Your hash bucket was tasty

CSE 250
Lecture 30

Hash Tables

The University at Buffalo, SUNY

Recap: So Far

- Current Design: Hash Table with Chaining
  - Array of Buckets
  - Each bucket is the head of a linked list (a “chain”)

Recap: apply(x)

- Expected Cost
  - Find the bucket: $O(c_{hash})$
  - Find the record: $O(\alpha \cdot c_{equality})$
  - **Total**: $O(c_{hash} + \alpha \cdot c_{equality}) \approx O(1 + 1) = O(1)$
- Worst-Case Cost
  - Find the record: $O(n \cdot c_{equality})$
  - **Total**: $O(c_{hash} + n \cdot c_{equality}) \approx O(1 + n) = O(n)$

Recap: remove(x)

- Expected Cost
  - Find the bucket: $O(c_{hash})$
  - Find the record: $O(\alpha \cdot c_{equality})$
  - Remove from linked-list: $O(1)$
  - **Total**: $O(c_{hash} + \alpha \cdot c_{equality} + 1) \approx O(1 + 1 + 1) = O(1)$
- Worst-Case Cost
  - Find the record: $O(n \cdot c_{equality})$
  - **Total**: $O(c_{hash} + n \cdot c_{equality} + 1) \approx O(1 + n + 1) = O(n)$

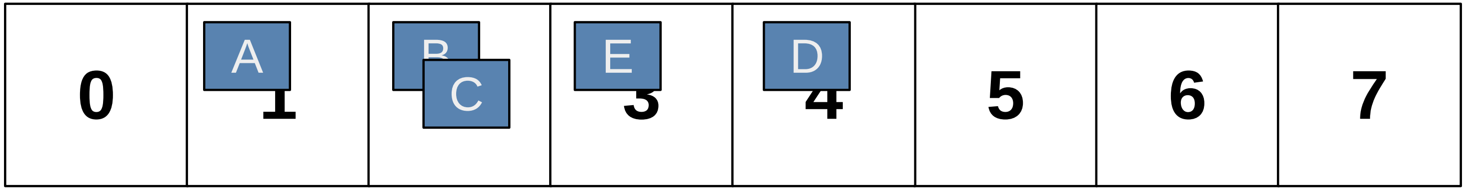
Recap: insert(x)

- Expected Cost
  - Find the bucket: $O(c_{hash})$
  - Remove the key, if present: $O(\alpha \cdot c_{equality} + 1)$
  - Prepend to linked-list: $O(1)$
  - Rehash: $O(n \cdot c_{hash} + N)$; amortized: $O(1)$
  - **Total**: $O(c_{hash} + \alpha \cdot c_{equality} + 1) \approx O(1 + 1 + 2) = O(1)$
- Worst-Case Cost (amortized)
  - Remove the key, if present: $O(n \cdot c_{equality} + 1)$
  - **Total**: $O(c_{hash} + n \cdot c_{equality} + 1 + 1) \approx O(1 + n + 2) = O(n)$

Variations

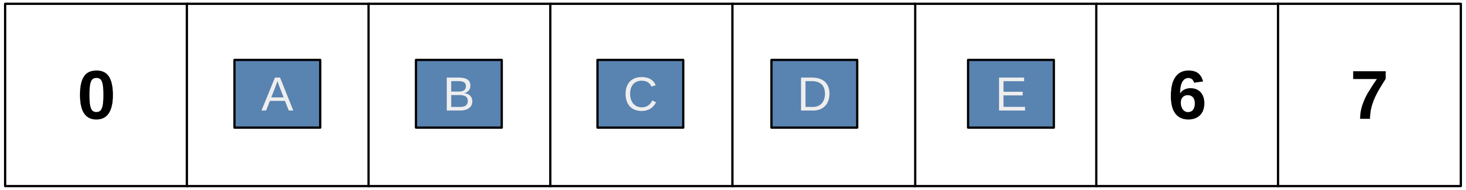
- **Hash Table with Chaining**
  - ... but re-use empty hash buckets instead of chaining
    - **Hash Table with Open Addressing**
    - **Cuckoo Hashing** (Double Hashing)
  - ... but avoid bursty rehashing costs
    - **Dynamic Hashing**
  - ... but avoid O(N) iteration cost
    - **Linked Hash Table**

