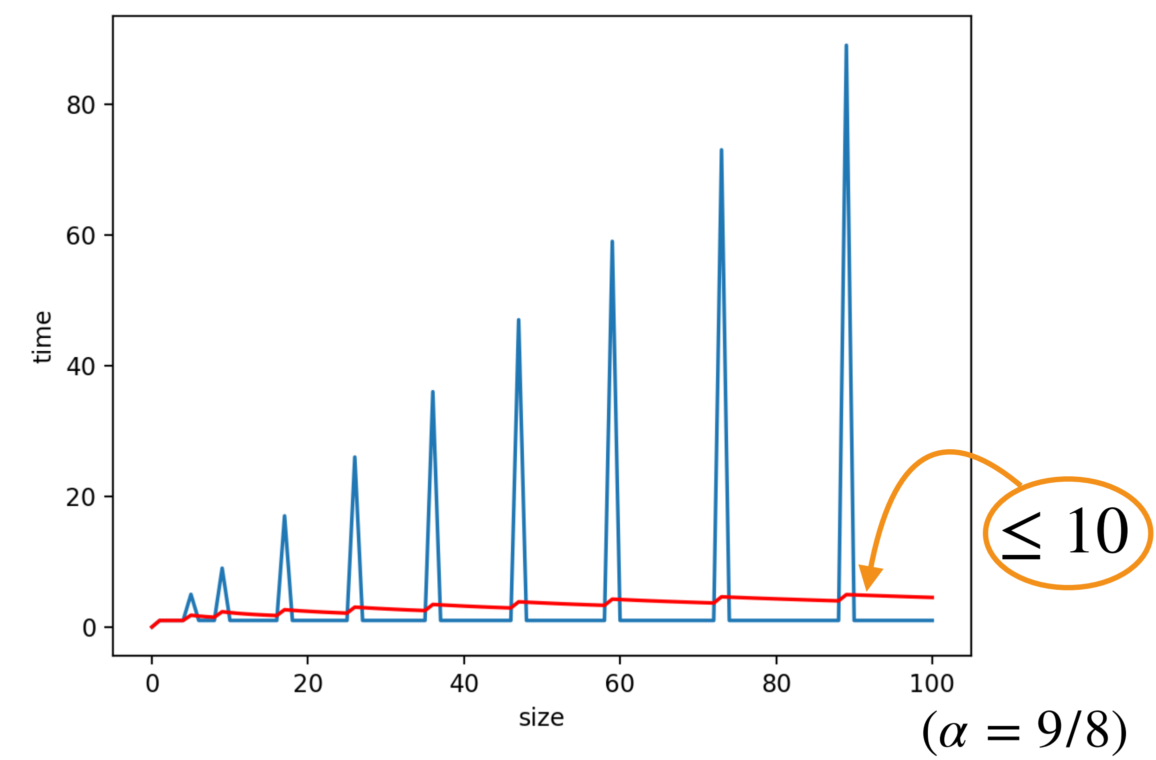
Total cost for N append: $C_N \leq N + \sum_{t_k \leq N} t_k$.

Thm. Amortized cost bounded by

$$C_N/N \leq 1 + \frac{\alpha}{\alpha - 1}.$$

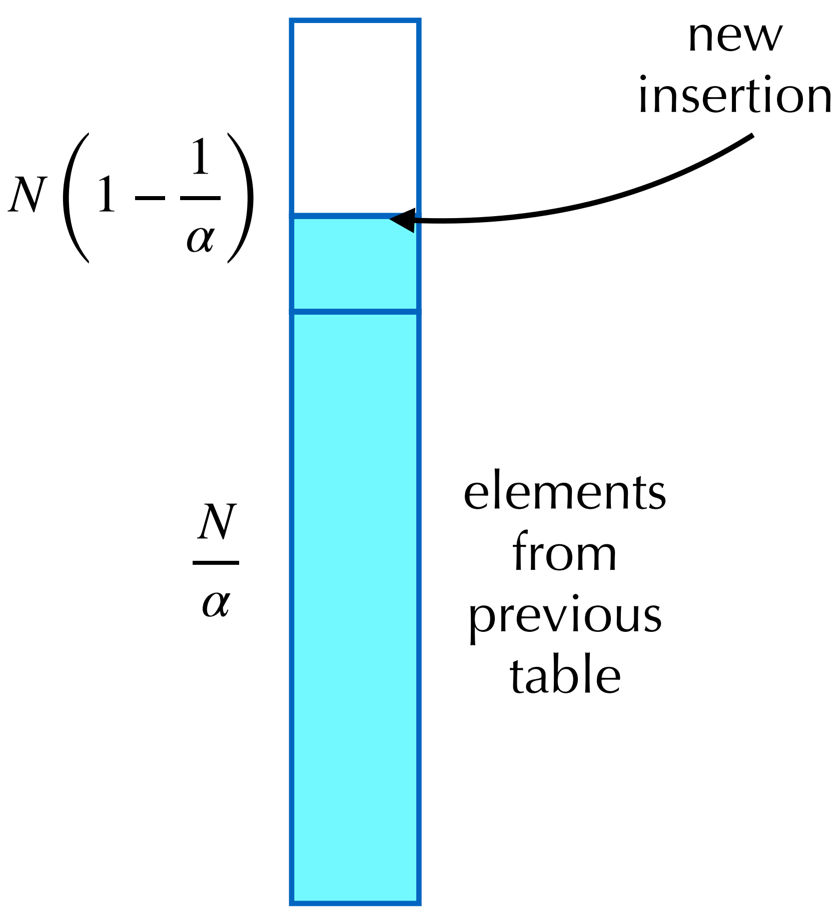
A larger α lowers the constant, but penalizes small tables.

Proof on the blackboard.



$(\alpha = 9/8)$

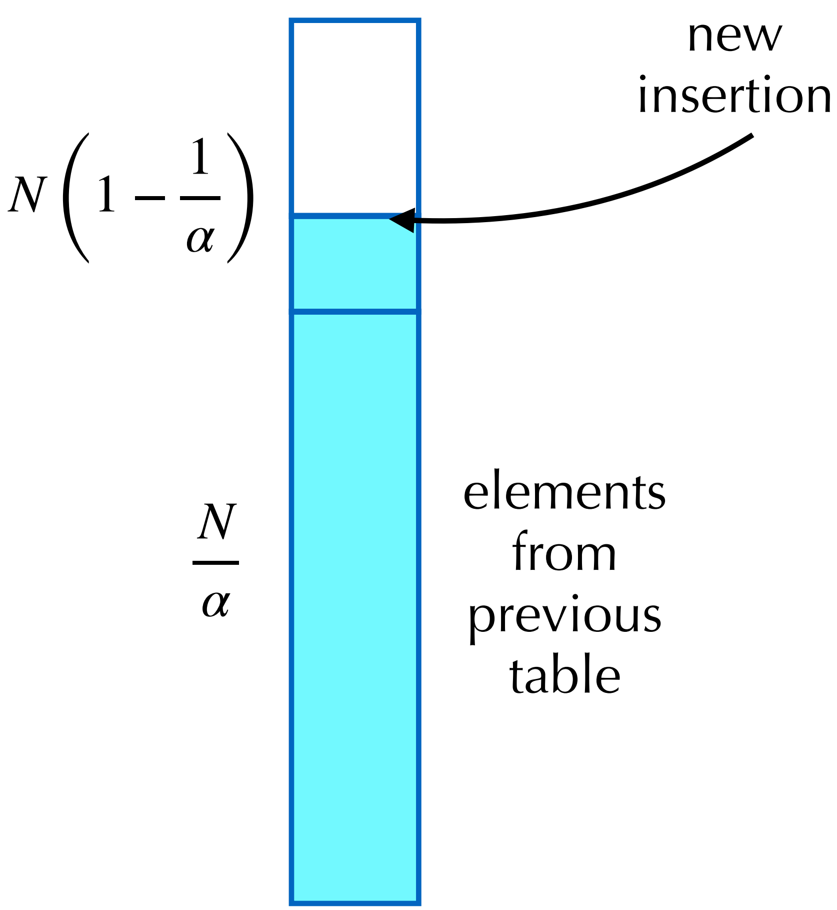
Interpretation by Accounting Method



Array of capacity N

When a new element is inserted, it is charged:

Interpretation by Accounting Method

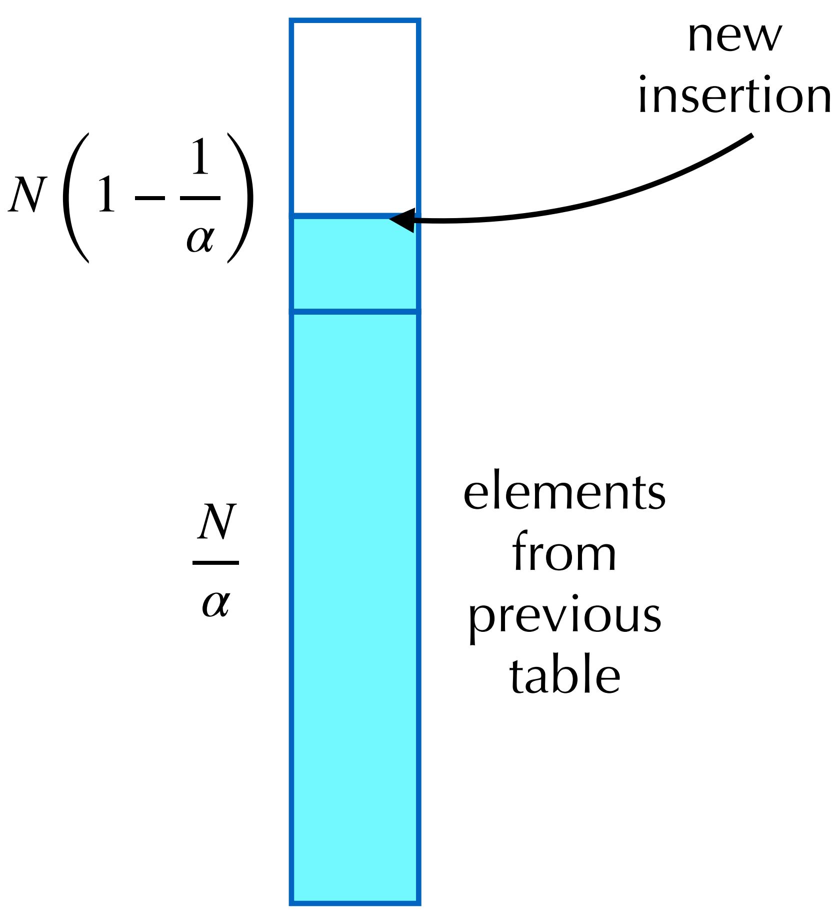


Array of capacity N

When a new element is inserted,
it is charged:

1 for its own insert

Interpretation by Accounting Method



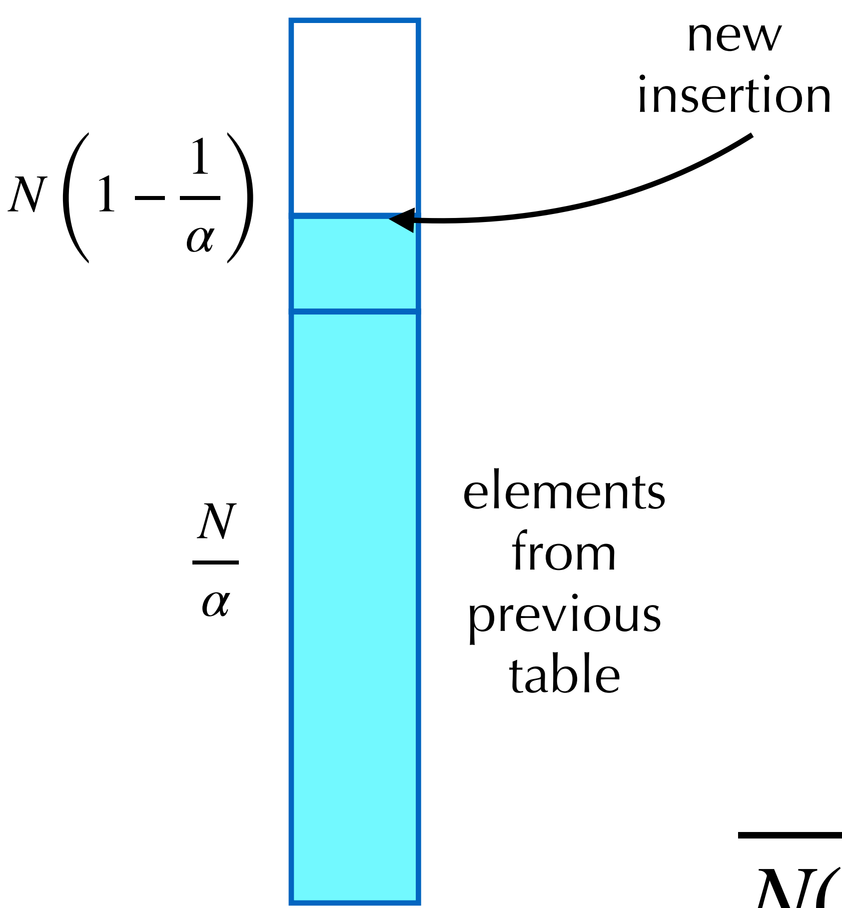
Array of capacity N

When a new element is inserted,
it is charged:

1 for its own insert

1 for its future copy
when the table next grows

Interpretation by Accounting Method



Array of capacity N

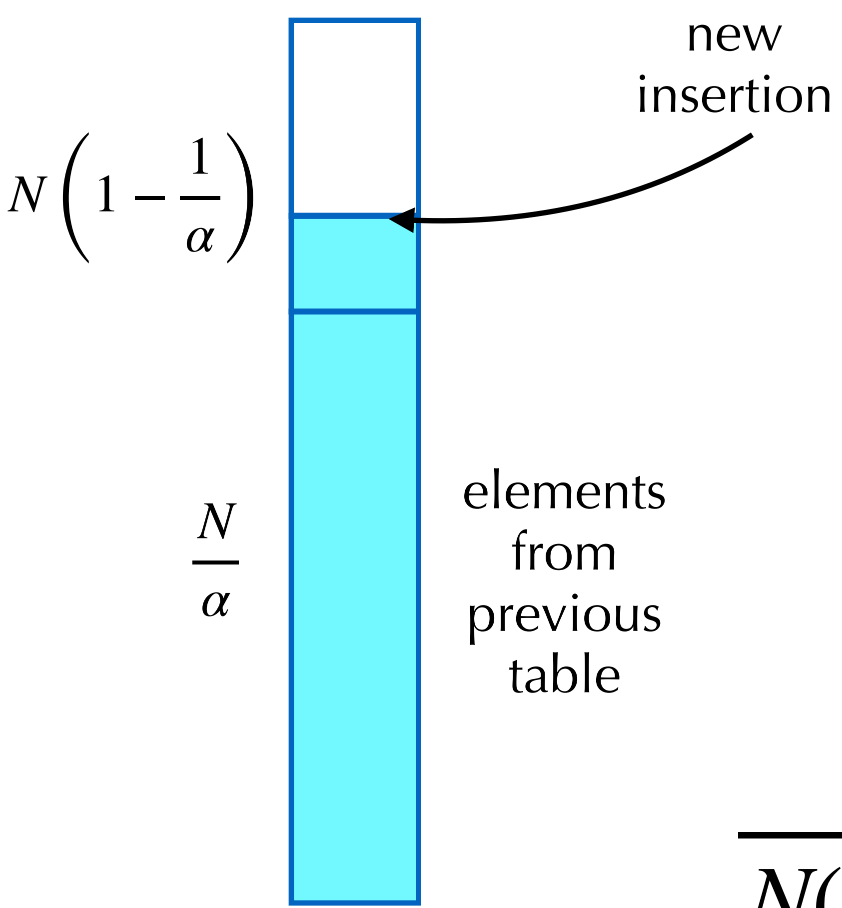
When a new element is inserted, it is charged:

1 for its own insert

1 for its future copy when the table next grows

$$\frac{N/\alpha}{N(1-1/\alpha)} = 1/(\alpha - 1)$$ for its share of the future copy of the previous table

Interpretation by Accounting Method



Array of capacity N

When a new element is inserted, it is charged:

1 for its own insert

1 for its future copy when the table next grows

$\frac{N/\alpha}{N(1-1/\alpha)} = 1/(\alpha - 1)$ for its share of the future copy of the previous table

$$\text{Total: } 1 + 1 + \frac{1}{\alpha - 1} = 1 + \frac{\alpha}{\alpha - 1}.$$

The cost of future copies is prepaid.

Deletion

Retrieve memory when the `size` of the table decreases

Dangerous scenario:

increase by a factor α when full;
decrease by a factor $1/\alpha$ when possible.

Append t_m times, then ADDAADD… copies too often.

Deletion

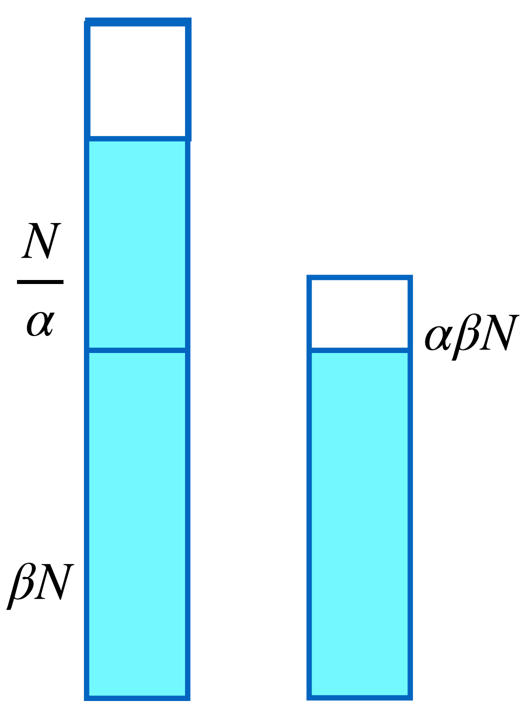
Retrieve memory when the `size` of the table decreases

Dangerous scenario:

increase by a factor α when full;
decrease by a factor $1/\alpha$ when possible.

Append t_m times, then ADDAADD… copies too often.

Solution: leave space to prepay for the next growth.




```
def pop(self):
    if self.size==0: raise IndexError
    res = self.table[self.size]
    self.resize(self.size-1)
    return res

def resize(self,newsize):
    if newsize > self.capacity or \
      newsize < self.capacity/2:
       self.realloc((int)(α*newsize))
    self.size=newsize
```

Python's choice for β.

Deletion

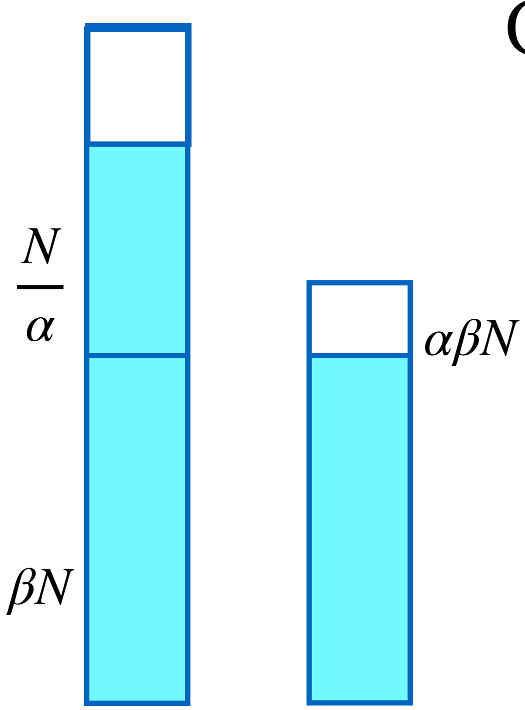
Retrieve memory when the `size` of the table decreases

Dangerous scenario:

increase by a factor α when full;
decrease by a factor $1/\alpha$ when possible.

Append t_m times, then ADDAADD… copies too often.

Solution: leave space to prepay for the next growth.



Charge for Insert unchanged:

$$1 + \frac{\alpha}{\alpha - 1}$$

Charge for Delete:

$$1 + \frac{\alpha\beta}{1 - \alpha\beta} \rightarrow \frac{\beta \cdot N}{(1/\alpha - \beta) \cdot N}$$

```
def pop(self):
    if self.size==0: raise IndexError
    res = self.table[self.size]
    self.resize(self.size-1)
    return res

def resize(self,newsize):
    if newsize > self.capacity or \
       newsize < self.capacity/2:
        self.realloc((int)(α*newsize))
    self.size=newsize
```

Python's choice for β.

Deletion

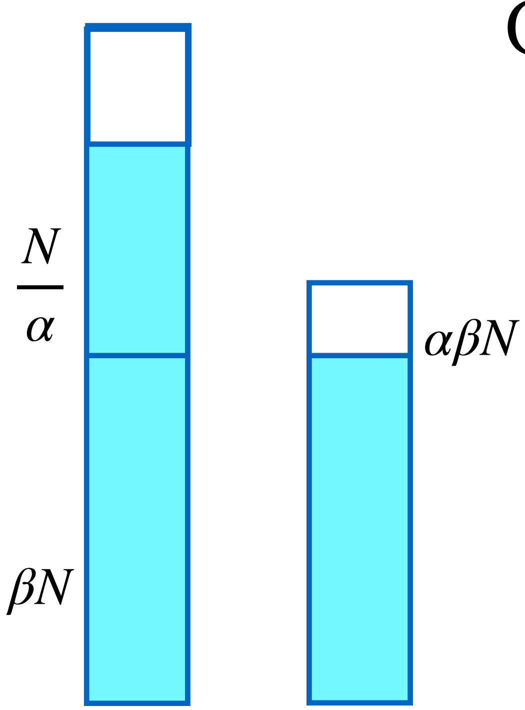
Retrieve memory when the `size` of the table decreases

Dangerous scenario:

increase by a factor α when full;
decrease by a factor $1/\alpha$ when possible.

Append t_m times, then ADDAADD… copies too often.

Solution: leave space to prepay for the next growth.



Charge for Insert unchanged:

$$1 + \frac{\alpha}{\alpha - 1}$$

Charge for Delete:

$$1 + \frac{\alpha\beta}{1 - \alpha\beta} \rightarrow \frac{\beta \cdot N}{(1/\alpha - \beta) \cdot N}$$

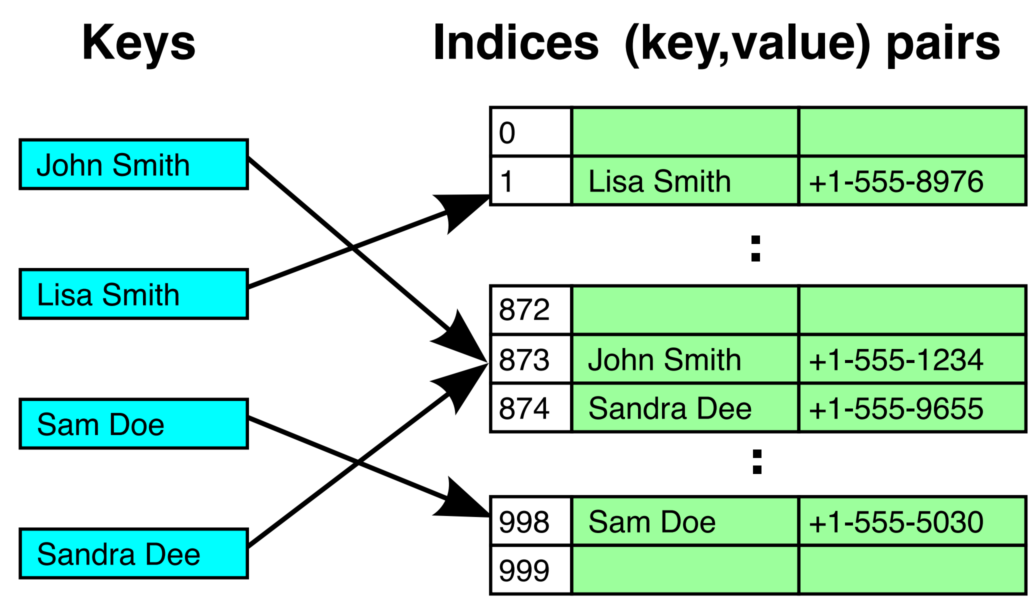
```
def pop(self):
    if self.size==0: raise IndexError
    res = self.table[self.size]
    self.resize(self.size-1)
    return res

def resize(self,newsize):
    if newsize > self.capacity or \
       newsize < self.capacity/2:
        self.realloc((int)(α*newsize))
    self.size=newsize
```

Python's choice for β.

Amortized cost O(1) per operation.

Application to Hash Tables

Hash tables with linear probing require a filling ratio bounded away from 1.
Implemented with dynamic tables.



