Hash Tables

Tables so far

		set()	get()	delete()
BST	Average	O(lg n)	O(lg n)	O(lg n)
	Worst	O(n)	O(n)	O(n)
RB Tree	Average	O(lg n)	O(lg n)	O(lg n)
	Worst	O(lg n)	O(lg n)	O(lg n)

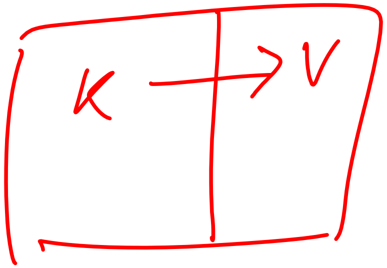


Table naïve array implementation

- "Direct addressing"
- Worst case O(1) access cost
- But likely to waste space

2 → A

4 → B

5 → C

get	O(1)
set	O(1)
search	O(1)

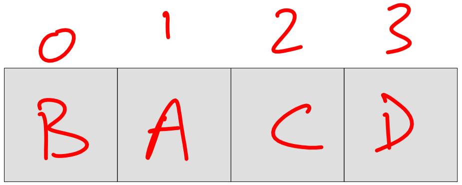
0	1	2	3	4	5	6
X	X	A	X	B	C	X

Hashing

- A hash function is just a function h(k) that takes in a key and spits out an integer between 0 and some other integer M
- For a table:
  - Create an array of size M
  - h(key) => index into array
- E.g. Division hash: h(k)=k mod m

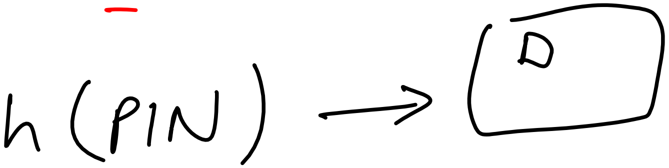
k mod 4

Key	Value	h(k)
2	C	2
3	D	3
8	B	0
9	A	1



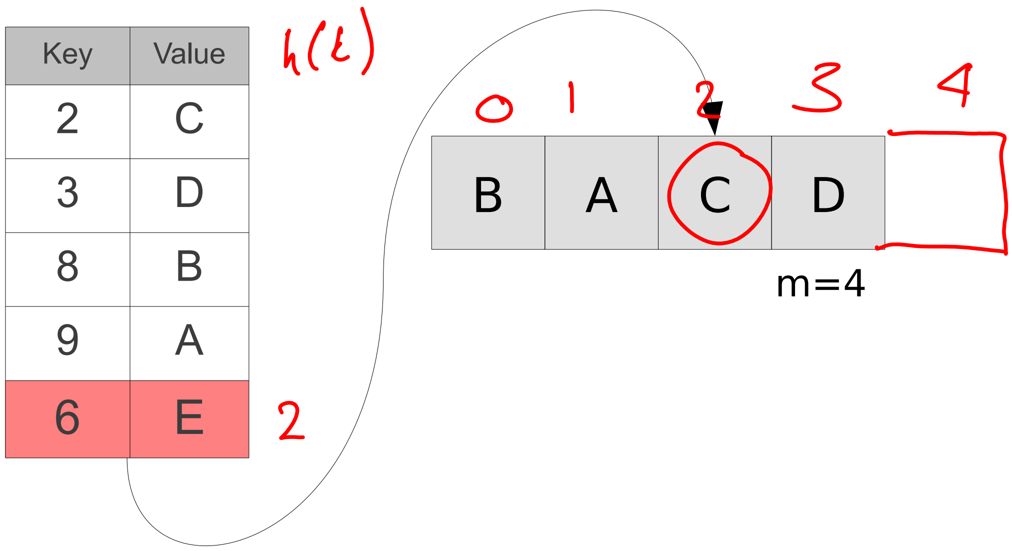
0	1	2	3
B	A	C	D

m=4



Collisions

- Set of all possible keys, U
- Set of actual keys, n
- We usually expect |n|<<|U| so we would like M<<|U|
- Inevitably, multiple keys must map to the same hash value: **Collisions**



