CH08-320201

Algorithms and Data Structures

ADS

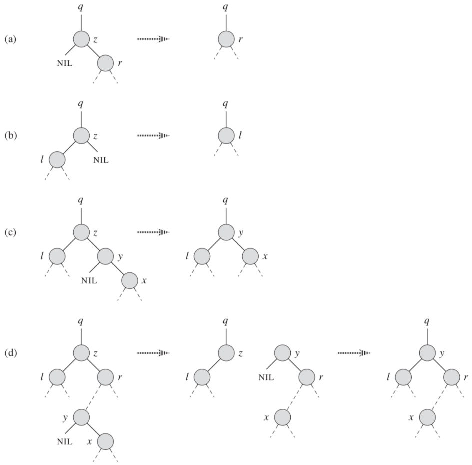
Lecture 16

Dr. Kinga Lipskoch

Spring 2019

Deletion (Remember BST)

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```



Deletion (RB) (1)

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Deletion (RB) (2)

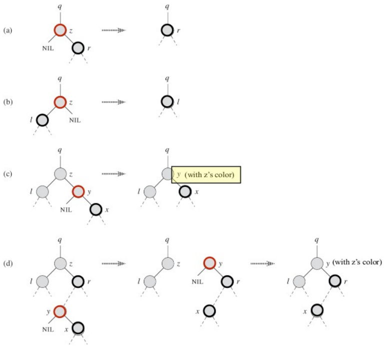
- node y
  - either removed (a/b)
  - or moved in the tree (c/d)
  - y-original-color
- node x
  - the node that moves into y's original position
  - x.p points to y's original parent (since it moves into y's position, note special case in 12/13)

RB-DELETE(T, z)

```
1   y = z
2   y-original-color = y.color
3   if z.left == T.nil
4       x = z.right
5       RB-TRANSPLANT(T, z, z.right)
6   elseif z.right == T.nil
7       x = z.left
8       RB-TRANSPLANT(T, z, z.left)
9   else y = TREE-MINIMUM(z.right)
10      y-original-color = y.color
11      x = y.right
12      if y.p == z
13          x.p = y
14      else RB-TRANSPLANT(T, y, y.right)
15          y.right = z.right
16          y.right.p = y
17      RB-TRANSPLANT(T, z, y)
18      y.left = z.left
19      y.left.p = y
20      y.color = z.color
21  if y-original-color == BLACK
22      RB-DELETE-FIXUP(T, x)
```

Deletion (RB) (3)

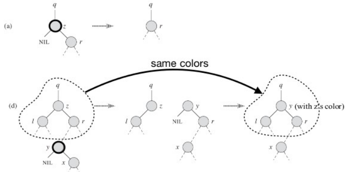
- y-original-color == red




```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Deletion (RB) (4)

- y-original-color == red
  - no problem
- y-original-color == black
  - violations might occur (2,4,5)
  - main idea to fix
    - x gets an "**extra black**" & needs to get rid of it
  - 4 cases



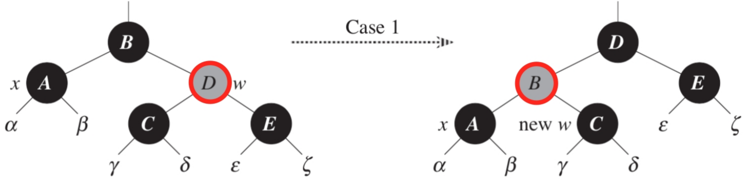
RB-DELETE(T, z)

```
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

Fixing Red-Black Tree Properties (1)

Case 1: x's sibling w is red.

Transform to Case 2, 3, or 4 by left rotation and changing colors of nodes B and D.

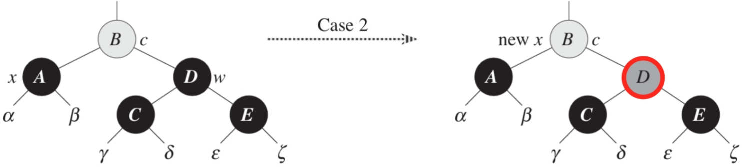


x = node with extra black
w = x's sibling

if $w.color$ == RED
$w.color$ = BLACK
$x.p.color$ = RED
LEFT-ROTATE($T, x.p$)
$w = x.p.right$

Fixing Red-Black Tree Properties (2)

Case 2: x's sibling w is black and the children of w are black. Set color of w to red and propagate upwards.

