

,

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

Announcements

Quiz 2 posted today by 5 pm

Sw hw in progress

Monday . . .

Hw 2 written: tomorrow ✓



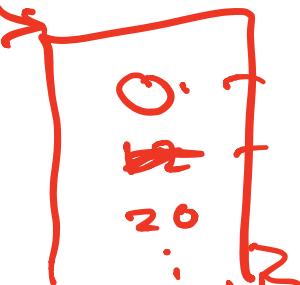
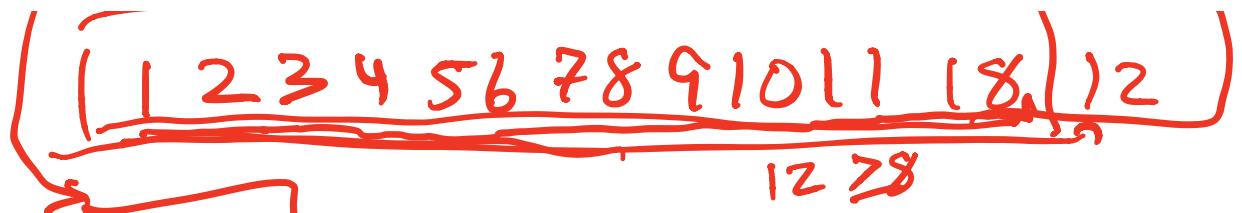
$r_{\text{min}} \rightarrow 0 \rightarrow 12346\sqrt{8}10.$

- 8 - 123467810

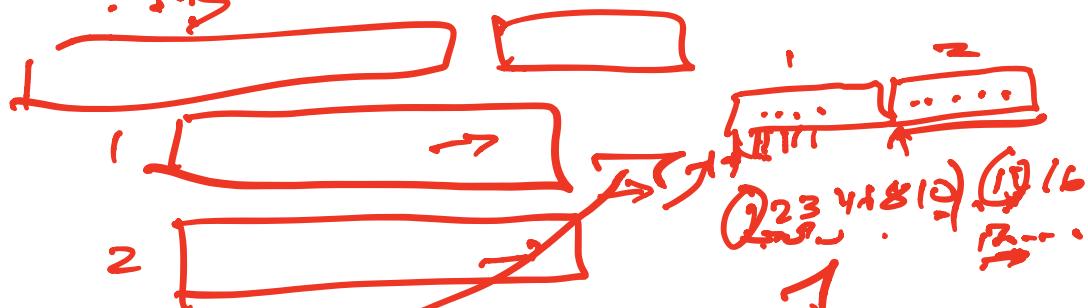
- 16 -

- 24 -

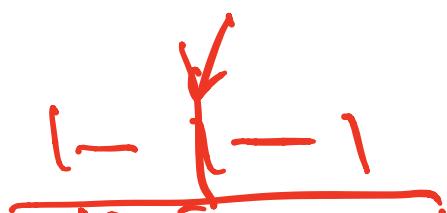
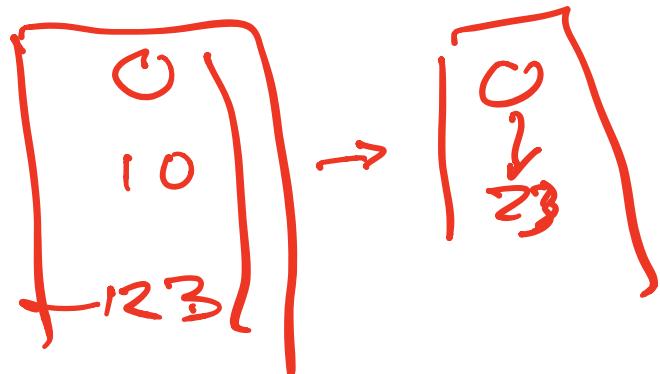
Red line: $y = \frac{1}{2}x^2 + 1$



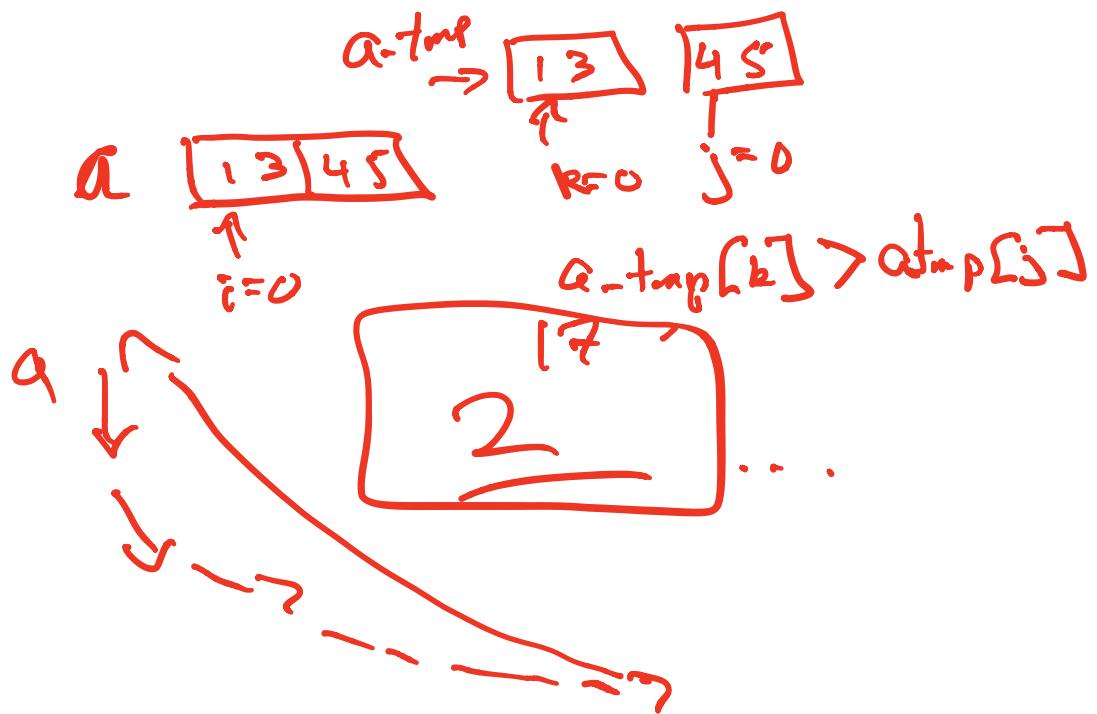
Collapse.



