EL9343

Data Structure and Algorithm

Lecture 6: Hash Tables, Binary Search Tree

Instructor: Pei Liu

Tentative for Midterm

- Midterm
  - Tentatively on the morning March 27
- I will briefly review topics for midterm next week
- Today's lecture will be in the midterm
- Next week's lecture will NOT be in the midterm
- No lecture on the Wednesday before the midterm.

The Search Problem

- Find items with keys matching a given search key in a dynamic set of data records
  - Given an array A, containing n records, and a search key x, find the index i such as x=key(A[I])
  - As in the case of sorting, a key could be part of a large record.

example of a record

Key	**other data**

Special Case: Dictionaries

- Dynamic data sets includes list/tree/array/queue
- **Dictionary:** Abstract Data Type (ADT) — maintain a set of items, each with a key, subject to
  - Insert(item): add item to set
  - Delete(item): remove item from set
  - Search(key): return item with key if it exists

Applications

- Keeping track of customer account information at a bank
  - Search through records to check balances and perform transactions
- Search engine
  - Looks for all documents containing a given word
- ...

Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe U = {0, 1, . . . , m - 1}
- Idea:
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NIL

Direct Addressing: Operations

Alg.: DIRECT-ADDRESS-SEARCH(T, k)

return T[k]

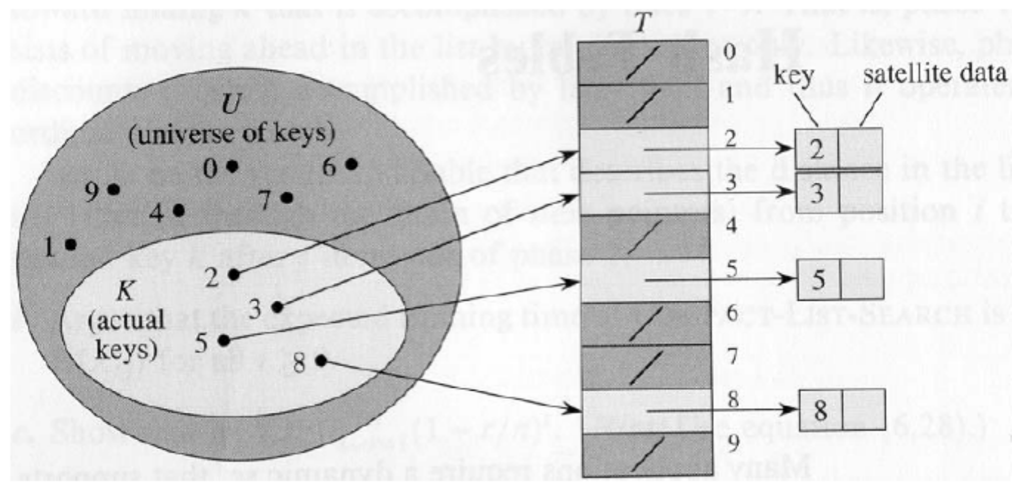
Alg.: DIRECT-ADDRESS-INSERT(T, x)

T[key[x]] ← x

Alg.: DIRECT-ADDRESS-DELETE(T, x)

T[key[x]] ← NIL

Running time for these operations: O(1)



(insert/delete in O(1) time)

Example

Example 1:

- 100 records with distinct integer keys ranging from 1 to 100,
- create an array A of 100 items, store item with key i in A[i]

Example 2:

- keys are nine-digit social security numbers
- create an array A of 10^9 items to store 100 items!
- number of items much smaller than key value range

Hash Tables

- When |K| is much smaller than |U|, a hash table requires much less space than a direct-address table
  - Can reduce storage requirements to |K|
  - Can still get O(1) search time, but on the average case, not the worst case