Resizing the table requires to rehash all the entries.

In Python, the hash function is computed once as a 64-bit integer, and stored with the object. Only its value mod the new size is recomputed.

II. Union-Find

Recall Union-Find (CSE103)

Abstract Data Type for Equivalence Classes

Main operations:

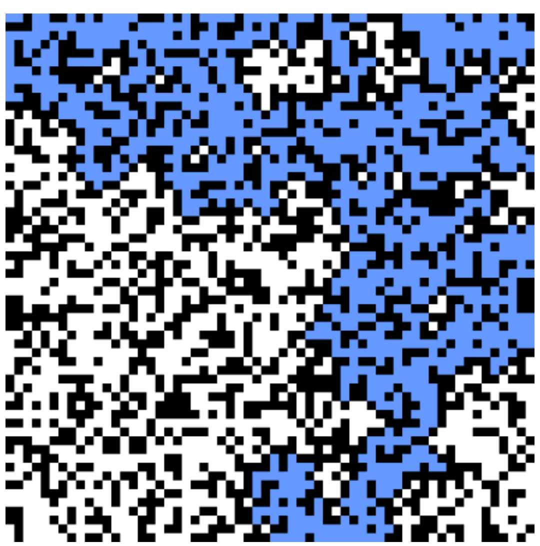
Find(p):	identifier for the equivalence class of p
Union(p, q):	add the relation $p \sim q$

Recall Union-Find (CSE103)

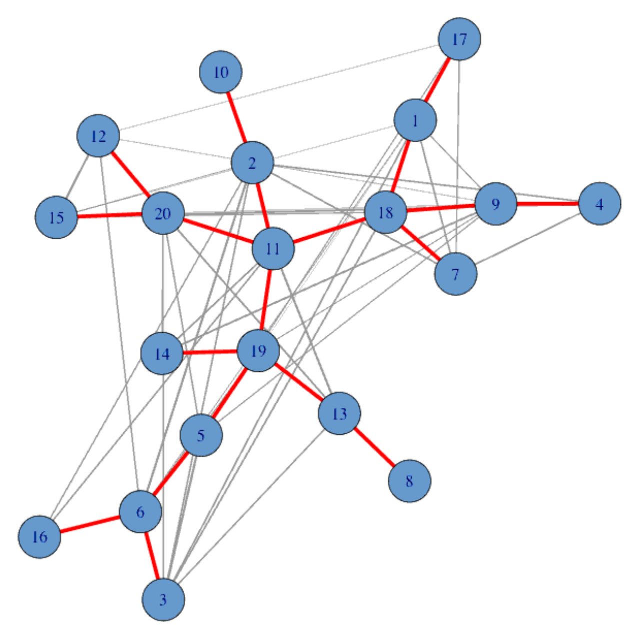
Abstract Data Type for Equivalence Classes

Main operations:

Find(p):	identifier for the equivalence class of p
Union(p, q):	add the relation $p \sim q$



Connected components in a graph as equivalence classes



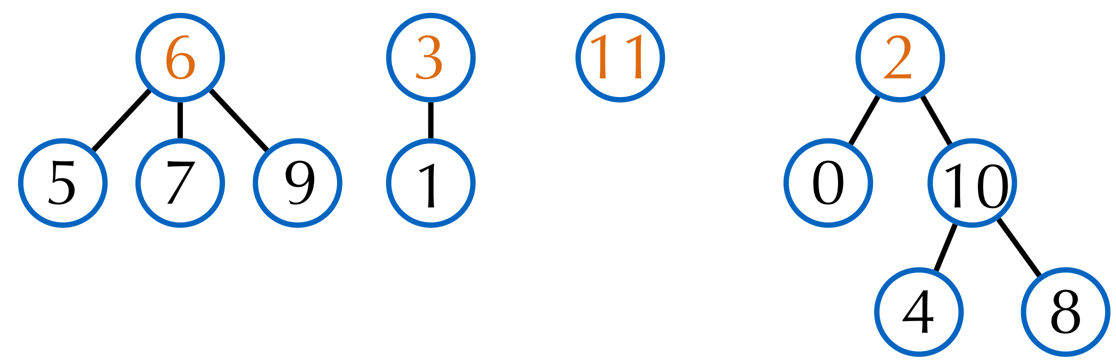
Kruskal's algorithm for the minimum spanning tree joins trees in a forest

Forests in Arrays

2	3	2	3	10	6	6	6	10	6	2	11

$p[i] := \text{parent}(i)$

(init with $p[i] := i$)



current equivalence classes

First version

```
def find(p,a):
    while p[a]!=a: a=p[a]
    return a

def union(p,a,b):
    link(p,find(p,a),find(p,b))

def link(p,a,b):
    p[a]=b
```

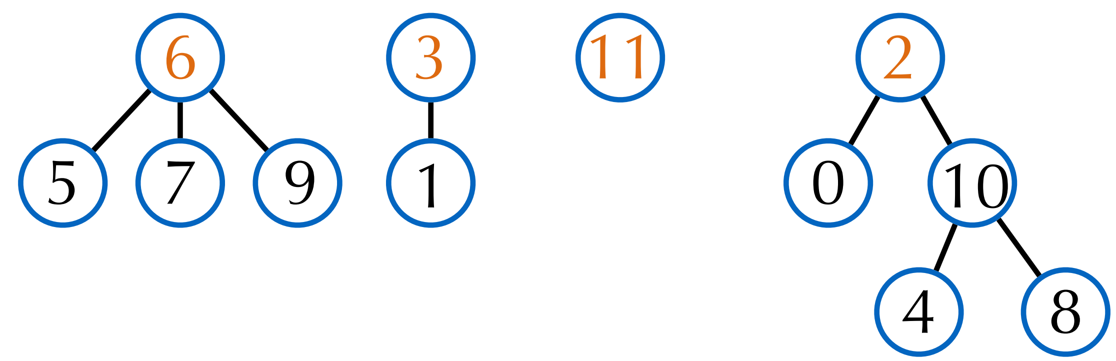
Only `find` uses more than $O(1)$ array accesses

Worst-case:

Forests in Arrays

2	3	2	3	10	6	6	6	10	6	2	11

$p[i] := \text{parent}(i)$
(init with $p[i] := i$)



current equivalence classes

First version

```
def find(p,a):
    while p[a]!=a: a=p[a]
    return a

def union(p,a,b):
    link(p,find(p,a),find(p,b))

def link(p,a,b):
    p[a]=b
```

Only `find` uses more than $O(1)$ array accesses

Worst-case:

```
for i in range(N):
    union(p,0,i)
```

uses $O(N^2)$ array accesses

Union by Rank

Maintain rank (=height).
Link short trees to higher ones.

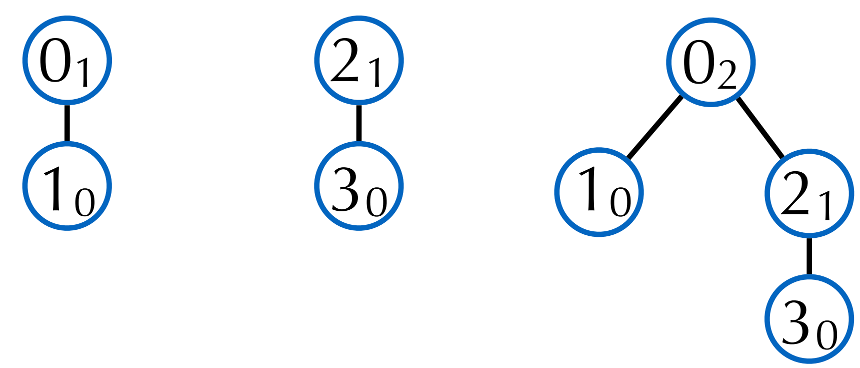
```
def link(p,a,b):
    if a == b: return
    if rk[b]>rk[a]: p[a]=b
    else: p[b]=a
    if rk[a]==rk[b]: rk[a]+=1
```

Starting from 0_0 1_0 2_0 3_0

rank denoted by an index

$0 \sim 1$
$2 \sim 3$
$0 \sim 3$

produce successively



Exercise: join this tree to another one of its shape

Union by Rank

Maintain rank (=height).
Link short trees to higher ones.

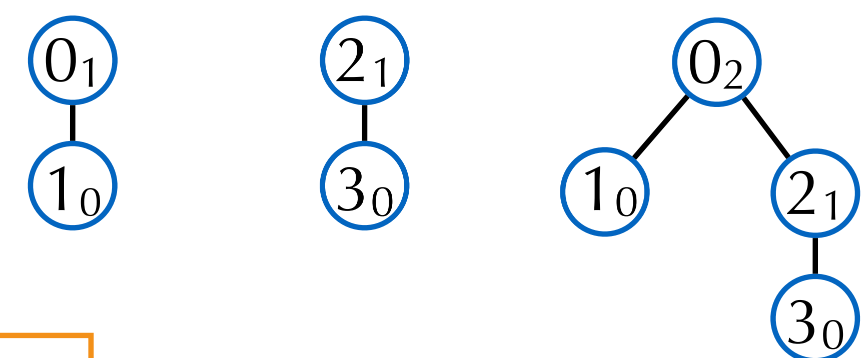
```
def link(p,a,b):
    if a == b: return
    if rk[b]>rk[a]: p[a]=b
    else: p[b]=a
    if rk[a]==rk[b]: rk[a]+=1
```

Starting from 0_0 1_0 2_0 3_0

rank denoted by an index

$0 \sim 1$
$2 \sim 3$
$0 \sim 3$

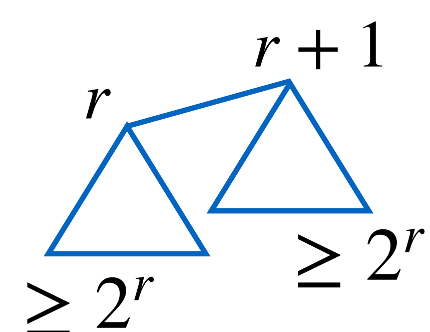
produce successively



Exercise: join this tree to another one of its shape

Properties.
. rank increases from leaf to root;
. size of tree $\geq 2^{\text{rank(root)}}$;
. num nodes of rank $r \leq n/2^r$.

Proof by induction.



Union by Rank

Maintain rank (=height).
Link short trees to higher ones.

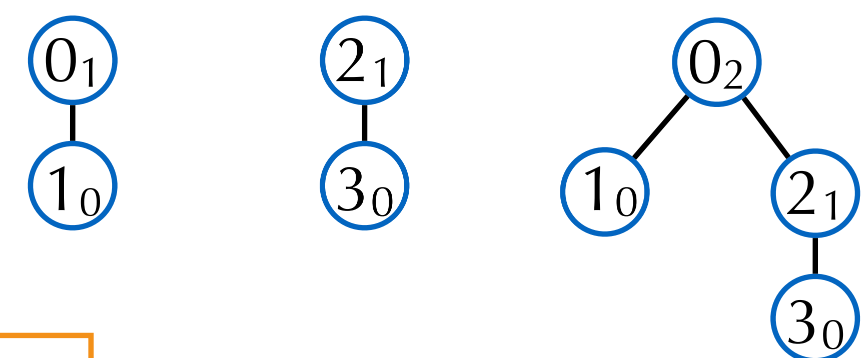
```
def link(p,a,b):
    if a == b: return
    if rk[b]>rk[a]: p[a]=b
    else: p[b]=a
    if rk[a]==rk[b]: rk[a]+=1
```

Starting from 0_0 1_0 2_0 3_0

rank denoted by an index

$0 \sim 1$
$2 \sim 3$
$0 \sim 3$

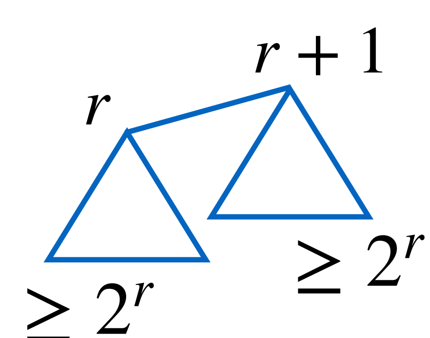
produce successively



Exercise: join this tree to another one of its shape

Properties.
. rank increases from leaf to root;
. size of tree $\geq 2^{\text{rank(root)}}$;
. num nodes of rank $r \leq n/2^r$.

Proof by induction.



$\Longrightarrow$ Worst case for find:

Union by Rank

Maintain rank (=height).
Link short trees to higher ones.