merge



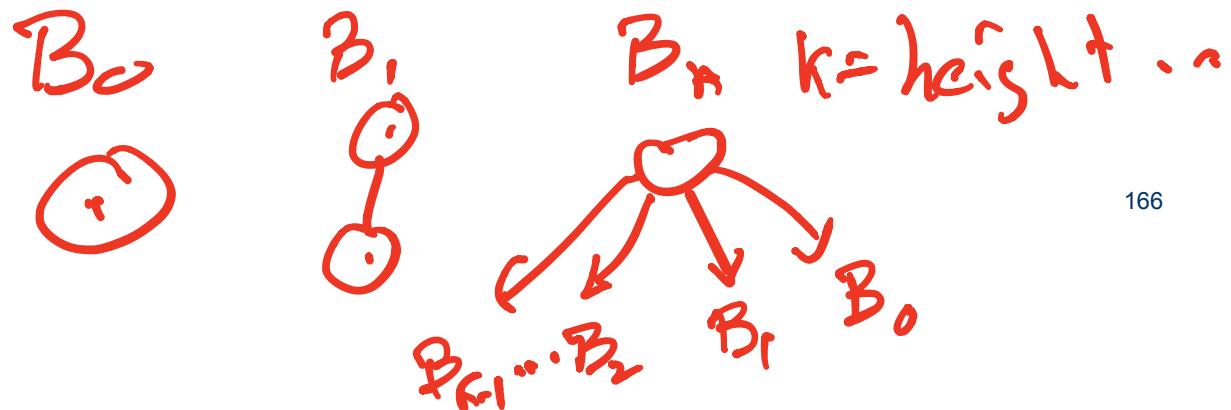
$a[i], a_{\text{temp}}[i]$

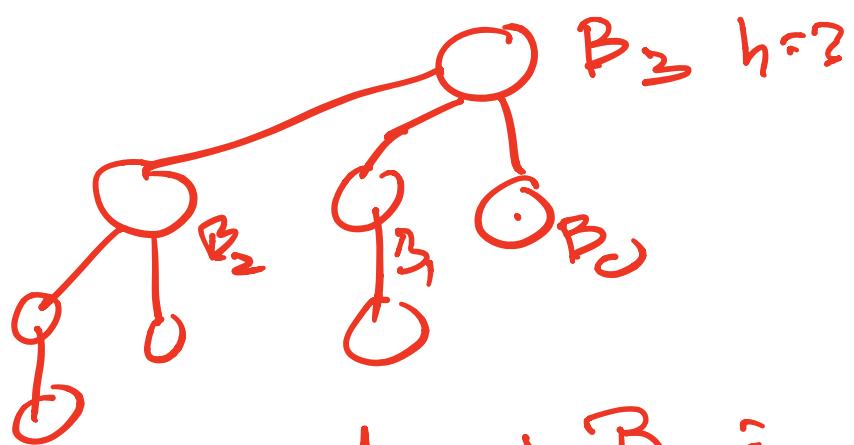
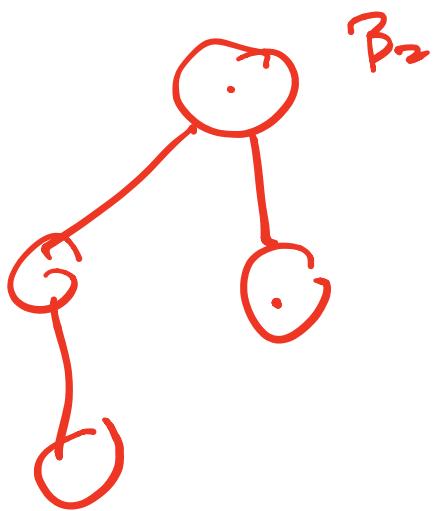


Binomial Heaps

- 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

$\log(n) \leq k$
 $n \rightarrow 2^k$

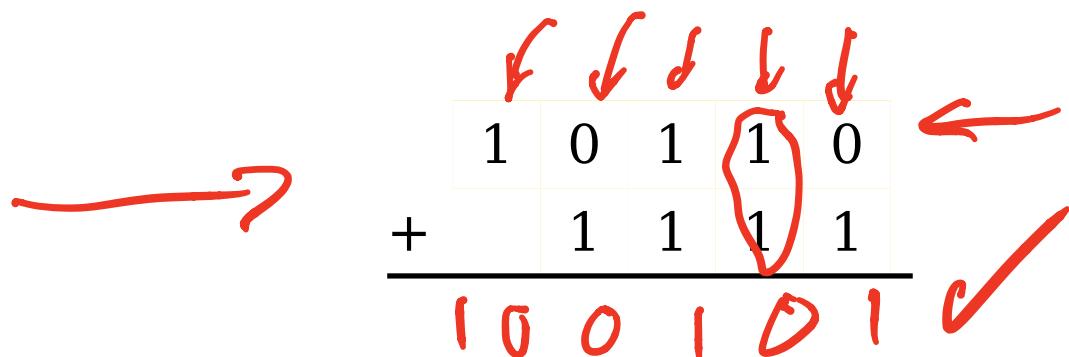




nodes in B_K =
 2^k .

Foundations: Binary Arithmetic

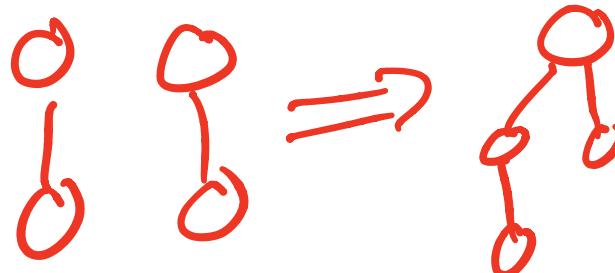
- Given the binary representations of two numbers n and m , we can add those numbers in time $\Theta(\max\{\log m, \log n\})$



A binary addition diagram showing the sum of two 5-bit binary numbers. The top number is 10110 and the bottom number is 11111. The sum is 100101. Red arrows and annotations highlight the carry operation: a red arrow points from the bottom-left to the fifth column of the grid; five red arrows point down from the top row to the bottom row, indicating the addition of bits; a red circle highlights the fourth column of the grid, where a carry is generated from the bottom bit; a red checkmark is placed at the end of the bottom row.

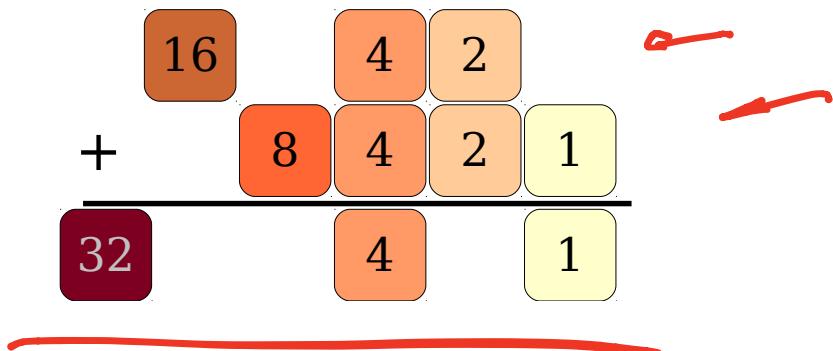
$$\begin{array}{r} 10110 \\ + 11111 \\ \hline 100101 \end{array}$$

.