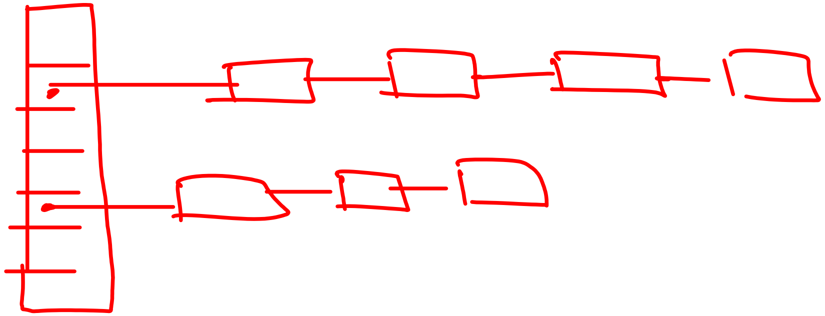
Key	Value
2	C
3	D
8	B
9	A
6	E

h(k)

2

Chaining

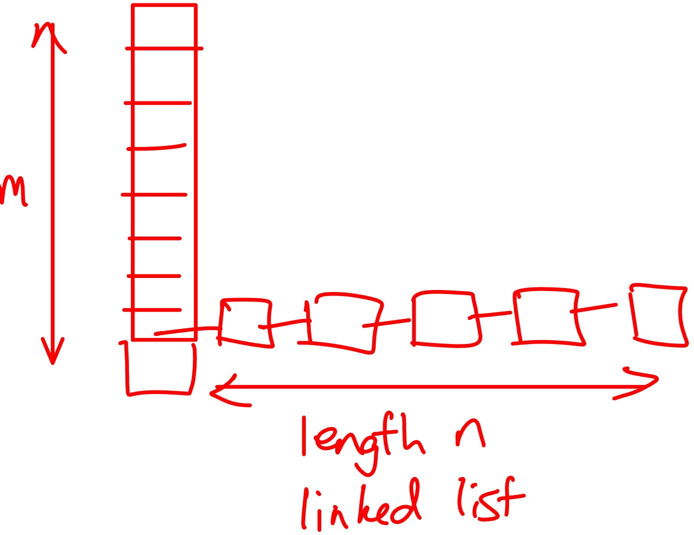
- Each hash table slot is actually a linked list of keys



- Analysis of costs
  - Depends on hash function and input distribution!
  - Can make progress by considering **uniform hashing:** h(k) is equally likely to be any of the M outputs.