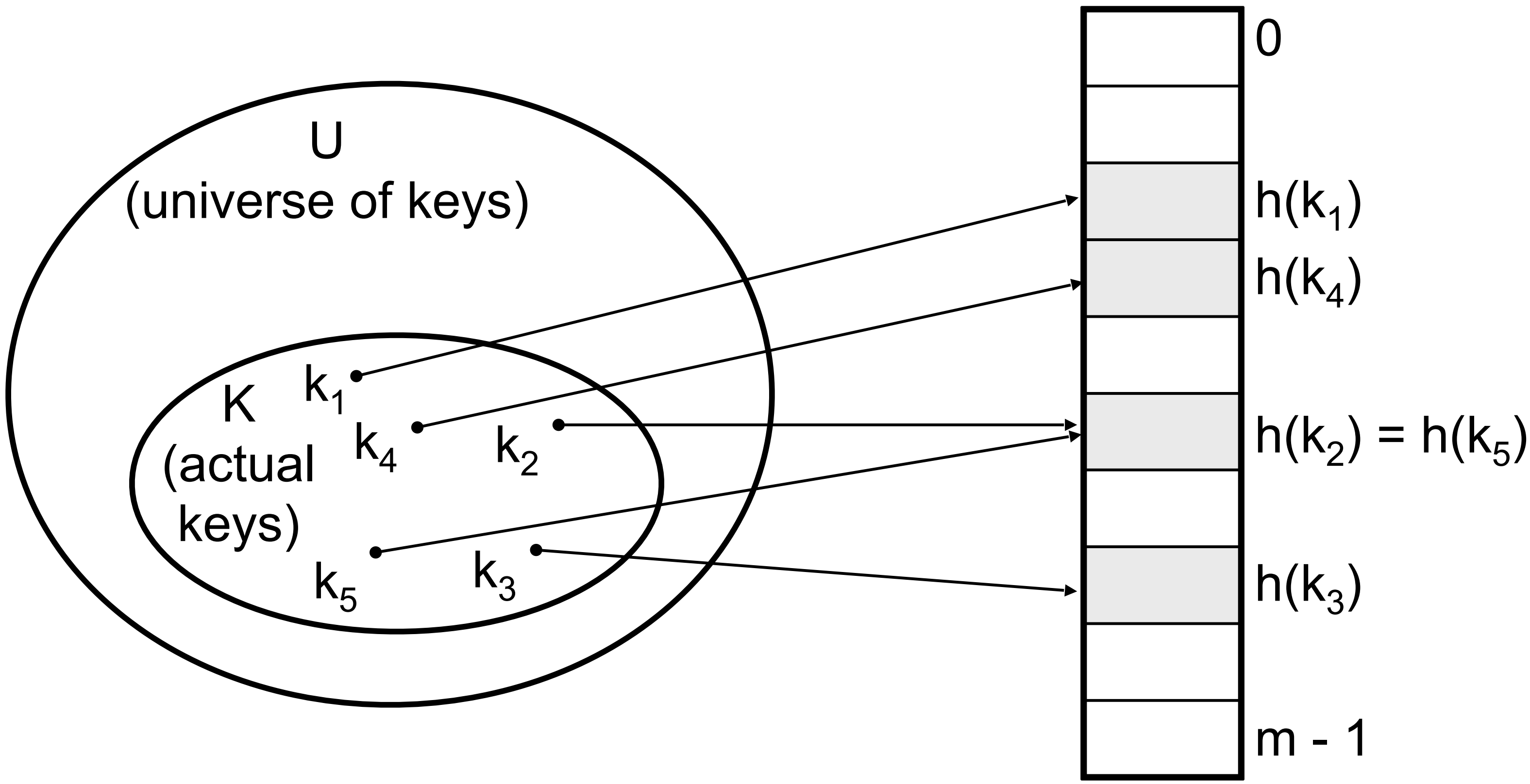
Hash Tables

- Idea
  - Use a compress function h to compute the slot for each key
  - Store the element in slot h(k)
- A hash function h transforms a key into an index in a hash table T[0…m-1]

$$h : U \rightarrow \{0, 1, . . . , m - 1\}$$

- We say that k hashes to slot h(k)
- Advantages
  - Reduce the range of array indices handled: m instead of |U|
  - Storage is also reduced