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



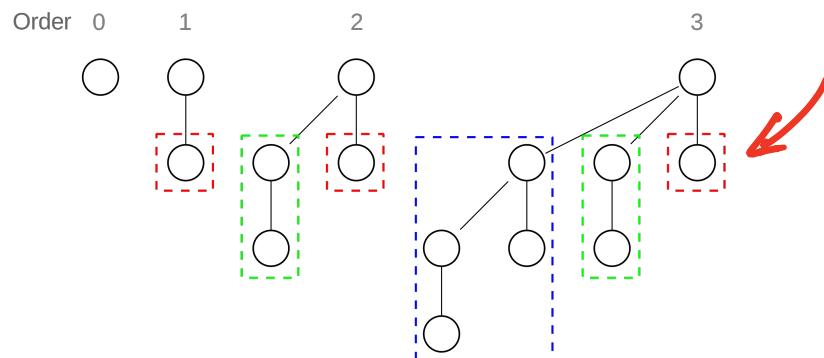
Idea for Priority Queues for Easy Merge

- 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

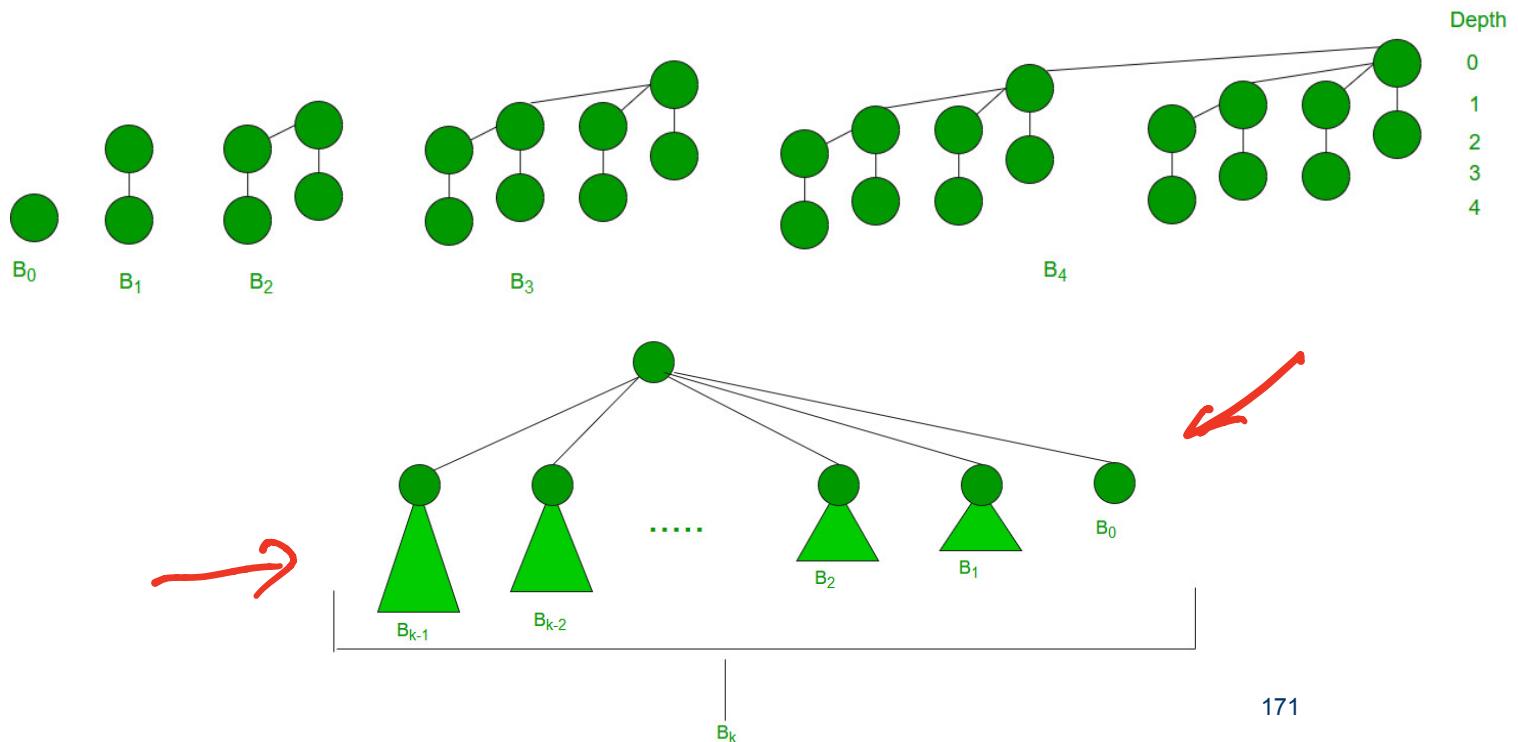
($\text{insert}, \text{delete min} \rightarrow O(1)$)
merge

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 $0, 1, 2, \dots, k - 1$
- Here are the first few binomial trees:



Binomial Trees



Binomial Trees

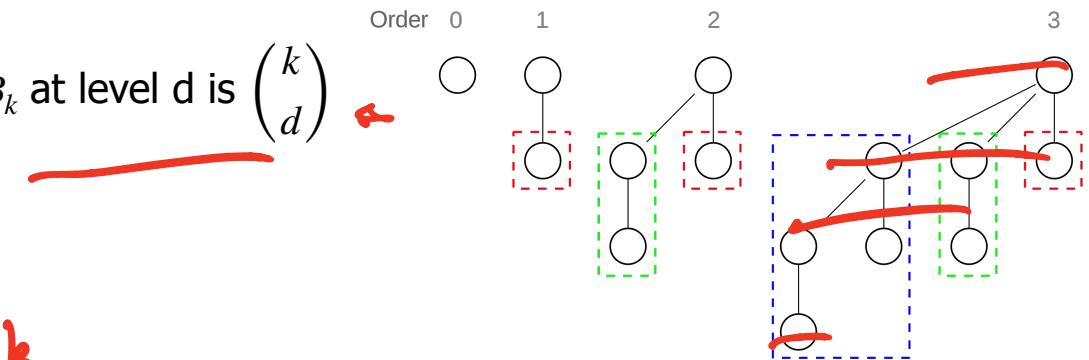
- Theorem: A binomial tree of order k has 2^k nodes

