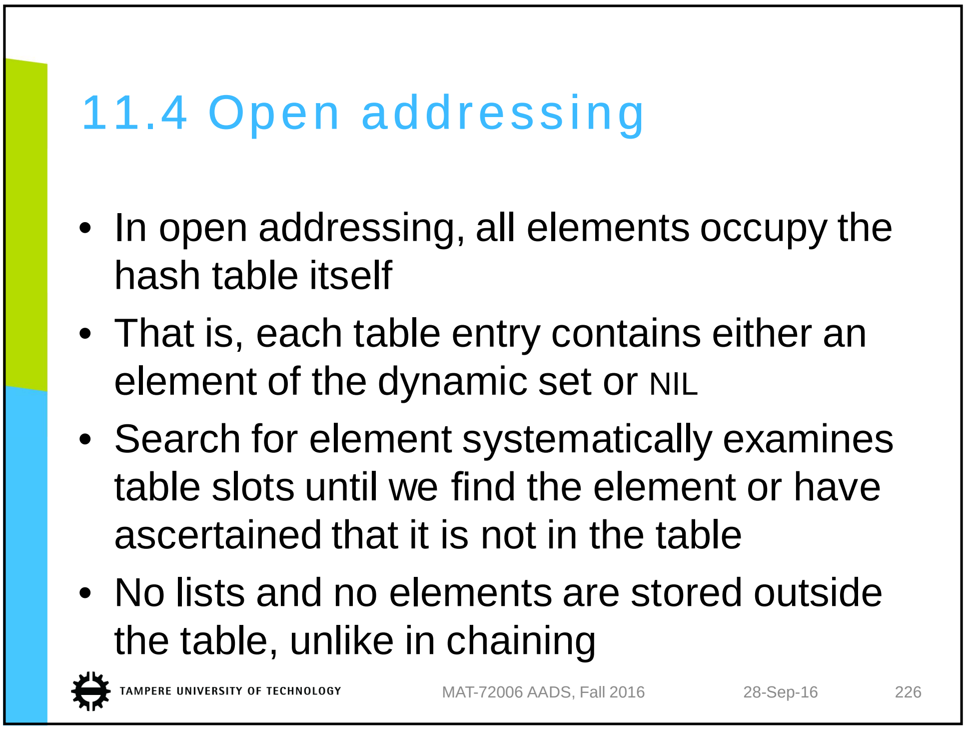
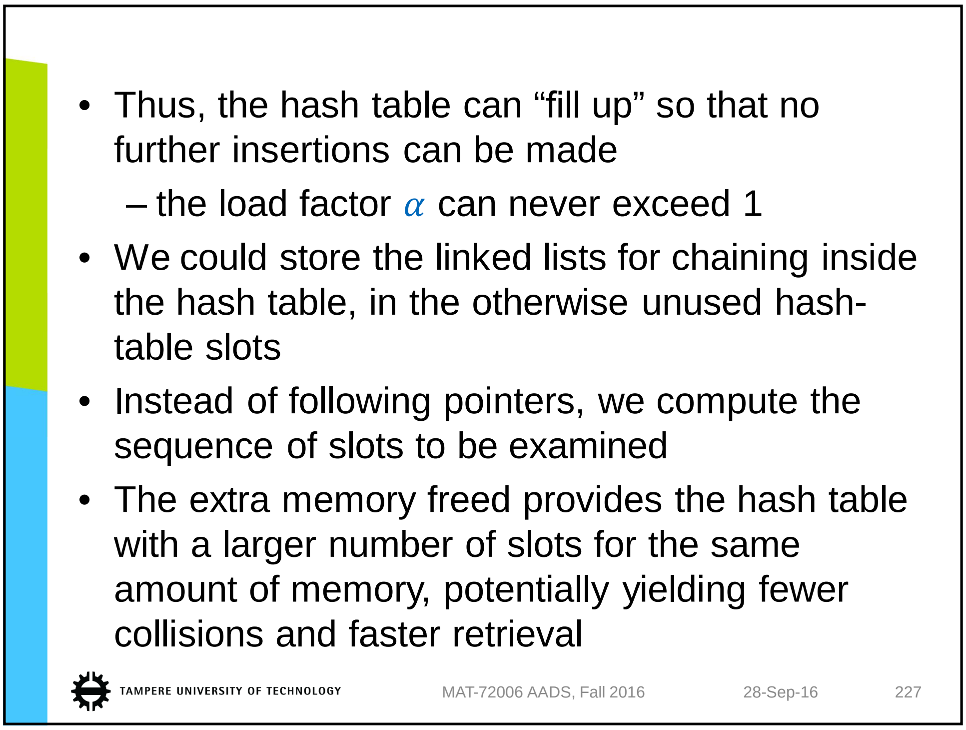


