COS 423 Lecture 13
Analysis of path Compression

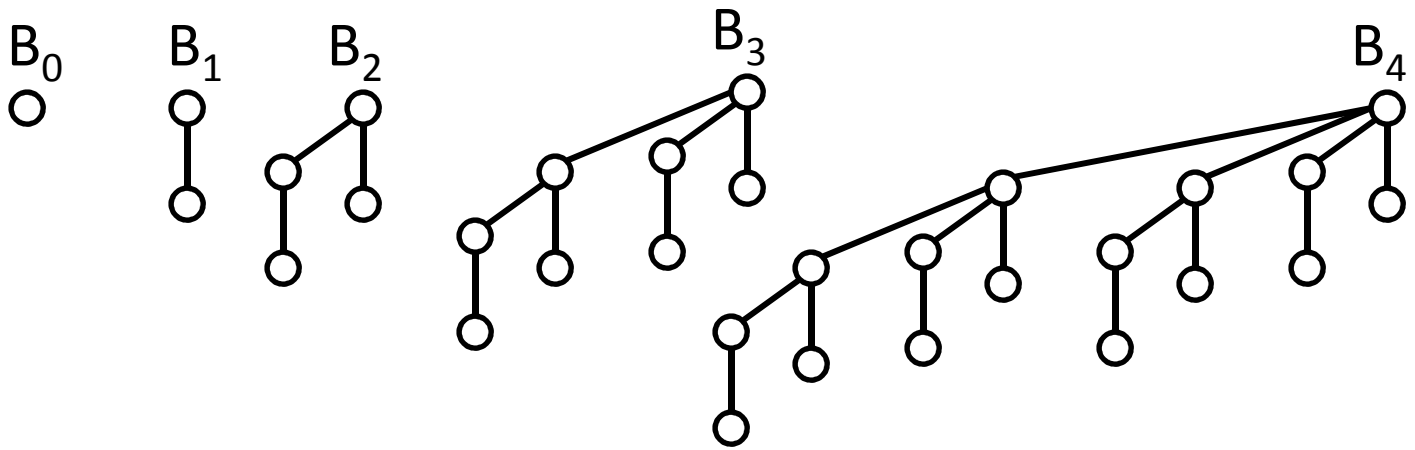
Path compression with naïve linking

Bad example for path compression?

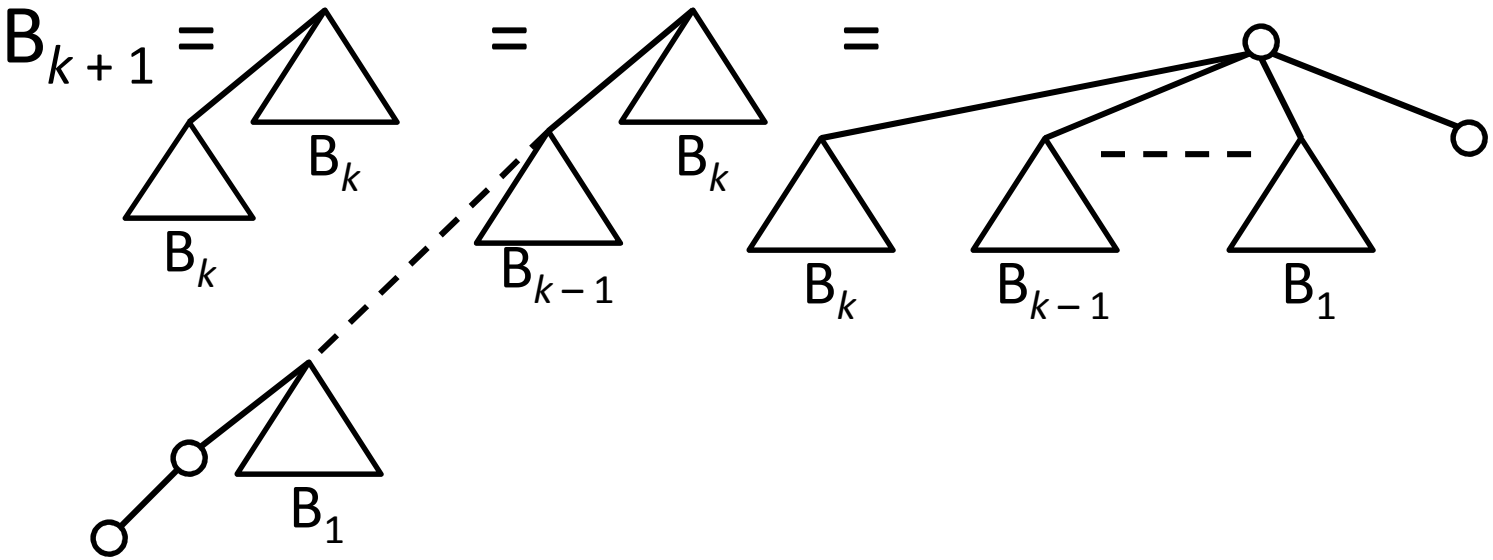
A path of *n* nodes can result in one find path of *n* nodes, but compression flattens the tree: not repeatable

