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

Binomial Heaps

- Forest of binomial trees (Note: not binary!)
  - Each in heap order
  - Each of a different height
- Forest: a collection of trees
- A binomial tree B_k of height k consists two B_{k-1} binomial trees
- The root of one B_{k-1} tree is the child of the root of the other B_{k-1} tree

Foundations: Binary Arithmetic

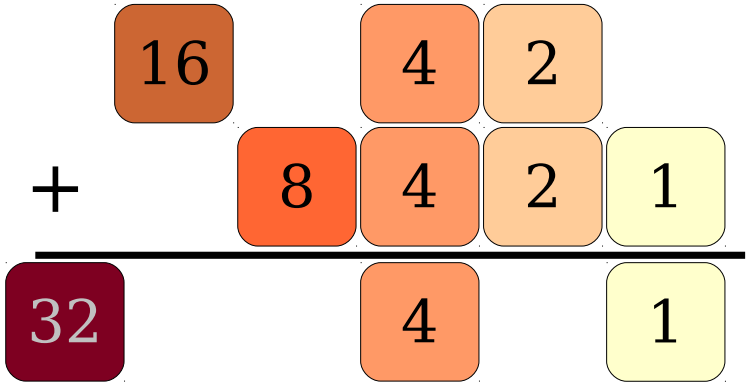
- Given the binary representations of two numbers n and m, we can add those numbers in time Θ(max{log m, log n})

	1	0	1	1	0
+		1	1	1	1

-

A Different View

- Represent n and m as a collection of "packets" whose sizes are powers of two
- Adding together n and m can then be thought of as combining the packets together, eliminating duplicates



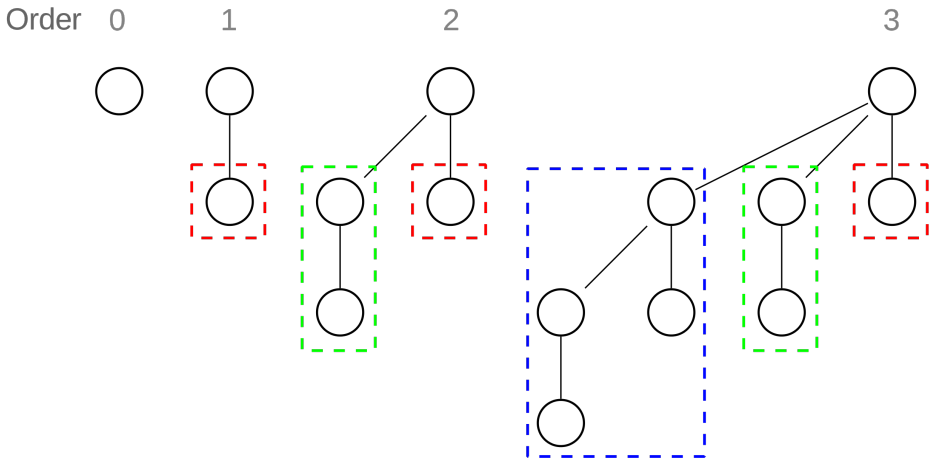
Idea for Priority Queues for Easy Merge

- Store elements in the priority queue in “packets” whose sizes are powers of two
- Store packets in ascending size order
- We'll choose a representation of a packet so that two packets of the same size can easily be fused together.
- Adding together n and m can then be thought of as combining the packets together, eliminating duplicates