11.4 Open addressing

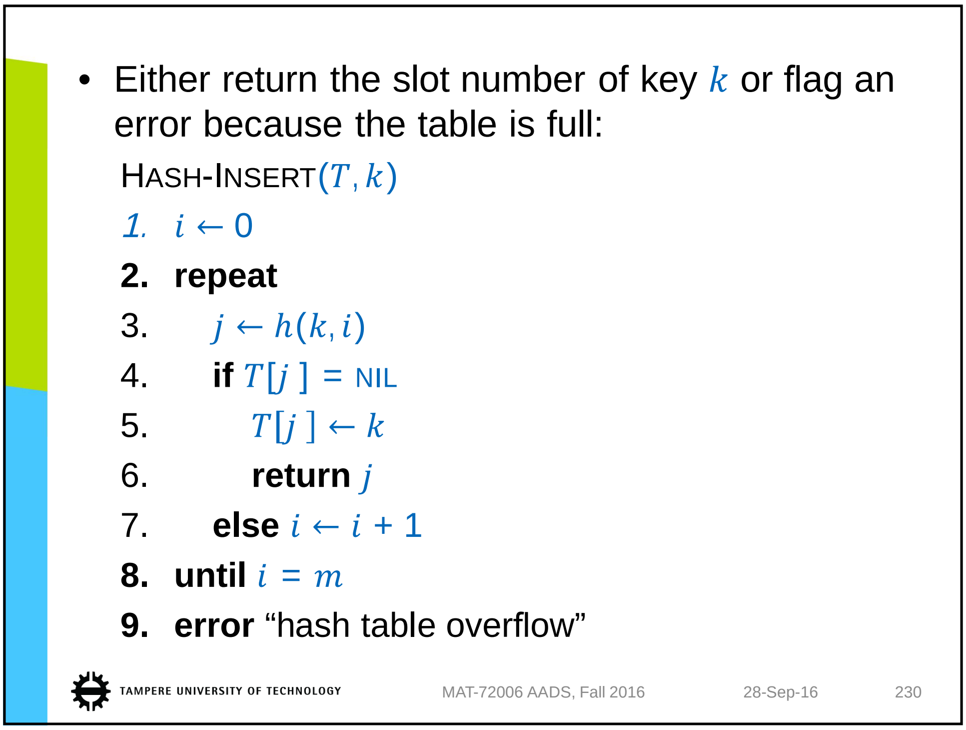
- In open addressing, all elements occupy the hash table itself
- That is, each table entry contains either an element of the dynamic set or NIL
- Search for element systematically examines table slots until we find the element or have ascertained that it is not in the table
- No lists and no elements are stored outside the table, unlike in chaining



- Thus, the hash table can “fill up” so that no further insertions can be made
  - the load factor α can never exceed 1
- We could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots
- Instead of following pointers, we compute the sequence of slots to be examined
- The extra memory freed provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval

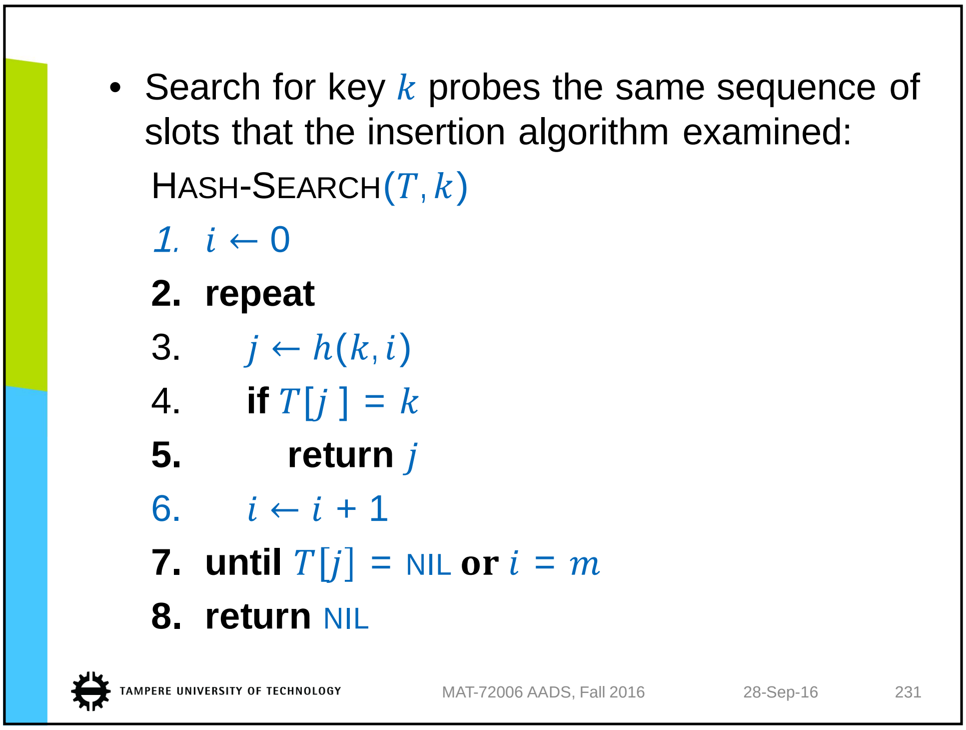
- To perform insertion, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key
- We are not being fixed in order $0,1,\ldots,m-1$, which requires $\Theta(n)$ search time
- Rather, the sequence of positions probed depends upon the key being inserted
- To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input

- Thus, the hash function becomes
  $$h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$$
- With open addressing, we require that for every key k, the probe sequence
  $$h(k,0), h(k,1), \ldots, h(k,m-1)$$
  be a permutation of $\langle 0,1,\ldots,m-1\rangle$
- Every position is eventually considered as a slot for a new key as the table fills up
- Let us assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k



- Either return the slot number of key k or flag an error because the table is full:

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



- Search for key k probes the same sequence of slots that the insertion algorithm examined:

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

- When we delete a key from slot i, we cannot simply mark it as empty by storing NIL in it
  - We might be unable to retrieve any key k during whose insertion we had probed slot i
- Instead we mark the slot with value DELETED
- Modify HASH-INSERT to treat such a slot as empty so that we can insert a new key there
- HASH-SEARCH passes over DELETED values
- When we use DELETED value, search times no longer depend on the load factor α
- Therefore chaining is more commonly selected as a collision resolution technique

- We assume ***uniform hashing*** (UH):
  - the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0,1,\ldots,m-1\rangle$
- UH generalizes the notion of SUH that produces not just a single number, but a whole probe sequence
- True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used

- We examine three common techniques to compute the probe sequences required for open addressing: 1) linear probing, 2) quadratic probing, and 3) double hashing
- These techniques all guarantee that $\langle h(k,0), h(k,1), \dots, h(k,m-1)\rangle$ is a permutation of $\langle 0,1,\dots,m-1\rangle$ for each key k
- No technique fulfills the assumption of UH;
  - none of them is capable of generating more than m^2 different probe sequences
- Double hashing has the greatest number of probe sequences and gives the best results

Linear probing

- Given a hash function $h' : U \to \{0,1,\dots,m-1\}$, an ***auxiliary hash function***, use the function

$$h(k,i) = (h'(k) + i) \bmod m$$

- Given key k, we first probe the slot given by the auxiliary hash function $T[h'(k)]$
- We next probe slots $T[h'(k)+1], \dots, T[m-1]$
- Wrap around to $T[0], T[1], \dots, T[h'(k)-1]$
- Initial probe determines the entire probe sequence, there are only m distinct sequences

- Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***
- Long runs of occupied slots build up, increasing the average search time
- Clusters arise because an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$
- Long runs of occupied slots tend to get longer, and the average search time increases

Quadratic probing

- Use a hash function of the form

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

where c_1, c_2 are positive auxiliary constants
- Initial position probed is $T[h'(k)]$; later positions are offset by amounts that depend in a quadratic manner on the probe number i
- This works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2, and m are constrained

- Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$
- This property leads to a milder form of clustering, called ***secondary clustering***
- As in linear probing, the initial probe determines the entire sequence, and so only m distinct probe sequences are used