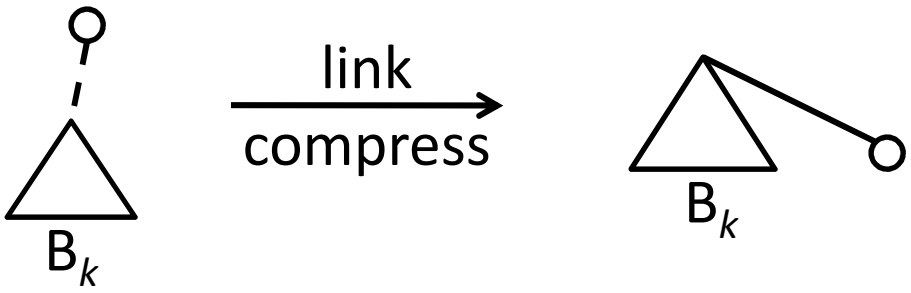
Need a class of trees preserved by path compression: *binomial trees*

Binomial trees



$B_0 = \circ$,





Given $n = 2^k$, build a B_{k-1}. Then repeat $n/2$ times: link with a singleton, do a find on deepest element: $\Theta(\lg n)$ time per find

Let the *density of finds* $d = \lceil m/n \rceil$. As d increases, the amortized time per find decreases: $\Theta(\log_{d+1} n)$

Lower bound: class of trees preserved by a link with a singleton followed by d finds (generalized binomial trees: exercise)

Upper bound: debit argument

Count changes of parent once a node becomes a non-root: undercounts number of nodes on each find path by 2

For purposes of the analysis we give each node a rank: when make-set(x) occurs, $r(x) \leftarrow 0$; when $a(y) \leftarrow x$ in a link, $r(x) \leftarrow \max\{r(x), r(y) + 1\}$

Without compression, $r(x) = h(x)$; with compression, $r(x) \geq h(x)$. With or without compression, $r(x) < r(a(x))$; $r(a(x))$ never decreases

Let x be a non-root. We charge a change in $a(x)$ during a find to the corresponding increase in $r(a(x))$. We define $k(x)$ and $j(x)$, the *level of* x and the *index of* x, as follows:

$$k(x) = \max\{k \mid (d+1)^k \leq r(a(x)) - r(x)\}$$

$$j(x) = \max\{j \mid j(d+1)^{k(x)} \leq r(a(x)) - r(x)\}$$

$0 \leq k(x) \leq \lfloor \log_{d+1} n \rfloor$, $1 \leq j(x) \leq d$

When a find occurs, if x is a node whose parent changes, we give x one debit unless it is the last node in its level along the find path.

Along each find path, there is at most one node per level that is last in the level, totaling $1 + \lfloor \log_{d+1} n \rfloor$ per find. The remaining nodes whose parents change are debited for the change.

How many debits in total?

Let x be a node. Then $k(x)$ never decreases.