- 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



k

1 3 3 1

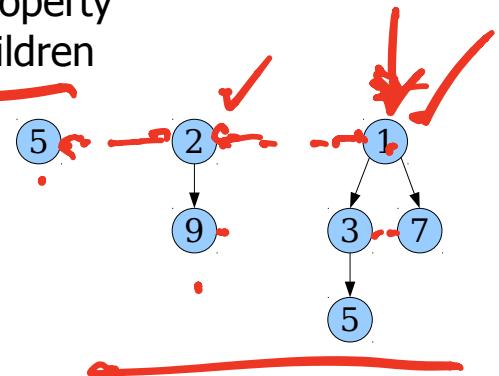
1 2 1

1 1

172

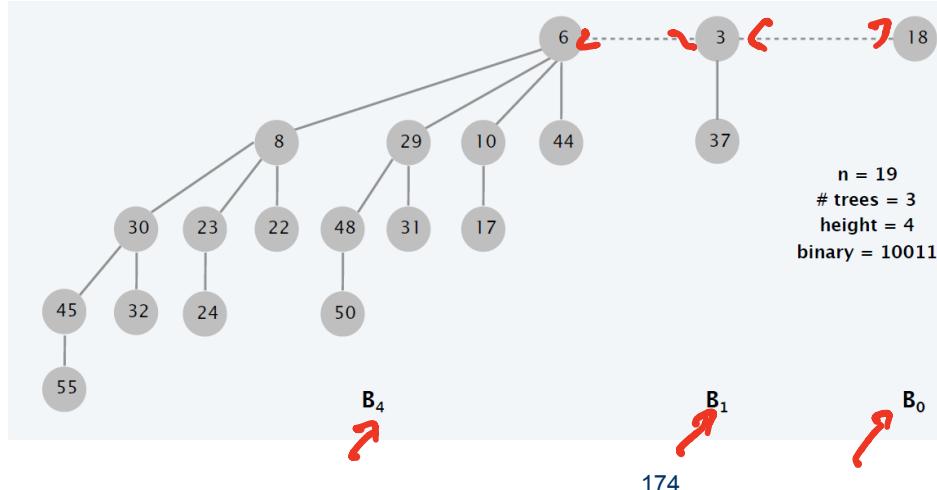
Heap-Ordered Binomial Trees

- 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

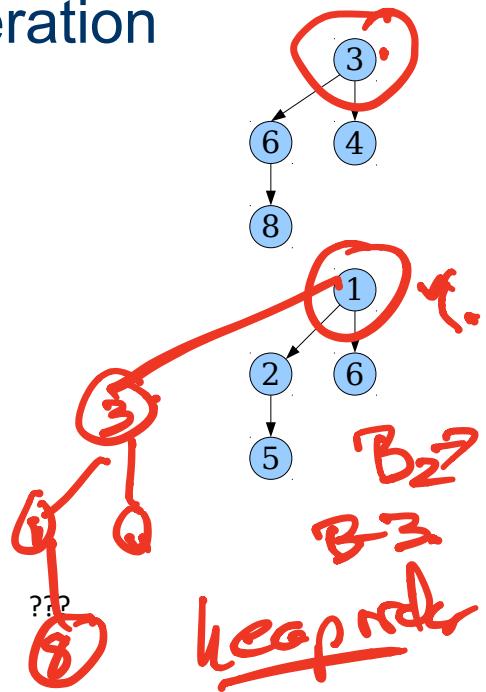
- Properties of Binomial heap with n nodes
 - The node containing the min element is a root of $B_k, B_{k-1}, \dots, B_1, B_0$
 - It contains the binomial tree B_k iff $b_k = 1$, where $b_k b_{k-1} \dots b_0$ is binary representation of n.
 - It has no more than $\lfloor \log_2 n \rfloor + 1$ binomial trees.
 - Its max height is less than $\lfloor \log_2 n \rfloor$



