EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Hash Tables

- Hashing
  - Technique supporting insertion, deletion and search in average-case constant time
  - Operations requiring elements to be sorted (e.g., FindMin) are not efficiently supported

- Generalizes an ordinary array,
  - Key property: direct addressing
  - An array is a direct-address table: Key value is position of data in array

- Main idea: Transform key into index, compute the index, then use an array of size N
  - Key k: data stored at h(k) (hashing)

- Basic operation is in O(1)!

Hash Function

- Mapping from key to array index is called a hash function
  - Typically, many-to-one mapping
  - Different keys map to different indices
  - Distributes keys evenly over table
- Collision occurs when hash function maps two keys to the same array index

Collision Resolution - 2

- Open addressing
  - If slot is busy, design sequence of other slots to be searched
  - probe alternative cell $h_1(K), h_2(K), \ldots,$until an empty cell is found.
  - $h_i(K)$= (hash(K) + f(i)) mod m, with f(0) = 0
  - f: collision resolution strategy

- Several approaches
  - Linear Probing: f(k) = k
  - Quadratic Probing: f(k) = k^2
  - Double Hashing: two hash functions
  - Cuckoo Hashing: (more to come)

Linear Probing

- f(i) =i
  - cells are probed sequentially (with wrap-around)
  - h_i(K) = (hash(K) + i) mod m

- Insertion:
  - Let K be the new key to be inserted, compute hash(K)
  - For i = 0 to m-1
    - compute L = (hash(K) + I) mod m
    - If T[L] is empty, then we put K there and stop
  - If we cannot find an empty entry to put K, it means that the table is full and we should report an error: Table is full

- Problem: We no longer have O(1) find, insert for worst case.

Example

- E.g, inserting keys 89, 18, 49, 58, 69 with hash(K)=K mod 10 (not prime!)
  - h_i(K) = (hash(K) + i) mod m
-

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratic Probing

- Two keys with different home positions will have different probe sequences
  - e.g. m=101, h(k1)=30, h(k2)=29
  - probe sequence for k1: 30,30+1, 30+4, 30+9
  - probe sequence for k2: 29, 29+1, 29+4, 29+9

- If the table size is prime, then a new key can always be inserted if the table is at least half empty (see proof in text book)
  - Secondary clustering
    - Keys that hash to the same home position will probe the same alternative cells
    - Simulation results suggest that it generally causes less than an extra half probe per search
    - To avoid secondary clustering, the probe sequence need to be a function of the original key value, not the home position

Quadratic probing

- E.g, inserting keys 89, 18, 49, 58, 69 with hash(K)=K mod 10 (not prime!)
  - $h_i(K) = (hash(K) + i^2) \bmod m$
-

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Double Hashing

- To alleviate the problem of clustering, the sequence of probes for a key should be independent of its primary position => use two hash functions: hash() and hash2()
  - f(i) = i * $hash_2(K)$
  - E.g. $hash_2(K)$ = R - (K mod R), with R is a prime smaller than m

- $hash_2(K)$ must never evaluate to zero
  - For any key K, $hash_2(K)$ must be relatively prime to the table size m. Otherwise, we will only be able to examine a fraction of the table entries.
  - One solution is to make m prime, and choose R to be a prime smaller than m

Double Hashing

- E.g, inserting keys 89, 18, 49, 58, 69 with hash(K)=K mod 10 (not prime!)
  - h_2(K) = (hash(K) + i * $h_2(K)$) mod m, where $h_2(K) = K$ mod 7
-

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Rehashing

- When hash table is near full, increase size, rehash all entries
  - Amortized still O(1)…

- When to rehash
  - When table is half full ($\lambda = 0.5$)
  - When an insertion fails
  - When load factor reaches some threshold
  - Works for chaining and open addressing

-

Big Problem with Open Address: Deletion

- If you delete, you break the chain of insertions and lose ability to find!
  - Solution: Add an extra bit to each table entry, and mark a deleted slot by storing a special value DELETED

- $hash_2(K)$ must never evaluate to zero
  - For any key K, $hash_2(K)$ must be relatively prime to the table size m. Otherwise, we will only be able to examine a fraction of the table entries.
  - One solution is to make m prime, and choose R to be a prime smaller than m
-

Perfect Hashing

- Choose a hash function with no collisions: Hard!
  - The expected cost of a lookup in a chained hash table is $O(1 + \alpha)$ for any load factor α
  - Expected cost of a lookup in these tables is not the same as the expected worst-case cost
- Theorem: Assuming truly random hash functions, the expected worst-case cost of a lookup in a linear probing hash table is $\Omega(\log n)$.
- Theorem: Assuming truly random hash functions, the expected worst-case cost of a lookup in a chained hash table is $\Theta(\log n / \log \log n)$.
  - Proofs: CLRS, exercise 11-1 and 11-2.

Perfect Hashing - 2

- Bottom line: perfect hashing needs O(1) inserts, lookups
  - Chaining and linear probing are a long way from this
  - This is for expected worst case, not actual worst case (which is worse!)
  - Need creative techniques to approach perfect hashing