Chaining



hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

Open Addressing

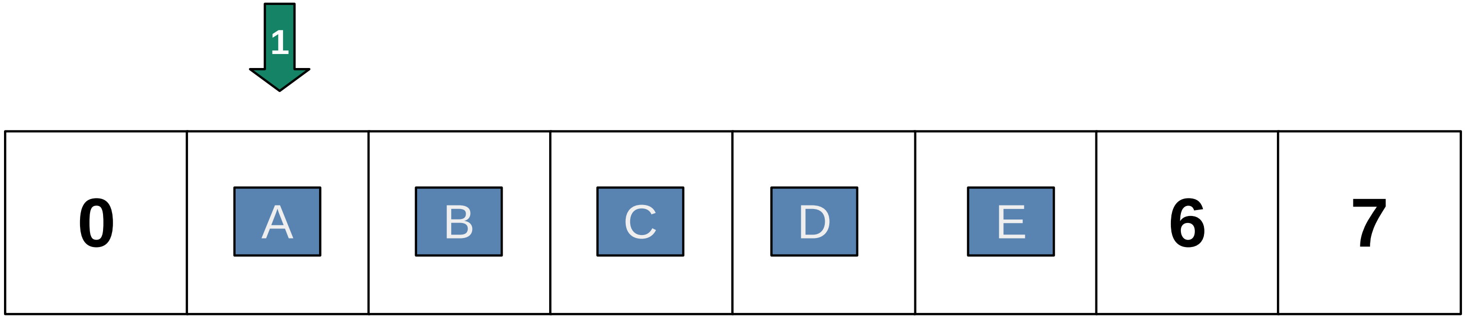


hash(A) = 1
hash(B) = 2
hash(C) = 2 !
hash(D) = 4
hash(E) = 3 !

"Cascade" collisions to the next available spot

Open Addressing

apply(A)

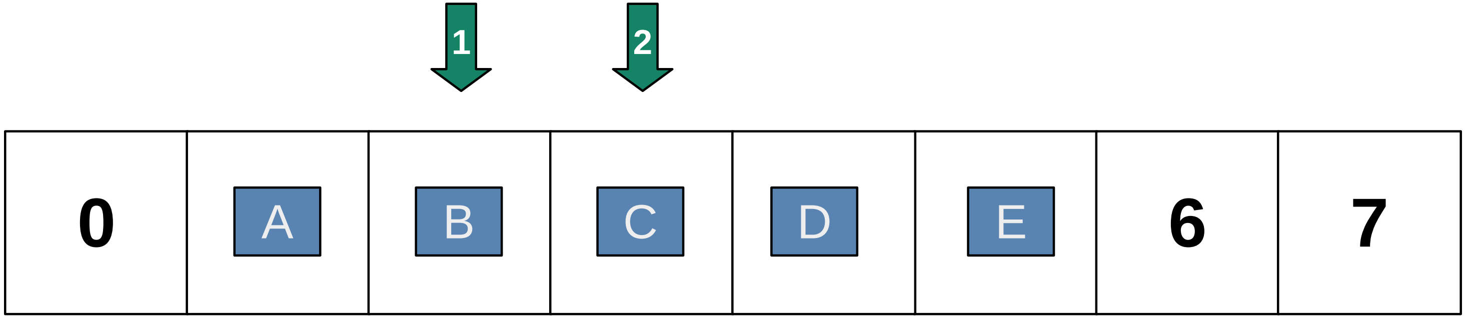


hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

"Cascade" collisions to the next available spot

Open Addressing

apply(C)