1 0 0 1 1 ↙

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

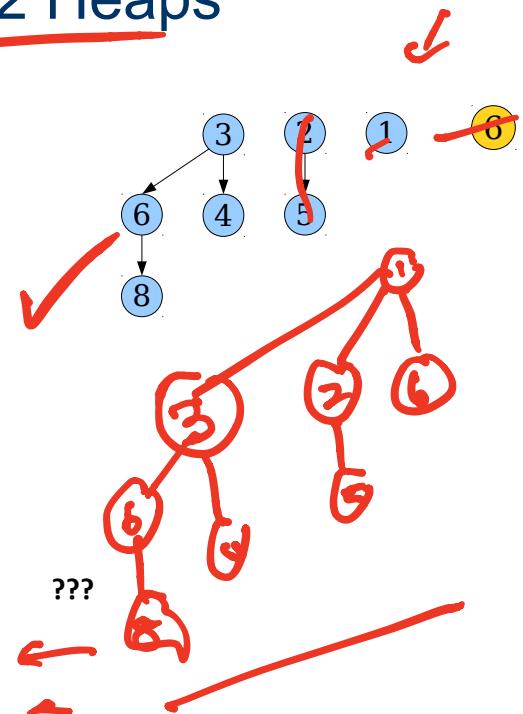


Binomial Heap: Merge 2 Heaps

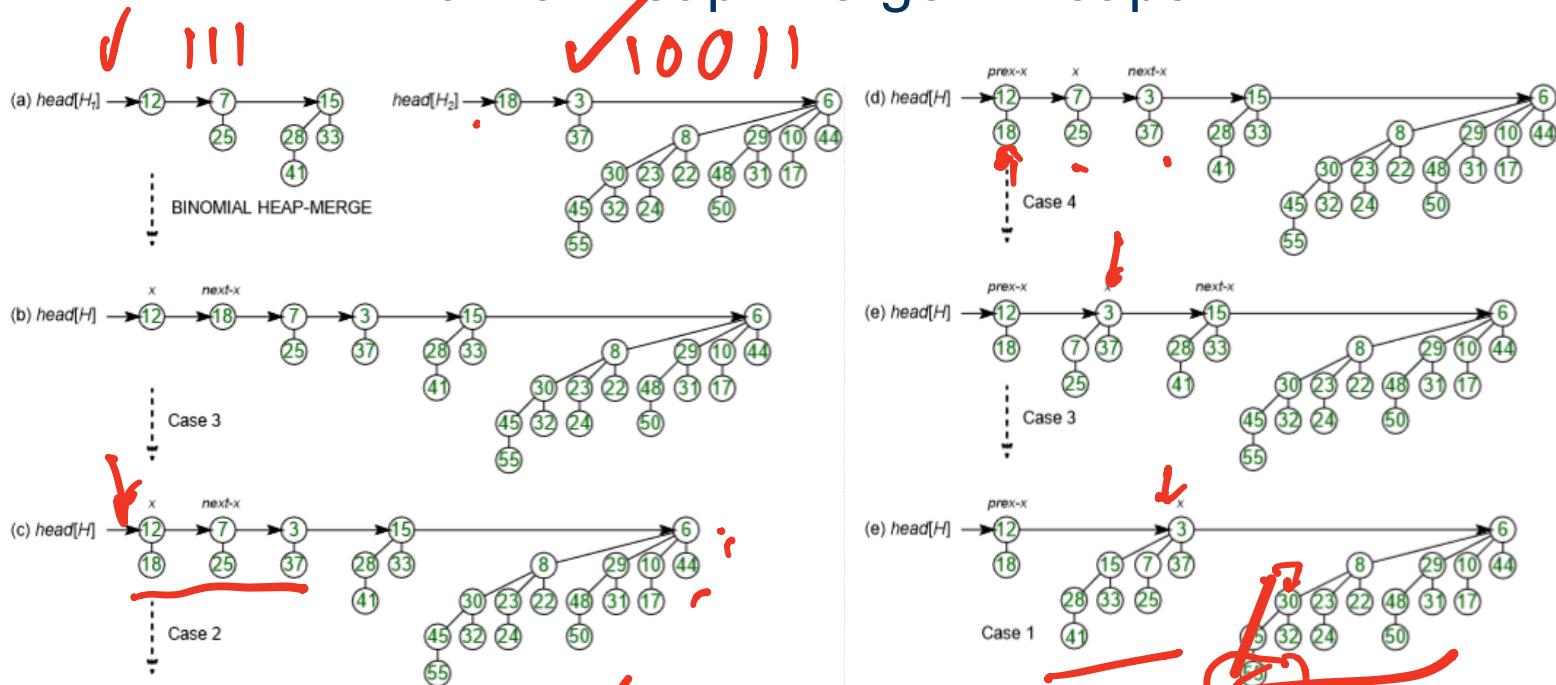
- Heap 1: Trees $B_0^1, B_1^1, \dots, B_k^1$
- Heap 2: Trees $B_0^2, B_1^2, \dots, B_m^2$
- Step 1. Add all the trees to the same list
 - May be duplicates in size
- Step 2: Starting from the smallest size, 0, if there are two or three of them, merge two of them into a tree of a higher order
 - Leaves at most one of that size behind
 - Repeats for increasing order

$\mathcal{O}(log(n))$

$$\begin{array}{r} 1 1 1 \\ + 1 1 1 \\ \hline 1 0 0 1 0 1 \end{array}$$



Binomial Heap: Merge 2 Heaps



Dijkstra's

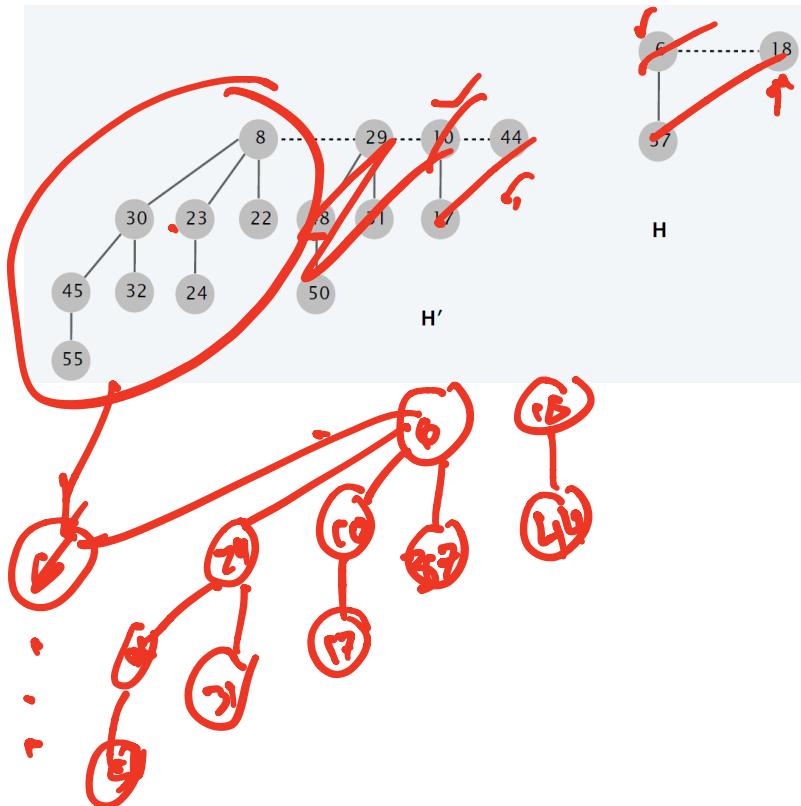
$\leftarrow \text{last}(u)$

177

$x \notin LD^8$

$\vec{B} \rightarrow | D \dots$

Binomial Heap: Insert ~~Merge~~

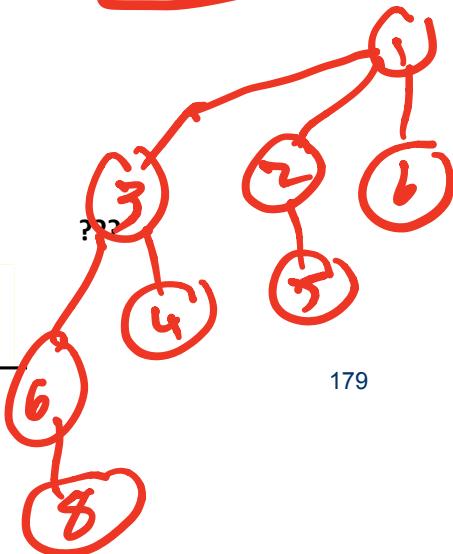
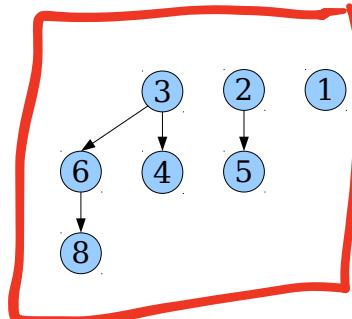