Hash Tables: Example

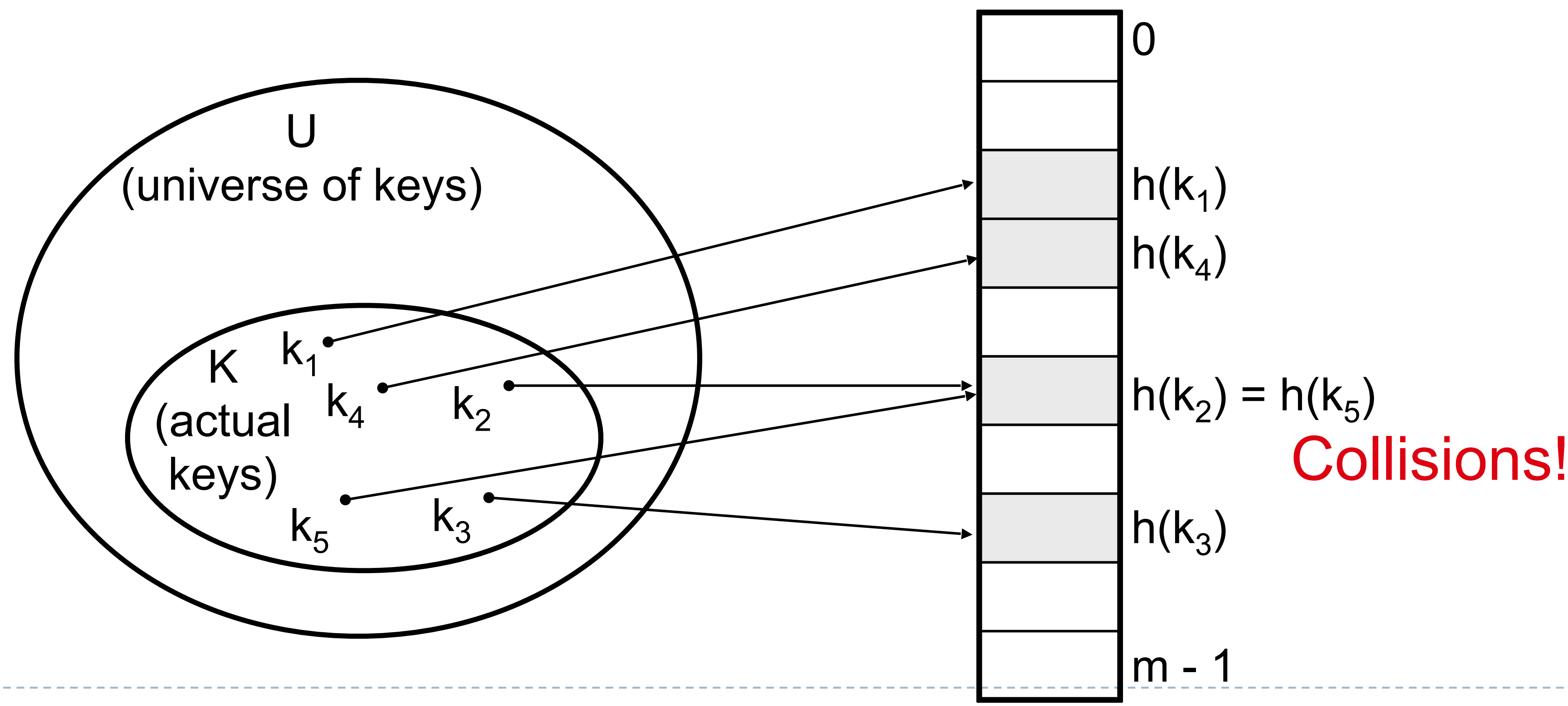


Revisit Example 2

Suppose keys are 9-digit social security numbers

Possible Hash Functions

h(ssn)=ssn mod 100 (last 2 digits of ssn)
h(103-224-511)=11=h(201-789-611)



Collisions

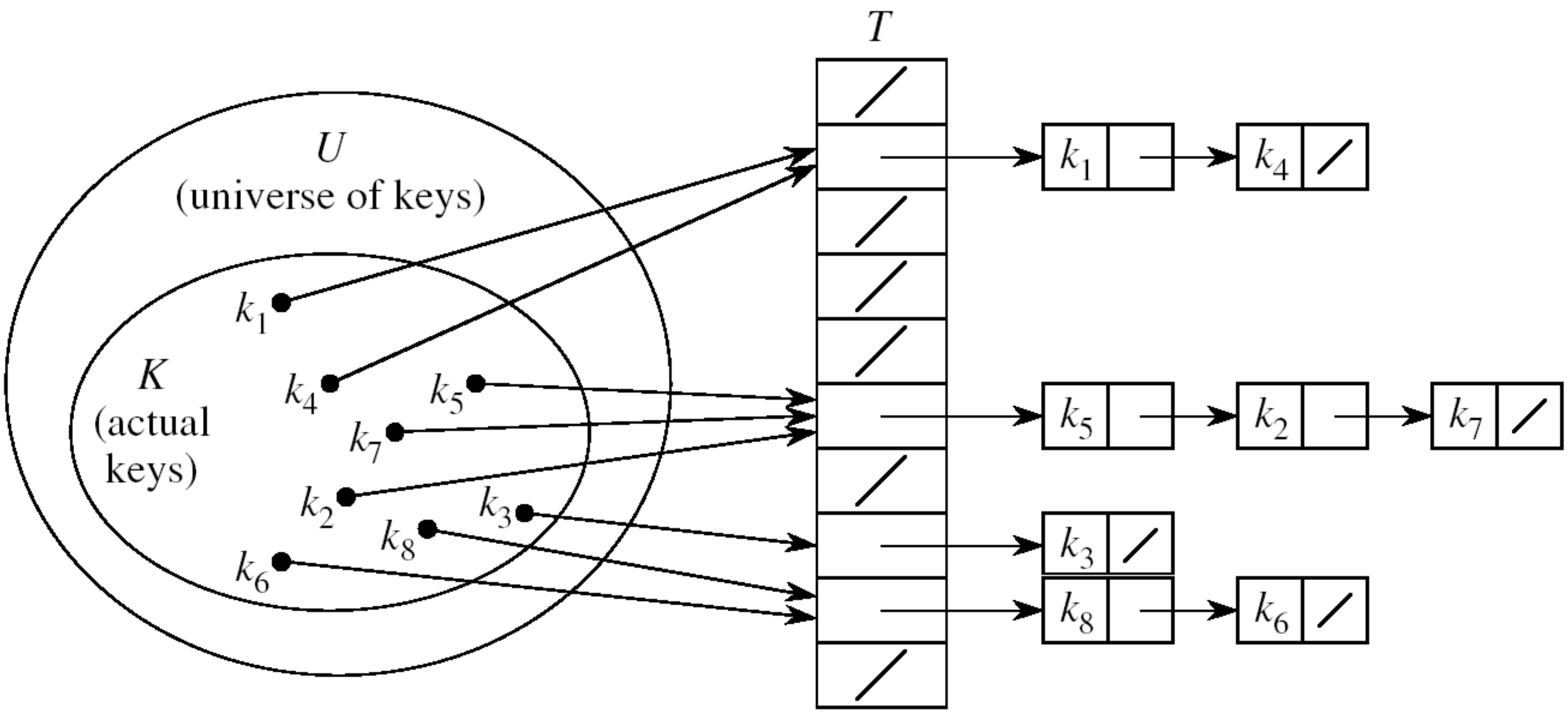
- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function