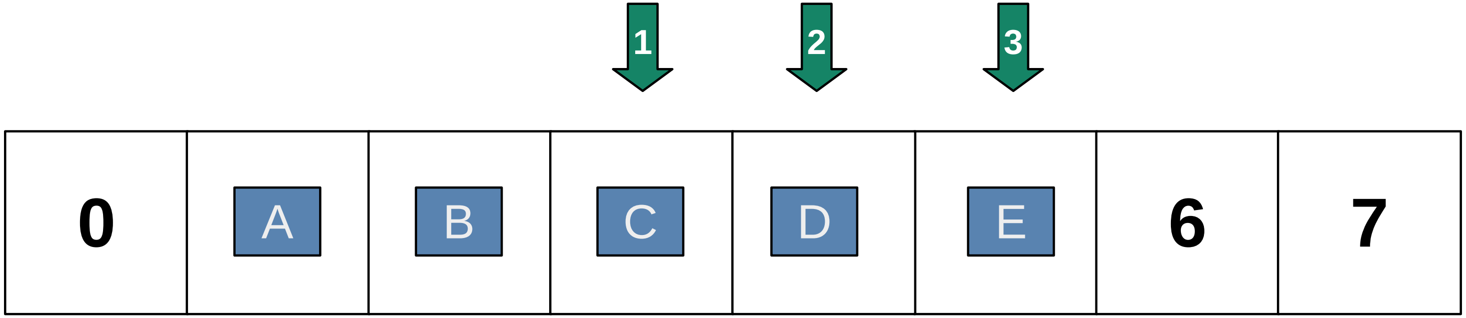


hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

“Cascade” collisions to the next available spot

Open Addressing

apply(E)



hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

“Cascade” collisions to the next available spot

Open Addressing

- insert(X)
  - While bucket hash(X)+i %N is occupied, i = i + 1
  - Insert at bucket hash(X)+i %N
- apply(X)
  - While bucket hash(X)+i %N is occupied
    - If the element at bucket hash(X)+i %N is X, return it
    - Otherwise i = i + 1
  - Element not found

Open Addressing

- remove(X)
  - While bucket hash(X)+i is occupied
    - If the element at bucket hash(X)+i is X, remove it
    - Otherwise i = i + 1 %N

What about elements that were cascaded ?

Removals Under Open Addressing

- Check each element in a contiguous block, starting at hash(X)
  - Move elements up
    - Don’t move any element Y ahead of hash(Y)

Open Addressing

- **Linear Probing**: Offset to hash(X)+ci for some constant c
- **Quadratic Probing**: Offset to hash(X)+ci^2 for some constant c
- Follow Probing Strategy to find the next bucket

- Runtime Costs
  - Chaining: Dominated by following chain
  - Open Addressing: Dominated by probing
- With a low enough α_{max}, operations still O(1)

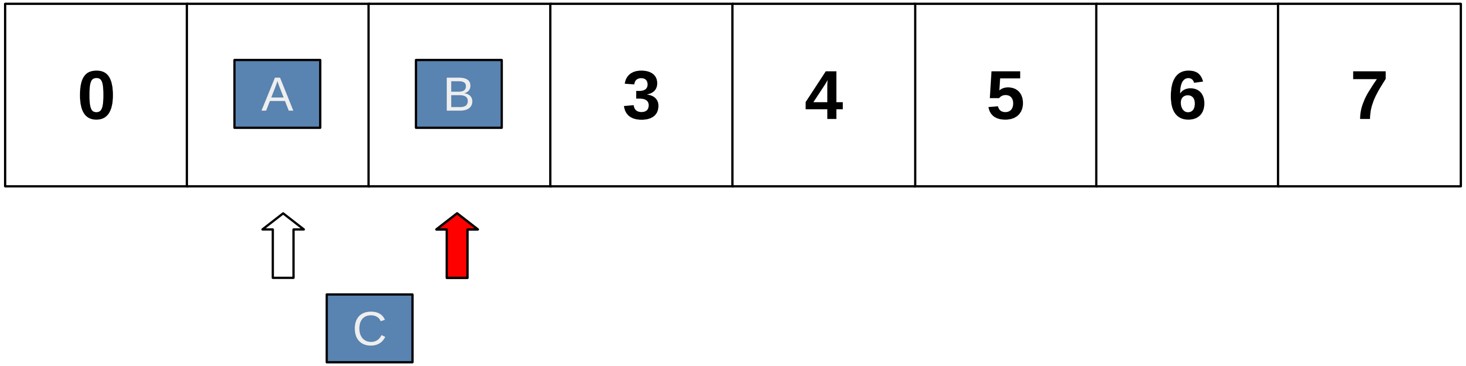
Cuckoo Hashing

- Dynamic Hashing can have arbitrarily long cascade chains
  - Can we reduce the chance of a cascade chain for some operations?