Double hashing

- One of the best methods available for open addressing, the permutations produced have many characteristics of random ones
- Uses a hash function of the form

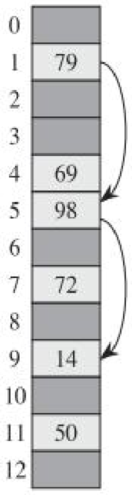
$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

  where h_1 and h_2 are auxiliary hash functions
- The initial probe goes to position $T[h_1(k)]$; successive probes are offset from previous positions by the amount $h_2(k)$, modulo m

- Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$
- Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot

$$h_1(k) + 2h_2(k) = 9,$$

after examining slots $h_1(k) = 1$ and $h_1(k) + h_2(k) = 5$ and finding them to be occupied



- The value $h_2(k)$ must be relatively prime to m for the entire hash table to be searched
- A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number
- Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m
- For example, we could choose m prime and let $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$, where m' is slightly less than m (say, $m - 1$)

- E.g., if $k = 123{,}456$, $m = 701$, and $m' = 700$, we have $h_1(k) = 80$ and $h_2(k) = 257$
- We first probe position 80, and then we examine every 257th slot (modulo m) until we find the key or have examined every slot
- When m is prime or a power of 2, double hashing improves over linear or quadratic probing in that $\Theta(m^2)$ probe sequences are used, rather than $\Theta(m)$
- Each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence

Analysis of open-address hashing

- Let us express our analysis of in terms of the load factor $\alpha = n/m$ of the hash table
- Now at most one element occupies each slot, and thus $n \leq m$, which implies $\alpha \leq 1$
- Assume that we are using uniform hashing
- In this idealized scheme, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0,1, \ldots, m-1\rangle$

Theorem 11.6 *Given an open-address hash table with* $\alpha = n/m < 1$, *the expected number of probes in an* ***unsuccessful*** *search is at most* $1/(1-\alpha)$, *assuming UH.*

Proof Every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty.

Define the random variable X to be the number of probes made in an unsuccessful search, and also define the event A_i, $i = 1, 2, \ldots$, to be the event that an ith probe occurs and it is to an occupied slot.

The event $\{X \geq i\}$ is the intersection of events $A_1 \cap A_2 \cap \cdots \cap A_{i-1}$.

Bound $\Pr\{X \geq i\}$ by $\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$ which by Exercise C.2-5

$$\Pr\{A_{i-1} | A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}$$
$$= \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \cdots$$

There are n elements and m slots, so

$$\Pr\{A_1\} = n/m$$

For $j > 1$, the probability that there is a jth probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is

$$(n-j+1)/(m-j+1)$$

This probability follows because we would be finding one of the remaining $(n - j + 1)$ elements in one of the $(m - j + 1)$ unexamined slots, and by the assumption of UH, the probability is the ratio of these quantities.

$n < m$ implies that $(n - j)/(m - j) \leq n/m$ for all $0 \leq j < m$.

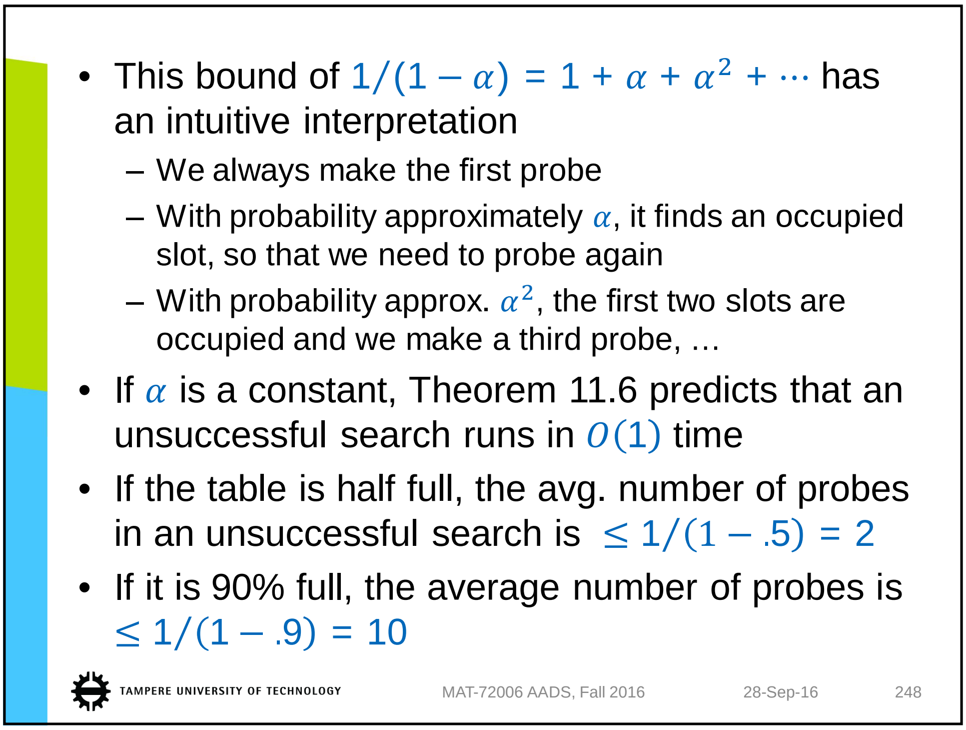
Therefore, we have for all $1 \leq i \leq m$,