Suppose x gets a debit. Then there is a node y after x on the find path such that $k(x) = k(y)$. Let a, a', respectively, be the parent functions before and after the compression. Then

$$\begin{aligned} r(a'(x)) - r(x) &\geq r(a(y)) - r(x) \\ &\geq r(a(y)) - r(y) + r(a(x)) - r(x) \\ &\geq (d+1)^{k(x)} + j(x)(d+1)^{k(x)} \\ &\geq (j(x)+1)(d+1)^{k(x)} \end{aligned}$$

Thus when x incurs a debit, its index or its level increases. Since $j(x)$ can only increase $d - 1$ times before $k(x)$ increases, x can incur at most $d \lfloor \log_{d+1} n \rfloor$ debits. Summing over all nodes, #debits = $O(m \log_{d+1} n)$

→ amortized time per find = $O(\log_{d+1} n)$

Path compression with linking by rank

History of bounds (amortized time per find)

- 1971 O(1) (false)
- 1972 O(lglg$n$) M. Fisher
- 1973 O(lg*n) Hopcroft & Ullman
- 1975 Θ(α(n, d)) Tarjan
- later Ω(lglg$n$) (false)
- 2005 top-down analysis Seidel & Sharir

Ackermann's function (Péter & Robinson)

$A(k, j) = j + 1$ if $k = 0$

$= A(k - 1, 1)$ if $k > 0, j = 0$

$= A(k - 1, A(k, j - 1))$ if $k > 0, j > 0$

$A(1, j) = j + 2$, $A(2, j) = 2j + 3$, $A(3, j) > 2^j$, $A(4, j) >$ tower of j 2's, $A(4, 2)$ has 19,729 decimal digits

A(k, j) is strictly increasing in both arguments

$\alpha(n, d) = \min\{k > 0 | A(k, d) > n\}$

Upper bound: debit argument

Count changes of parent once a node becomes a non-root: undercounts number of nodes on each find path by 2. Charge a change in $a(x)$ to the corresponding increase in $r(a(x))$.

If $r(x) \geq d$, we define $k(x)$ and $j(x)$, the *level of* x, and the *index of* x, as follows:

$$k(x) = \max\{k \mid A(k, r(x)) \leq r(a(x))\}$$

$$j(x) = \max\{j \mid A(k(x) + 1, j) \leq r(a(x))\}$$

$A(0, r(x)) = r(x) + 1 \rightarrow k(x) \geq 0$

$A(\alpha(n, d), d) > n \rightarrow k(x) < \alpha(n, d)$ $(r(x) \geq d)$

$A(k(x) + 1, 0) = A(k(x), 1) \rightarrow j(x) \geq 0$ $(r(x) \geq 1)$

$A(k(x) + 1, r(x)) > r(a(x)) \rightarrow j(x) < r(x)$

$\rightarrow 0 \leq k(x) < \alpha(n, d), 0 \leq j(x) < r(x)$ if $r(x) \geq d$

We charge for nodes whose parent changes as a result of a find. If x is such a node, we give x a debit if $r(x) < d$ and $r(a(x)) < d$, or if $r(x) \geq d$ and x is not last in its level on the find path. Every other node on the find path is either last in its level, at most $\alpha(n, d)$ nodes per find, or its rank is $< d$ but the rank of its parent is $\geq d$, at most one node per find. Thus at most $\alpha(n, d) + 1$ nodes per find change parent but do not accrue a debit.

Each charged node x of rank $< d$ can accumulate a charge of at most $d - 1$ before its parent has rank $\geq d$ and it is never charged again. Thus the total number of debits accrued by such nodes is at most $n(d - 1) = O(m)$.

It remains to bound the debits accrued by nodes of rank $\geq d$. Suppose such a node x accrues a debit. Let y be a node after x on the find path with $k(y) = k(x)$. Let α, α' be the parent functions before and after the find, respectively.

$r(a'(x)) \geq r(a(y)) \geq A(k(x), r(y)) \geq A(k(x), r(a(x)))$

$\geq A(k(x), A(k(x) + 1, j(x))$

$= A(k(x) + 1, j(x) + 1)$

→ $j(x)$ or $k(x)$ increases as a result of the find

→ #debits accrued by nodes of high rank

$\leq \alpha(n, d)r$ per node of rank $r \geq d$

The sum of node ranks is at most n (Why?)