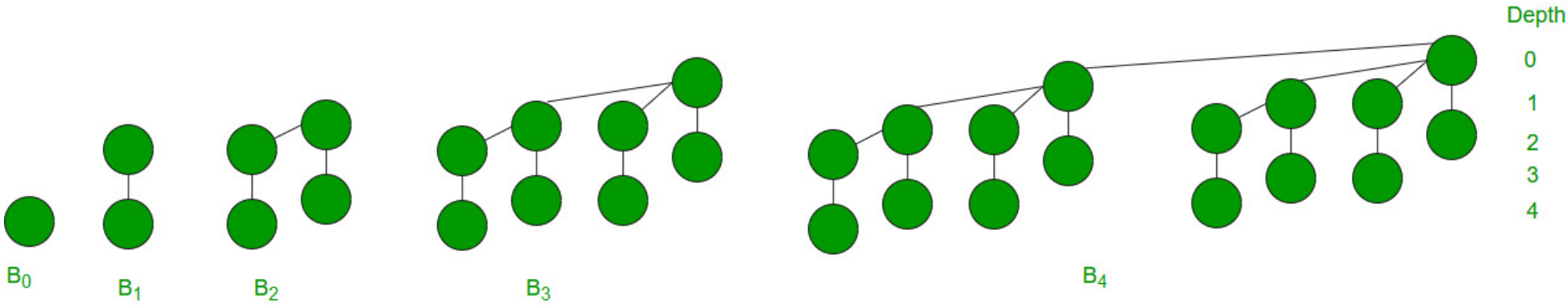
Right Packets: Binomial Trees

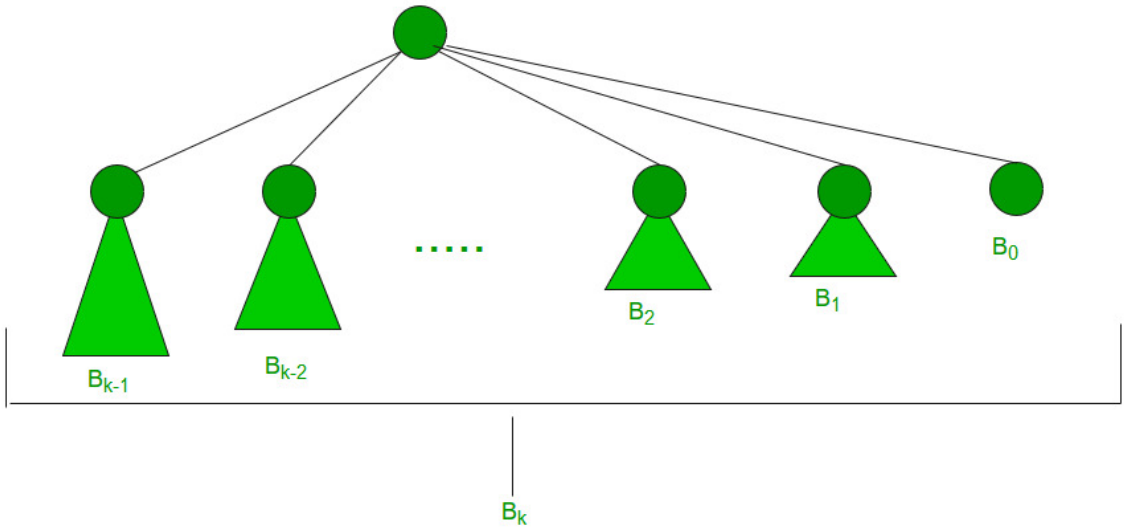
- A binomial tree of order k is a type of tree recursively defined as follows:
  - A binomial tree of order k is a single node whose children are binomial trees of order 0, 1, 2, ..., k – 1

- Here are the first few binomial trees:



Binomial Trees





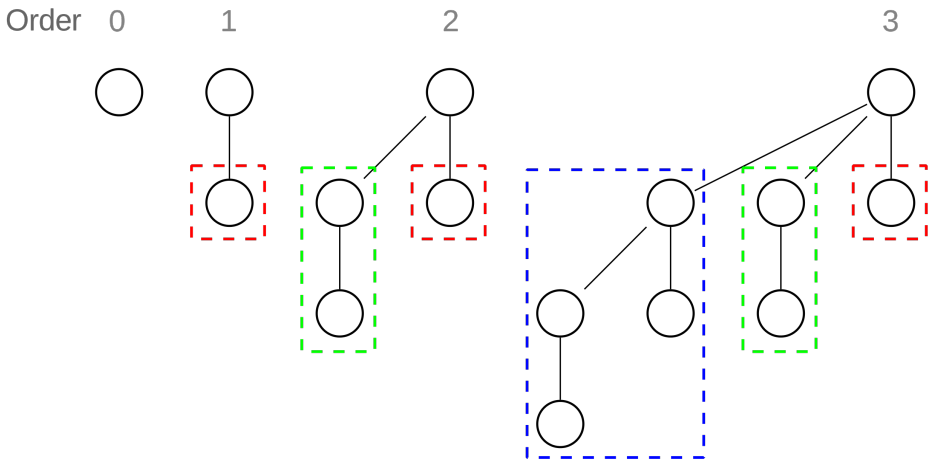
Binomial Trees

- Theorem: A binomial tree of order k has 2^k nodes
  - Proof by induction:

$$\text{Nodes in } B_k = \text{root} + \sum_{j=0}^{k-1} \#(B_j) = 1 + \sum_{j=0}^{k-1} 2^j = 1 + \frac{2^k - 1}{2 - 1} = 2^k$$

Number of nodes in B_k at level d is $\binom{k}{d}$

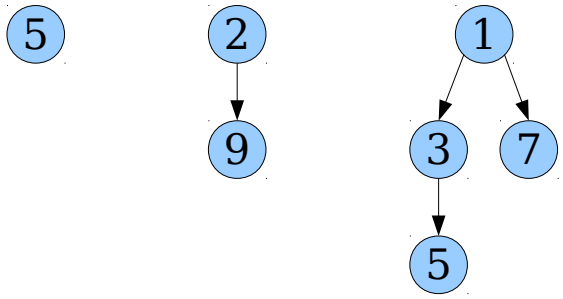
B_k has height k



Heap-Ordered Binomial Trees

- Heap-ordered binomial trees: Trees satisfy heap property
  - Key of parent is less than or equal to keys of children