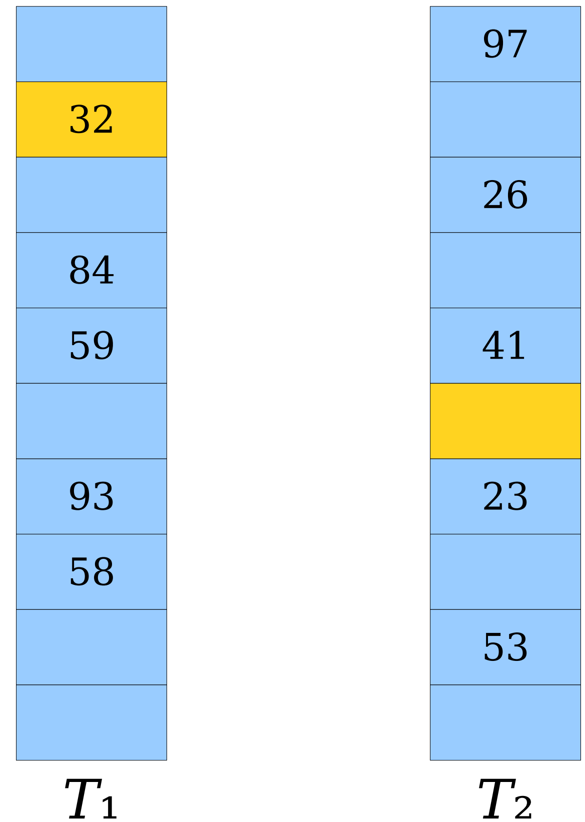
- Let’s try a new idea: Cuckoo Hashing

Cuckoo Hashing

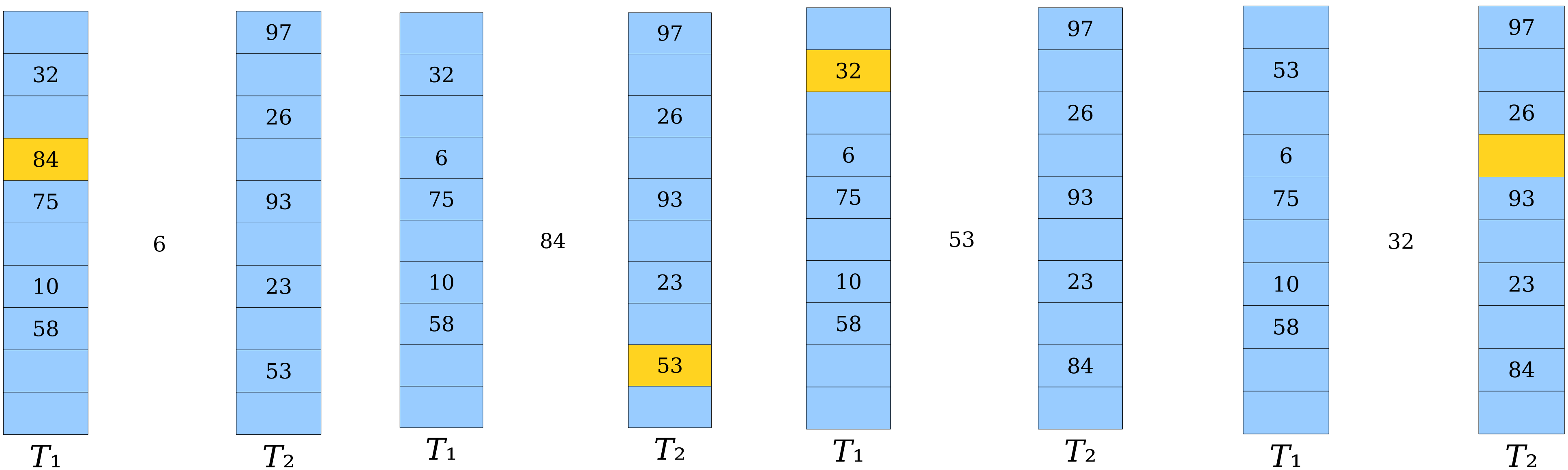
- Cuckoo hashing is a simple hash table where
  - Lookups are worst-case O(1).
  - Deletions are worst-case O(1).
  - Insertions are amortized, expected O(1).
  - Insertions are amortized O(1) with reasonably high probability.
- Key idea: Maintain two table, each with m elements
  - Two hash functions $h_1(K), h_2(K)$
  - Every element K will be either in position $h_1(K)$ in the first table or $h_2(K)$ in the second table

Cuckoo Hashing

- Lookups are O(1) because only 2 locations must be searched
- Deletions take O(1) because only 2 locations must be searched
- To insert an element, try placing it in $h_1(K)$. If empty, place it there
- If not empty, place it there, and kick out existing one (J) and try in $h_2(J)$
- If $h_2(J)$ is busy, place J there, kick out L and try placing it in $h_1(L)$
- Repeat, alternating, until items stabilize

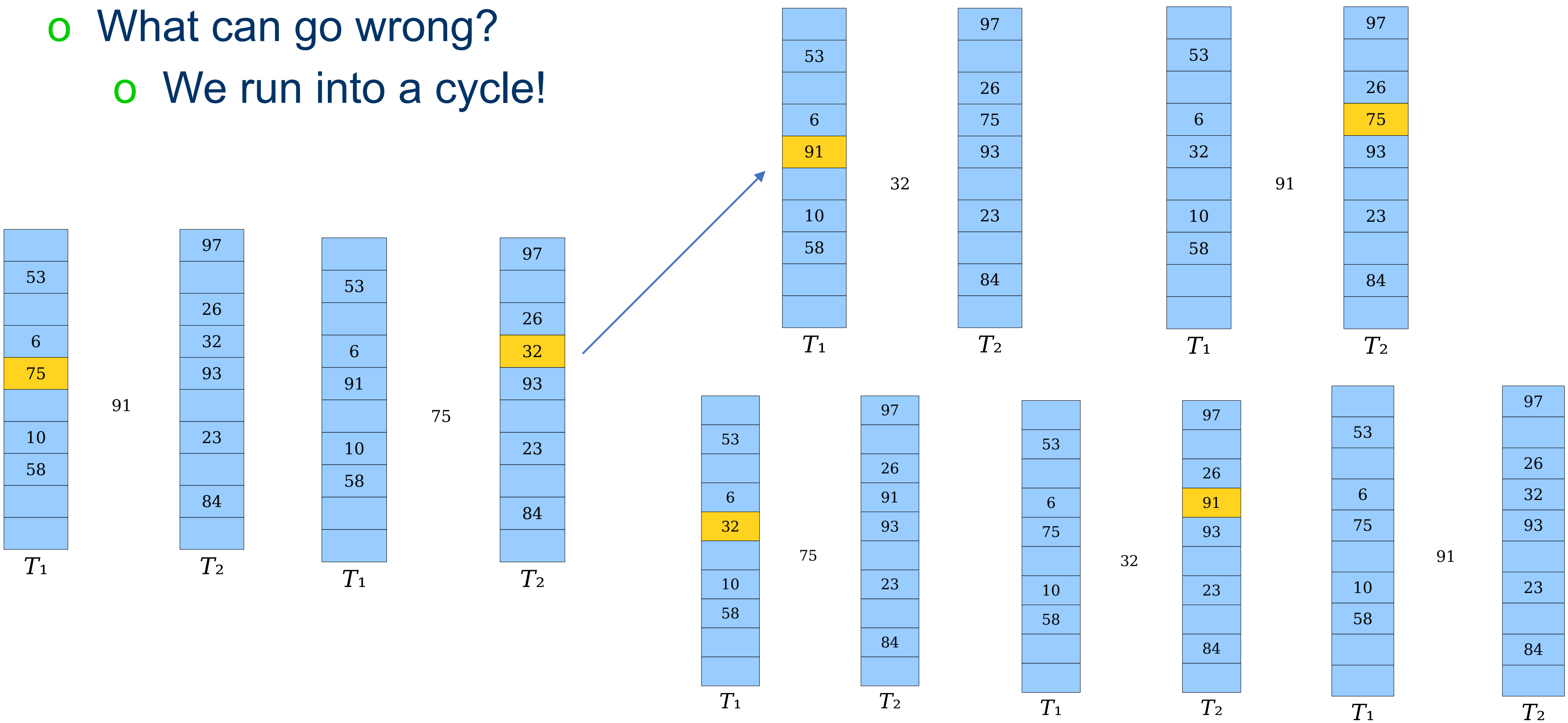


Cuckoo Hashing Example



Cuckoo Hashing

- What can go wrong?
  - We run into a cycle!

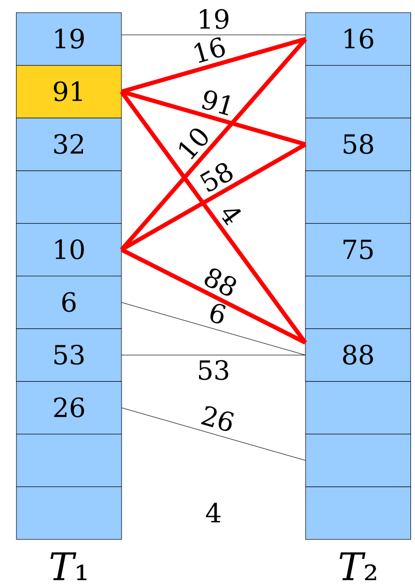


Cuckoo Hashing

- What can go wrong?
  - We run into a cycle!
- If that happens, perform a rehash by choosing two new hash functions and inserting all elements back into the tables
- Multiple rehashes might be necessary before this succeeds
- Cycles only arise if we revisit the same slot with the same element to insert

Cuckoo Hashing Results

- Hard probabilistic analysis: based on random bipartite graphs
  - Uses the Cuckoo graph
  - Beyond scope of our course — still a topic for research
- m: size of one of the hash tables. n:the number of edges between the two tables



- Theorem: If m = (1 + ε)n for ε > 0, the probability that cuckoo graph contains a complex connected component is O(1 / m)

  Keep a little over half the cells empty: $\frac{n}{2m} = \frac{1}{2 + 2\epsilon}$
- Theorem: The expected, amortized cost of an insertion into a cuckoo hash table is O(1 + ε + ε-1)

Cuckoo Hashing Variations

- The hash functions chosen need to have a high degree of independence for results to hold
  - Once numbers of keys gets close to 1/2, failure is imminent!
- Cuckoo hashing with $k \geq 3$ tables tends to perform much better than Cuckoo hashing with $k = 2$ tables
  - With $k = 3$, you can load tables up to 90% before you run into cycles with enough probability
- Another idea: slots in a cuckoo hash table can store multiple elements
  - When displacing an element, choose a random one to move and move it.
  - Works well, makes it unlikely to have long chains

Cuckoo Hashing

- Tricky to analyze
  - Everything moves around, two tables, change hash functions, reinsert
- If that happens, perform a rehash by choosing two new hash functions and inserting all elements back into the tables
- Multiple rehashes might be necessary before this succeeds
- Cycles only arise if we revisit the same slot with the same element to insert

Evolution of Hashing

- Hashing is fast, O(1) inserts, deletes, find
  - But lacks ability for successor, predecessor, find min, …O(n)!