Binomial Heap: Insert

- 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

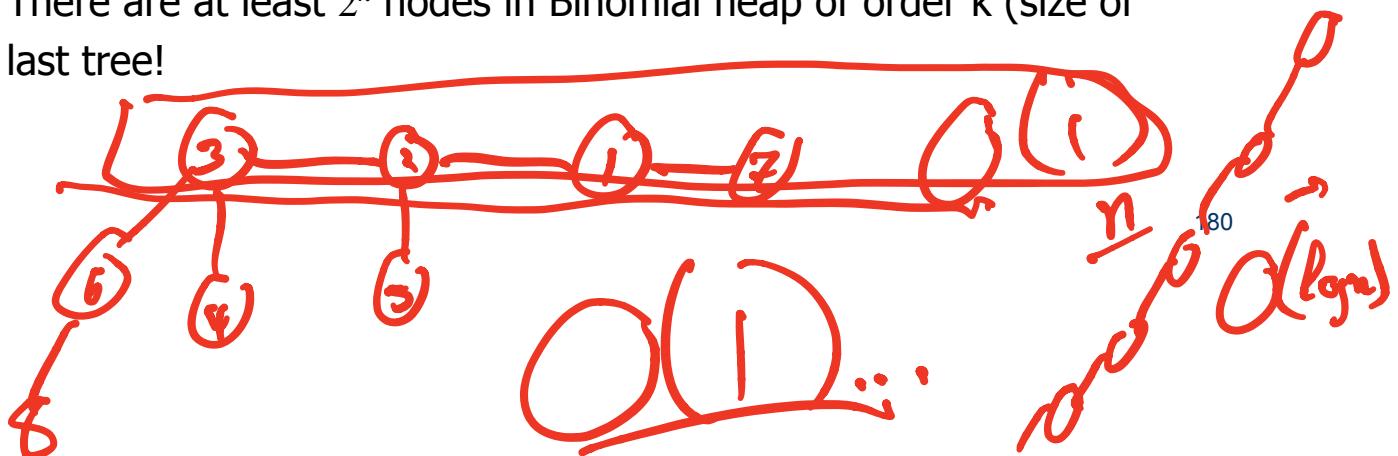
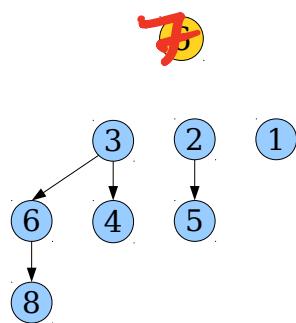
$O(\log(n))$.

$$\begin{array}{r} 1 & 0 & 1 & 1 & 0 \\ + & 1 & 1 & 1 & 1 \\ \hline \end{array}$$



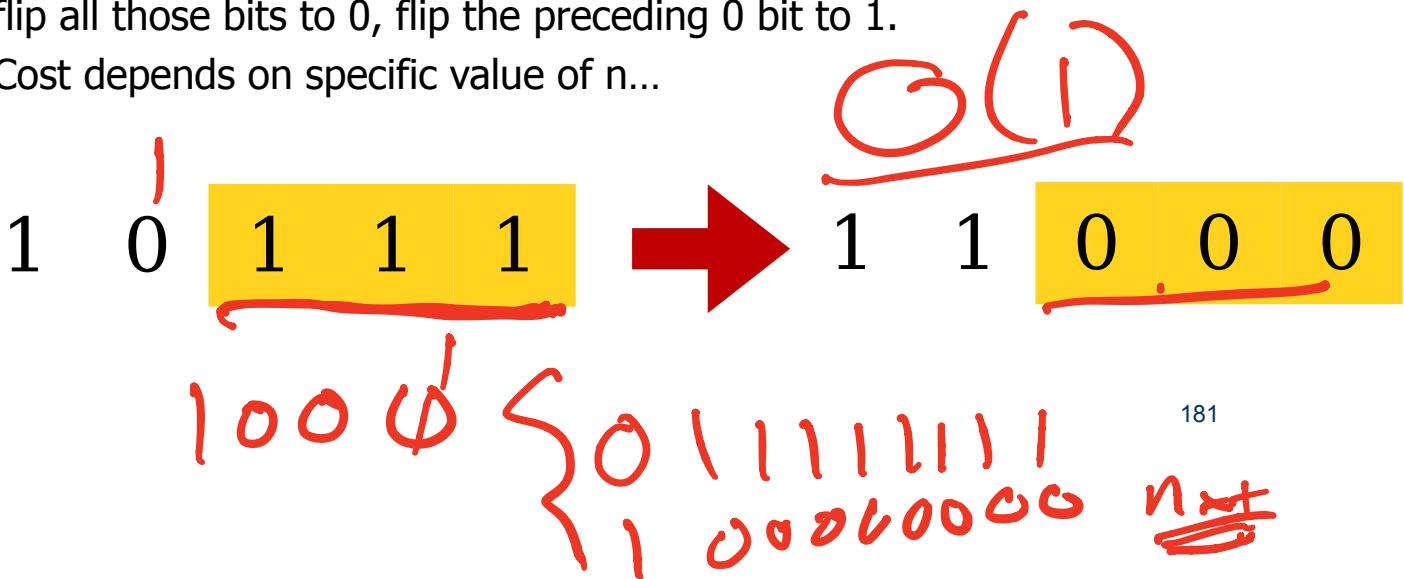
Binomial Heap: Insert Complexity

- 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 = $j + 1 + \Phi(n+1) - \Phi(n) = 2 \rightarrow O(1)$

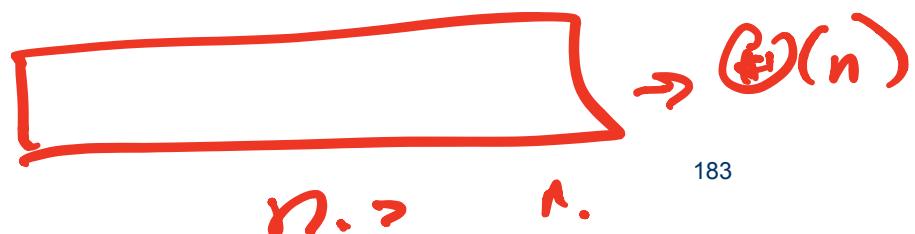
1 0 0 0 0

- Catch: cannot decrement! Only increment!

$n \leftarrow$
 $n - i$!

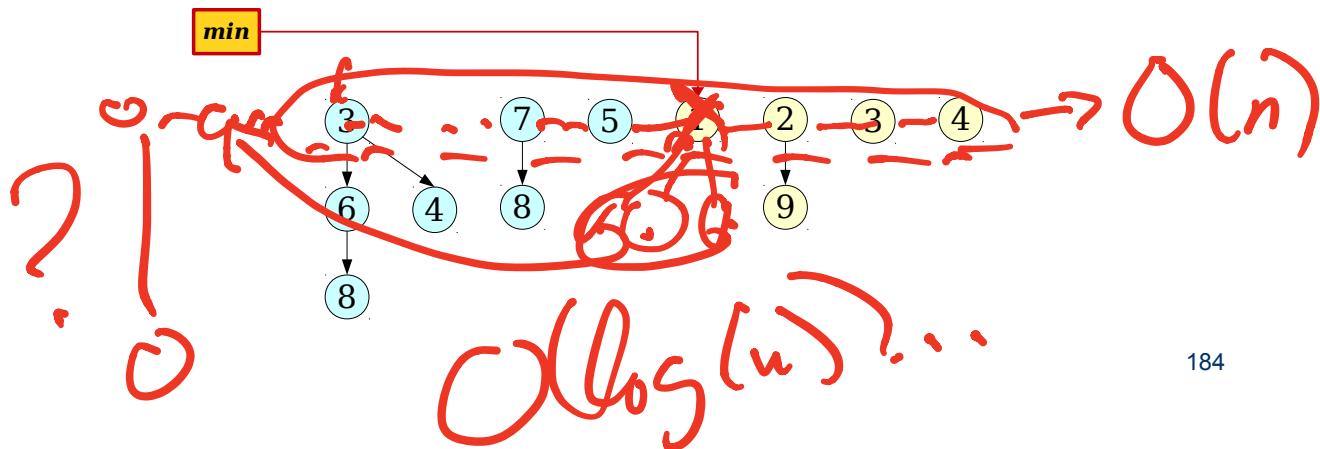
Binomial Heap: Amortized Insert Complexity