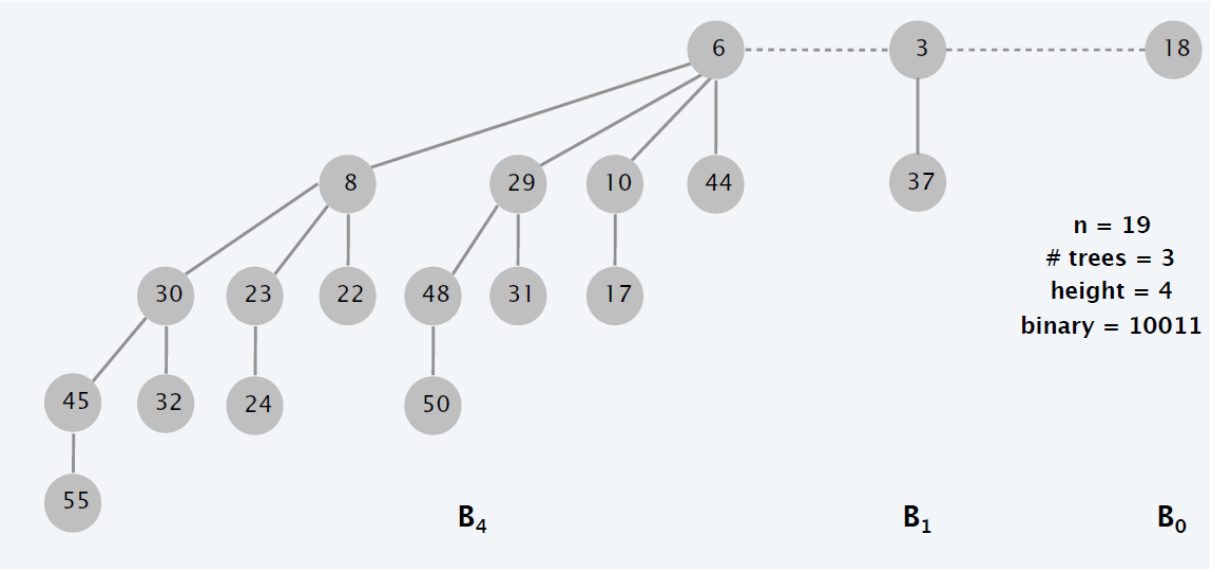
- **Binomial heap**: A collection of heap-ordered binomial trees
  - Pointer to the tree with the min value
  - No two binomial trees are of the same order
    - Boolean representation



- Children of any node are arranged in a doubly linked list (easy to link/unlink). Useful in delete…

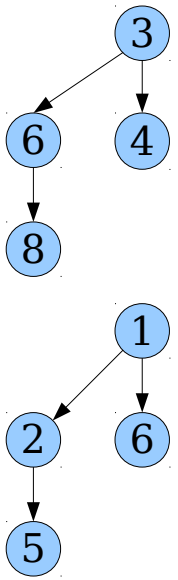
Binomial Heaps

- Properties of Binomial heap with n nodes
  - The node containing the min element is a root of $B_k, B_{k-1}, \ldots, B_1, B_0$
  - It contains the binomial tree B_k iff $b_k = 1$, where $b_k b_{k-1} \cdots b_0$ is binary representation of n.
  - It has no more than
    ⌊log2 n⌋ + 1 binomial trees.
  - Its max height is less than
    ⌊log2 n⌋



Binomial Heap Basic Operation

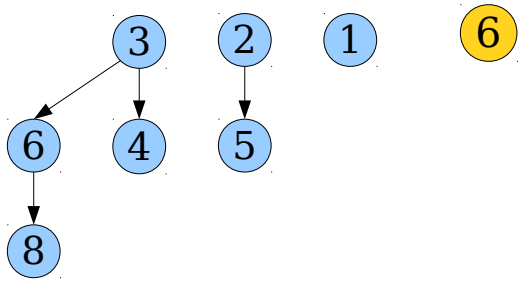
- Linking two binomial trees of the same order
  - If you have two binomial trees with the same order k, (e.g. B_1, B_1'), you can merge them (like addition) into binomial tree of order k+1
  - Find one with the smaller root, hook up the other one as subtree as leftmost child
  - Can do recursively (boolean addition) until there is at most only one tree of each order



???