Handling Collisions

- Collisions can be resolved by:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build "good" hash functions.

Handling Collisions Using Chaining

- **Idea**
  - Put all elements that hash to the same slot into a linked list



  - Slot j contains a pointer to the head of the list of all elements that hash to j

Collision with Chaining - Discussion

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Insertion in Hash Tables

Alg.: CHAINED-HASH-INSERT(T, x)

insert x at the head of list T[h(key[x])]

- Worst-case running time is O(1)
- Assumes that the element being inserted isn't already in the list
- It would take an additional search to check if it was already inserted

Deletion in Hash Tables

Alg.: CHAINED-HASH-DELETE(T, x)

delete x from the list T[h(key[x])]

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in Hash Tables

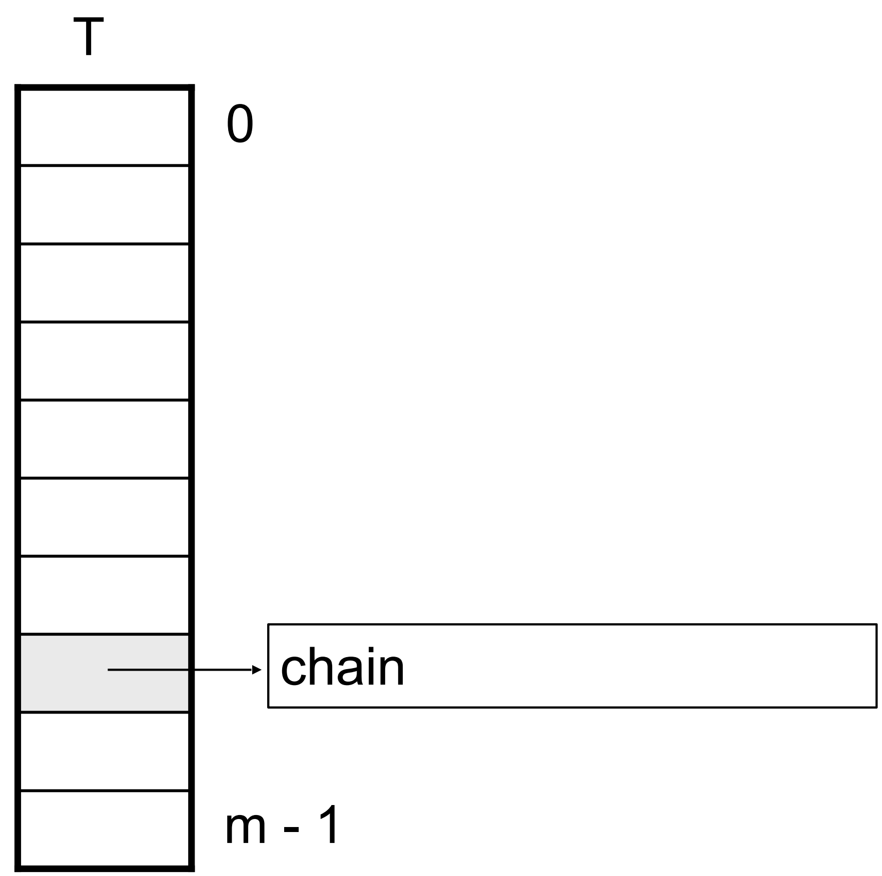
Alg.: CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list T[h(k)]