- 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 heap is $\Theta(n)$



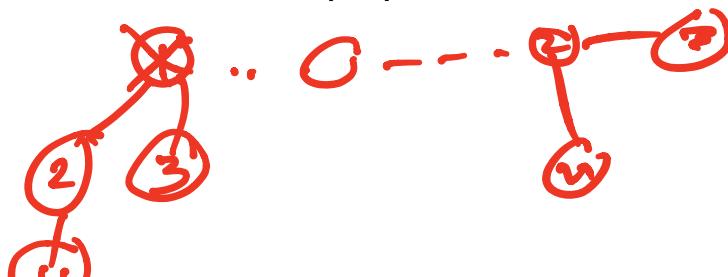
Binomial Heap: Lazy Merge

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

Amortized Analysis of Delete Minimum

- Potential method: $\Phi(\mathcal{H})$ is number of trees in Binomial heap \mathcal{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 \mathcal{H} , this is $\Theta(T)$
- If there are T current trees in \mathcal{H} , current potential $\Phi(\mathcal{H})$ is T
- After coalescing, potential $\Phi(\mathcal{H}) = \log(n)$
- Amortized cost: $O(\log(n)) + \Theta(T) - O(1)T + O(\log(n)) = O(\log(n)) !!!$

Overall Analysis of Lazy Binomial Heaps

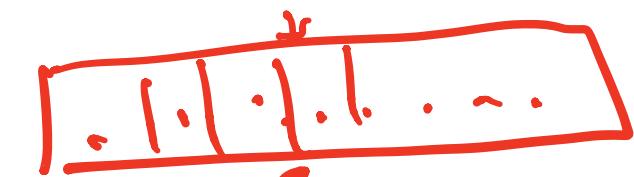
- The amortized costs of operations:

- Insert: $O(1)$
- Merge: $O(1)$
- Find Min: $O(1)$
- Delete Min: $O(\log(n))$

!
~~delete~~

- 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, otrs
 - Must do upHeap in that binomial tree: describe how in next data structure



K keys

Hash Tables

n not large

O(1).

- o Hashing

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

- o Generalizes an ordinary array,

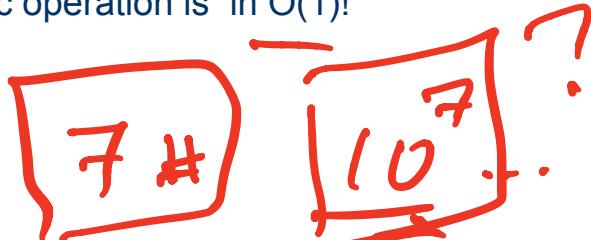
- o Key property: direct addressing
- o An array is a direct-address table: Key value is position of data in array

- o Main idea: Transform key into index, compute the index, then use an array of size N

- o Key k: data stored at $h(k)$ (hashing)

hash function.

- o Basic operation is in $O(1)$!



$$3 \times 10^4$$

ID# (234567 % 10) Price

Hashing

- o Hashing
 - o Technique supporting insertion, deletion and search in average-case constant time
 - o Operations requiring elements to be sorted (e.g., FindMin) are not efficiently supported
- o Generalizes an ordinary array,
 - o Key property: direct addressing
 - o An array is a direct-address table: Key value is position of data in array
- o Main idea: Transform key into index, compute the index, then use an array of size N
 - o Key k : data stored at $h(k)$ (hashing)
- o Basic operation is in $O(1)$!

Hash Table

- o Hash table is a data structure that support
 - o Finds, insertions, deletions (deletions may be unnecessary in some applications)
 - o The implementation of hash tables is called hashing
 - o A technique which allows the executions of above operations in constant average time
 - o Tree operations that requires any ordering information among elements are not supported
 - o findMin and findMax
 - o Successor and predecessor
 - o Report data within a given range
 - o List out the data in order

General Idea

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

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	<...>

1 2 3 4 5 6 7
7 2 3 4 6 8 1

Index

Hash Function

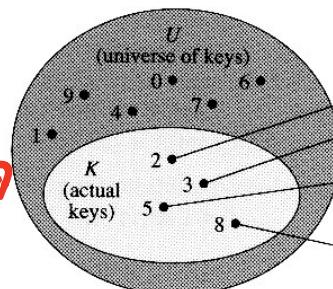
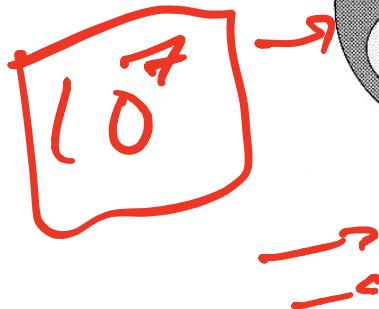
- o Mapping from key to array index is called a hash function

- o Typically, many-to-one mapping
- o Different keys map to different indices
- o Distributes keys evenly over table

- o Collision occurs when hash function maps two keys to the same array index

Unrealistic solution

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



T	key	satellite data
0		
1		
2	2	
3	3	
4		
5	5	
6		
7		
8	8	
9		

.94

Applications

- o Compilers use hash tables (symbol table) to keep track of declared variables.
- o 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. *30 sec.*
- o 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.

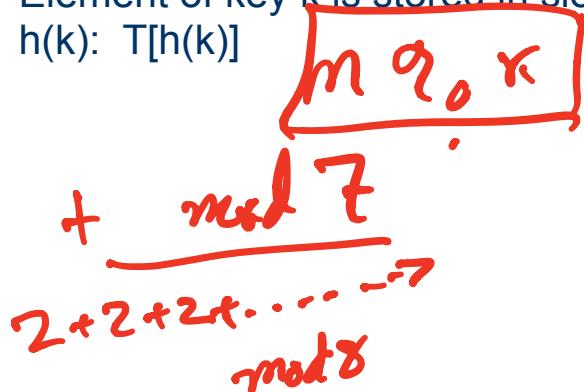
Python dictionary!