Binomial Heap: Merge 2 Heaps

- Heap 1: Trees $B_0^1, B_1^1, \ldots, B_k^1$
- Heap 2: Trees $B_0^2, B_1^2, \ldots, B_m^2$

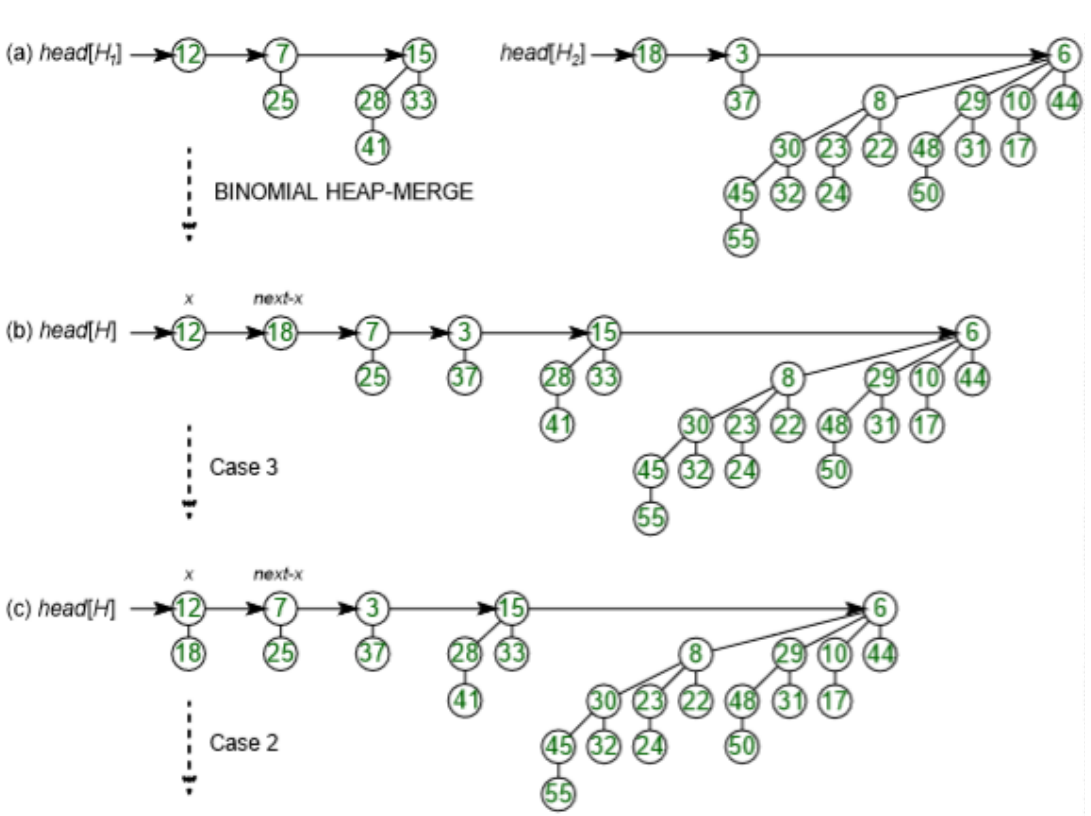


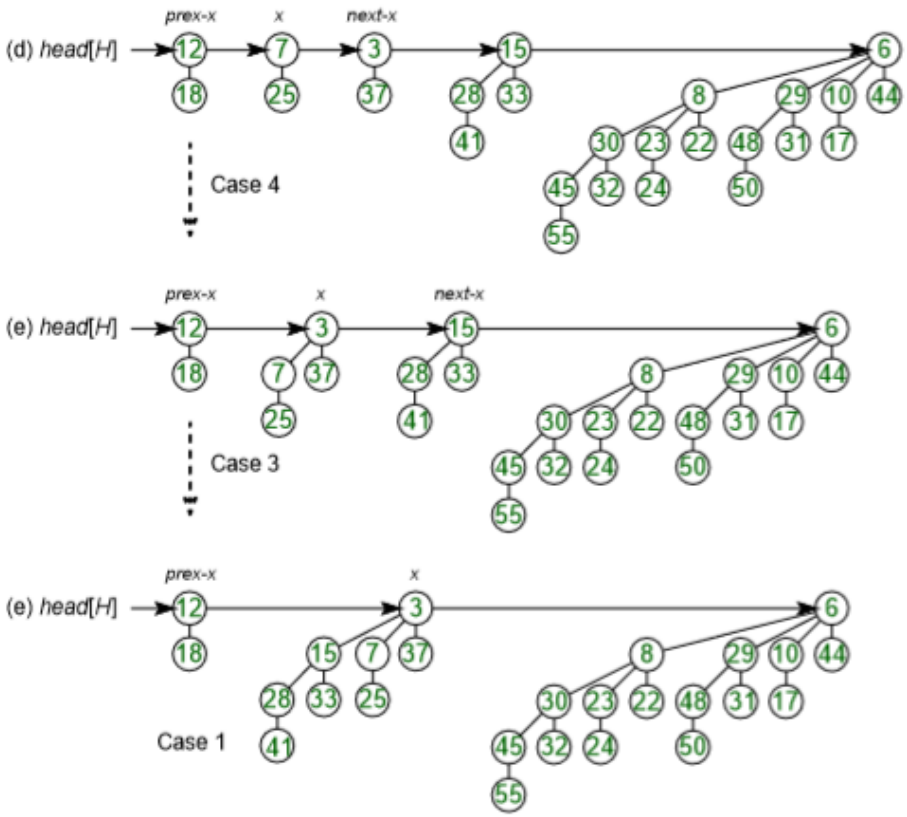
- Step 1. Add all the trees to the same list
  - May be duplicates in size
- Step 2: Starting from the smallest size, 0, if there are two or three of them, merge two of them into a tree of a higher order
  - Leaves at most one of that size behind
  - Repeats for increasing order

???