- Running time is proportional to the length of the list of elements in slot h(k)

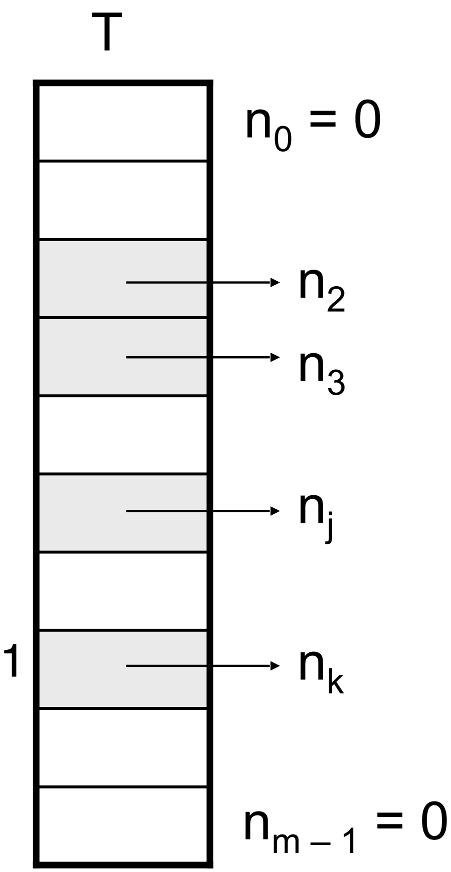
Analysis of Hashing with Chaining:Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All n keys hash to the same slot
  - Worst-case time to search is $\Theta(n)$, plus time to compute the hash function



Analysis of Hashing with Chaining:Average Case

- Average case
  - depends on how well the hash function distributes the n keys among the m slots
- Simple uniform hashing assumption
  - Any given element is equally likely to hash into any of the m slots (i.e., probability of collision $Pr(h(x)=h(y))$, is $1/m$)
- Length of a list: $T[j] = n_j$, $j = 0, 1, . . . , m – 1$
- Number of keys in the table: $n = n_0 + n_1 + \cdots + n_{m-1}$
- Average value of n_j: $E[n_j] = \alpha = n/m$

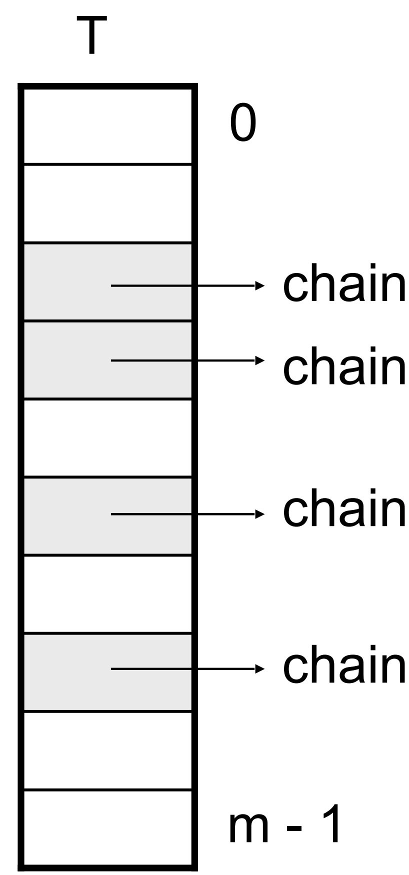


Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

  - n = # of elements stored in the table
  - m = # of slots in the table = # of linked lists
- α encodes the average number of elements stored in a chain
- α can be <, =, > 1



Case 1: Unsuccessful Search (i.e., item not stored in the table)

Theorem

- An unsuccessful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)

Proof

- Searching unsuccessfully for any key k
  - need to search to the end of the list T[h(k)]
- Expected length of the list:
  - $E[n_{h(k)}] = \alpha = n/m$
- Expected number of elements examined in an unsuccessful search is α
- Total time required is:
  - O(1) (for computing the hash function) + $\alpha \longrightarrow \Theta(1+\alpha)$

Case 2: Successful Search

Theorem

- An successful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing

Proof: let x_i be the i-th element inserted to the hash table,

define X_{ij} to be the indicator random variable that element i and j will be hashed to the same value, then the expected number of elements examined in a successful search is:

$$
\begin{aligned}
& \mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] \\
& \quad = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \mathrm{E}\left[X_{ij}\right]\right) = 1+\frac{n-1}{2m} \\
& \quad = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