$$\begin{aligned} \Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1} \end{aligned}$$

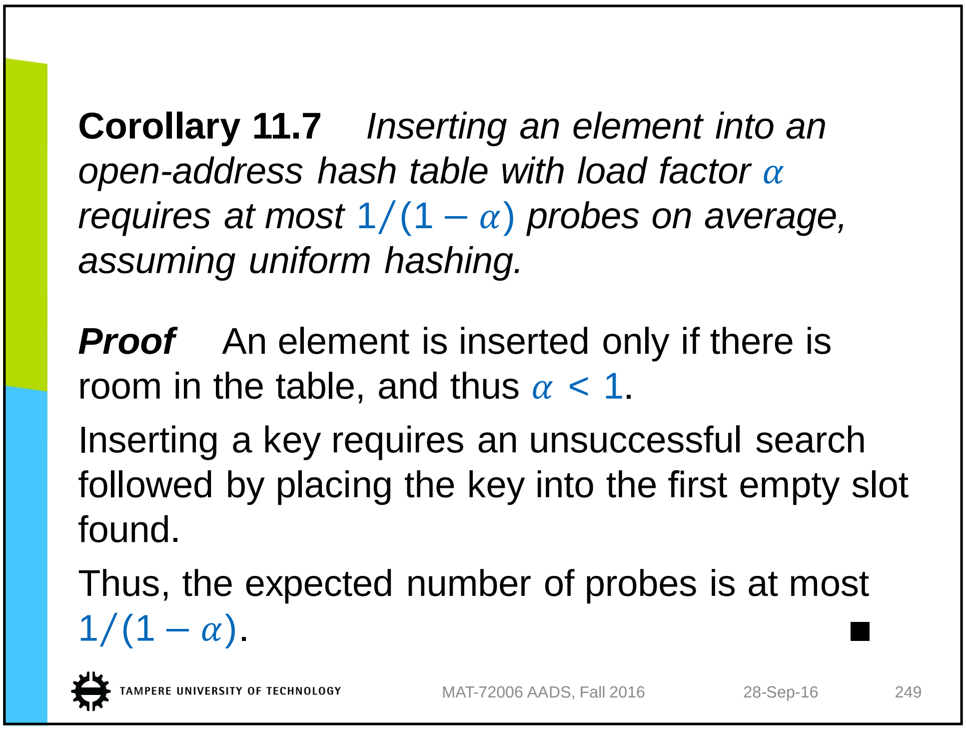
- Now, because $\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} \end{aligned}$$

■



- This bound of $1/(1-\alpha) = 1 + \alpha + \alpha^2 + \cdots$ has an intuitive interpretation
  - We always make the first probe
  - With probability approximately α, it finds an occupied slot, so that we need to probe again
  - With probability approx. α^2, the first two slots are occupied and we make a third probe, …
- If α is a constant, Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time
- If the table is half full, the avg. number of probes in an unsuccessful search is $\leq 1/(1-.5) = 2$
- If it is 90% full, the average number of probes is $\leq 1/(1-.9) = 10$



Corollary 11.7 *Inserting an element into an open-address hash table with load factor* α *requires at most* $1/(1-\alpha)$ *probes on average, assuming uniform hashing.*

Proof An element is inserted only if there is room in the table, and thus $\alpha < 1$.

Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found.