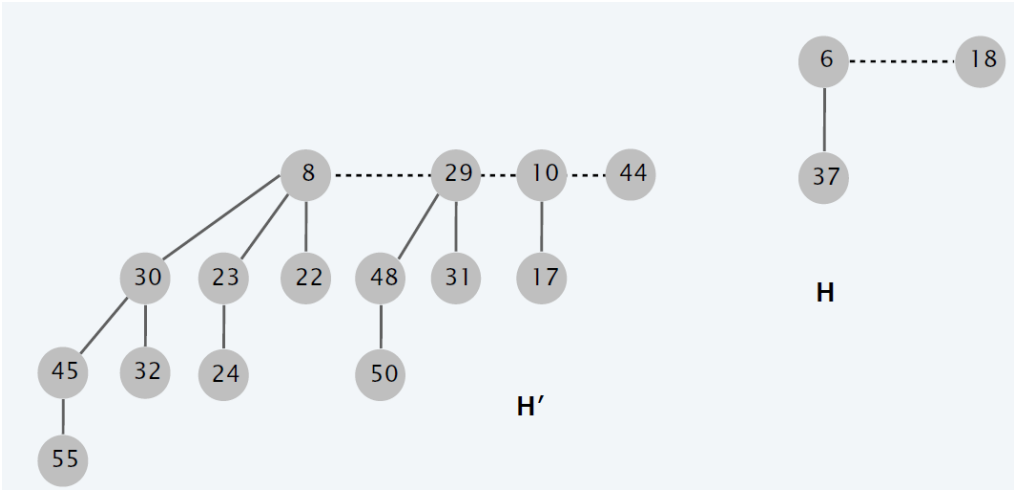
	1	0	1	1	0
+		1	1	1	1

Binomial Heap: Merge 2 Heaps



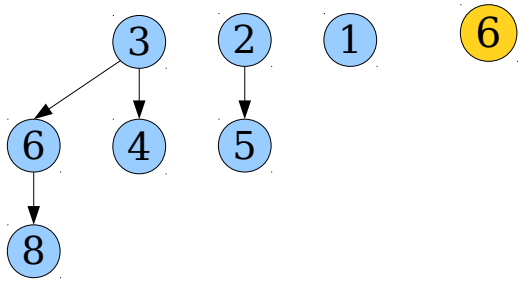


Binomial Heap: Insert



Binomial Heap: Insert

- Insert: create single element heap, merge it with existing binomial heap



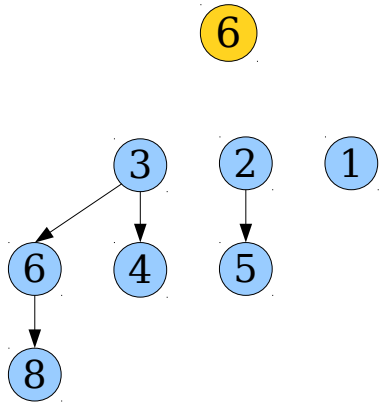
- Step 1. Add all the trees to the same list
  - May be duplicates in size
- Step 2: Starting from the smallest size, 0, if there are two or three of them, merge two of them into a tree of a higher order
  - Leaves at most one of that size behind
  - Repeats for increasing order

???

	1	0	1	1	0
+		1	1	1	1

Binomial Heap: Insert Complexity

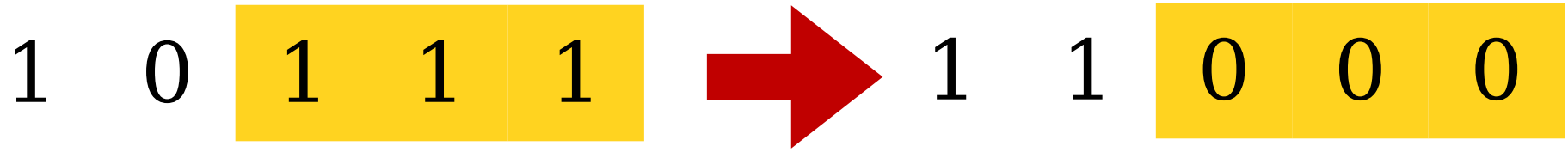
- Lazy insert: just leave the single nodes alone, allow duplicates
  - Merge later when you do something else (e.g. delete minimum)
  - Insert is O(1)! Just add to linked list, update pointer to minimum



- Full insert: O(k), where k is number of trees in Binomial heap
  - Number of trees is O(log(n)), where n is number of entries in binomial heap
  - There are at least 2^k nodes in Binomial heap of order k (size of last tree!