Chaining Analysis

Worst case



Insert

$$\overset{\text{hash}}{O(1)} + \overset{\text{mem access}}{O(1)} + O(n)$$

$$O(n)$$

Search

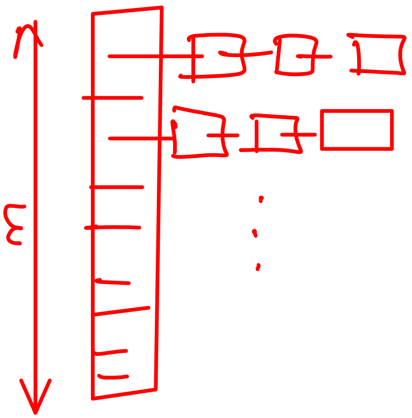
$$O(1) + O(1) + O(n) \quad O(n)$$

Delete

$$O(1) + O(1) + O(n) \quad O(n)$$

Chaining Analysis

Average case



Uniform ⇒ length $\frac{n}{m}$ lists

Insert

$O(1) + O(1) + O(\frac{n}{m}) = O(1 + \frac{n}{m})$

$= O(1 + \alpha)$

Search

$O(1 + \alpha)$

Delete

$O(1 + \alpha)$

The Load Factor

$\alpha = \frac{n}{m}$

0	Fast	waste memory
1	Fast	efficient
∞	Slow	semi-efficient memory

Try to keep α constant

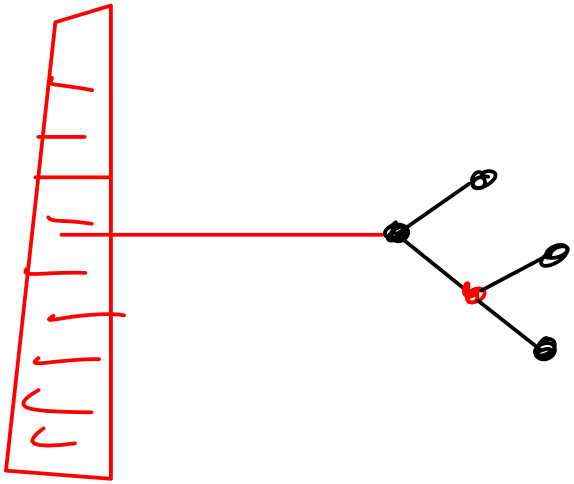
$\sim \underline{\underline{0.75}}$

$O(1+\alpha)$

$\sim O(1)$

Variants

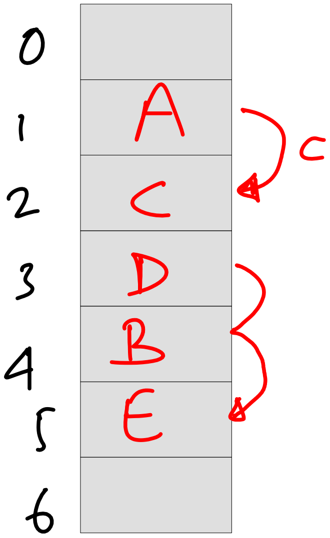
- Sometimes speedy lookup is an absolute requirement e.g. real-time systems
- Sometimes see variants of chaining where the linked list is replaced with a BST or Red-Black tree or similar
- (What does this do to the complexities?)



Open Addressing

- Instead of chaining, we could simply use the next unassigned slot in our array.

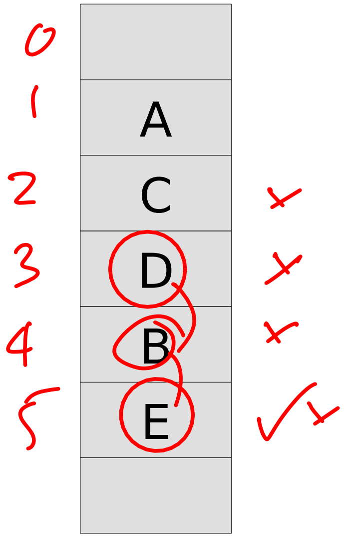
Keys: A,B,C,D,E
h(A)=1
h(B)=4
h(C)=1
h(D)=3
h(E)=3



Open Addressing

- Instead of chaining, we could simply use the next unassigned slot in our array.