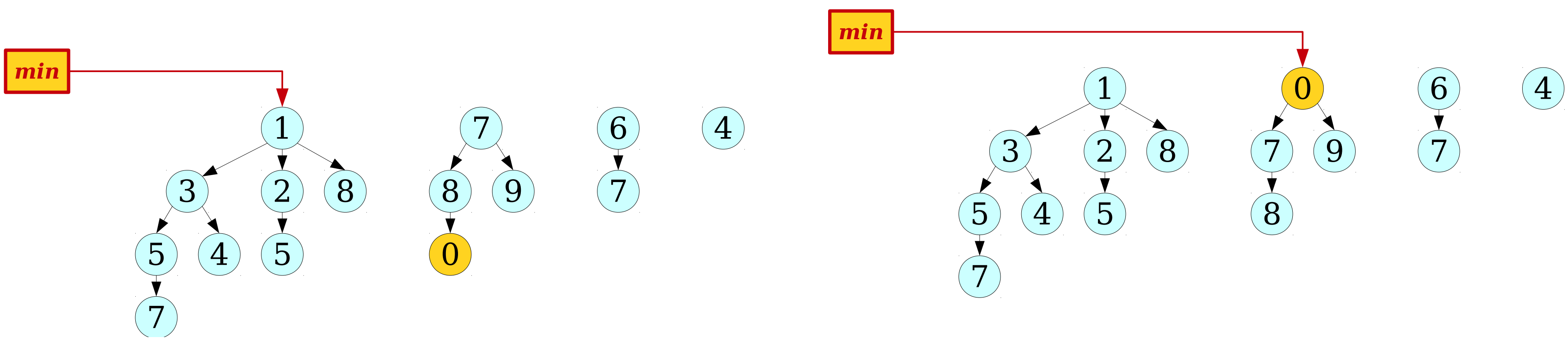
- New data structures developed for fast operations
  - If data range limited to [0,U]: can do bitwise orderings (similar to Radix sort)
  - Van Emde Boas trees
  - Structures using **tries**: we'll discuss later
  - x-fast tries: uses cuckoo hashing together with bitmap tries: find in O(1), insert in O(log(U)), find min O(log(log(U))
  - y-fast tries: x-fast trie on top of forest of red-black trees with all operations O(log(log(U)))

Fibonacci Heaps

- Fredman-Tarjan 1986
  - CLRS Chapter 19
- Binary heaps: Insert: O(log(n)), Merge: O(n), DeleteMin: O(log(n)); DecreaseKey: O(log(n))
- Lazy Binomial heals: Insert: O(1); Merge: O(1); DeleteMin: O(log(n)) (amortized) DecreaseKey: O(log(n))
- Network optimization algorithms require priority queues where keys are decreased much more often than inserts, deletes, pop minima
  - Can we find a data structure to make DecreaseKey O(1) amortized?

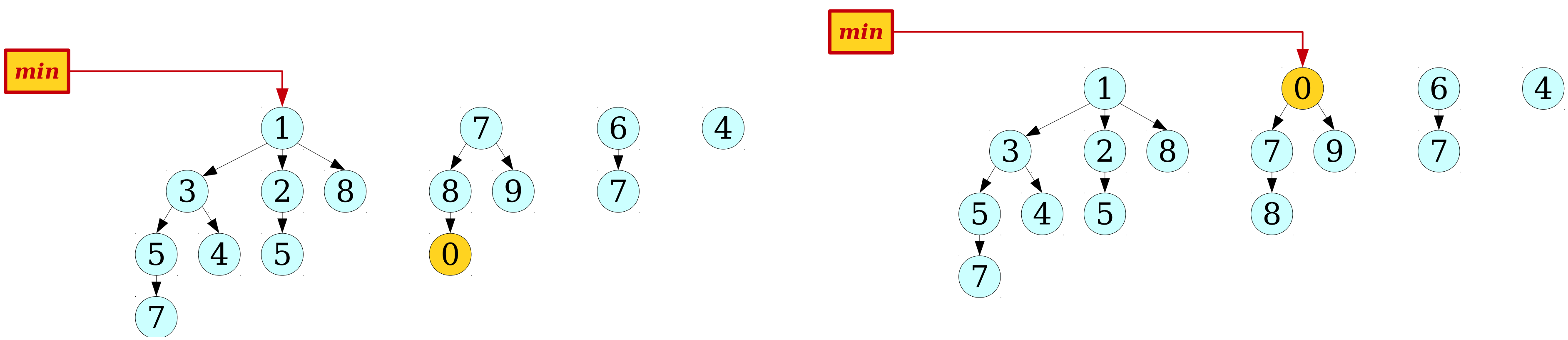
DecreaseKey in Binomial Heaps

- Assume you can find the in the Binomial Heap in O(1) (may require a dictionary or hashmap)
  - Decrease key and up-heap it (O(log(n)))
  - If key is at root of subtree, update minimum pointer.



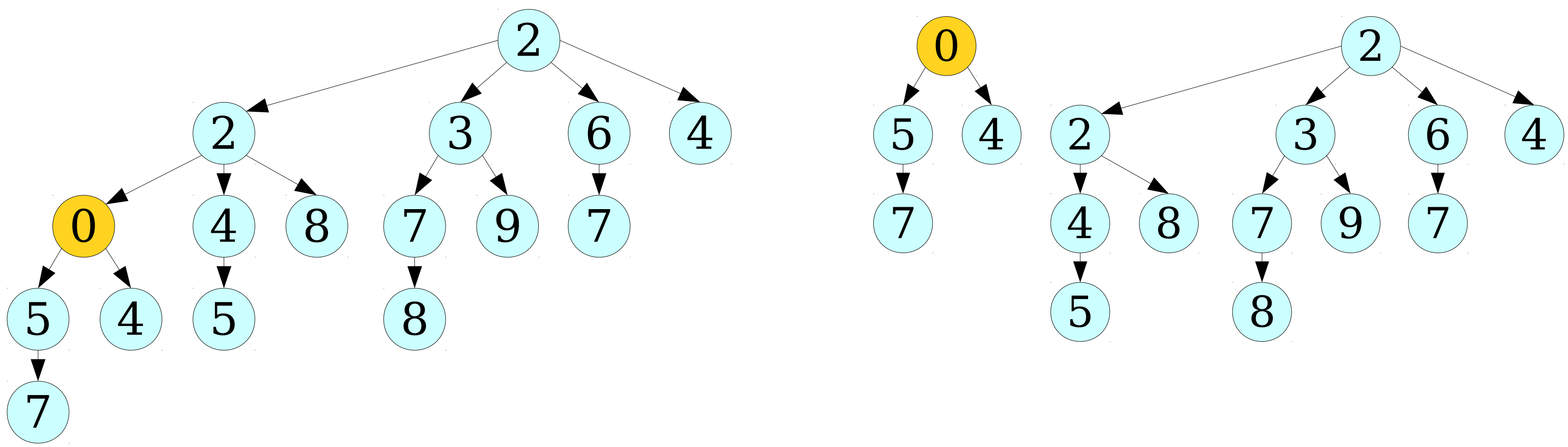
DecreaseKey in Binomial Heaps

- Assume you can find the in the Binomial Heap in O(1) (may require a dictionary or hashmap)
  - Decrease key and up-heap it (O(log(n)))
  - If key is at root of subtree, update minimum pointer.