Hash Function

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

$\{K_0, K_1, \dots, K_{N-1}\}$ possible keys
↓
hash function h
T[0, ..., m - 1]

$m q_0, x$



Resolving Collisions

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

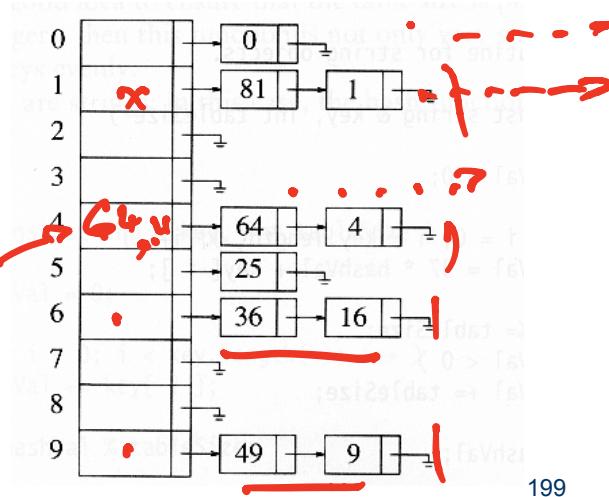
Design Hash Function

- o A simple and reasonable strategy: $h(k) = k \bmod m$
 - o e.g. $m=12$, $k=100$, $h(k)=4$ ✓
 - o Requires only a single division operation (quite fast)
- o Certain values of m should be avoided
 - o e.g. if $\underline{m=2^p}$, then $h(k)$ is just the p lowest-order bits of k ; does not depend on all the bits
 - o if the keys are decimal numbers, should not set m to be a power of 10
- o It's a good practice to set the table size m to be a prime number ✓
- o Good values for m : primes not too close to exact powers of 2
 - o 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 $m=701$
- o For strings, map keys to ascii integers...

$k^n \bmod p \rightarrow 0, 1, \dots, p-1$

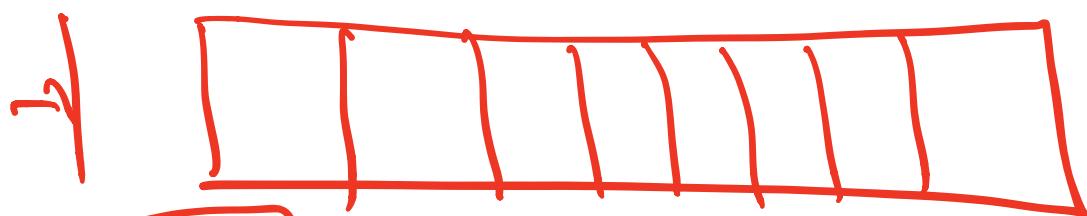
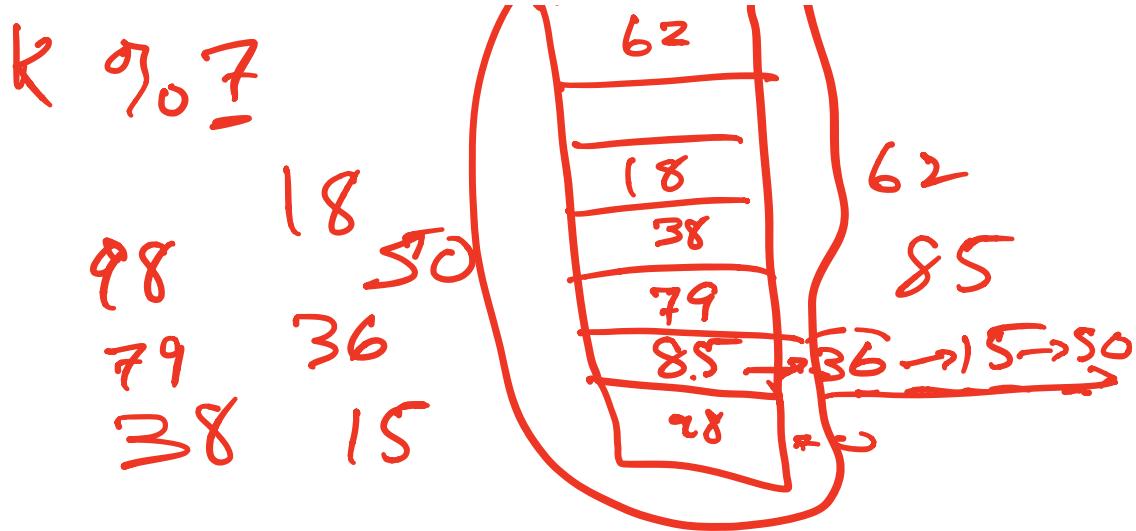
Collision Resolution

- o Chaining: Like 'equivalent classes' or clock numbers in math
 - o Instead of a hash table, we use a table of linked list
 - o keep a linked list of keys that hash to the same value
 - o Problem: can get to be $O(n)$ worst case
- o Typical approach: if storing M keys, make hash number the next largest prime number



199

hash fun.



$m < n$

$$P(x = k) = \frac{1}{n} \quad | \quad n \uparrow, m \downarrow$$

$P()$ collision?

$P(\text{no collision})$ $n \rightarrow 50$

$$\left(1 - \frac{n-1}{n} \cdot \frac{n-2}{n} \cdots \frac{n-k}{n} \right) \rightarrow \text{fast} \quad | \quad 10^{-2}$$

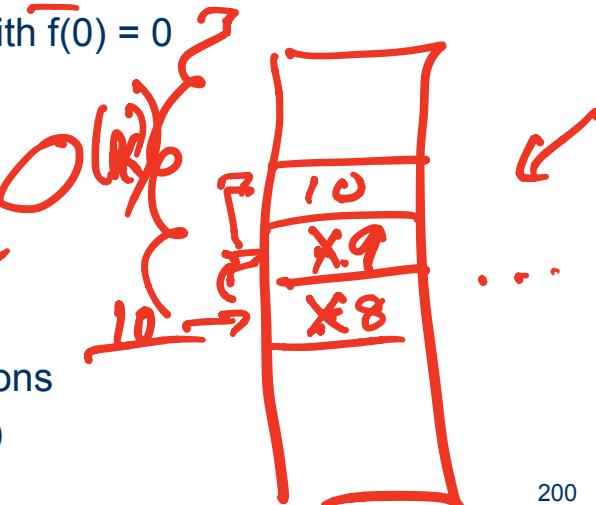
Collision Resolution - 2

Open addressing

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

Several approaches

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



New Data Structure: Fibonacci Heaps

- Fredman-Tarjan 1986
 - CLRS Chapter 19