Keys: A,B,C,D,E
h(A)=1
h(B)=4
h(C)=1
h(D)=3
h(E)=3
h(X)=2



Search for E

$h(E) = 3$ 3 lookups

Search for X

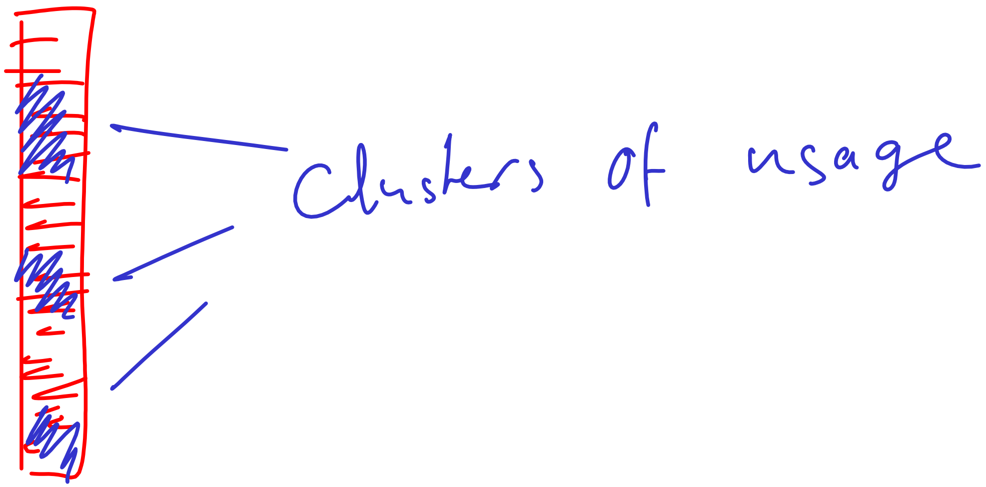
$h(X) = 2$

X not found

5 lookups

Linear Probing

- We call this **Linear Probing** with a step size of one (you probe the array until you find an empty slot)
- Basically 'randomises' the start of the sequence and then proceeds incrementally
- Simples :-)
- Get long runs of occupied slots separated by empty slots => **“Primary Clustering”**



Better Probing

- We can extend our idea to a more general probe sequence
  - Rather than jumping to the next slot, we jump around (the more pseudorandom the better)
  - So each key has some (hopefully unique) probe sequence: an ordered list of slots it will try
  - As before, operations involve following the sequence until an element is found (hit) or an empty slot is found (miss) or the sequence ends (full).
  - So we need some function to generate the sequence for a given key

  - Linear probing would have:

$$S_i(k) = (h(k) + i) \bmod m$$

Better Probing

- Quadratic Probing

$$S_i(k) = (h(k) + c_1 i + c_2 i^2) \bmod m$$

⇒ pseudorandom jumping around ✓✓

⇒ No primary clustering

But Don't visit every slot

- Two keys with the same hash have the same probe sequence => **"Secondary Clustering"**

Better Probing

- Double Hashing

$$S_i(k) = (h_1(k) + ih_2(k)) \bmod m$$

⇒ 2 Different hash functions

⇒ Hard to choose h_1 h_2

⇒ No primary clustering

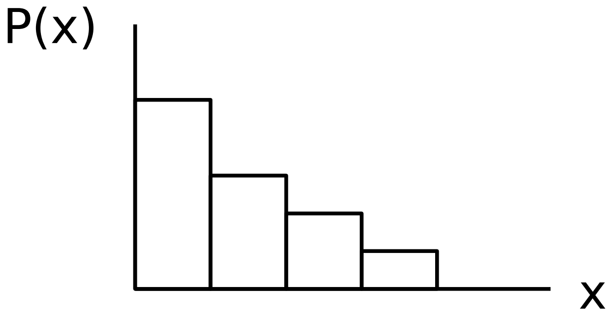
⇒ No sec. clustering.

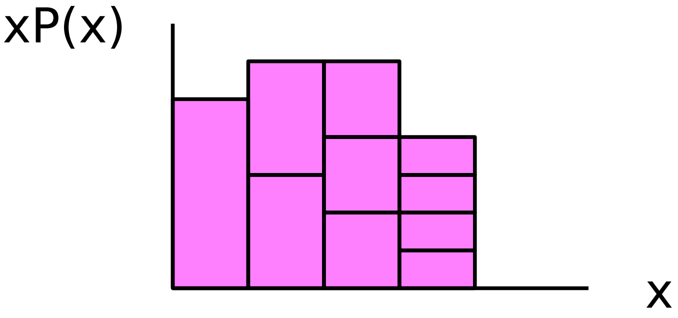
Analysis

- Let x = no. of probes needed
- What is E(x)?