An Unusual Idea

- Cut subtrees if heap order violated: O(1)
  - Loses binomial tree structure…
  - But we may be able to get by with that

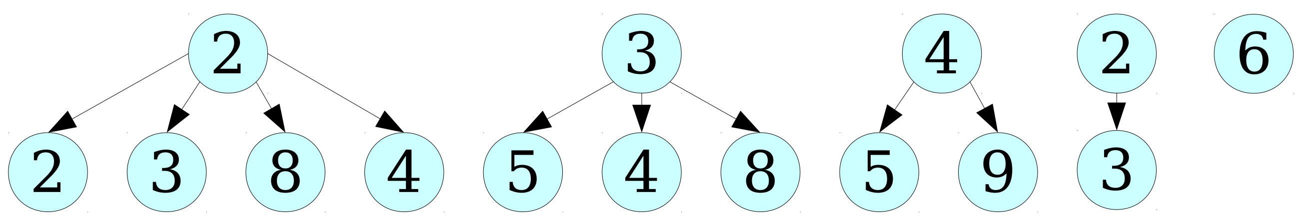


Fibonacci Heaps

- Similar to Binomial Heaps
  - Forest of heap-ordered trees, but trees do not have to be in binomial shape
  - Defer consolidation as in lazy binomial trees
  - DecreaseKey by breaking off the subtree and adding subtree to forest of trees
- Binary heaps: Insert: O(log(n)), Merge: O(n), DeleteMin: O(log(n)); DecreaseKey: O(log(n))
- Lazy Binomial heals: Insert: O(1); Merge: O(1); DeleteMin: O(log(n)) (amortized) DecreaseKey: O(log(n))
- Problem: $\Theta(n)$ number of trees in worst case…need to manage complexity

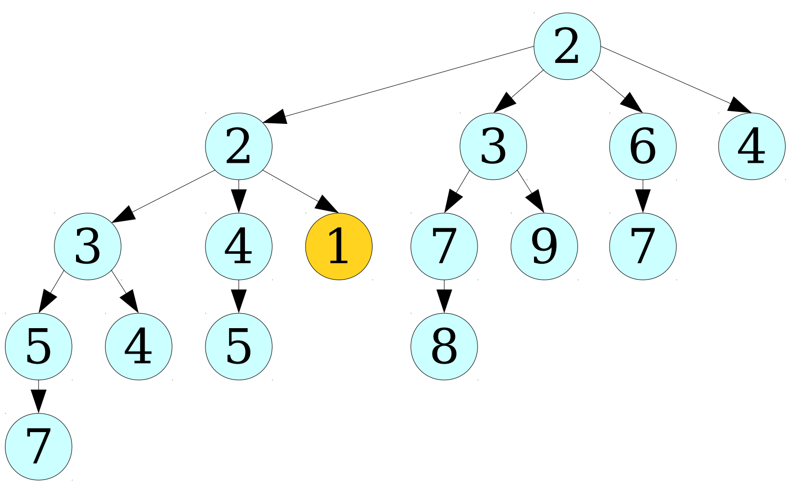
Fibonacci Heaps

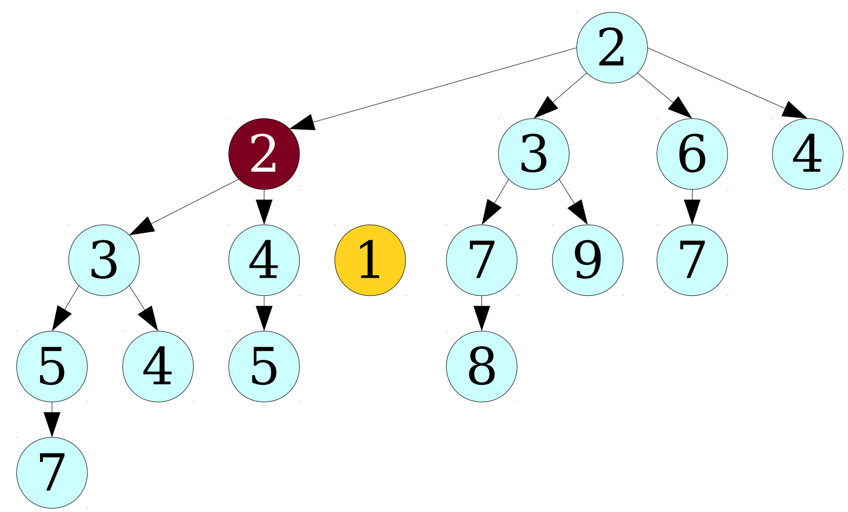
- Notation: Order of a node = number of children
  - In binomial trees with root of order h, there are 2^h nodes
  - If cut trees are no longer binomial, they may have fewer keys
- e.g. number of nodes $\Theta(k^2)$, number of trees $\Theta(k)$
  - Need to avoid this! Want number of nodes exponential in number of trees
  - Must impose some structural constraints



Fibonacci Heaps

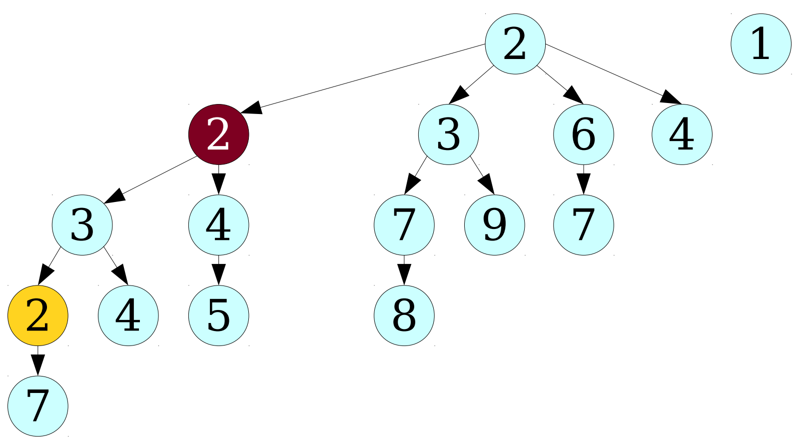
- Structural constraint: limit number of cuts of children to a non-root node
  - At most one cut can be done without restructuring
  - Mark node that has lost one child
  - If a non-root node loses a second child, we cut the node from its parent also
    - May be recursive…