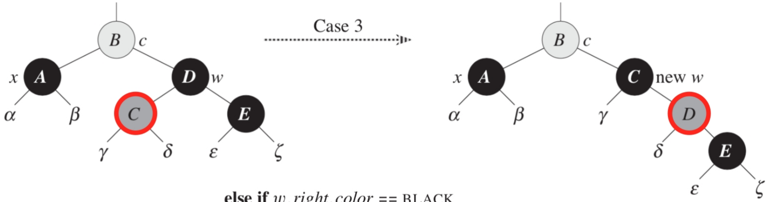


x = node with extra black
w = x's sibling
c = color of the node

if $w.left.color$ == BLACK and $w.right.color$ == BLACK
$w.color = \text{RED}$
$x = x.p$

Fixing Red-Black Tree Properties (3)

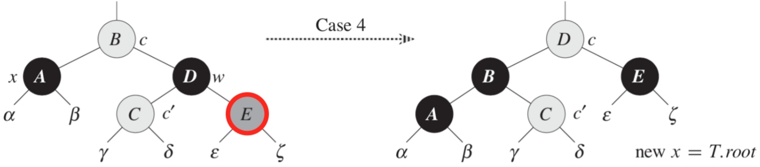
Case 3: x's sibling w is black and the left child of w is red, while the right child of w is black.
Transform to Case 4 by right rotation and changing colors of nodes C and D.




```
else if w.right.color == BLACK
        w.left.color = BLACK
        w.color = RED
        RIGHT-ROTATE(T, w)
        w = x.p.right
```

Fixing Red-Black Tree Properties (4)

Case 4: x's sibling w is black and the right child of w is red. Perform a left-rotate and change colors of B, D, and E. Then, the loop terminates.



$w.color = x.p.color$
$x.p.color =$ BLACK
$w.right.color =$ BLACK
LEFT-ROTATE$(T, x.p)$

Fixing Red-Black Tree Properties (5)

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                  // case 1
            x.p.color = RED                                  // case 1
            LEFT-ROTATE(T, x.p)                              // case 1
            w = x.p.right                                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                    // case 2
            x = x.p                                          // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                         // case 3
                w.color = RED                                // case 3
                RIGHT-ROTATE(T, w)                           // case 3
                w = x.p.right                                // case 3
            w.color = x.p.color                              // case 4
            x.p.color = BLACK                                // case 4
            w.right.color = BLACK                            // case 4
            LEFT-ROTATE(T, x.p)                              // case 4
            x = T.root                                       // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Time complexity: $O(h) = O(\lg n)$

Conclusion

Modifying operations on red-black trees can be executed in $O(\lg n)$ time.

Direct Access Table

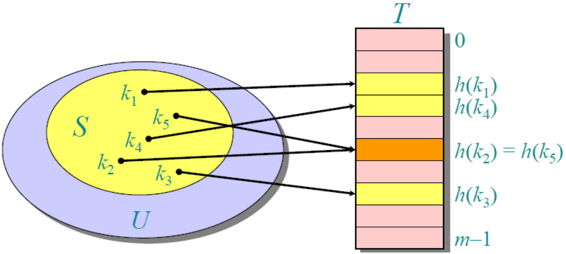
- The idea of a direct access table is that objects are directly accessed via their key.
- Assuming that keys are out of $U = \{0, 1, ..., m-1\}$.
- Moreover, assume that keys are distinct.
- Then, we can set up an array $T[0..m-1]$ with

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } key[x] = k \\ \text{NIL} & \text{otherwise.} \end{cases}$$

- Time complexity: With this set-up, we can have the dynamic-set operations (Search, Insert, Delete, ...) in $\Theta(1)$.
- Problem: m is often large. For example, for 64-bit numbers we have $18,446,744,073,709,551,616$ different keys.

Hash Function

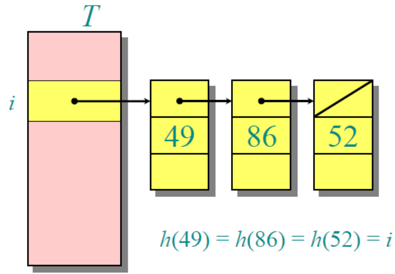
- Use a function h that maps U to a smaller set $\{0, 1, ..., m-1\}$.