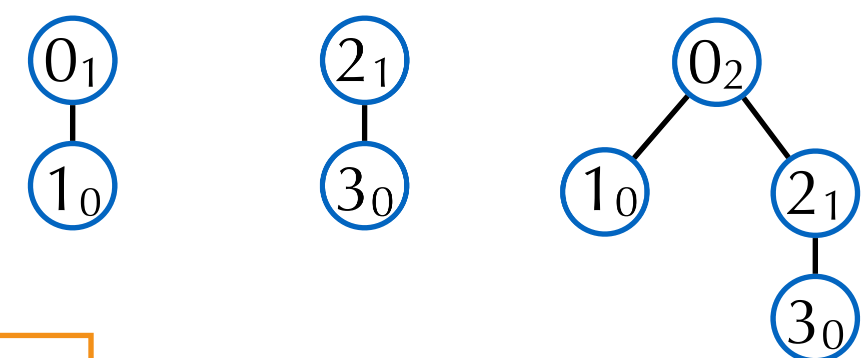
```
def link(p,a,b):
    if a == b: return
    if rk[b]>rk[a]: p[a]=b
    else: p[b]=a
    if rk[a]==rk[b]: rk[a]+=1
```

Starting from 0_0 1_0 2_0 3_0

rank denoted by an index

$0 \sim 1$
$2 \sim 3$
$0 \sim 3$

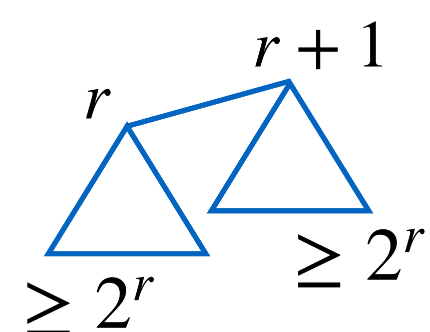
produce successively



Exercise: join this tree to another one of its shape

Properties.
. rank increases from leaf to root;
. size of tree $\geq 2^{\text{rank(root)}}$;
. num nodes of rank $r \leq n/2^r$.

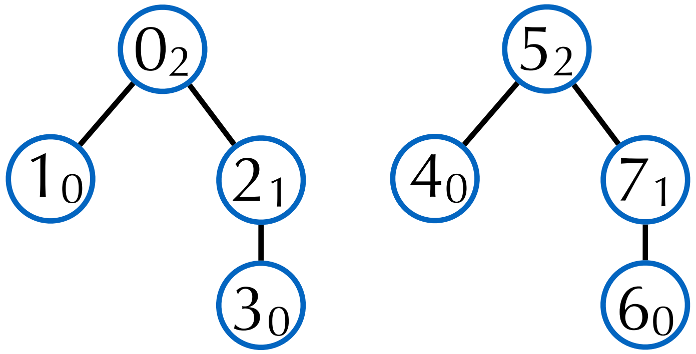
Proof by induction.

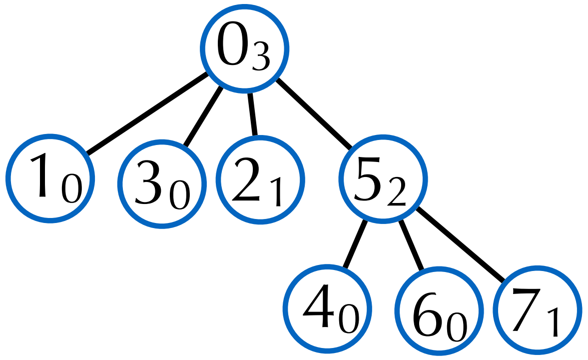


$\Longrightarrow$ Worst case for find: $O(\log n)$.

Path Compression

Every `find` branches all the nodes it visits to their root.

From 

$3 \sim 6$ gives 


```
def find(p,a):
    if p[a]!=a: p[a]=find(p,p[a])
    return p[a]
```

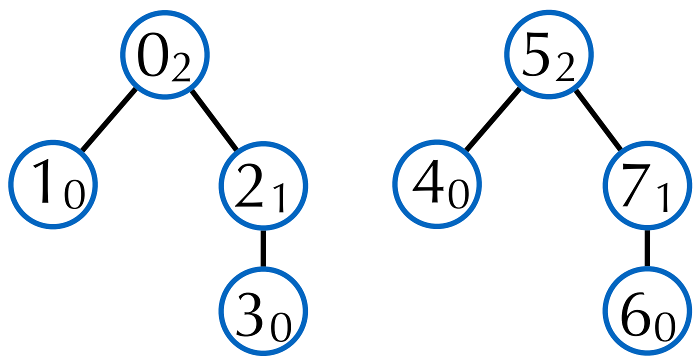
Preserves the properties of rank (becomes an upper bound on height)

Worst-case for find unchanged.

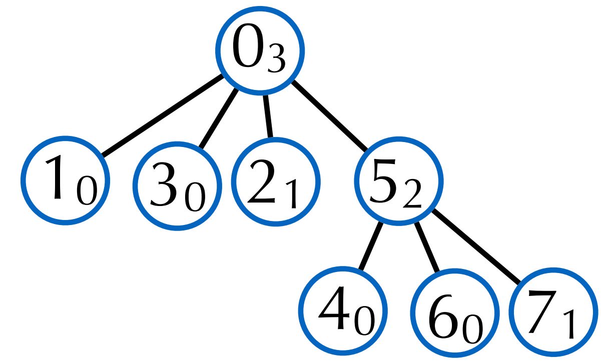
Path Compression

Every `find` branches all the nodes it visits to their root.

From



$3 \sim 6$ gives




```
def find(p,a):
    if p[a]!=a: p[a]=find(p,p[a])
    return p[a]
```

Preserves the properties of rank (becomes an upper bound on height)

Worst-case for find unchanged.

Thm. A sequence of $m \geq n$ `union` or `find` operations uses $O(m \log^\star n)$ array accesses.

Very good amortized complexity

Proof next 4 slides.

$\log^\star n$: number of iterations of $\log_2$ before reaching ≤ 1.

$\log^\star 2 = 1, \ \log^\star 4 = 2, \ \log^\star 16 = 3,$

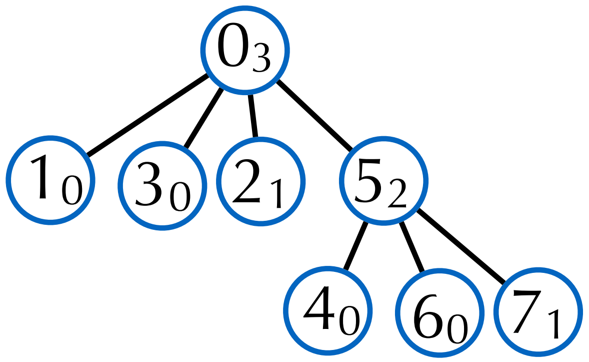
$\log^\star 65536 = 4, \ \log^\star 10^{19000} = 5.$

Constant in practice.

Strategy for the Amortized Analysis

We analyse a sequence of $m \geq n$ union or find.

Difficulty in the analysis:
a node can change parents several times



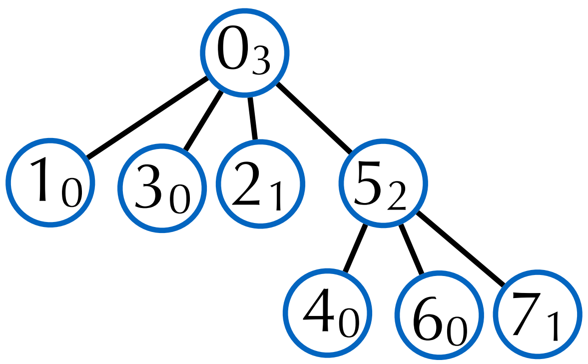
Idea 1: analyze another algorithm with the same cost, easier to handle.

Idea 2: treat high-ranking elements separately, recursively.

$$\#\text{array accesses} = O(m + \#\text{parent changes})$$

Link & Compress

1. Rewrite the sequence of m `union` or `find` as a sequence of $O(m)$ `link` or `compress`



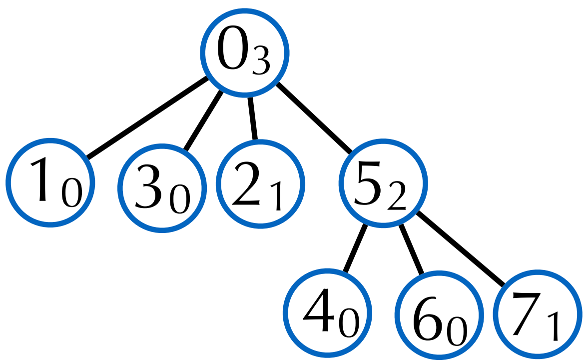
$$l(0,1), l(2,3), l(0,2), l(5,4),$$
$$l(7,6), l(5,7), \underbrace{c(3,0), c(6,5), l(0,5)}_{\text{union}(3,6)}$$

```
def compress(p,a,b):
# b ancestor of a
    if a!=b:
        compress(p,p[a],b)
        p[a]=p[b]
```

Links determine the ranks

Link & Compress

1. Rewrite the sequence of m `union` or `find` as a sequence of $O(m)$ `link` or `compress`

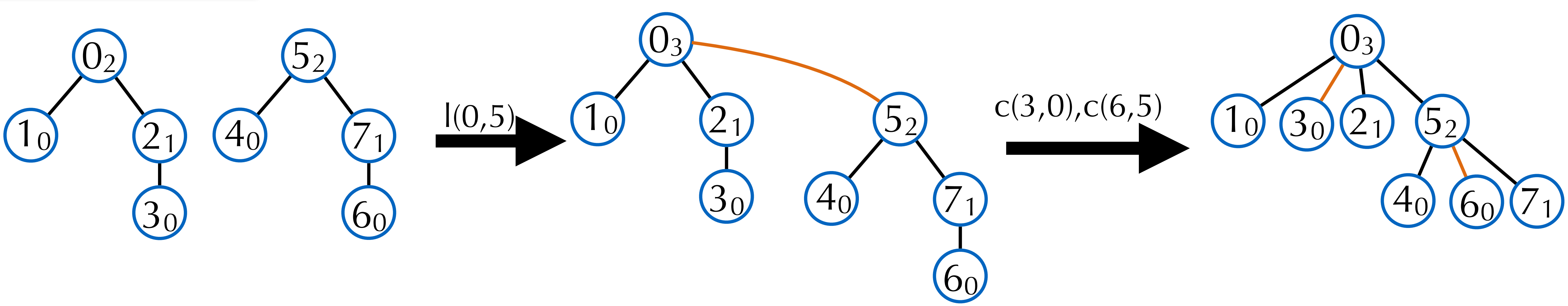


$$l(0,1), l(2,3), l(0,2), l(5,4),$$
$$l(7,6), l(5,7), \underbrace{c(3,0), c(6,5), l(0,5)}_{\text{union}(3,6)}$$

```
def compress(p,a,b):
# b ancestor of a
    if a!=b:
        compress(p,p[a],b)
        p[a]=p[b]
```

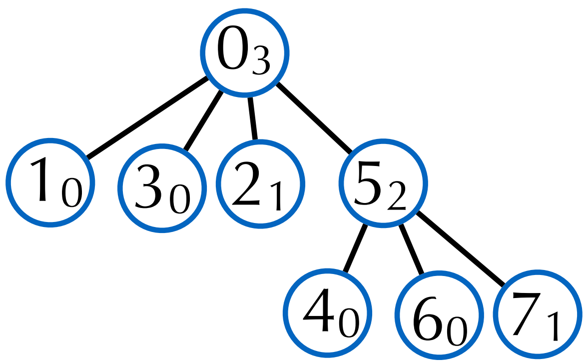
Links determine the ranks

2. Perform the links first (each in $O(1)$ operations)



Link & Compress

1. Rewrite the sequence of m `union` or `find` as a sequence of $O(m)$ `link` or `compress`

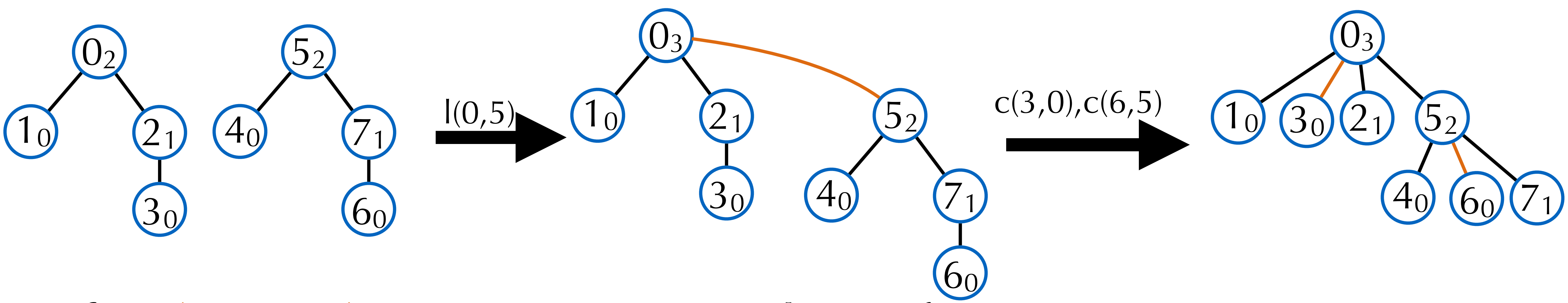


$$l(0,1), l(2,3), l(0,2), l(5,4),$$
$$l(7,6), l(5,7), \underbrace{c(3,0), c(6,5), l(0,5)}_{\text{union}(3,6)}$$

```
def compress(p,a,b):
# b ancestor of a
    if a!=b:
        compress(p,p[a],b)
        p[a]=p[b]
```