Analysis of Search in Hash Table

- If m (# of slots) is proportional to n (# of elements in the table):
  - $n = O(m)$
  - $\alpha = n/m = O(m)/m = O(1)$
- Searching takes constant time on average

Hash Functions

- A hash function transforms a key into a table address
- What makes a good hash function?
  - Easy to compute
  - Approximates a random function: for every input, every output is equally likely (simple uniform hashing)
- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., we don't know in advance the probability distribution that keys are drawn from

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as pt and pts should hash to different slots

- Derive a hash value that is independent from any patterns that may exist in the distribution of the keys

The Division Method

- Idea
  - Map a key k into one of the m slots by taking the remainder of k divided by m

$$h(k) = k \bmod m$$

- Advantage
  - Fast, requires only one operation
- Disadvantage
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

The Division Method: Example

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow h(k) = \{0, 1\}$, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow h(k) = \{0, 1, 2, 3\}$, least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

The Multiplication Method

Idea

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m \underbrace{(k\,A \bmod 1)}_{\text{fractional part of } kA \,=\, kA - \lfloor kA \rfloor} \rfloor$$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, e.g., typically 2^p

The Multiplication Method: Example

- The value of m is not critical now (e.g., $m = 2^p$)

assume $m = 2^3$

.101101 (A)
110101 (k)

1001010.0110011 (kA)