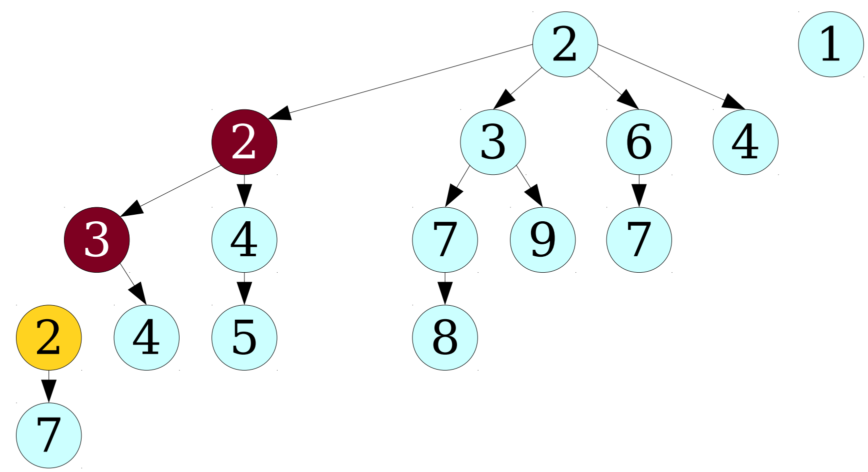




Fibonacci Heaps

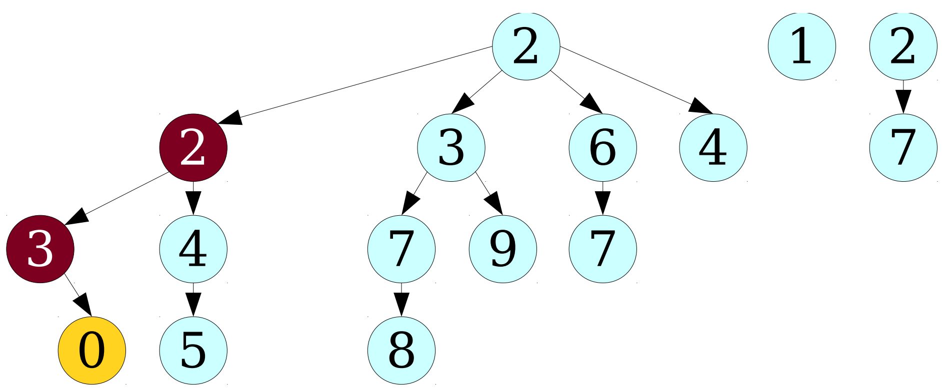
- Structural constraint: limit number of cuts of children to a non-root node
    - At most one cut can be done without restructuring
    - Mark node that has lost one child
    - If a non-root node loses a second child, we cut the node from its parent also
        - May be recursive…

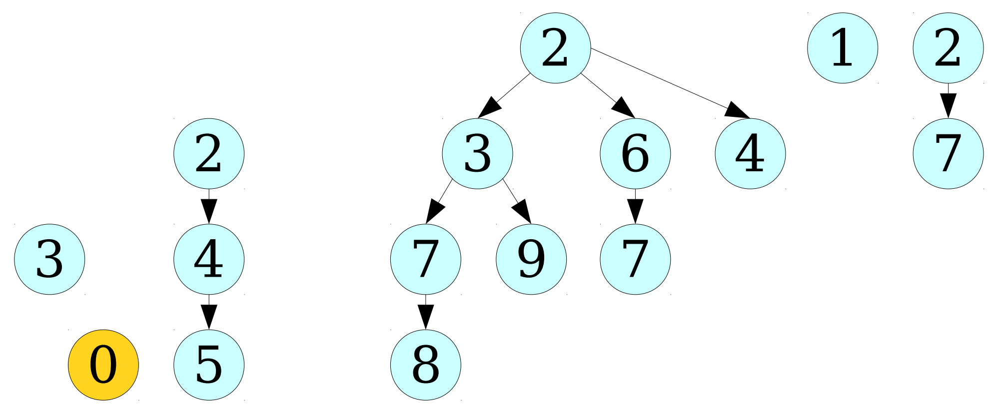




Fibonacci Heaps

- Structural constraint: limit number of cuts of children to a non-root node
  - Mark node that has lost one child
  - If a non-root node loses a second child, we cut the node from its parent also
    - May be recursive…



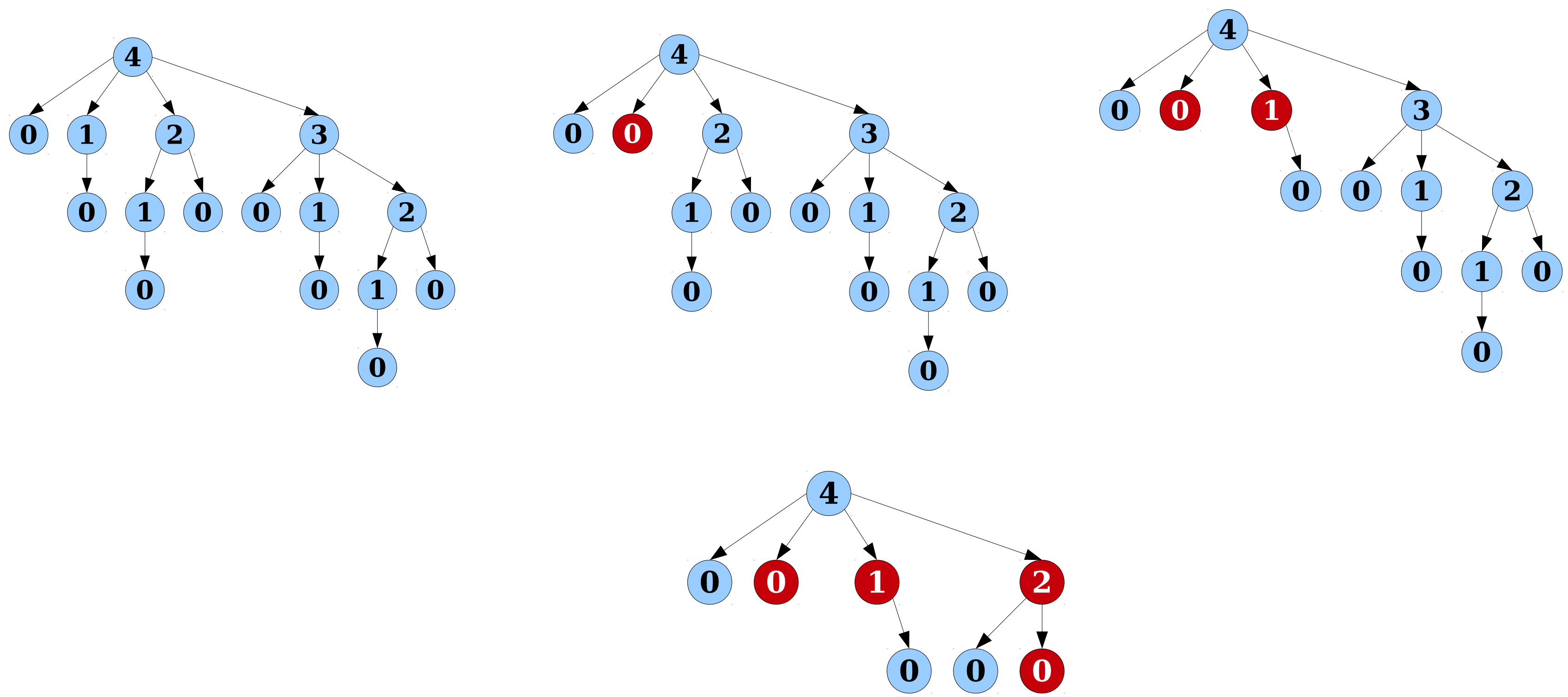


Fibonacci Heaps

- Cut operation: cut node v from parent p
  - Unmark v. Cut v from p
  - If p is not marked and is not the root of a tree, mark it
  - If p was already marked, recursively cut p from its parent
- If we do a few decrease-keys, then the tree won't lose “too many” nodes.
  - If we do many decrease-keys, the information slowly propagates to the root
- DeleteMin: complexity O(h), where h is height of tallest tree
  - In Binomial heaps, $h \in O(\log(n))$
  - What is it now?