→#debits per find = $O(\alpha(n, d))$

= amortized time per find

This argument can be tightened: the function A can grow even faster; the inverse function α can grow even more slowly, e. g. $\alpha(n, d) = \min\{k > 0 | A(k, d) > \lg n\}$, but this only improves the bound by an additive constant.

Seidel and Sharir have shown that for any feasible problem size, the number of parent changes during compressions is at most $m + 2n$

Can extend the bound to the case $m = o(n)$; can tighten the bound to $\alpha(n', d)$, where n' is the number of elements in the set on which the find is done

The bound holds for some one-pass variants of path compression: *path halving*, *path splitting*

Is the bound tight?

We use **double** induction to build examples that change k pointers per find, for any k. The number of nodes is $B(k, j)$, defined as follows:

$B(k, j) = 1$ if $k = 0$

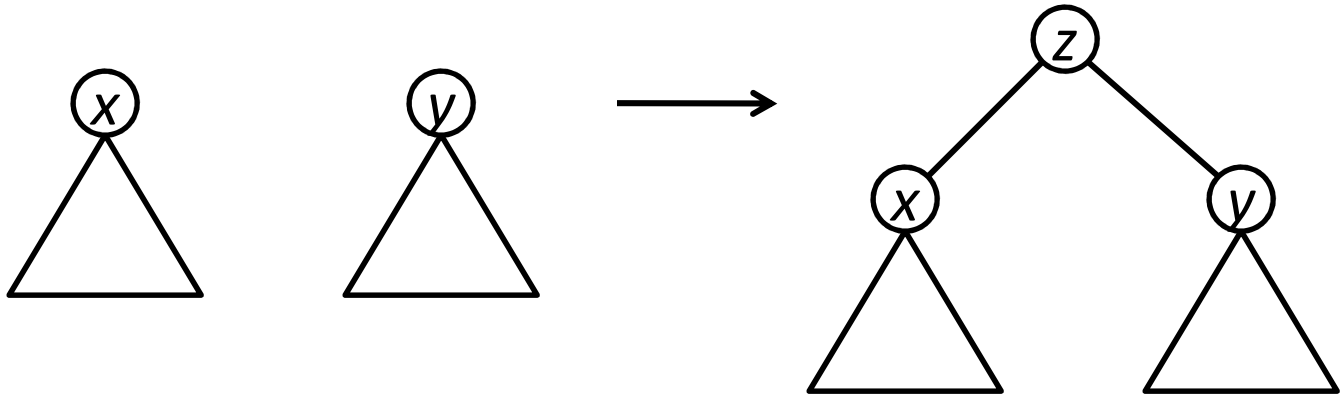
$= 2B(k - 1, 2)$ if $k > 0, j = 1$

$= B(k, j - 1) \times B(k - 1, B(k, j - 1))$ if $k > 0, j > 1$

$B(k, j)$ grows even faster than $A(k, j)$, but the inverses are within an additive constant.

To simplify the argument, we change the way linking is done:

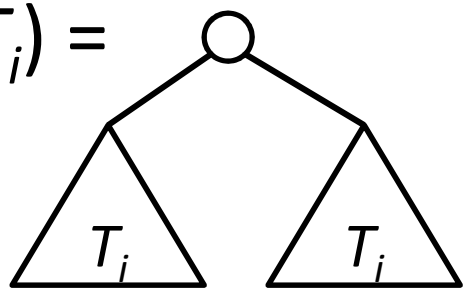
link(x, y): let z be a new root; $a(x) \leftarrow z$; $a(y) \leftarrow z$



We shall only link identical trees. Given a tree T,

$T_0 = T$,

$T_{i+1} = link(T_i, T_i) =$



By such links we can build perfect binary trees.
(Such trees can be Borůvka trees.)

Theorem: Let T be an tree with j leaves other than the root. If $s = B(k, j)$, then starting with s copies of T, there is a sequence of intermixed links and finds such that each find changes at least k pointers, each link combines two T_i trees to form a T_{i+1} tree, and there are js finds, one on each leaf of an original copy of T.

Proof: By double induction on k and j.