- Such a function is called a hash function.
- The table T is called a hash table.
- If two keys are mapped to the same location, we have a collision.

Resolving Collisions

- Collisions can be resolved by storing the colliding mappings in a (singly-)linked list.



- Worst case: All keys are mapped to the same location. Then, access time is $\Theta(n)$.

Average Case Analysis (1)

- Assumption (simple uniform hashing): Each key is equally likely to be hashed to any slot of the table, independent of where other keys are hashed.
- Let n be the number of keys.
- Let m be the number of slots.
- The load factor $\alpha = n/m$ represents the average number of keys per slot.

Average Case Analysis (2)

Theorem:
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.
Proof:

- Any key k not already stored in the table is equally likely to hash to any of the m slots.
- The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$.
- Expected length of the list is $E[n_{h(k)}] = \alpha$.
- Time for computing $h(k) = O(1) \Rightarrow$ overall time $\Theta(1+\alpha)$.

Average Case Analysis (3)

- Runtime for unsuccessful search:
  The expected time for an unsuccessful search is $\Theta(1+\alpha)$ including applying the hash function and accessing the slot and searching the list.
- What does this mean?
  - $m \sim n$, i.e., if $n = O(m) \Rightarrow \alpha = n/m = O(m)/m = O(1)$
  - Thus, search time is $O(1)$
- A successful search has the same asymptotic bound.

Choosing a Hash Function (1)

- What makes a good hash function?
  - The goal for creating a hash function is to distribute the keys as uniformly as possible to the slots.
- Division method
  - Define hashing function $h(k) = k \bmod m$.
  - Deficiency: Do not pick an m that has a small divisor d, as a prevalence of keys with the same modulo d can negatively effect uniformity.
  - Example: if m is a power of 2, the hash function only depends on a few bits: If $k = 1011000111011010$ and $m = 2^6$, then $h(k) = 011010$.

Choosing a Hash Function (2)

- Division method (continue)
  - Common choice: Pick m to be a prime not too close to a power of 2 or 10 and not otherwise prominently used in computing environments.
  - Example: $n = 2000$; we are ok with average 3 elements in our collision chain $\Rightarrow m = 701$ (a prime number close to $2000/3$), $h(k) = k \bmod 701$.

Choosing a Hash Function (3)

- Multiplication method
  - Assume all keys are integers, $m = 2^r$, and the computer uses w-bit words.
  - Define hash function $h(k) = (A \cdot k \bmod 2^w) >> (w - r)$, where ">>" is the right bit-shift operator and A is an odd integer with $2^{w-1} < A < 2^w$.
  - Note that these operations are faster than divisions.
  - Example: $m = 2^3 = 8$ and $w = 7$.

$$\begin{array}{lr l} & 1\,0\,1\,1\,0\,0\,1 & = A \\ \times & 1\,1\,0\,1\,0\,1\,1 & = k \\ \hline & 1\,0\,0\,1\,0\,1\,0\,\underbrace{0\,1\,1}_{h(k)}\,0\,0\,1\,1 & \end{array}$$

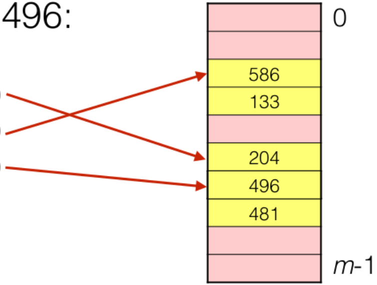
Resolving Collisions by Open Addressing

- No additional storage is used.
- Only store one element per slot.
- Insertion probes the table systematically until an empty slot is found.
- The hash function depends on the key and the probe number, i.e., $h : U \times \{0, 1, ..., m-1\} \to \{0, 1, ..., m-1\}$.
- The probe sequence $< h(k,0), h(k,1), ..., h(k,m-1) >$ should be a permutation of $\{0, 1, ..., m-1\}$.