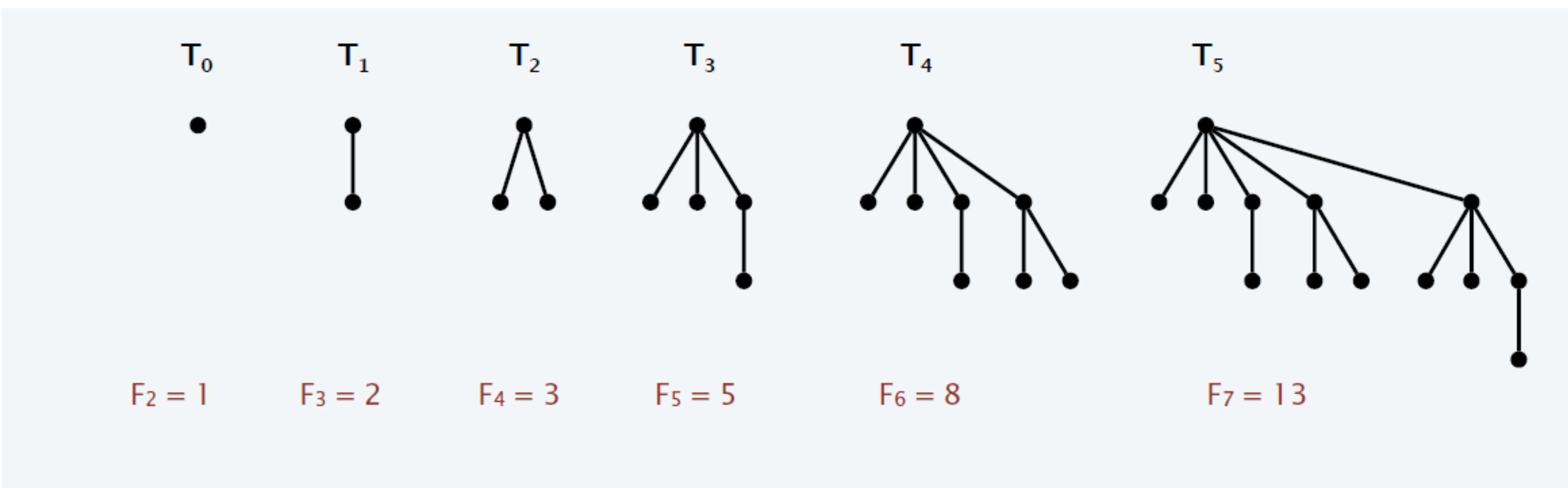
Fibonacci Heaps

- Minimum number of nodes in tree of rank h



Fibonacci Heaps

- Lemma: Number of keys in a tree of order k (root rank k) is F_{k+2}, the k+2 Fibonacci number
  - Implies that number of keys grows exponentially with root rank: $\left(\frac{1+\sqrt{5}}{2}\right)^k$
  - Max rank O(log(n)), height is no larger than rank



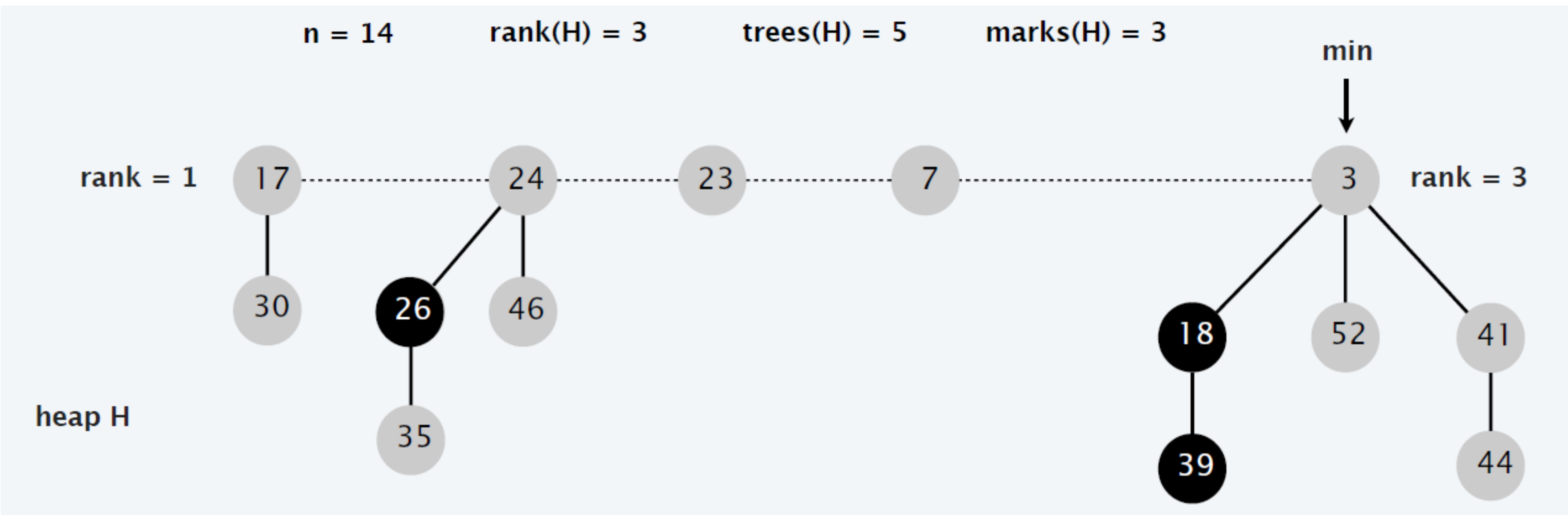
Fibonacci Heaps

- Operations
  - Insert: O(1), just add another tree to heap with insert node, update min
  - Merge: O(1), just link the two sets of trees, update min
  - Find_Min: O(1), just read it
  - Delete_Min: Delete min root, consolidate trees of the same rank. Analysis identical to Binomial Heap, O(log(n)) amortized
  - Decrease_Key: ???

- Amortized analysis: Potential function $\Phi(\mathscr{H})$ = number of trees + 2 * number of marked nodes

Amortized Analysis of DecreaseKey

- Amortized analysis: Potential function $\Phi(\mathscr{H})$ = number of trees + 2 * number of marked nodes