Cuckoo Hashing

- Use two hash functions: $hash_1$, $hash_2$
  - Each record is stored at one of the two
- insert(x)
  - If both buckets are available: pick at random
  - If one bucket is available: insert record there
  - If neither bucket is available, pick one at random
    - "Displace" the record there, move it to the other bucket
    - Repeat displacement until an empty bucket is found

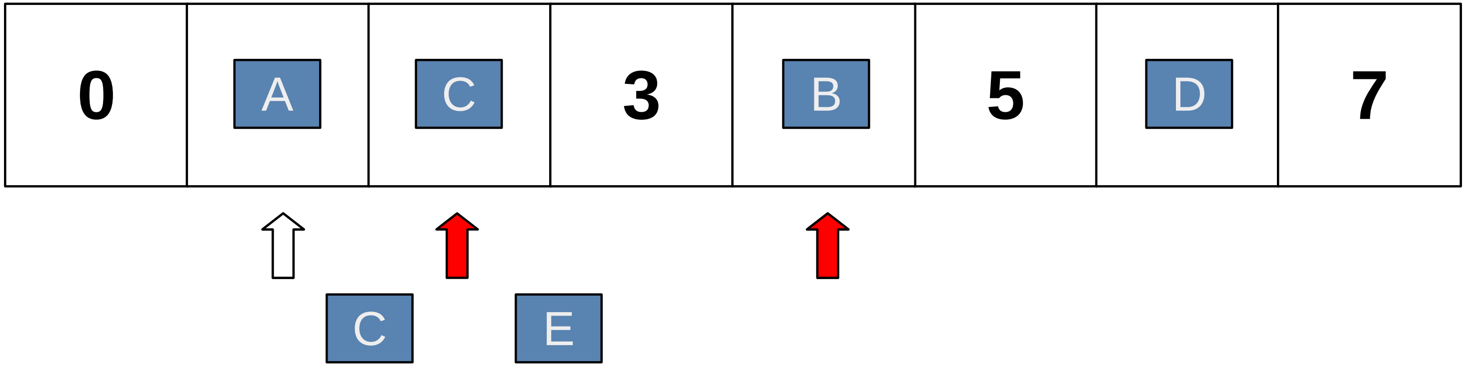
Cuckoo Hashing



$hash_1(A) = 1$
$hash_1(B) = 2$
$hash_1(C) = 2$!
$hash_1(D) = 4$
$hash_1(E) = 3$

$hash_2(A) = 3$
$hash_2(B) = 4$
$hash_2(C) = 1$!
$hash_2(D) = 6$
$hash_2(E) = 3$

Cuckoo Hashing



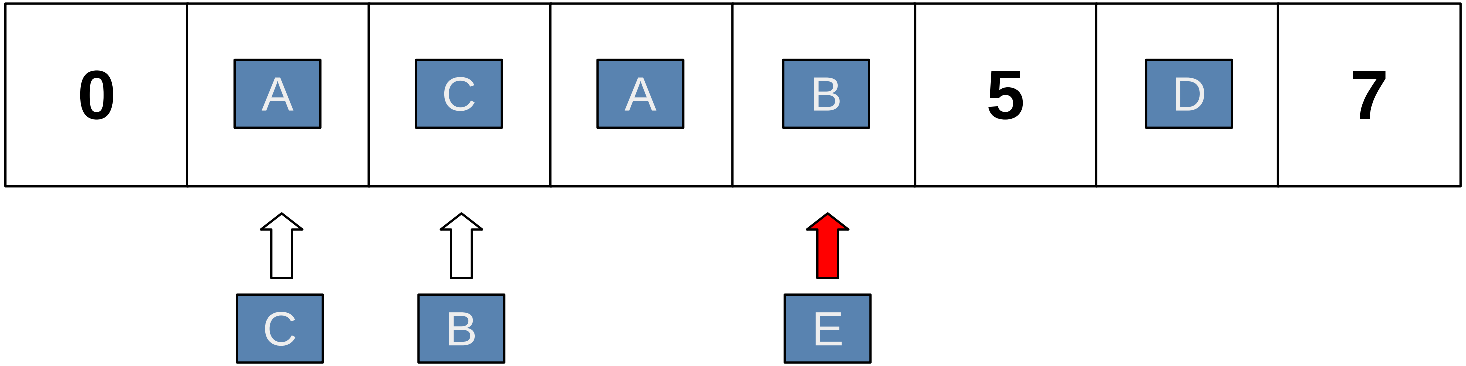
$hash_1(A) = 1$
$hash_1(B) = 2$
$hash_1(C) = 2$
$hash_1(D) = 4$
$hash_1(E) = 1$!