$E(x) = \sum x P(x)$

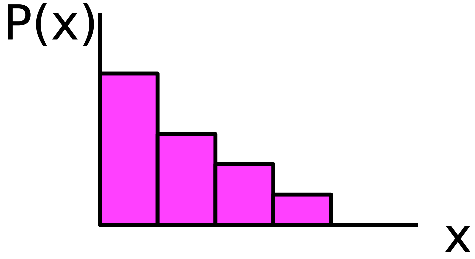
Aside: Expectation





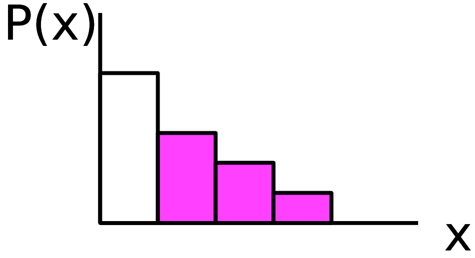
$$E(x)=\sum xP(x)$$

Aside: Expectation



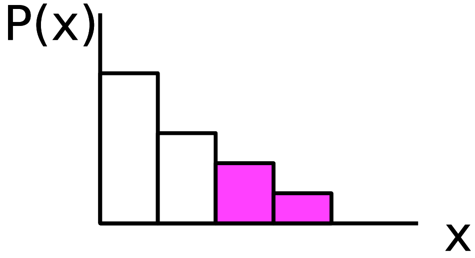
$$P\left(x \geq 1\right)$$

+



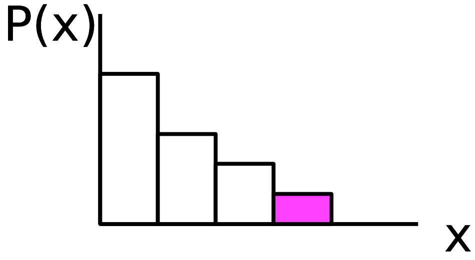
$$P\left(x \geq 2\right)$$

+



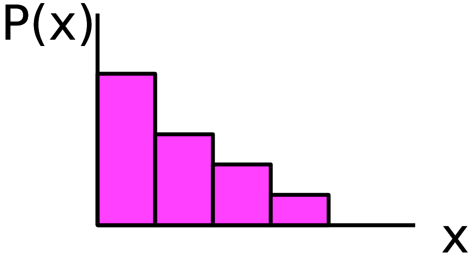
$$P\left(x \geq 3\right)$$

+



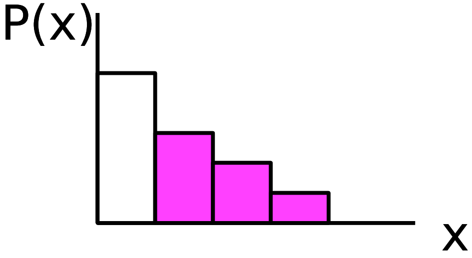
$$P\left(x \geq 4\right)$$

Aside: Expectation



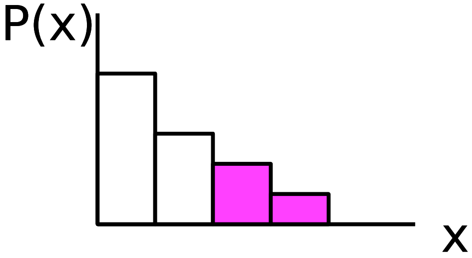
$$P\left(x \geq 1\right)$$

+



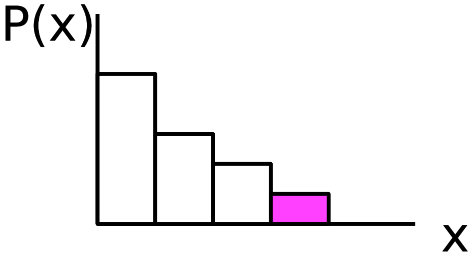
$$P\left(x \geq 2\right)$$

+



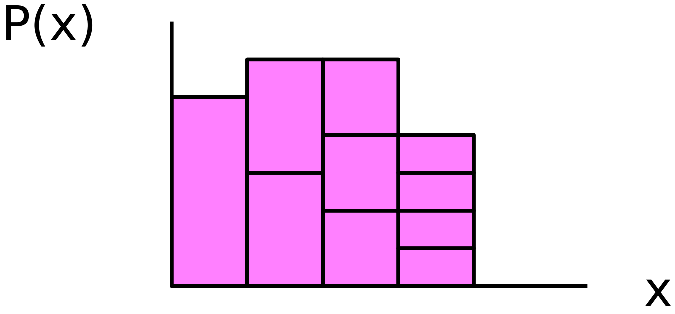
$$P\left(x \geq 3\right)$$

+



$$P\left(x \geq 4\right)$$

=



$$E\left(x\right) = \sum_i P\left(x \geq i\right)$$

Analysis

- Let x = no. of probes needed
- What is E(x)?
- What is P(x>=i)?

$$E(x) = \sum_i P(x \geq i)$$

Uniform hashing

$$P(x \geq i) = \frac{n}{m} \times \frac{n-1}{m-1} \times \frac{n-2}{m-2} \times \dots \times \frac{n-i+2}{m-i+2}$$