Binomial Heap: Amortized Insert Complexity

- Illustration: Suppose we have integer n (represented in binary) and we want to add 1 to it (a binary counter).
  - What is the amortized complexity of this addition?
  - Algorithm: find the longest string of all 1's on the right of the binary expansion, flip all those bits to 0, flip the preceding 0 bit to 1.
  - Cost depends on specific value of n…



Binomial Heap: Amortized Insert Complexity

- Claim: amortized analysis of incrementing binary n is O(1)
  - Idea: Potential function $\Phi(n)$ is number of 1's in binary expansion

- Cost of incrementing n:
  - Actual cost: number of starting 1's (j) + 1
  - Change in potential: $\Phi(n+1) - \Phi(n)$ = 1 - j

$\Phi = 0$ 0 0 0 0 0

$\Phi = 1$ 0 0 0 0 1

- Amortized cost = j + 1 + $\Phi(n+1) - \Phi(n)$ = 2 —> O(1)

- Catch: cannot decrement! Only increment!

Binomial Heap: Amortized Insert Complexity

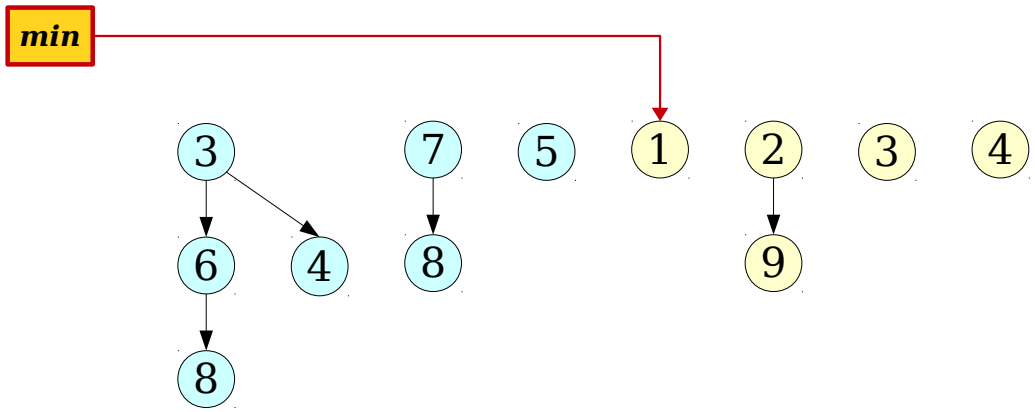
- Analysis extends to Binomial Heaps
  - As long as we are doing inserts in a row, amortized cost is O(n)

- Compare with BuildHeap for binary heaps: If we are given array of n numbers, can build to binary heap in $\Theta(n)$ (must know n)

- For Binomial heaps, even if we don't know n, cost of inserting n elements into Binomial heal is $\Theta(n)$

Binomial Heap: Lazy Merge

- Binomial heap stores pointer to root of tree with smallest element
- Don’t merge trees as long as there is no delete minimum
- Inserts are O(1), heap merges are O(1)
  - But, number of trees in heap may be O(n)



Binomial Heap: Delete Minimum

- Minimum is root of one of the Binomial Trees in the heap
  - We know which one
  - Delete it, and form another Binomial heap from the children
  - Merge the two Binomial heaps

- If we did not do lazy inserts: Complexity: O(log(n))
  - Maximum number of binomial trees is log(n)

- With lazy inserts, number of trees can be $\Theta(n)$! How would this work?
  - Amortize! Expensive operation is deleteMin
  - But, to create trees, we must do cheap operations: insert, merge

Amortized Analysis of Delete Minimum

- Potential method: $\Phi(\mathscr{H})$ is number of trees in Binomial heap $\mathscr{H}$
- **Lazy insertion**: adds a tree. Operations 1, change in potential +1 —> amortized cost O(1)
- **Join** two binomial heaps, with potentials $\Phi(\mathscr{H}_1)$, $\Phi(\mathscr{H}_2)$:
  - New heap $\mathscr{H}$ has potential $\Phi(\mathscr{H}_1) + \Phi(\mathscr{H}_2)$, and heaps $\mathscr{H}_1$, $\mathscr{H}_2$ have 0 potential
  - Net change in potential is 0: amortized cost is O(1)

Amortized Analysis of Delete Minimum

- Potential method: $\Phi(\mathscr{H})$ is number of trees in Binomial heap $\mathscr{H}$
- Delete Min: Find minimum element using pointer (O(1)), create new binomial trees from children of minimum element (O(log(n)) max children), and coalesce
  - If T current trees in $\mathscr{H}$, this is $\Theta(T)$
- If there are T current trees in $\mathscr{H}$, current potential $\Phi(\mathscr{H})$ is T
- After coalescing, potential $\Phi(\mathscr{H})$ = log(n)
- Amortized cost: O(log(n)) + $\Theta(T)$ -O(1) T + O(log(n)) = O(log(n)) !!!