$hash_2(A) = 3$
$hash_2(B) = 4$
$hash_2(C) = 1$
$hash_2(D) = 6$
$hash_2(E) = 4$!

Cuckoo Hashing

$hash_1(A) = 1$
$hash_1(B) = 2$
$hash_1(C) = 2$
$hash_1(D) = 4$
$hash_1(E) = 1$!

$hash_2(A) = 3$
$hash_2(B) = 4$
$hash_2(C) = 1$
$hash_2(D) = 6$
$hash_2(E) = 4$!



Cuckoo Hashing

$hash_1(A) = 1$
$hash_1(B) = 2$
$hash_1(C) = 2$
$hash_1(D) = 4$
$hash_1(E) = 1$ **!**

$hash_2(A) = 3$
$hash_2(B) = 4$
$hash_2(C) = 1$
$hash_2(D) = 6$
$hash_2(E) = 4$ **!**

0	C	B	A	E	**5**	D	**7**

apply(x) and remove(x) is guaranteed O(1)

insert(x) is <u>expected</u> O(1) if α is low enough

Dynamic Hashing

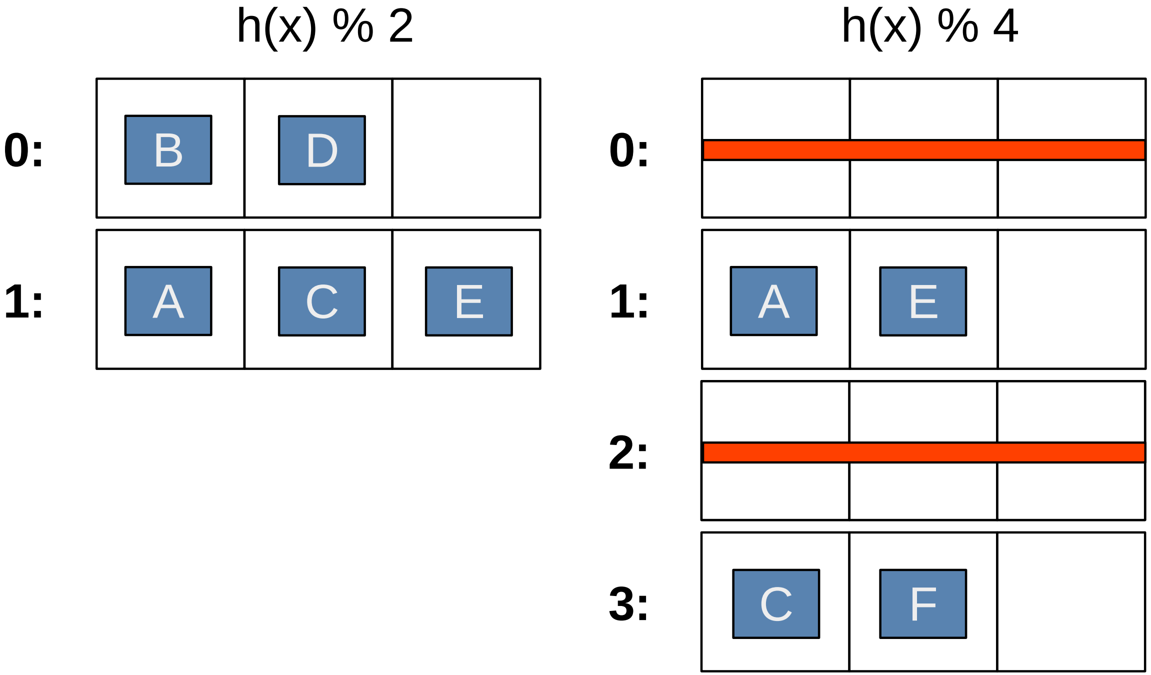
- Rehash is expensive!
  - Amortized cost of rehash is still O(1)
  - ... but every so often everything grinds to a halt!