Links determine the ranks

2. Perform the links first (each in $O(1)$ operations)

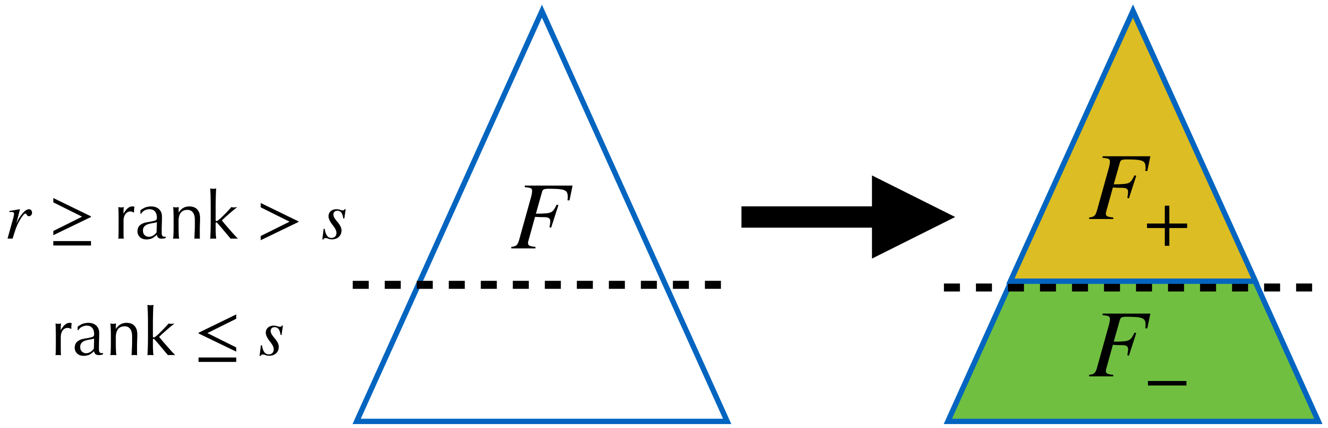


Def. $T(m, n, r)$ worst-case number of parent changes in $\leq m$ `compress` in a forest of $\leq n$ nodes, each of rank $\leq r$.

Simple bound $T(m, n, r) \leq nr$.

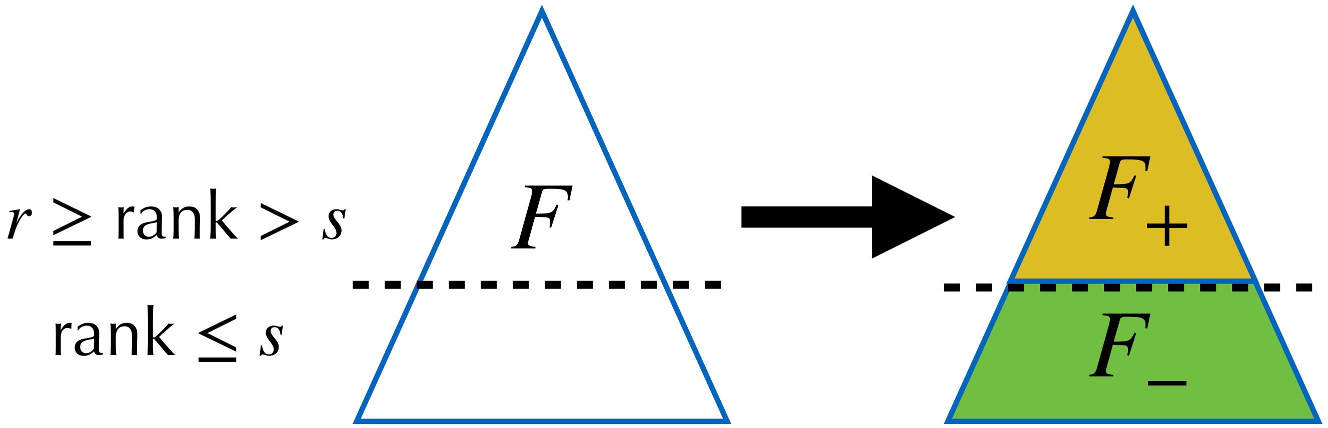
High and Low Forests

Idea: Most of the compressions take place in small rank

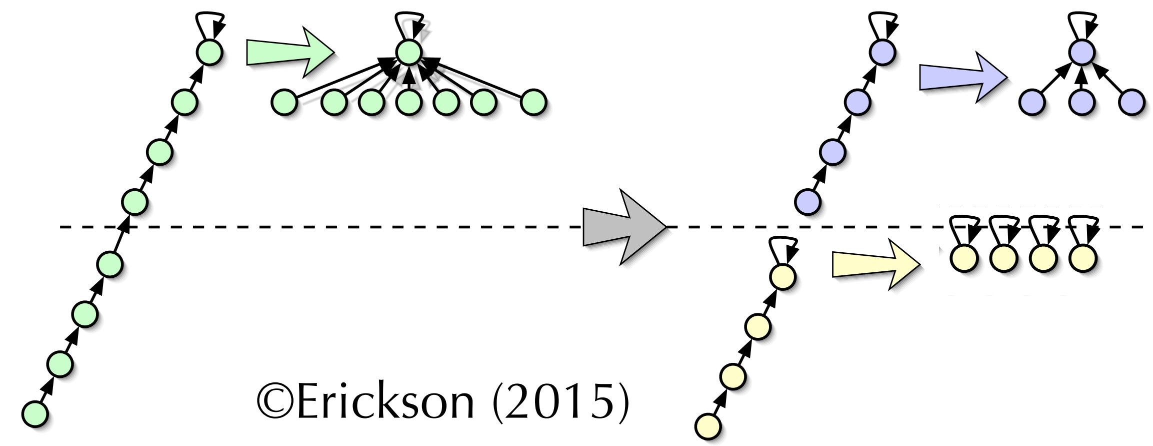


High and Low Forests

Idea: Most of the compressions take place in small rank

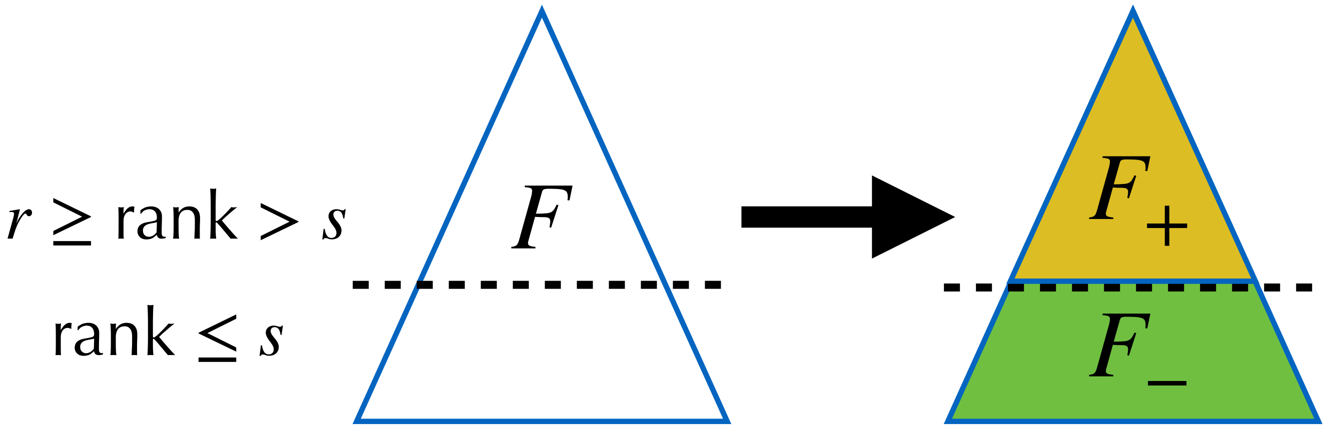


Split compress:

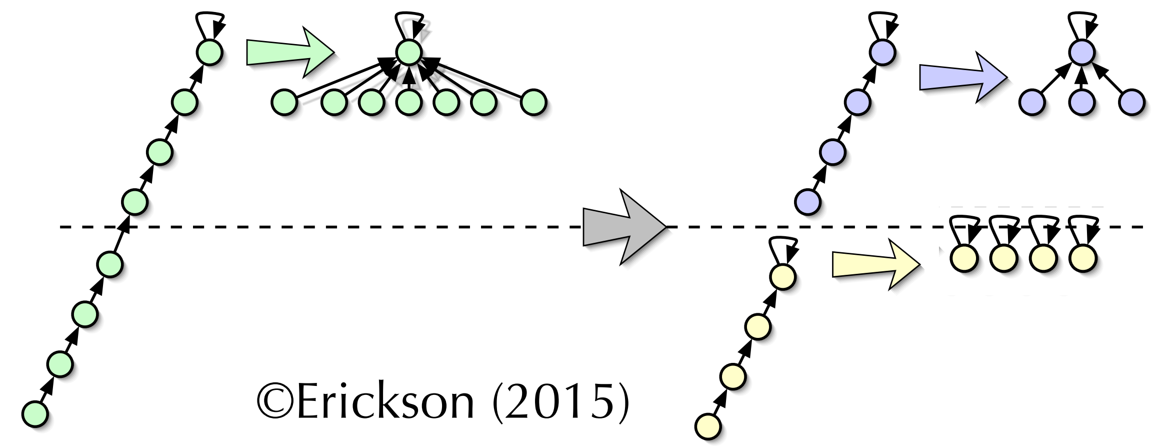


High and Low Forests

Idea: Most of the compressions take place in small rank



Split compress:

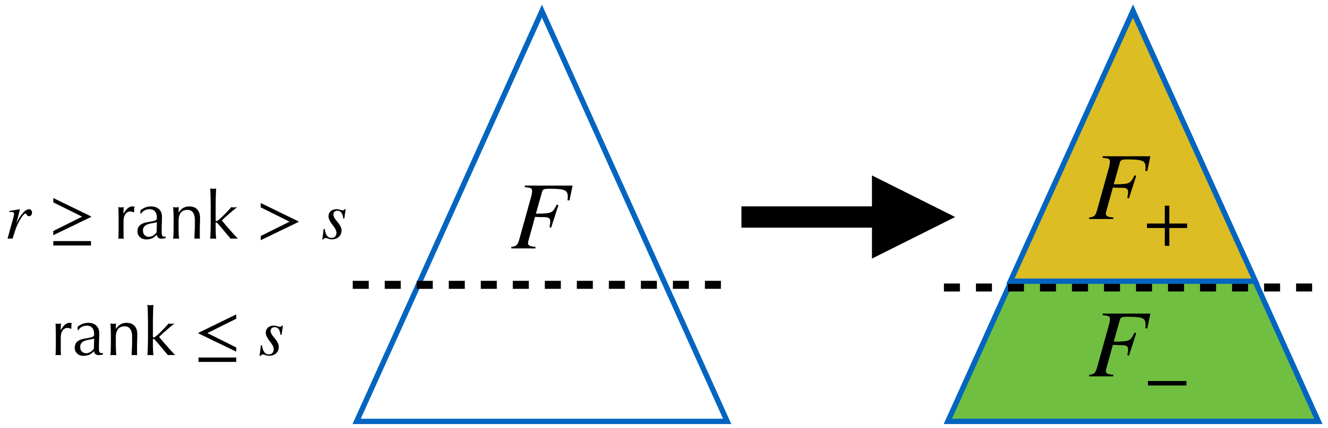



```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F₊)
elif rk[b]≤s then Compress2(a,b,F₋)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F₊)
        Shatter(a,x,F₋)
        p[x]=x
```