Thus, the expected number of probes is at most $1/(1-\alpha)$. ■

Theorem 11.8 *Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a* ***successful*** *search is at most*

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

assuming UH and assuming that each key in the table is equally likely to be searched for.

- If the hash table is half full, the expected number of probes in a successful search is < 1.387
- If the hash table is 90 percent full, the expected number of probes is < 2.559

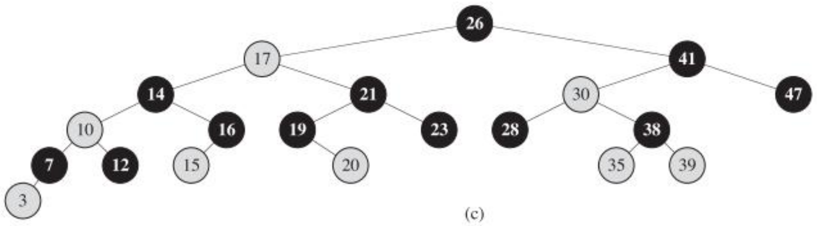
---

13 Red-Black Trees

- A red-black tree (RBT) is a BST with one extra bit of storage per node: color, either **RED** or **BLACK**
- Constraining the node colors on any path from the root to a leaf
  - Ensures that no such path is more than twice as long as any other, so that the tree is approximately balanced

- A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either **RED** or **BLACK**
2. The root is **BLACK**
3. Every leaf (NIL) is **BLACK**
4. If a node is **RED**, then both its children are **BLACK**
5. For each node, all simple paths from the node to descendant leaves contain the same number of **BLACK** nodes



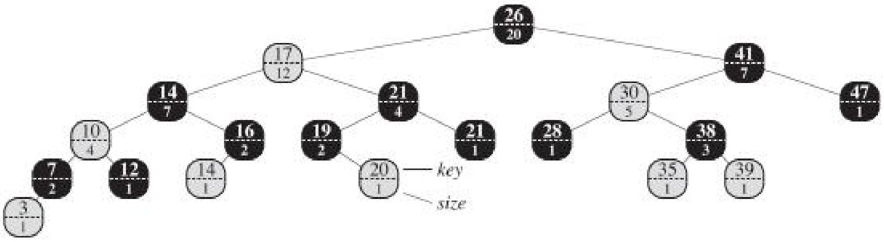
(c)

14 Augmenting Data Structures

- Some engineering situations require more than a “textbook” data structure
- Usually it suffices to augment a textbook data structure by storing additional information in it
- You can then program new operations for the data structure to support the desired application
- The added information must be updated and maintained by the ordinary operations on the data structure

14.1 Dynamic order statistics

- Let us see how to modify red-black trees (RBTs) so that we can determine any order statistic for a dynamic set in $O(\lg n)$ time
- We shall also see how to compute the ***rank*** of an element—its position in the linear order of the set—in $O(\lg n)$ time
- An ***order-statistic tree*** T is simply a red-black tree with additional information stored in each node



- Besides the usual RBT attributes $x.key$, $x.color$, $x.p$, $x.left$, and $x.right$ in a node x, we have another attribute,

$$x.size$$

- This attribute contains the number of (internal) nodes in the subtree rooted at x (including x itself), that is, the size of the subtree
- If we define the sentinel's size to be 0—that is, we set $T.nil.size$ to be 0—then we have the identity

$$x.size = x.left.size + x.right.size + 1$$

- We do not require keys to be distinct in an order-statistic tree
- In the presence of equal keys, the above notion of rank is not well defined
- We remove this ambiguity for an order-statistic tree by defining the rank of an element as the position at which it would be printed in an inorder walk of the tree
- In previous figure, e.g., the key 14 stored in a black node has rank 5, and the key 14 stored in a red node has rank 6

Retrieving an element with a given rank

- Let us begin with an operation that retrieves an element with a given rank
- Procedure OS-SELECT(x, i) returns a pointer to the node containing the ith smallest key in the subtree rooted at x
- To find the node with the ith smallest key in an order-statistic tree T, we call

$$\text{OS-SELECT}(T.root, i)$$

```
OS-SELECT$(x, i)$
$r \leftarrow x.left.size + 1$
if $i = r$
    return $x$
elseif $i < r$
    return OS-SELECT$(x.left, i)$
else return OS-SELECT$(x.right, i - r)$
```