Let $k = 0$. Then can do finds on all leaves of $T = T_0$. Each find changes no pointers; no links are needed.

Suppose true for $k - 1$, any j. Let T have one leaf other than the root. Link two copies of T to form T_1. Let T' be T_1 with its two leaves deleted. Then T' has two leaves, the parents of the deleted leaves. By the induction hypothesis there is a sequence of intermixed links and finds on $B(k - 1, 2)$ copies of T' that does finds on all the leaves of the copies, each of find changing $k - 1$ pointers.

The corresponding sequence of intermixed links of copies of T_1 and finds on leaves of the copies changes k pointers per find: each find on a parent is replaced by a find on its child; one more pointer is changed since the find path is one edge longer. Thus the theorem holds for k, $j = 1$: $B(k, 1) = 2B(k - 1, 2)$.

Suppose true for $k - 1$, any j; and for k, $j - 1$. Let T be a tree with j leaves. Starting with $B(k, j - 1)$ copies of T, can do links intermixed with finds on $j - 1$ leaves (all but one) in each copy of T.

Each find changes k pointers, and the final tree T' is a compressed version of T_i for $i = \lg B(k, j-1)$. Repeat this process $B(k-1, B(k, j-1))$ times, resulting in this many copies of T'. Let T'' be T' with all nodes deleted except for proper ancestors of the original leaves which finds have not yet been done. Then T'' has $B(k, j-1)$ leaves. Starting with $B(k-1, B(k, j-1))$ copies of T'', can do links intermixed with finds on all the leaves, each of which changes $k-1$ pointers. Instead do the corresponding sequence of operations on the copies of T', replacing each find by a find on its child that was an original leaf. Each of these finds changes k pointers.

Thus the theorem is true for k, j:

$$B(k, j) = B(k, j-1) \times B(k-1, B(k, j-1))$$

Corollary: Starting with $B(k+1, j)$ singletons, can do an intermixed sequence of links and finds such that there are j finds of each node and each find changes k pointers.

Corollary: Path compression with linking by rank takes $\Omega(n, d)$ amortized time per find.

Proof: Map original links to new links, add extra pointers (shortcuts) for free.

Upper bound by top-down analysis (extra)

Bound the number of parent changes by a divide-and-conquer recurrence

Solve the recurrence (or just plug it into itself repeatedly)

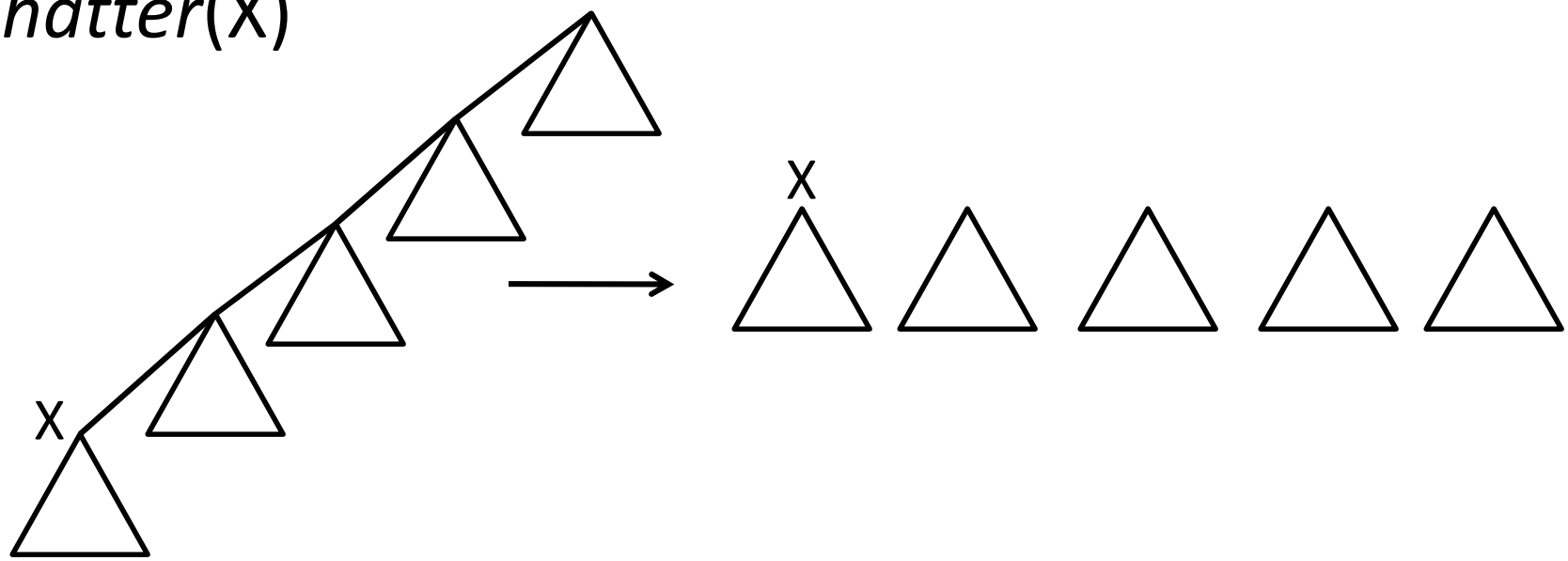
To obtain a closed recurrence, we need a "funny" form of compression