$$\leq \frac{n}{m} \times \frac{n}{m} \times \frac{n}{m} \times \dots \times \frac{n}{m}$$

$$\leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

$$0 \leq \alpha \leq 1$$

$$E(x) = \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \underline{\underline{\frac{1}{1-\alpha}}}$$

$$S_\infty = \frac{1}{1-r}$$

Analysis

Successful search

Imagine $(i+1)^{th}$ element inserted

Expected no. probes $= \frac{1}{1-\frac{i}{m}}$

Ave over n inserts

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} \longrightarrow \frac{1}{\alpha}\log\frac{1}{1-\alpha}$$

Open Addressing Performance

- Ave. number of probes in a failed search

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

- Ave. Number of probes in a successful search

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

- If we can keep n/m ~constant, then the searches run in O(1) still

Resizing your hash tables

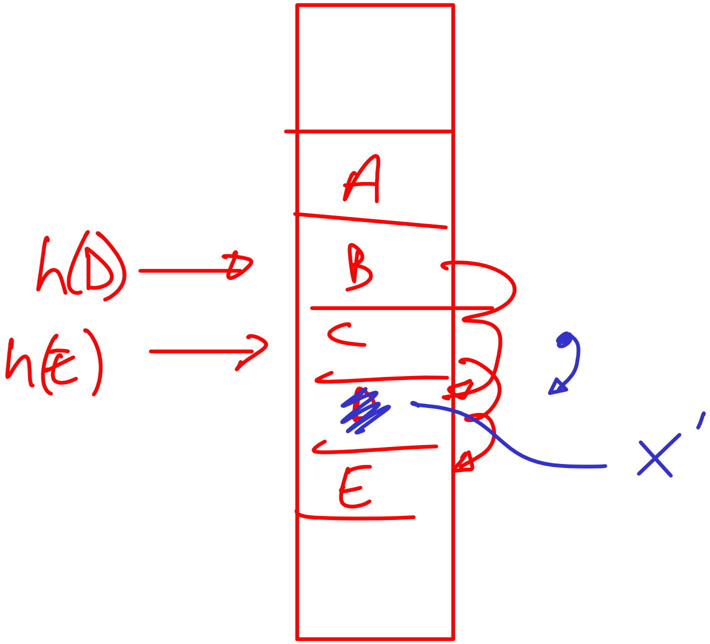
1. Allocate space $O(1)$
2. Rehash all elements $n \times O(1) = O(n)$
3. Copy in $n \times O(1) = O(n)$