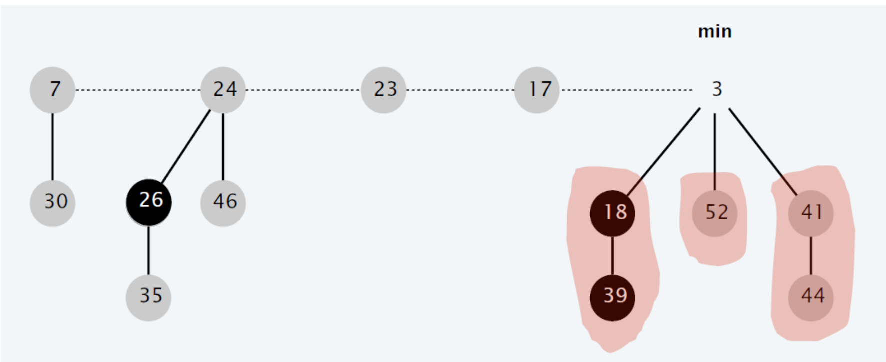


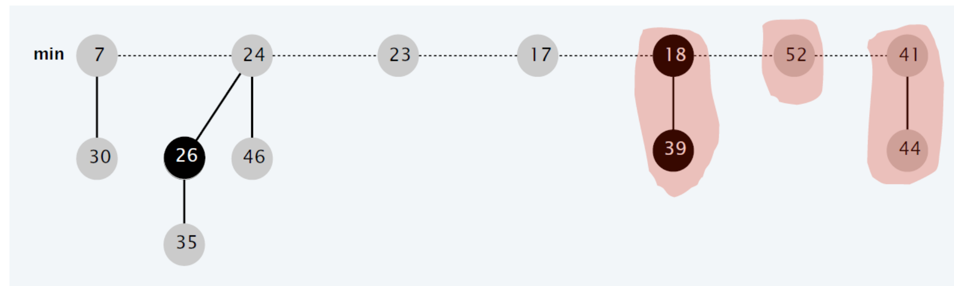
Amortized Analysis of DeleteMin

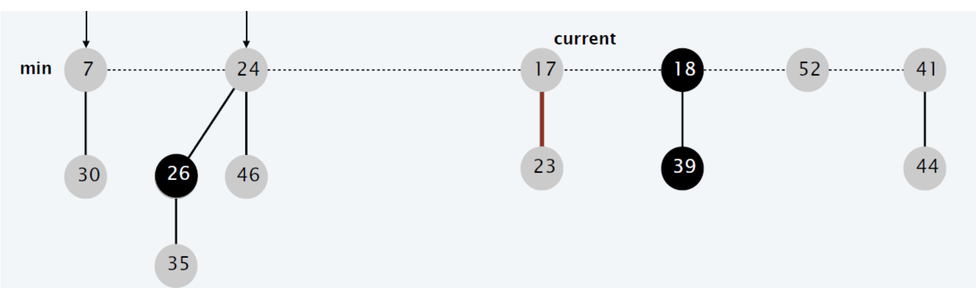
- Amortized analysis: Potential function $\Phi(\mathscr{H})$ = number of trees + 2 * number of marked nodes
  - Insert: increases potential by 1, O(1) work so amortized O(1)
  - Delete_min:
    - Promoting children of min root increases trees by max rank in $\mathscr{H}$: rank($\mathscr{H}$)
    - When heap $\mathscr{H}$ has k trees, consolidate is $\Theta(k)$+ rank($\mathscr{H}$)
    - Number of trees after consolidation: less than or equal to rank($\mathscr{H}'$) + 1
      - No repeated ranks
    - May lose some marks
    - Amortized cost: O(rank($\mathscr{H}$)) + O(rank($\mathscr{H}'$)), the latter of which is O(log(n))
    - Claim: O(rank($\mathscr{H}$)) is O(log(n)) also!