Dynamic Hashing

- Contrast h(x) % 4 with h(x) % 8
  - e.g. h(x) = 7069; h(x) % 8 = 5
- If we rehash from h(x) % N to h(x) % 2N either:
  - h(x) % 2N = h(x) % N

    or
  - h(x) % 2N = (h(x) % N) + N
- **Idea**: Only rehash "full" buckets
  - An element x can be located at any of the following buckets: h(x) % N   or   h(x) % 2N   or   h(x) % 4N or …

Dynamic Hashing



hash(A) = 1
hash(B) = 6
hash(C) = 3
hash(D) = 4
hash(E) = 9
hash(F) = 7

insert(x) is always O(1)

apply(x), remove(x) are O(log(n))

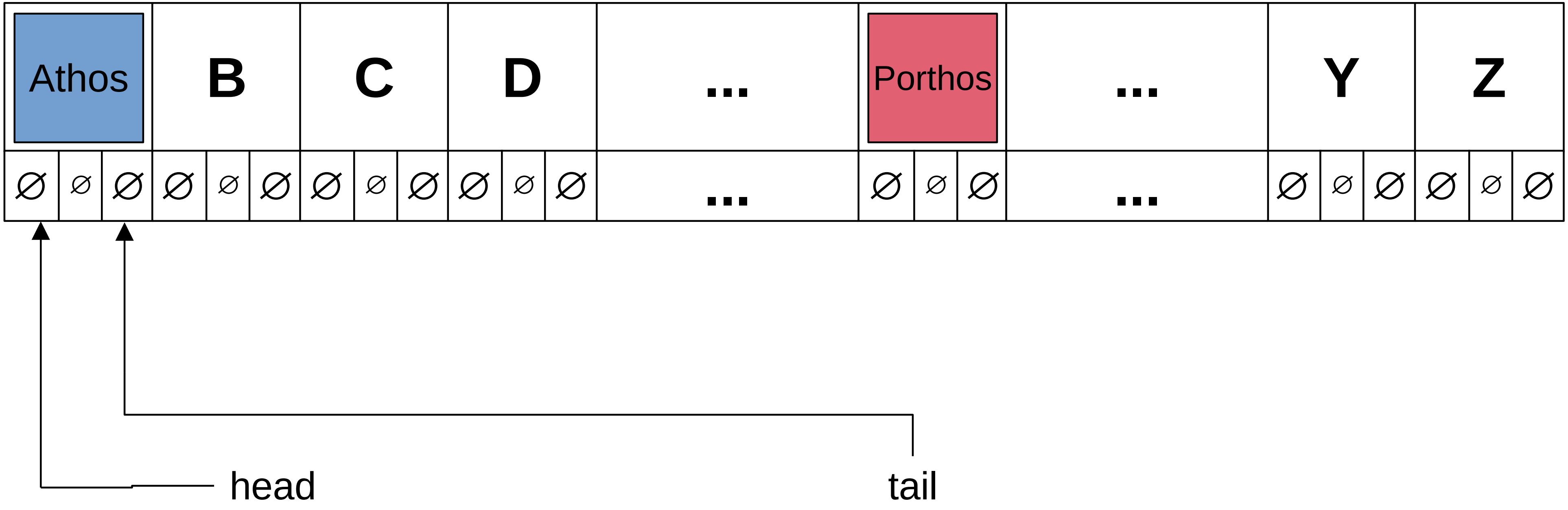
Dynamic Hashing

- Keep log(n) levels
  - Each level i contains hash buckets for h(x) % $2^i \cdot N$
  - Any record will be stored at exactly one level
    - When a level fills up, split its records at the next level
    - When a level empties out, merge with its counterpart
- Keep an array of $2^i \cdot N$ entries
  - Indicate which level h(x)%$2^i \cdot N$ is located at