shatter(x): make every ancestor of x a root, by setting its parent to null

Once $a(x)$ becomes null as a result of a shatter, x can no longer be linked; its tree is only subject to compressions and shatterings

The parent changes (to null) that occur during a shatter are counted outside the recursive subproblem in which the shatter occurs; within the subproblem, these changes are free

shatter(X)



To use the method if linking is naïve, we assign ranks to nodes as described previously

To bound parent changes, we partition nodes into *low* and *high* based on their final (maximum) ranks: if $r(x) < k$, x is low, otherwise x is high (k is a parameter)

Given the original problem, we form two subproblems, the *low* and *high* problems, on the low and high nodes, respectively

We do all the make-set operations before all the links and finds, and we give all nodes their final ranks (so that links do not change any ranks)

Each node in the low problem has the same rank as in the original problem; each node in the high problem has rank k less than its rank in the original problem.

Mapping of operations

Consider *link*(*x*, *y*) in the original problem. Let *y* be the new root. (Proceed symmetrically if *x* is the new root.) If *y* is low, do the link in the low problem, if *x* is high, do the link in the high problem, and if *x* is low and *y* is high, do nothing in either subproblem.

Suppose *find*(*x*) in the original problem returns node *y*. If *y* is low, do *find*(*x*) in the low problem. If *x* is high, do *find*(*x*) in the high problem. If *x* is low and *y* is high, let *z* be the first high node along the find path in the original problem; do *shatter*(*x*) in the low problem and *find*(*z*) in the high problem.

Each link in the original problem maps to a link in the low or the high problem or to nothing

Each find in the original problem maps to a find in the low or the high problem and, if to a find in the high problem, possibly to a shatter in the low problem.

If x is any non-root in either the low or the high problem, $r(x) < r(a(x))$.

If linking is naïve, in both the low and high problems there is at least one node per rank, from 0 up to the maximum rank.

If linking is by rank, the number of nodes of rank $\geq j$ in the both low and high problems is at most $1/2^j$ times the total number of nodes in the problem, and the number of nodes in the high problem is at most $1/2^k$ times the total number of nodes in the original problem.

The total cost of finds (number of parent changes) in the original problem is at most the total cost of finds in the high and low problems plus the number of nodes in the low problem plus the number of finds in the high problem:

Each find path that contains both low and high nodes contains one low node whose parent is already a high node, and zero or more nodes whose parent is low but whose new parent is high. Over all finds, this is at most one per find in the high problem plus at most one per node in the low problem.

The recurrence

$C(n, m, r) \leq C(n', m', r') + C(n'', m'', r'') + n' + m''$

Here n, m, and r are the number of nodes, the number of finds, and the maximum rank; single and double primes denote the low and high problems, respectively: $n = n' + n''$, $m = m' + m''$, $r = r' + r''$.

One can use this recurrence to bound the amortized time for finds with path compression, with naïve linking or linking by rank: **your challenge**