$O(n)$ resize array of n

Amortised time when doubling each time $= \underline{\underline{O(1)}}$

Issues with Hash Tables

- Worst-case performance is dreadful
- Deletion is slightly tricky if using open addressing



Priority Queues

Priority Queue ADT

- first() - get the smallest key-value (but leave it there)
- insert() - add a new key-value
- extractMin() - remove the smallest key-value
- decreaseKey() - reduce the key of a node
- merge() - merge two queues together

Sorted Array Implementation

- Put everything into an array
- Keep the array sorted by sorting after every operation

- first()

- insert()

- extractMin()

- decreaseKey()

- merge()

Binary Heap Implementation

- Could use a *min-heap* (like the max-heap we saw for heapsort)
- insert()
- first()

Binary Heap Implementation

- extractMin()
- decreaseKey()
- merge()

Limitations of the Binary Heap

- It's common to want to merge two priority queues together
- With a binary heap this is costly...

Binomial Heap Implementation

- First define a binomial **tree**
  - Order 0 is a single node
  - Order k is made by merging two binomial trees of order (k-1) such that the root of one remains as the overall root

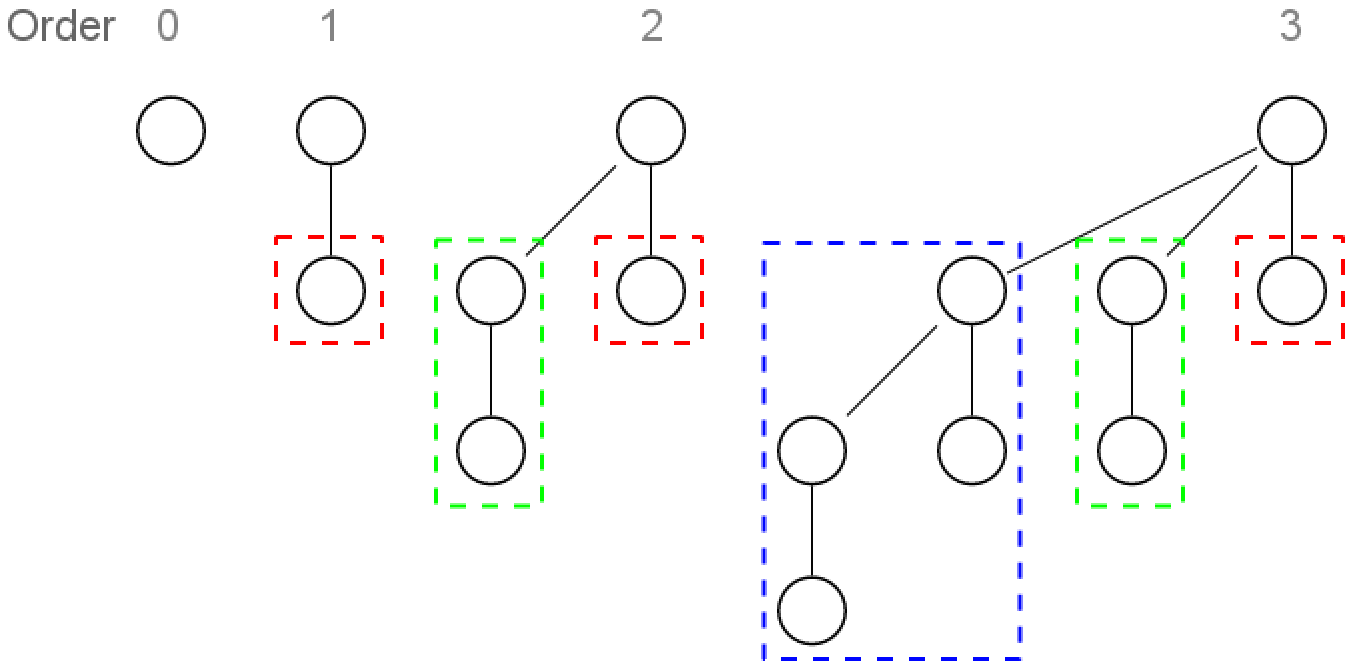


Image courtesy of wikipedia

Merging Trees

- Note that the definition means that two trees of order X are trivially made into one tree of order X+1

How Many Nodes in a Binomial Tree?