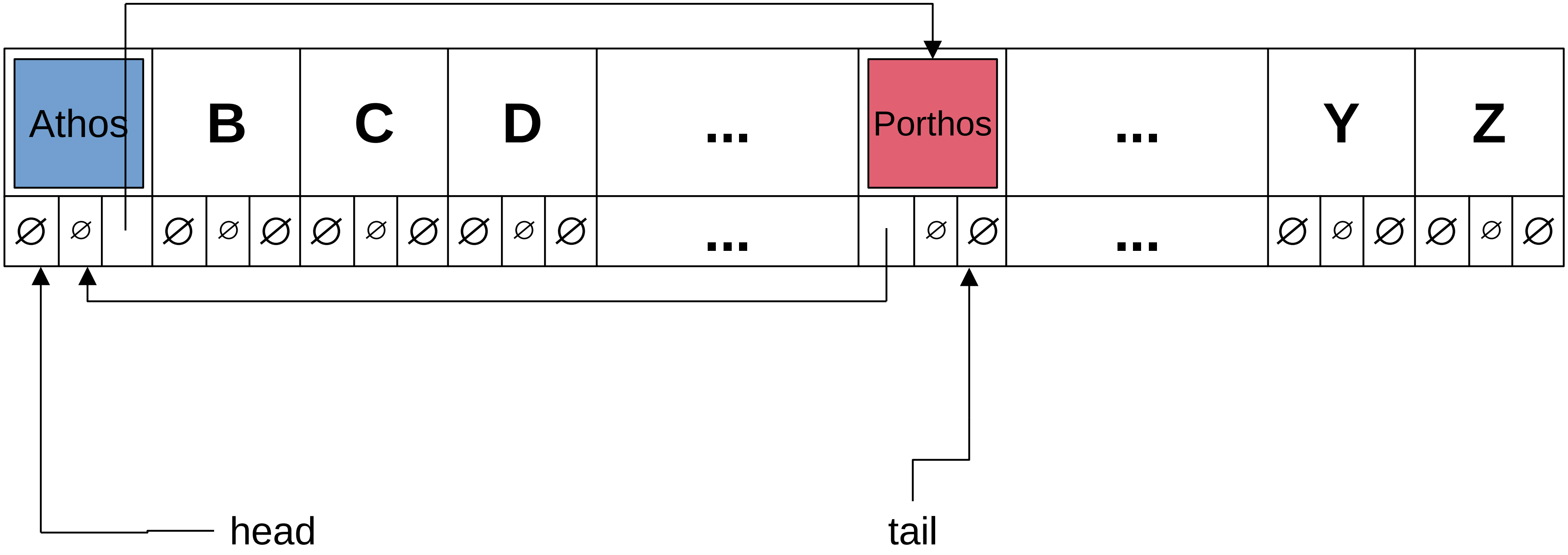
Linked Hash Table

- Iteration over Hash Table is O(N + n)
  - Can be much slower than O(n)
- **Idea**: Connect entries together in a Doubly Linked List