discard: 1001010

shift .0110011 by 3 bits to the left

011.0011

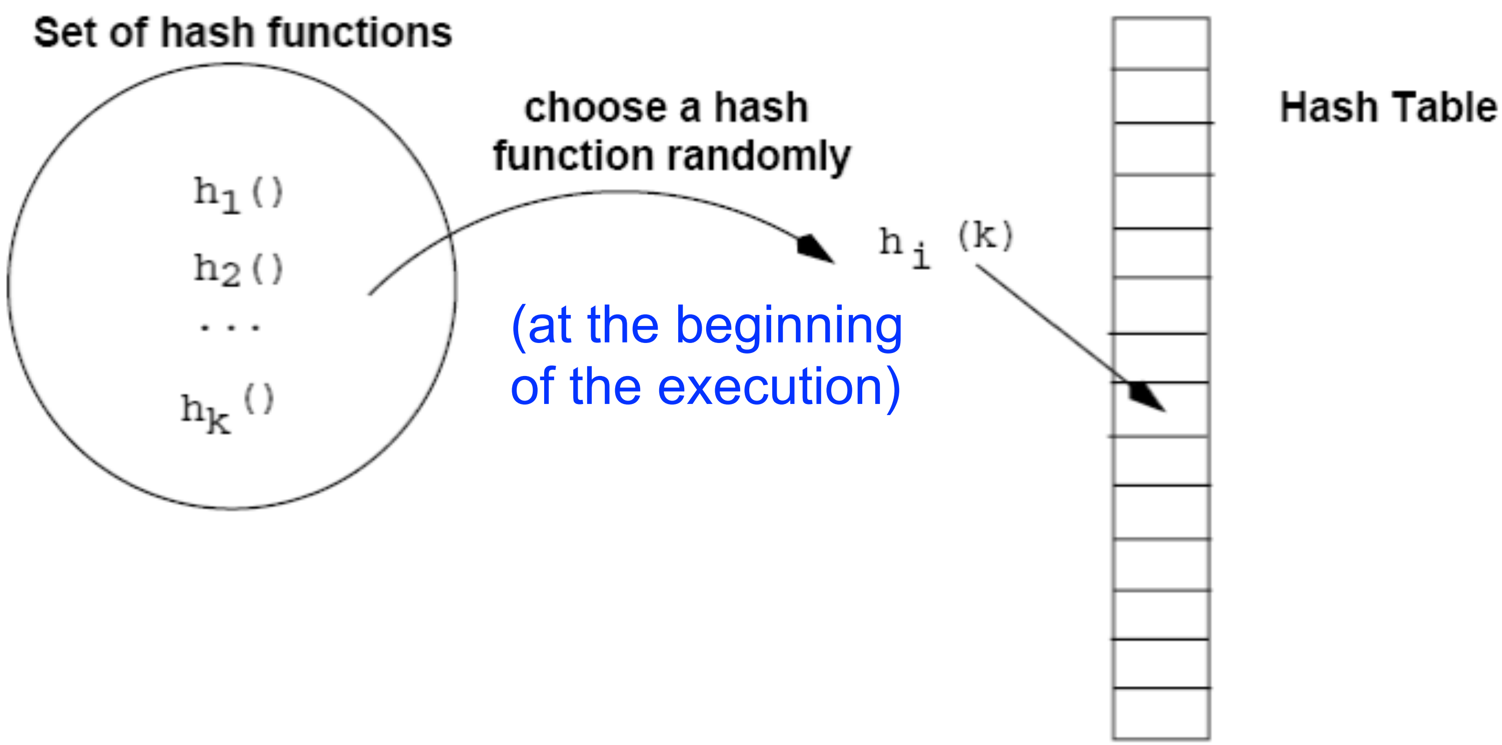
take integer part: 011

thus, h(110101)=011

Universal Hashing

- In practice, keys are not randomly distributed
- Any fixed hash function, adversary may construct a key sequence so that the search time is $\Theta(n)$
- Goal: hash functions that produce random table indices irrespective of the keys
- Idea: select a hash function at random, from a designed class of functions at the beginning of the execution

Universal Hashing



Definition of Universal Hash Functions

From the textbook:

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such a collection is said to be ***universal*** if for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$. In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys k and l is no more than the chance $1/m$ of a collision if $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m-1\}$.

Universal Hashing:Main Result

- With universal hashing the chance of collision between distinct keys k and l is no more than the chance 1/m of a collision if locations h(k) and h(l) were randomly and independently chosen from the set {0, 1, …, m – 1}

Designing a Universal Class of Hash Functions

- Choose a prime number p large enough so that every possible key k is in the range [0 ... p – 1]

  $Z_p = \{0, 1, \ldots, p - 1\}$ and $Z_p^* = \{1, \ldots, p - 1\}$

- Define the following hash function

  $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$

  $\forall\, a \in Z_p^*$ and $b \in Z_p$

- The family of all such hash functions is

  $H_{p,m} = \{h_{a,b} : a \in Z_p^* \text{ and } b \in Z_p\}$

The class $H_{p,m}$ of hash functions is universal

a , b: chosen randomly at the beginning of execution

Universal Hashing Function: Example

E.g.: $p = 17, m = 6$

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

$$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$$

$$= (28 \bmod 17) \bmod 6$$

$$= 11 \bmod 6$$

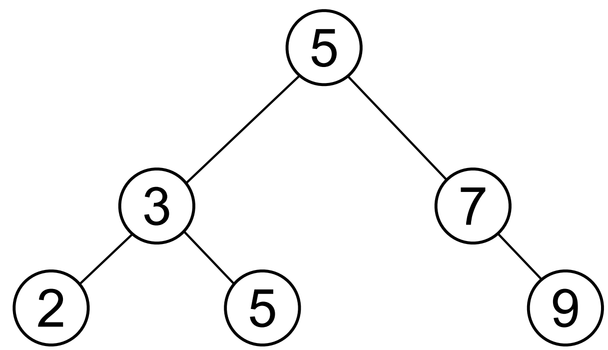
$$= 5$$

Universal Hashing Function: Advantages

- Universal hashing provides good results on *average* performance, independently of the keys to be stored
- Guarantees that no input will always elicit the worst-case performance
- Poor performance occurs only when the random choice returns an inefficient hash function – this has small probability

Binary Search Tree Property

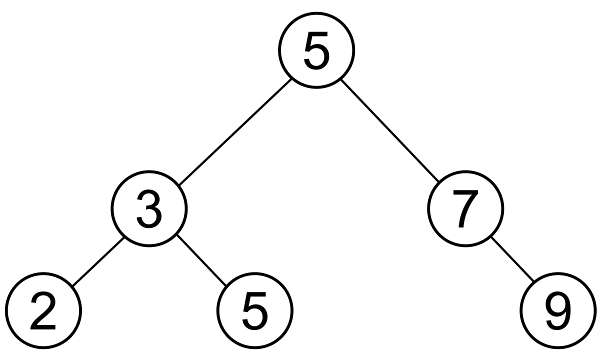
- Binary search tree:
  - binary tree
  - linked data structure, each node has
    - key, satellite data,
    - left/right child, parent
- Property:
  - If y is in left subtree of x,
    - then key [y] ≤ key [x]
  - If y is in right subtree of x,
    - then key [y] ≥ key [x]



$$key[leftSubtree(x)] \le key[x] \le key[rightSubtree(x)]$$

Traversing a Binary Search Tree

- **Inorder** tree walk:
  - Root is printed between the values of its left and right subtrees: left, root, right
  - Keys are printed in sorted order
- **Preorder** tree walk:
  - root printed first: root, left, right
- **Postorder** tree walk:
  - root printed last: left, right, root