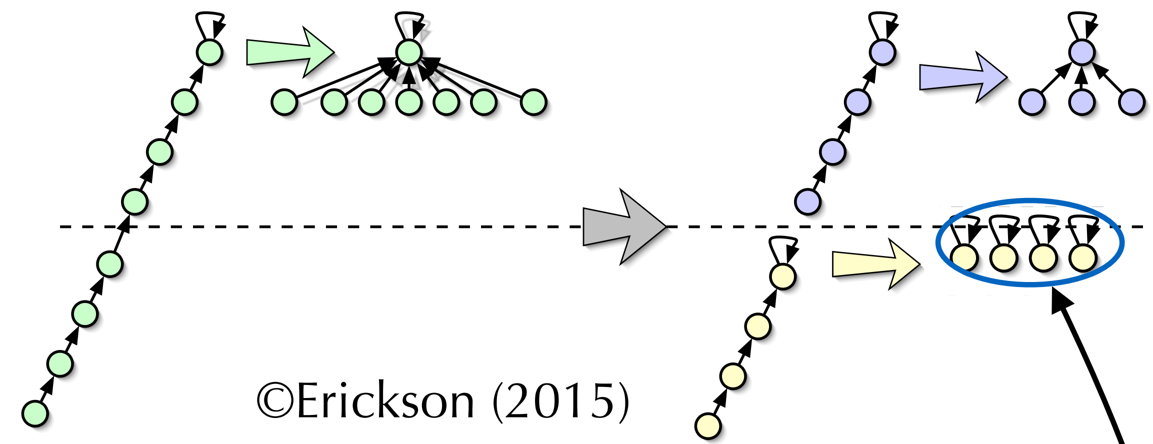
Upper bounds number of parent changes

High and Low Forests

Idea: Most of the compressions take place in small rank



Split compress:




```
Shatter(a,x,F₋):
if p[a] != x then
    Shatter(p[a],x,F₋)
    p[a] = a
```

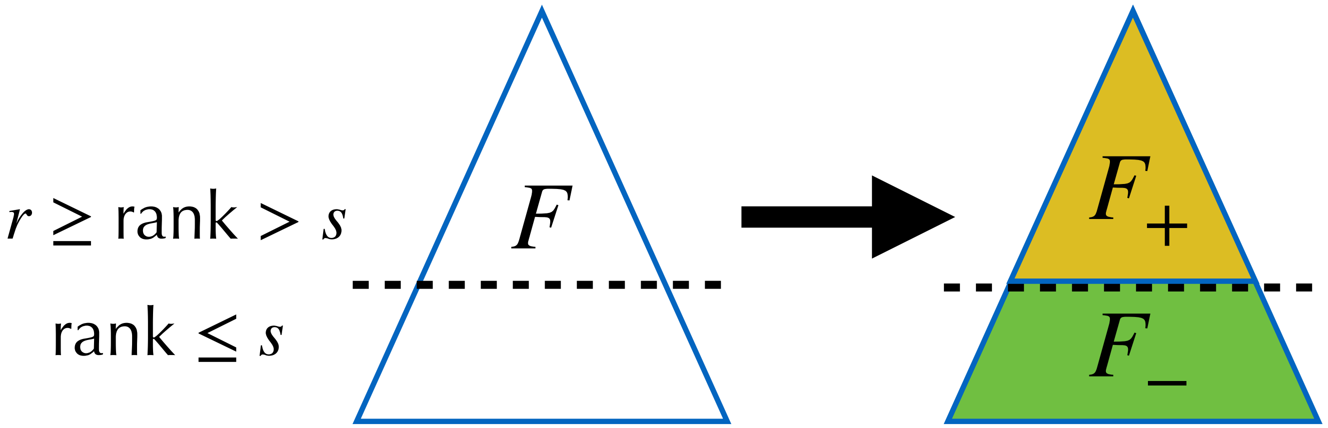
```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F₊)
elif rk[b]≤s then Compress2(a,b,F₋)
else
    x=a
    while rk[p[x]]≤s and p[x]!=x: x=p[x]
    Compress2(p[x],b,F₊)
    Shatter(a,x,F₋)
    p[x]=x
```

new parent in F_+

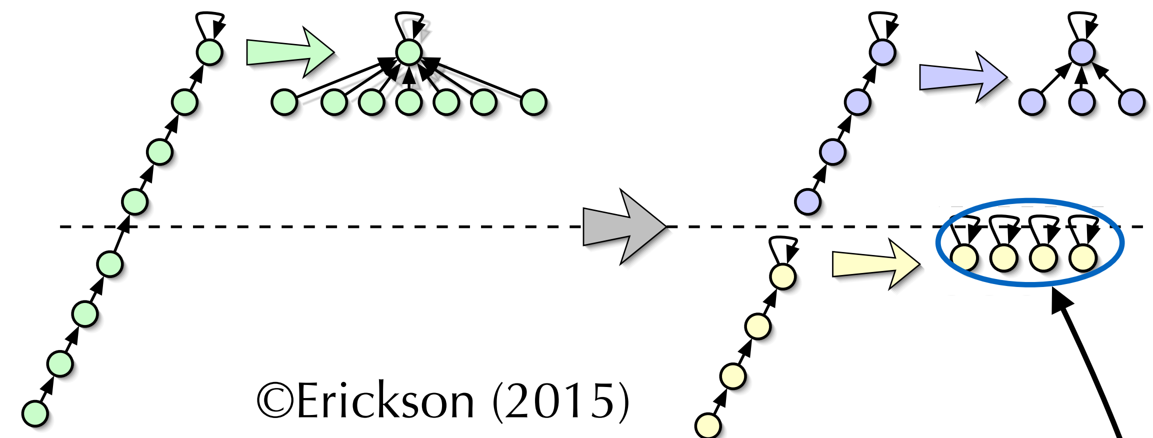
Upper bounds number of parent changes

High and Low Forests

Idea: Most of the compressions take place in small rank



Split compress:



©Erickson (2015)

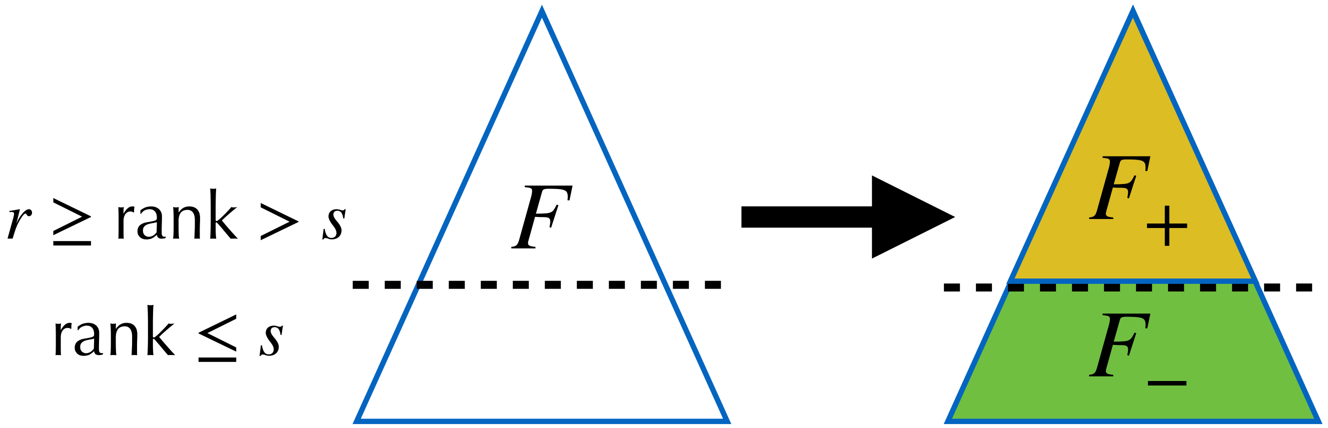
```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F₊)
elif rk[b]≤s then Compress2(a,b,F₋)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F₊)
        Shatter(a,x,F₋)
        p[x]=x
```

new parent in F_+

Upper bounds number of parent changes

High and Low Forests

Idea: Most of the compressions take place in small rank

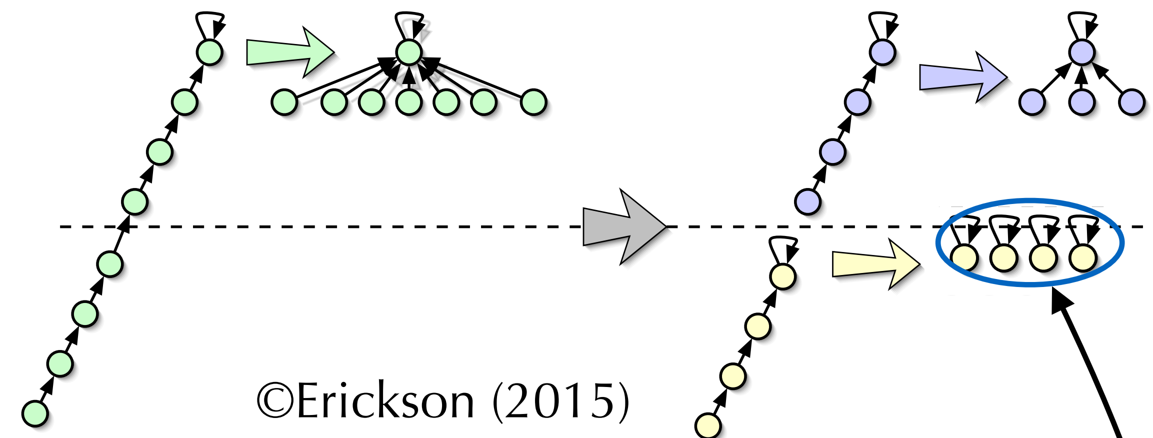


m_- compress purely inside F_-

$m_+ := m - m_-$

C sequence of m compress splits into

Split compress:



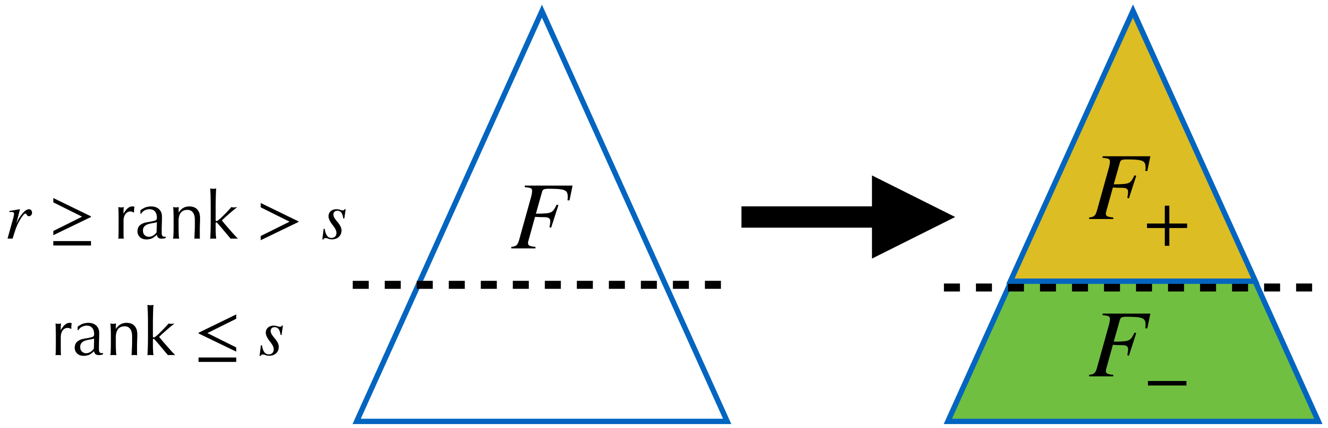

```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F+)
elif rk[b]≤s then Compress2(a,b,F-)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F+)
        Shatter(a,x,F-)
        p[x]=x
```

new parent in F_+

Upper bounds number of parent changes

High and Low Forests

Idea: Most of the compressions take place in small rank



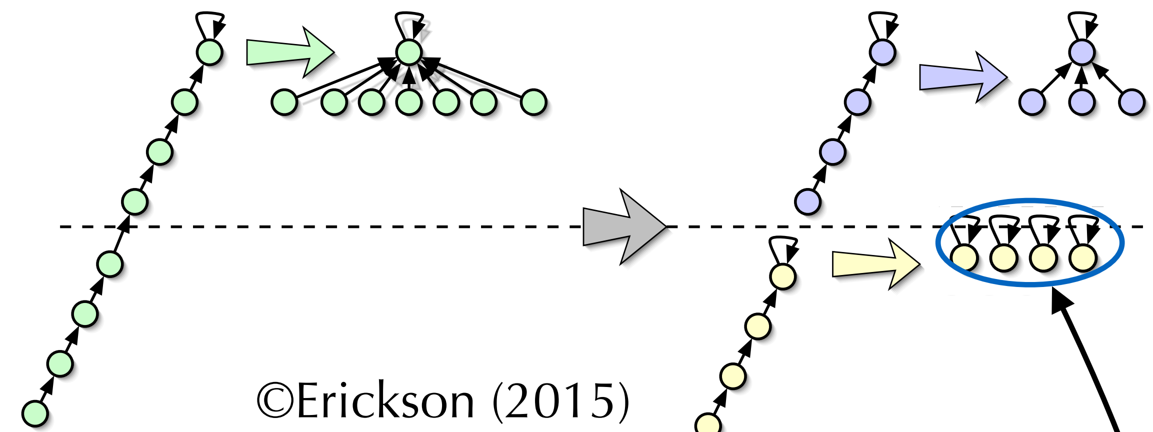
m_- compress purely inside F_-

$m_+ := m - m_-$

C sequence of m compress splits into

m_- compress in F_-, denoted C_-

Split compress:



©Erickson (2015)

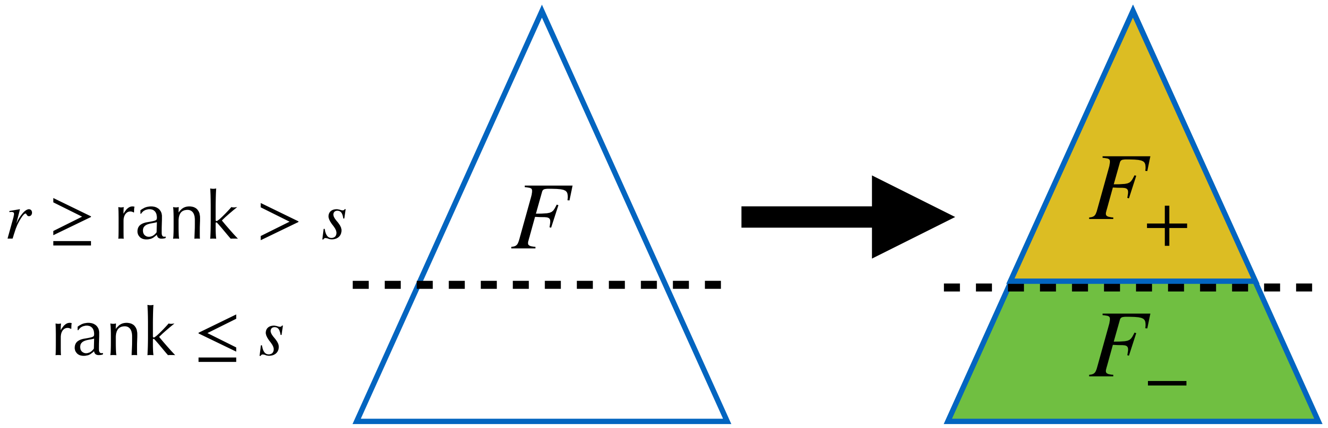
```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F+)
elif rk[b]≤s then Compress2(a,b,F–)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F+)
        Shatter(a,x,F–)
        p[x]=x
```

new parent in F_+

Upper bounds number of parent changes

High and Low Forests

Idea: Most of the compressions take place in small rank



m_- compress purely inside F_-

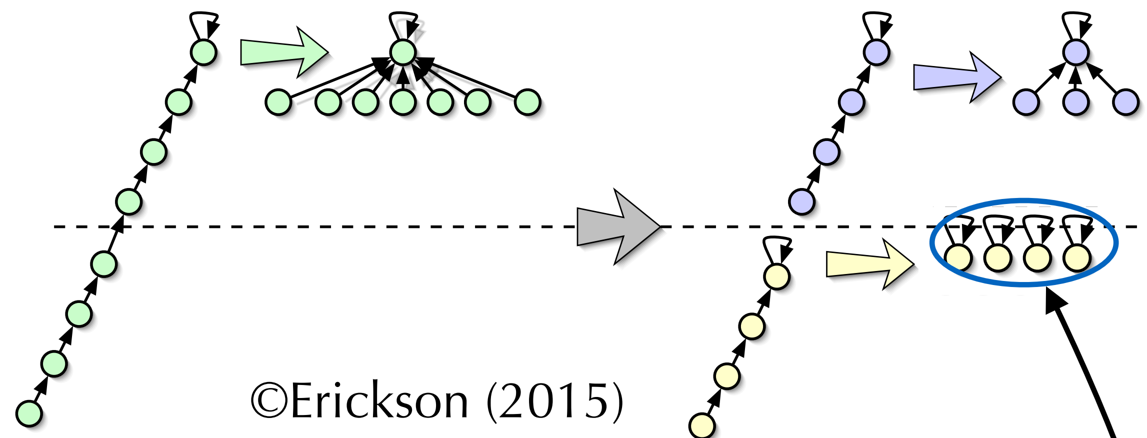
$m_+ := m - m_-$

C sequence of m compress splits into

m_- compress in F_-, denoted C_-

m_+ compress in F_+, denoted C_+

Split compress:



©Erickson (2015)

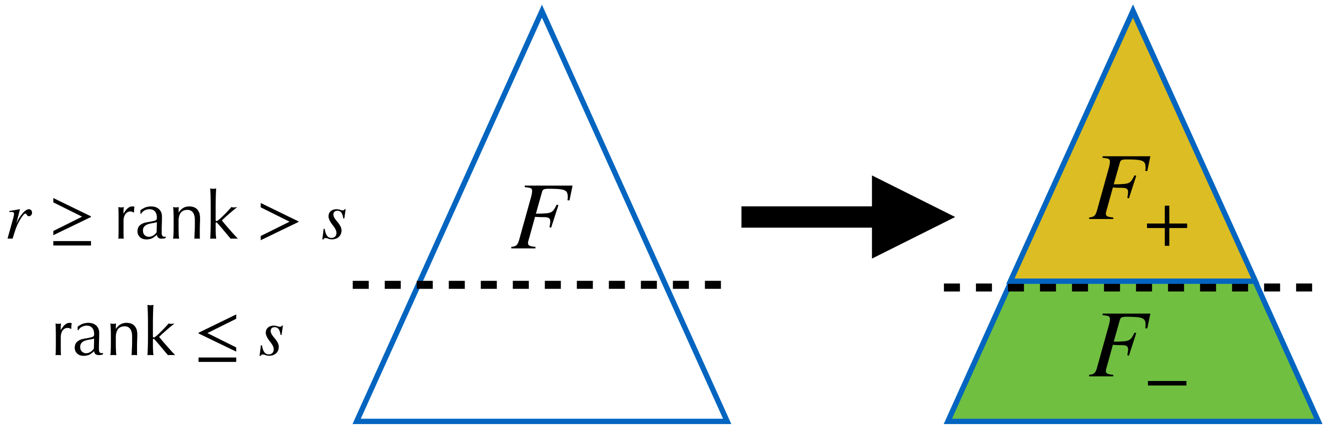
```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F₊)
elif rk[b]≤s then Compress2(a,b,F₋)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F₊)
        Shatter(a,x,F₋)
        p[x]=x
```

new parent in F_+

Upper bounds number of parent changes

High and Low Forests

Idea: Most of the compressions take place in small rank



m_- compress purely inside F_-

$m_+ := m - m_-$

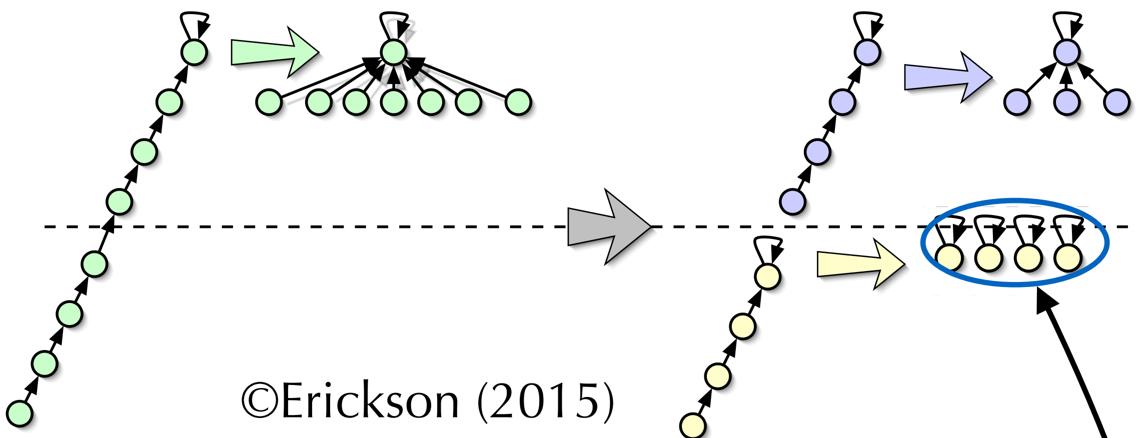
C sequence of m compress splits into

m_- compress in F_-, denoted C_-

m_+ compress in F_+, denoted C_+

$|F_-| \leq n$ parent changes in Shatter

Split compress:



©Erickson (2015)

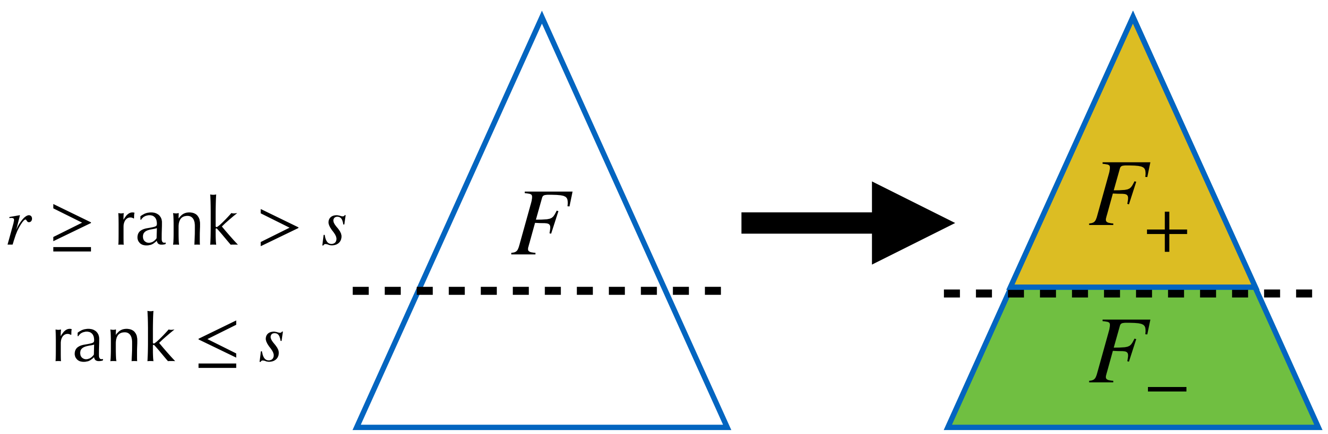
```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F+)
elif rk[b]≤s then Compress2(a,b,F-)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F+)
        Shatter(a,x,F-)
        p[x]=x
```

new parent in F_+

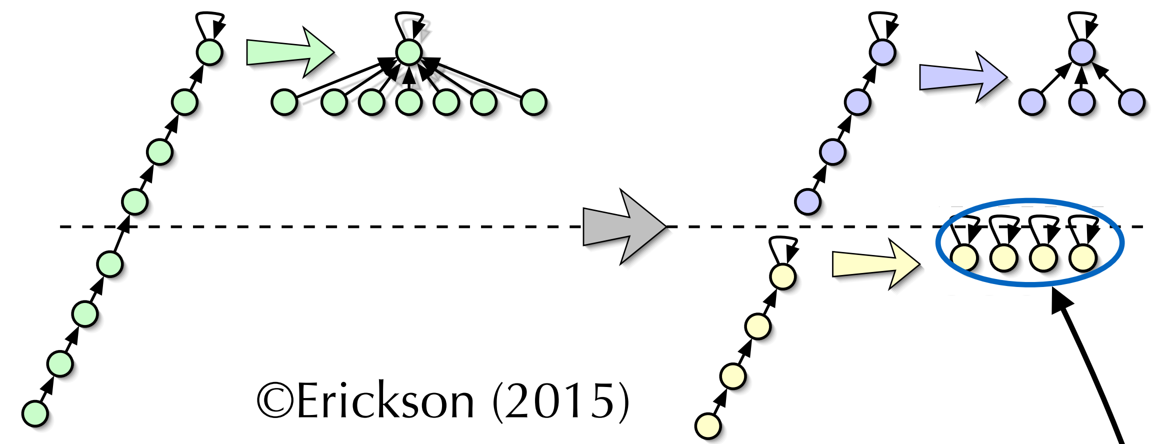
Upper bounds number of parent changes

High and Low Forests

Idea: Most of the compressions take place in small rank



Split compress:



©Erickson (2015)

m_- compress purely inside F_-

$m_+ := m - m_-$

C sequence of m compress splits into

m_- compress in F_-, denoted C_-

m_+ compress in F_+, denoted C_+

$|F_-| \leq n$ parent changes in Shatter

$\leq m_+$ parent changes within F_+

```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F+)
elif rk[b]≤s then Compress2(a,b,F–)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F+)
        Shatter(a,x,F–)
        p[x]=x
```

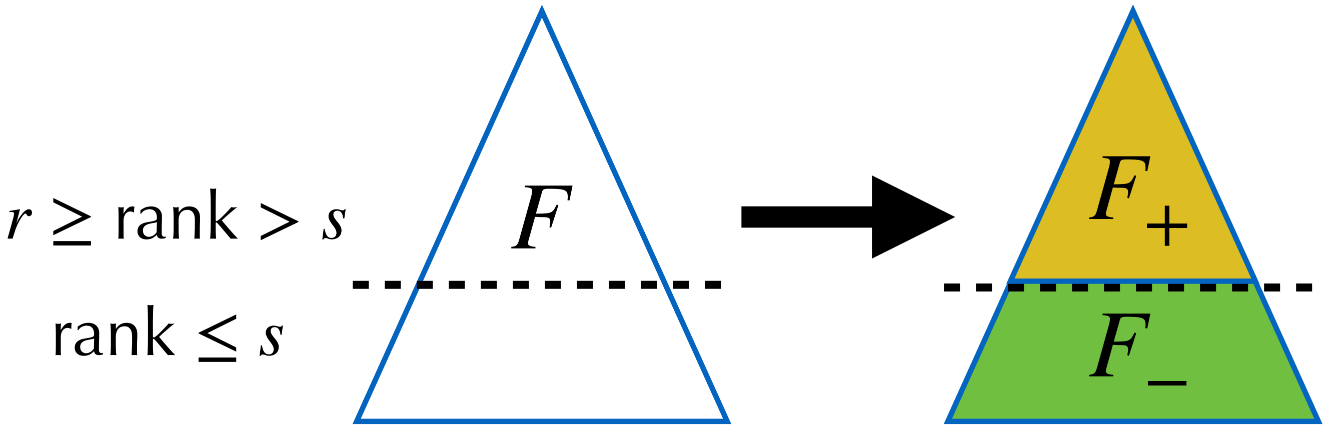
new parent in F_+

Upper bounds number of parent changes

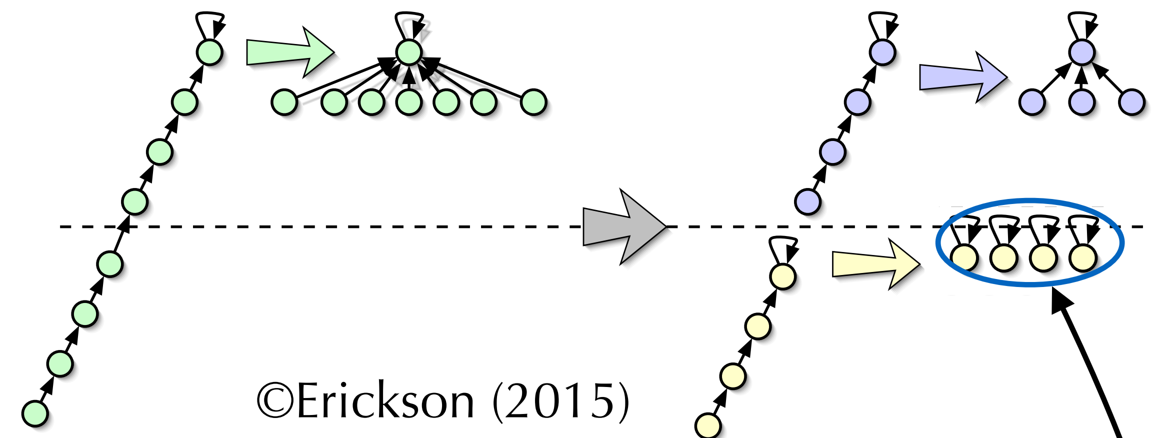
counts parent change within F_+

High and Low Forests

Idea: Most of the compressions take place in small rank



Split compress:



©Erickson (2015)

m_- compress purely inside F_-

$m_+ := m - m_-$

C sequence of m compress splits into

m_- compress in F_-, denoted C_-

m_+ compress in F_+, denoted C_+

$|F_-| \leq n$ parent changes in Shatter

$\leq m_+$ parent changes within F_+

```
Compress2(a,b,F):
if rk[a]>s then Compress2(a,b,F+)
elif rk[b]≤s then Compress2(a,b,F-)
else
        x=a
        while rk[p[x]]≤s and p[x]!=x: x=p[x]
        Compress2(p[x],b,F+)
        Shatter(a,x,F-)
        p[x]=x
```

new parent in F_+

Upper bounds number of parent changes

counts parent change within F_+

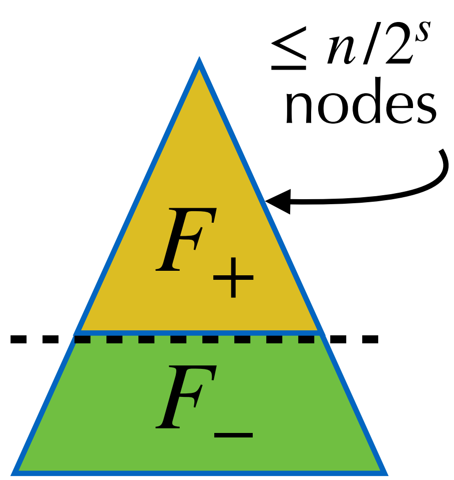
$$T(m,n,r) = T(F,C) \leq T(F_+,C_+) + T(F_-,C_-) + m_+ + n$$

Conclusion

For any sequence C of length $\leq m$
in a forest with n nodes of rank $\leq r$,

$$\underbrace{T(F, C) - m}_{\text{rk}\leq r} \leq \underbrace{T(F_-, C_-) - m_-}_{\text{rk}\leq s} + \underbrace{T(F_+, C_+)}_{\leq rn/2^s} + n$$

by the simple bound 2p. ago

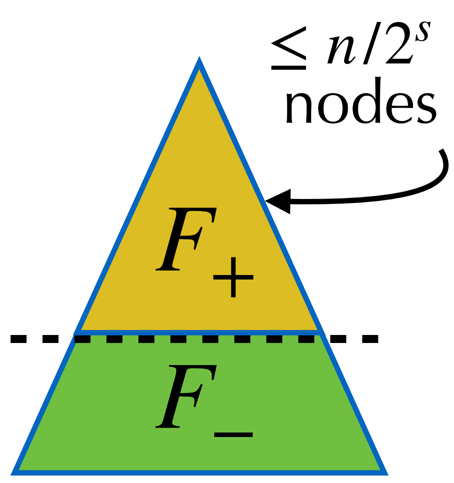


Conclusion

For any sequence C of length $\leq m$
in a forest with n nodes of rank $\leq r$,

$$\underbrace{T(F, C) - m}_{\text{rk}\leq r} \leq \underbrace{T(F_-, C_-) - m_-}_{\text{rk}\leq s} + \underbrace{T(F_+, C_+)}_{\leq rn/2^s} + n$$

by the simple bound 2p. ago



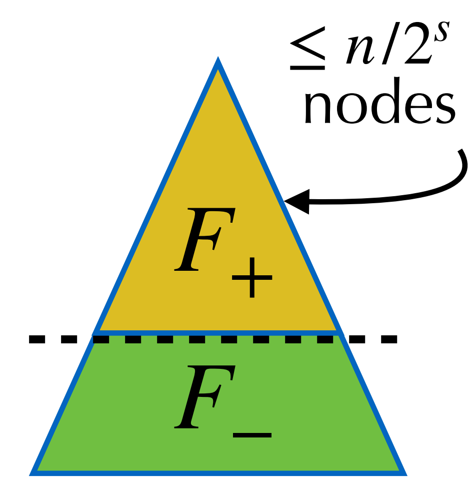
Choose $s = \log_2 r$

$$\underbrace{T(F, C) - m}_{\text{rk}\leq r} \leq \underbrace{T(F_-, C_-) - m_-}_{\text{rk}\leq \log_2 r} + 2n$$

Conclusion

For any sequence C of length $\leq m$
in a forest with n nodes of rank $\leq r$,

$$\underbrace{T(F,C) - m}_{\text{rk}\leq r} \leq \underbrace{T(F_-,C_-) - m_-}_{\text{rk}\leq s} + \underbrace{T(F_+,C_+)}_{\substack{\leq rn/2^s \\ \text{by the simple bound 2p. ago}}} + n$$



Choose $s = \log_2 r$

$$\underbrace{T(F,C) - m}_{\text{rk}\leq r} \leq \underbrace{T(F_-,C_-) - m_-}_{\text{rk}\leq \log_2 r} + 2n$$

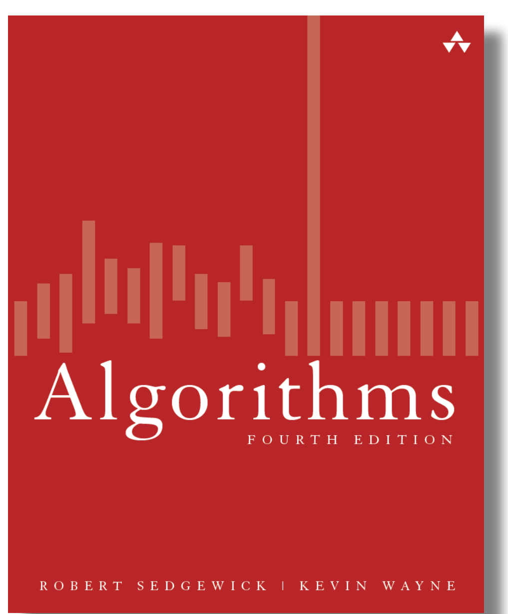
Iterating $\log^\star r$ times yields

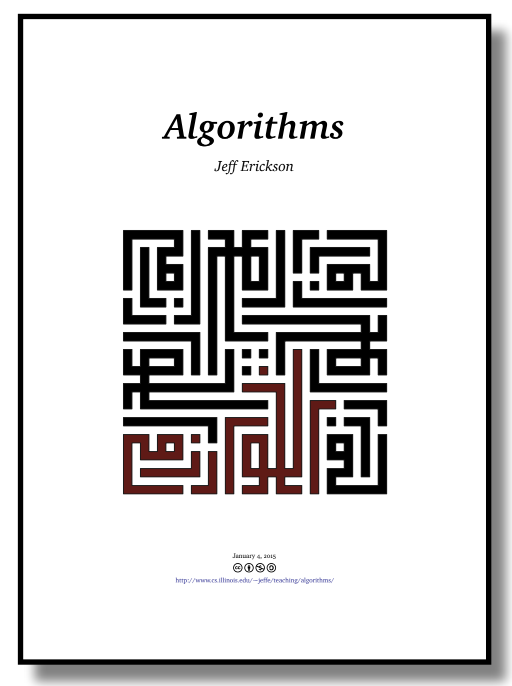
$$T(F,C) \leq m + 2n\log^\star r = O(m \log^\star n) \quad (m \geq n, r \leq n)\,.$$

References for this lecture

The slides are designed to be self-contained.

They were prepared using the following books that I recommend if you want to learn more:





Next

No Assignment

Next tutorial: midterm

Next week: Balancing against Worst-Case

Feedback

Moodle

Questions: constantin.enea@polytechnique.edu