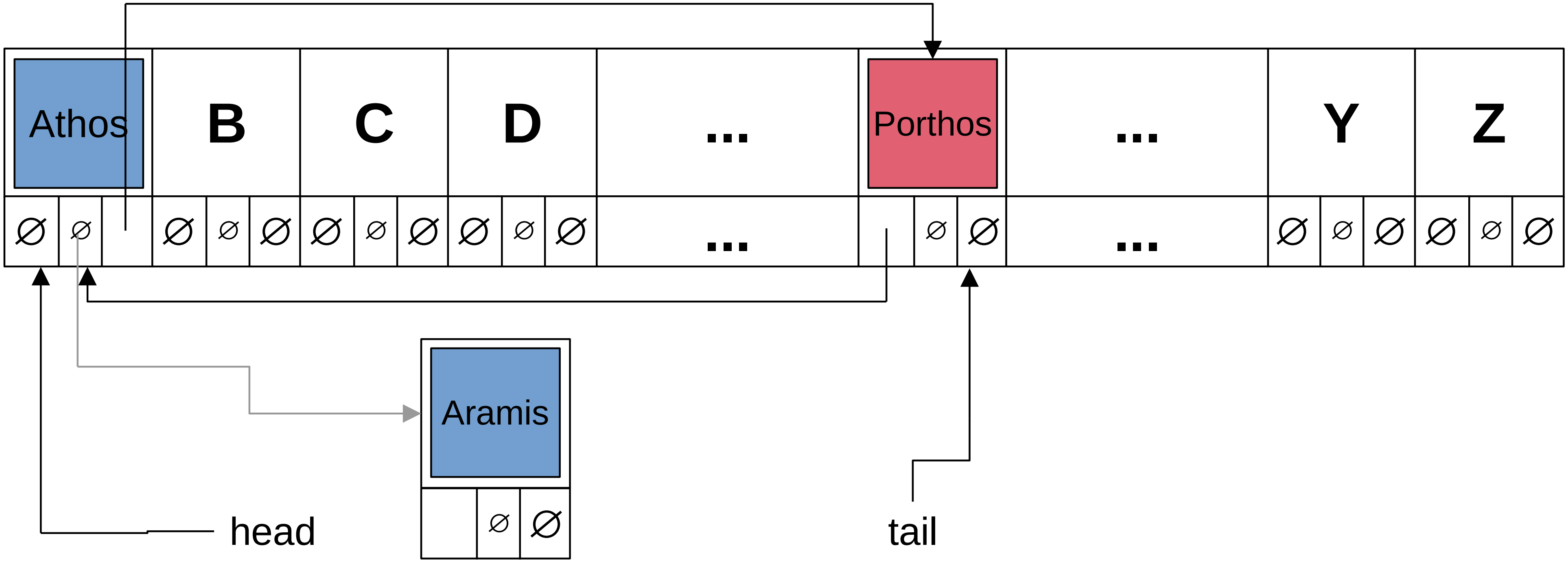
Linked Hash Table



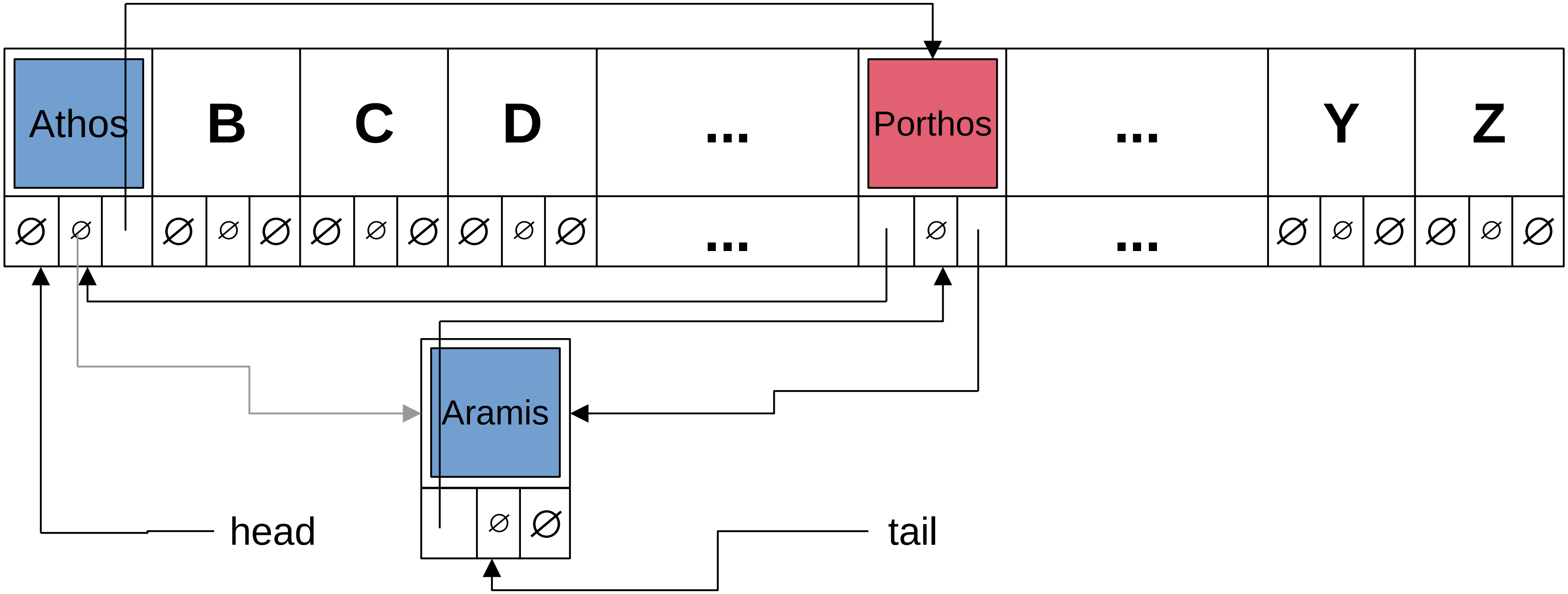
Linked Hash Table



Linked Hash Table



Linked Hash Table



Linked Hash Table

- O(n) Iteration
- apply(x)
  - O(1) increase in cost
- insert(x)
  - O(1) increase in cost
- remove(x)
  - O(1) increase in cost