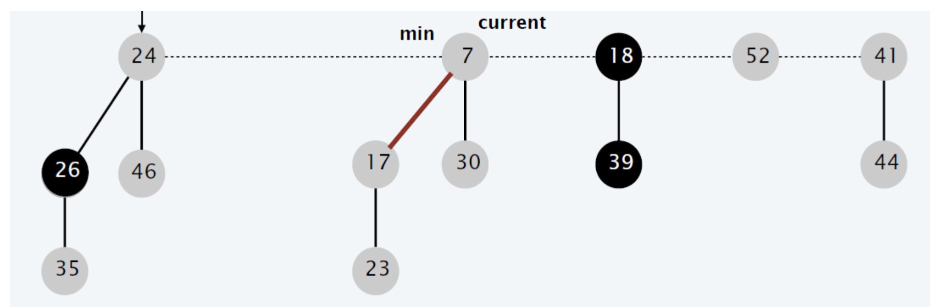
Amortized Analysis of DeleteMin

- DeleteMin





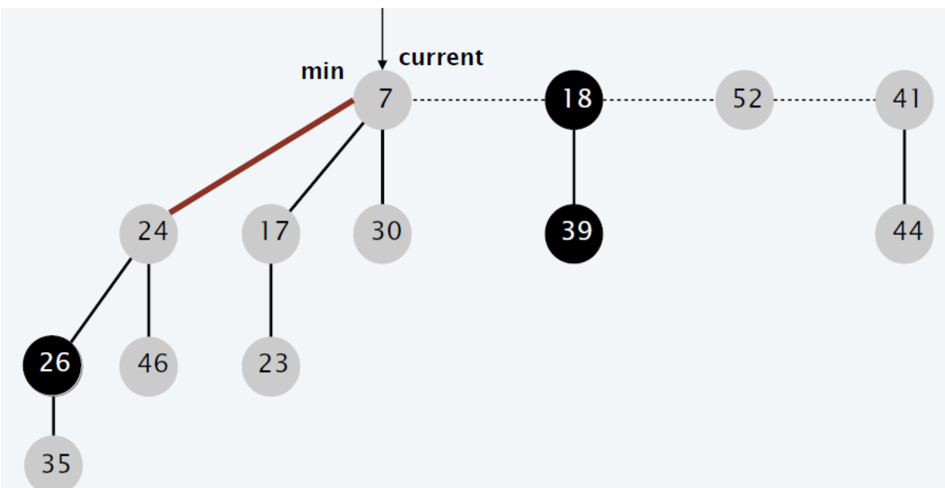


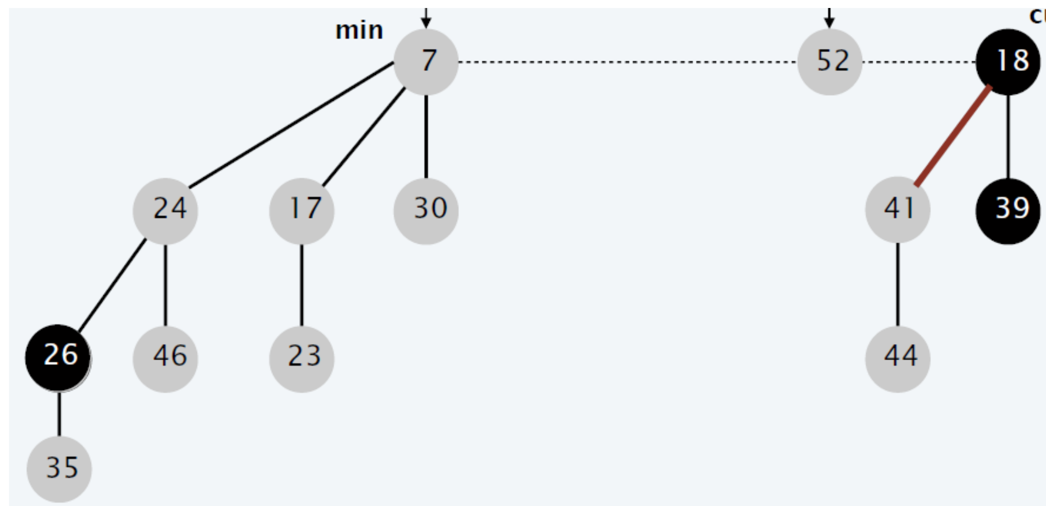


Amortized Analysis of DeleteMin

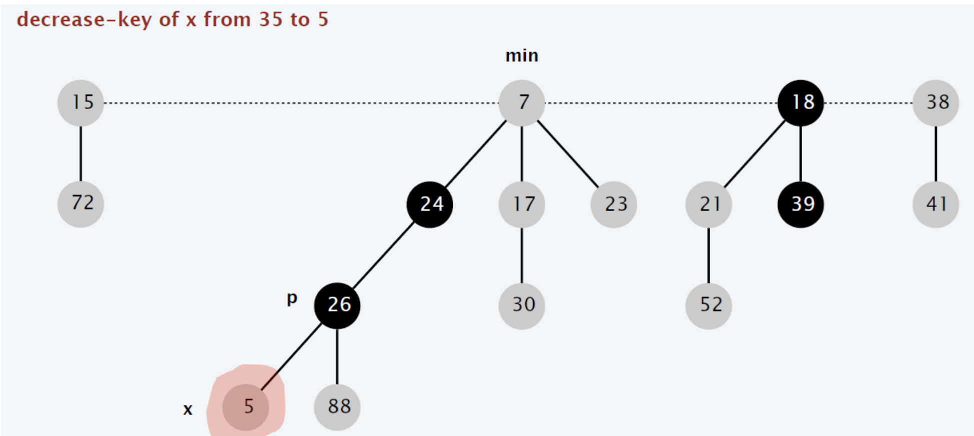
- DeleteMin

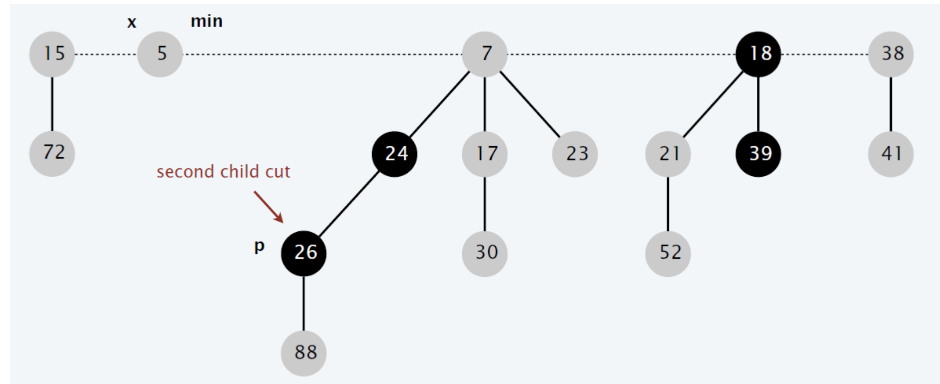
Delete mark

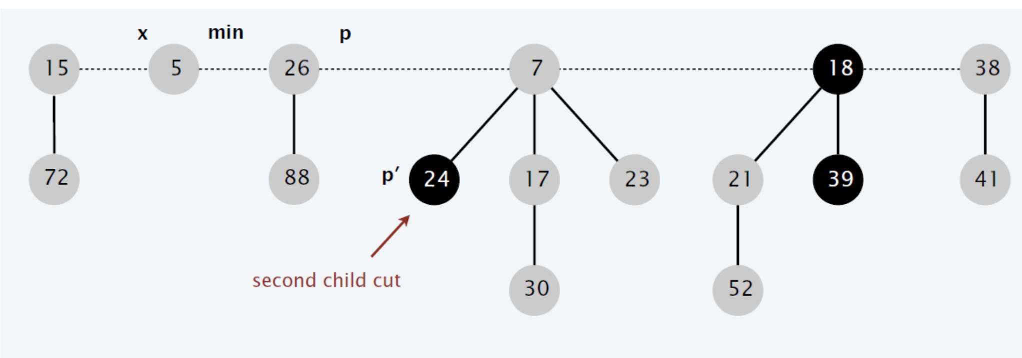


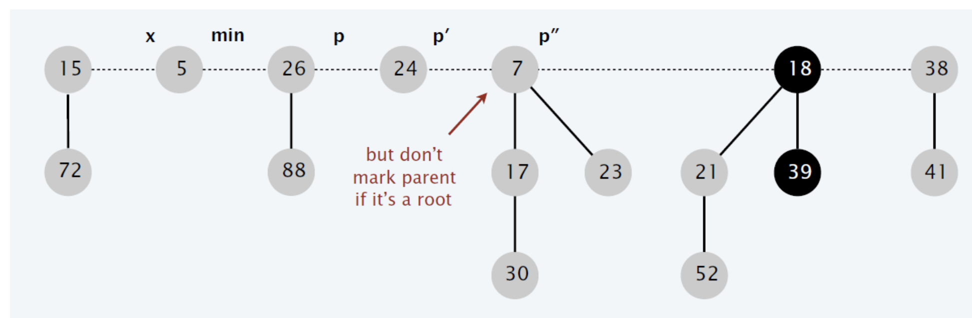


Amortized Analysis of DecreaseKey









Amortized Analysis of DecreaseKey

- DecreaseKey: Actual cost O(C) where C number of cuts
  - Number of trees increases by number of cuts: potential increase
  - Number of marked nodes decreases by number of cuts - 1: May mark one new node, remove marks from others cut

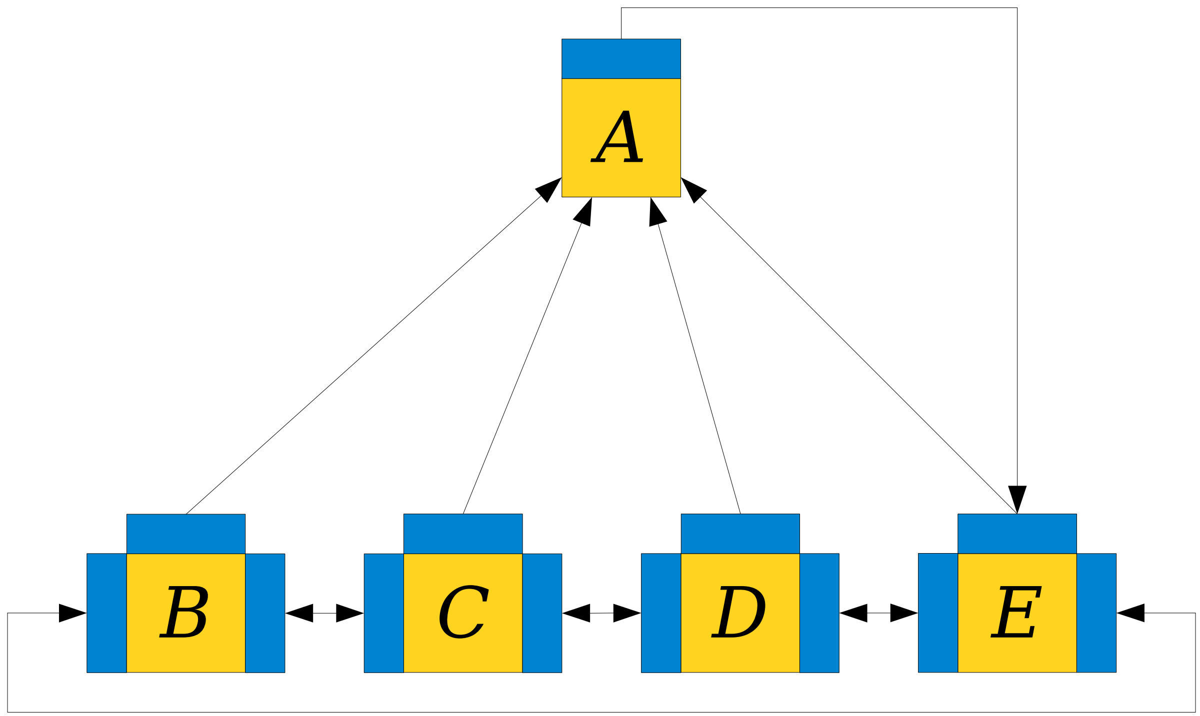
- Amortized cost: $\Theta(C) + \Delta\Phi = \Theta(C) + (2 - 2C + C) = \Theta(1)$

- Note: rank($\mathscr{H}$) does not increase! It can only decrease, and may only increase during DeleteMin.

- So, what's the problem?

Fibonacci Heaps: Implementation

- In order to do cuts efficiently O(1), must have very complex data structures
- Children in doubly-linked circular lists
  - Point to parent
  - Parent points to one child in list
- Awful linked lists!
  - But now, can do in O(1):
    - Cut node from parent
    - Add another child to node



Fibonacci Heaps: Implementation

- Size of Fibonacci Heap node: each node in a Fibonacci heap stores
  - A pointer to its parent.
  - A pointer to the next sibling.
  - A pointer to the previous sibling. A pointer to an arbitrary child.
  - A bit for whether it's marked.
  - Its order.
  - Its key.
  - Its element
- In practice, Fibonacci heaps are slower because of overheads
  - Good for theoretical analysis
  - Good research topic: Simpler heaps with O(1) DecreaseKey