- Because we combine two trees of the same size to make the next order tree, we double the nodes when we increase the order
- Hence:

Binomial Heap Implementation

- Binomial **heap**
- A set of binomial trees where every node is smaller than its children
- And there is at most one tree of each order attached to the root

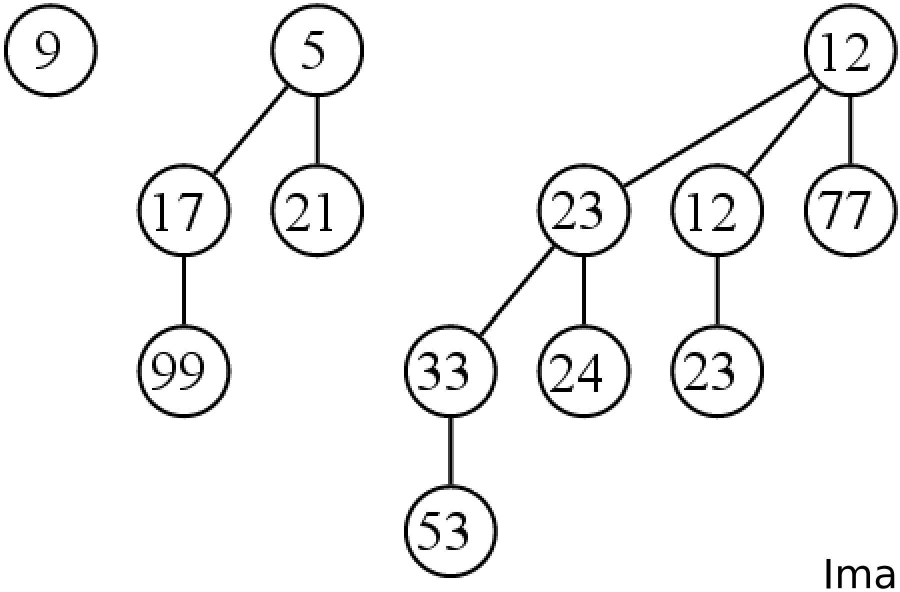


Image courtesy of wikipedia

Binomial Heaps as Priority Queues

- first()
    - The minimum node in each *tree* is the tree root so the heap minimum is the smallest root

How many roots in a binomial heap?

- For a heap with n nodes, how many root (or trees) do we expect?
- Because there are 2^k nodes in a tree of order k, the binary representation of n tells us which trees are present in a heap. E.g 100101
- The biggest tree present will be of order log n, which corresponds to the (log n +1)-th bit
  - So there can be no more than (log n +1) roots
- first() is O(no. of roots) = O(lg n)

Merging Heaps

- Merging two heaps is useful for the other priority queue operations
- First, link together the tree heads in increasing tree order

Merging Heaps

- Now check for duplicated tree orders and merge if necessary

Merging Heaps: Analogy

- This process is actually analogous to binary addition!

Merging Heaps: Costs

- Let H1 be a heap with n nodes and H2 a heap with m nodes

Priority Queue Operations

- insert()
  - Just create a zero-order tree and merge!

- extractMin()
  - Splice out the tree with the minimum
  - Form a new heap from the 2nd level of that tree
  - merge the resulting heap with the original

Priority Queue Operations

- decreaseKey()
  - Change the key value
  - Let it 'bubble' up to its new place
  - O(height of tree)

Priority Queue Operations

- deleteKey()
  - Decrease node value to be the minimum
  - Call extractMin() (!)