Overall Analysis of Lazy Binomial Heaps

- The amortized costs of operations:
  - Insert: O(1)
  - Merge: O(1)
  - Find Min: O(1)
  - Delete Min: O(log(n))

- Any series of m inserts mixed with j extract-mins will take time
  O(m + j log m)

- What operations don't work well yet? Decrease key of a node, othrs
  - Must do upHeap in that binomial tree: describe how in next data structure

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)!

Hashing

- 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 Table

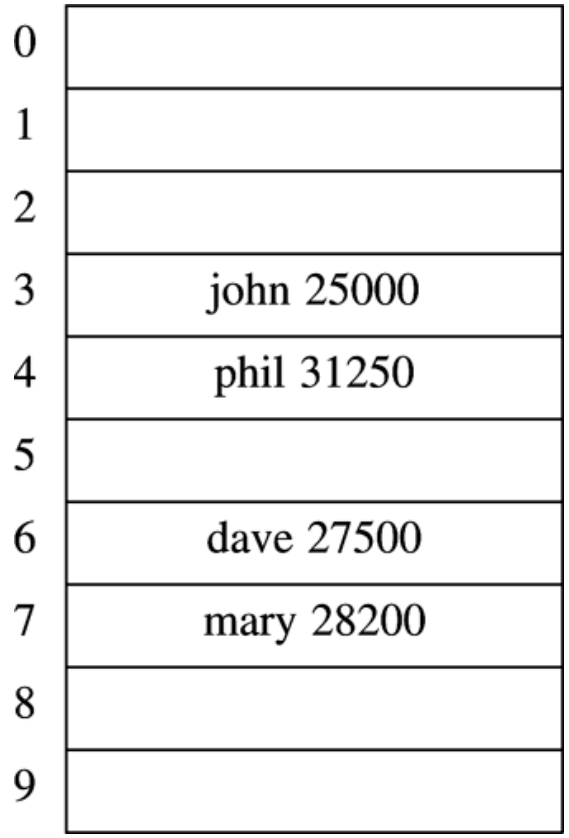
- Hash table is a data structure that support
  - Finds, insertions, deletions (deletions may be unnecessary in some applications)

- The implementation of hash tables is called hashing
  - A technique which allows the executions of above operations in constant average time

- Tree operations that requires any ordering information among elements are not supported
  - findMin and findMax
  - Successor and predecessor
  - Report data within a given range
  - List out the data in order

General Idea

- The ideal hash table data structure is an array of some fixed size, containing the items
- A search is performed based on key
- Each key is mapped into some position in the range 0 to TableSize-1
- The mapping is called hash function

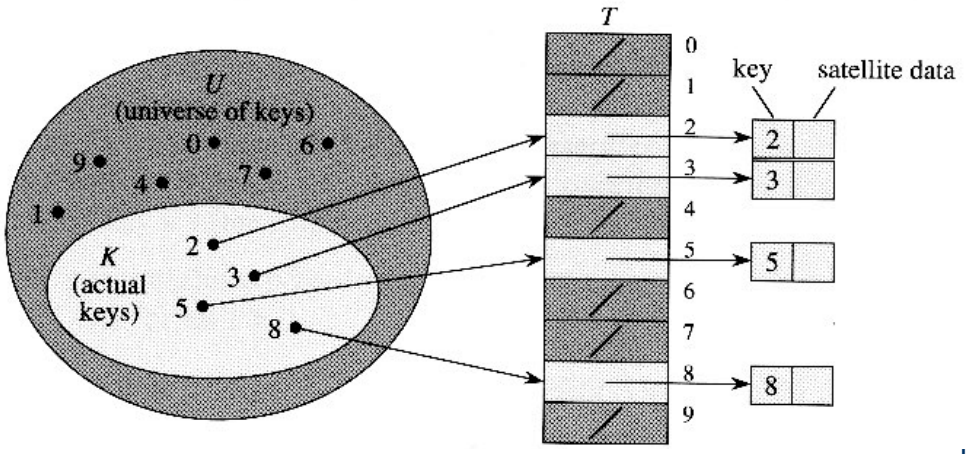


Hash Function

- 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

Unrealistic solution

- Huge hash tables
  - Each position (slot) corresponds to a key in the universe of keys
  - T[k] corresponds to an element with key k
  - If the set contains no element with key k, then T[k]=NULL
  - WASTE!
  - e.g BU IDs