Insert Example

- Insert key k = 496:

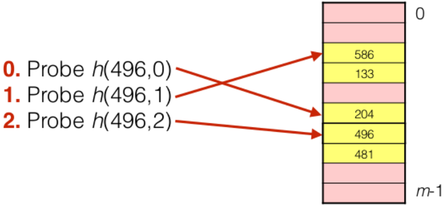
0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$




```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

Search Example

```
HASH-SEARCH(T, k)
  i = 0
  repeat
      j = h(k, i)
      if T[j] == k
          return j
      i = i + 1
  until T[j] == NIL or i == m
  return NIL
```



- Search key $k = 496$
  - Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot (or made it all the way through the list)
- What about delete?
  - Have a special node type: DELETED
  - Note though: search times no longer depend on load factor α
  - Chaining more commonly used when keys must also be deleted

Probing Strategies (1)

Linear probing:

- Given an ordinary hash function $h'(k)$, linear probing uses the hash function $h(k,i) = (h'(k) + i) \bmod m$.
- This is a simple computation.
- However, it may suffer from primary clustering, where long runs of occupied slots build up and tend to get longer.
  - empty slot preceded by i full slots gets filled next with probability $(i+1)/m$

Probing Strategies (2)

Quadratic probing:

- Quadratic probing uses the hash function $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$.
- Offset by amount that depends on quadratic manner, works much better than linear probing
- But, it may still suffer from secondary clustering: If two keys have initially the same value, then they also have the same probe sequence
- In addition c_1, c_2, and m need to be constrained to make full use of the hash table

Probing Strategies (3)

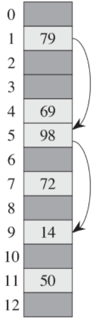
Double hashing:

- Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$.
- The initial probe goes to position $T[h_1(k)]$; successive probe positions are offset by $h_2(k) \rightarrow$ the initial probe position, the offset, or both, may vary
- This method generates excellent results, if $h_2(k)$ is "relatively prime" to the hash-table size m,

Probing Strategies (4)

Double hashing (continue):

- e.g., by making m a power of 2 and design $h_2(k)$ to only produce odd numbers.
- or let m be prime and design h_2 such that it always returns a positive integer less than m, e.g. let m' be slightly less than m:
  $h_1(k) = k \bmod m$
  $h_2(k) = 1 + (k \bmod m')$



$h_1(k) = k \bmod 13$
$h_2(k) = 1 + (k \bmod 11)$

—> k=14; $h_1(k)=1$, $h_2(k)=4$

—> k=27; $h_1(k)=1$, $h_2(k)=6$

Analysis of Open Addressing (1)

Theorem:

- Assume uniform hashing, i.e., each key is likely to have any one of the $m!$ permutations as its probe sequence.
- Given an open-addressed hash table with load factor $\alpha = n/m < 1$.
- The expected number of probes in an unsuccessful search is, at most, $1/(1 - \alpha)$.

Analysis of Open Addressing (2)

Proof:

- At least, one probe is always necessary.
- With probability n/m, the first probe hits an occupied slot, i.e., a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, i.e., a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, i.e., a fourth probe is necessary.
- ...

Analysis of Open Addressing (3)

Given that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$.

$$1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}\left(1 + \frac{n-2}{m-2}\left(\cdots\left(1 + \frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$

$$\leq 1 + \alpha(1 + \alpha(1 + \alpha(\cdots(1+\alpha)\cdots)))$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

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

Analysis of Open Addressing (4)

- The successful search takes less number of probes [expected number is $1/\alpha \ln 1/(1-\alpha)$].
- We conclude that if α is constant, then accessing an open-addressed hash table takes constant time.
- For example, if the table is half full, the expected number of probes is $1/(1-0.5) = 2$.
- Or, if the table is 90% full, the expected number of probes is $1/(1-0.9) = 10$.

Summary

- Dynamic sets with queries and modifying operations.
- Array: Random access, search in $O(\lg n)$, but modifying operations $O(n)$.
- Stack: LIFO only. Operations in $O(1)$.
- Queue: FIFO only. Operations in $O(1)$.
- Linked list: Modifying operations in $O(1)$, but search $O(n)$.
- BST: All operations in $O(h)$.
- Red-black trees: All operations in $O(\lg n)$.
- Heap: All operations in $O(\lg n)$.
- Hash tables: Operations in $O(1)$, but additional storage space.