- Consider a search for the 17th smallest element in the order-statistic tree of previous figure
- First, x is the root, whose key is 26, and $i = 17$
- The size of 26's left subtree is 12, its rank is 13
- Thus, the node with rank 17 is the $17 - 13 = 4$th smallest element in 26's right subtree
- Now, x becomes the node with key 41, and $i = 4$
- The size of 41's left subtree is 5, its rank within its subtree is 6
- Thus, the node with rank 4 is the 4th smallest element in 41's left subtree

- After the recursive call, x is the node with key 30, and its rank within its subtree is 2
- Thus, we recurse once again to find the $4 - 2 = 2$nd smallest element in the subtree rooted at the node with key 38
- We now find that its left subtree has size 1, which means it is the second smallest element
- Thus, the procedure returns a pointer to the node with key 38

- Because each recursive call goes down one level in the order-statistic tree, the total time for OS-SELECT is at worst proportional to the height of the tree
- Since the tree is a red-black tree, its height is $O(\lg n)$, where n is the number of nodes
- Thus, the running time of OS-SELECT is $O(\lg n)$ for a dynamic set of n elements

Determining the rank of an element

- OS-RANK returns the position of x in the linear order determined by an inorder tree walk of T

```
OS-RANK($T, x$)
$r \leftarrow x.left.size + 1$
$y \leftarrow x$
while $y \neq T.root$
    if $y = y.p.right$
        $r \leftarrow r + y.p.left.size + 1$
    $y \leftarrow y.p$
return $r$
```

- E.g., when we run OS-RANK to find the rank of key 38, we get the following sequence of values of at the top of the while loop:

iteration	$y.key$	r
1	38	2
2	30	4
3	41	4
4	26	17

- The procedure returns the rank 17
- the running time of OS-RANK is at worst proportional to the height of the tree: $O(\lg n)$ on an n-node order-statistic tree

Maintaining subtree sizes

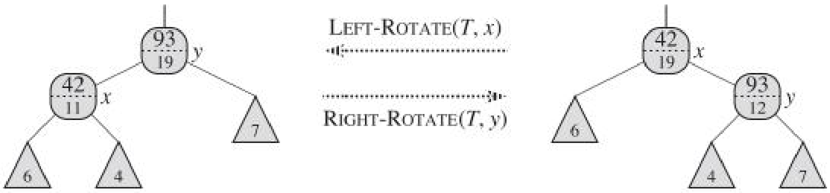
- Given the size attribute in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information
- We must be able to efficiently maintain these attributes within the basic modifying operations on red-black trees
- Let us see how to maintain subtree sizes for both insertion and deletion without affecting the asymptotic running time of either operation

---

- Insertion into a RBT consists of two phases
  - The first goes down the tree, inserting the new node as a child of an existing one
  - The second phase goes up the tree, changing colors and performing rotations to maintain the RB properties
- To maintain the subtree sizes in the first phase, we increment $x.size$ for each node x on the simple path traversed
- The new node added gets a size of 1
- Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the size attributes is $O(\lg n)$

- In the second phase, the only structural changes to the underlying RBT are caused by rotations, of which there are at most two
- Moreover, a rotation is a local operation: only two nodes have their size attributes invalidated
- The link around which the rotation is performed is incident on these two nodes
- Referring to the code for LEFT-ROTATE(T, x), we add the following lines:

```
13 $y.size \leftarrow x.size$
14 $x.size \leftarrow x.left.size + x.right.size + 1$
```



- Since at most two rotations are performed during insertion into a RBT, we spend only $O(1)$ additional time updating size attributes in the second phase
- Thus, the total time for insertion into an n-node order-statistic tree is $O(\lg n)$, which is asymptotically the same as for an ordinary RBT

- Deletion also consists of two phases:
  - the first operates on the underlying search tree
  - the second causes at most three rotations and otherwise performs no structural changes
- The first phase either removes one node y from the tree or moves it upward within the tree
- To update the subtree sizes, we simply traverse a simple path from node y (starting from its original position) up to the root, decrementing the $size$ attribute of each node on the path

- Since this path has length $O(\lg n)$ in an n-node red-black tree, the additional time spent maintaining $size$ attributes in the first phase is $O(\lg n)$
- We handle the $O(1)$ rotations in the second phase of deletion in the same manner as for insertion
- Thus, both insertion and deletion, including maintaining the $size$ attributes, take $O(\lg n)$ time for an n-node order-statistic tree