Applications

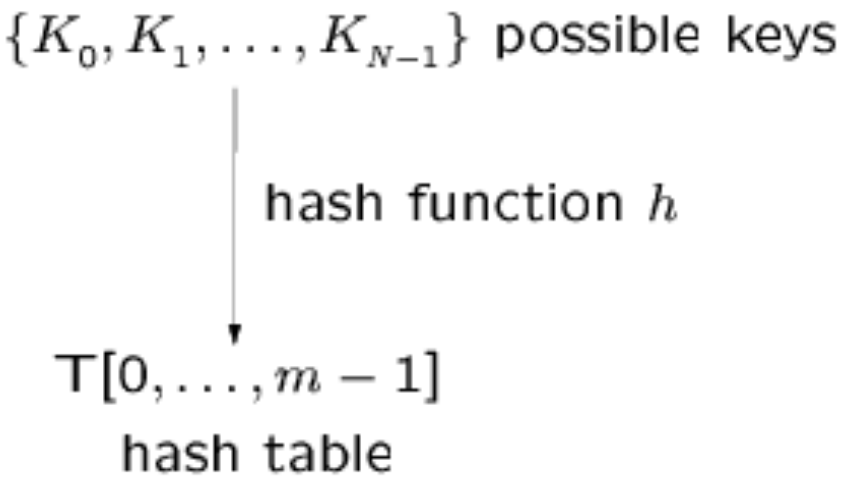
- Compilers use hash tables (symbol table) to keep track of declared variables.

- On-line spell checkers. After prehashing the entire dictionary, one can check each word in constant time and print out the misspelled word in order of their appearance in the document.

- Useful in applications when the input keys come in sorted order. This is a bad case for binary search tree. B+-trees are harder to implement and they are not necessarily more efficient.

Hash Function

- Simple hash
  - h(Key) = Key mod TableSize, which is typically a prime number
  - Assumes integer keys
- For random keys, h() distributes keys evenly over table
- h(Key) is the hash value of key
- Element of key k is stored in slot h(k): T[h(k)]



Resolving Collisions

- Problem: collision
  - two keys may hash to the same slot
  - can we ensure that any two distinct keys get different cells?
    - No, if N>m, where m is the size of the hash table
- Task 1: Design a good hash function
  - fast to compute and
  - can minimize the number of collisions
- Task 2: Design a method to resolve the collisions when they occur

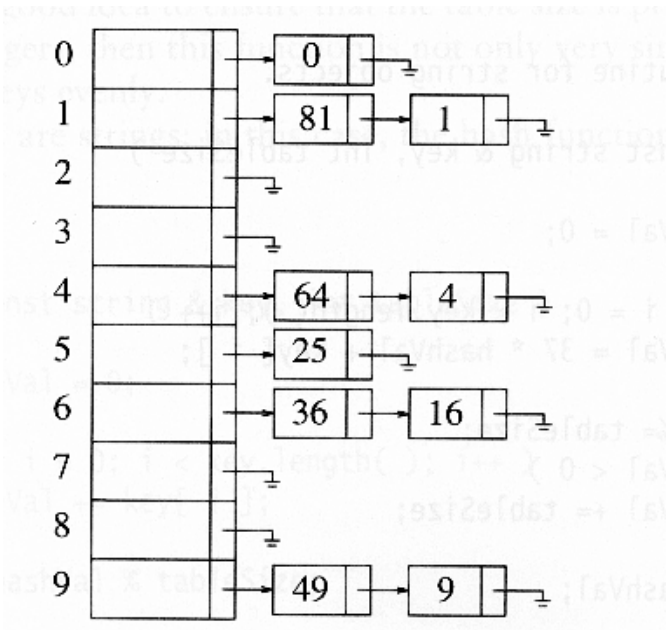
Design Hash Function

- A simple and reasonable strategy: h(k) = k mod m
  - e.g. m=12, k=100, h(k)=4
  - Requires only a single division operation (quite fast)
- Certain values of m should be avoided
  - e.g. if m=2^p, then h(k) is just the p lowest-order bits of k; does not depend on all the bits
  - if the keys are decimal numbers, should not set m to be a power of 10
- It's a good practice to set the table size m to be a prime number
- Good values for m: primes not too close to exact powers of 2
  - e.g. the hash table is to hold 2000 numbers, and we don't mind an average of 3 numbers being hashed to the same entry: choose m=701
- For strings, map keys to ascii integers…

Collision Resolution

- Chaining: Like ‘equivalent classes’ or clock numbers in math
  - Instead of a hash table, we use a table of linked list
  - keep a linked list of keys that hash to the same value
  - Problem: can get to be O(n) worst case

- Typical approach: if storing M keys, make hash number the next largest prime number



Collision Resolution - 2

- Open addressing
  - If slot is busy, design sequence of other slots to be searched
  - probe alternative cell $h_1(K), h_2(K)$, …,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)

New Data Structure: Fibonacci Heaps

- Fredman-Tarjan 1986
  - CLRS Chapter 19