Inorder: 2 3 5 5 7 9

Preorder: 5 3 2 5 7 9

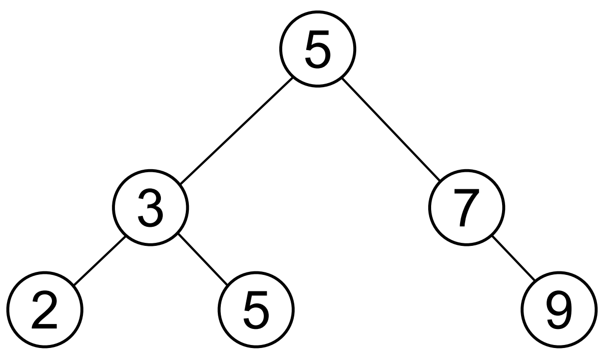
Postorder: 2 5 3 9 7 5

Inorder tree walk

Alg: INORDER-TREE-WALK(x)

1. **if** x ≠ NIL
2. INORDER-TREE-WALK (left [x])
3. print key [x]
4. INORDER-TREE-WALK (right [x])

E.g.:



Output: 2 3 5 5 7 9

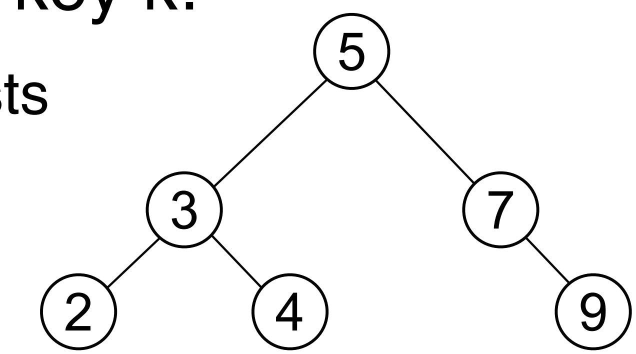
- Running time:
  - Θ(n), where n is the size of the tree rooted at x

Binary Search Trees

- Support many operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees
  - On average: $\Theta(\log n)$
    - The expected height of the tree is $\log n$
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of n nodes (very unbalanced)

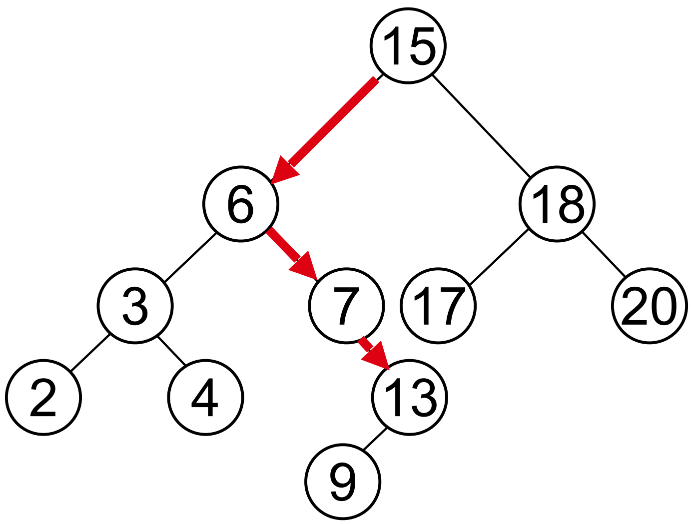
Searching for a Key

- Given a pointer to the root of a tree and a key k:
  - Return a pointer to a node with key k if one exists
  - Otherwise return NIL



- Idea
  - Starting at the root: trace down a path by comparing k with the key of the current node:
    - If the keys are equal: we have found the key
    - If k < key[x] search in the left subtree of x
    - If k > key[x] search in the right subtree of x

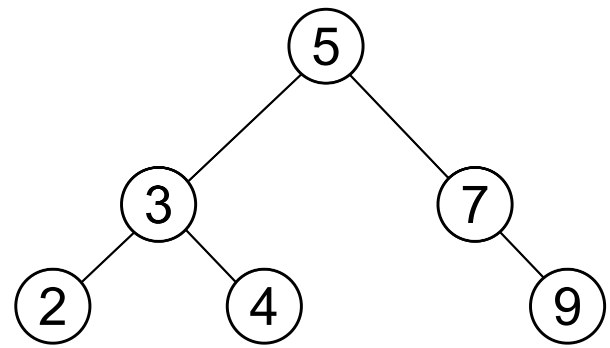
Searching for a Key: Example



- Search for key 13:
  - 15 → 6 → 7 → 13

Binary Search Trees

Alg: TREE-SEARCH(x, k)



1. **if** x = NIL or k = key [x]
2. **then return** x
3. **if** k < key [x]
4. **then return** TREE-SEARCH(left [x], k)
5. **else return** TREE-SEARCH(right [x], k)

Running Time: O (h),
h – the height of the tree

Binary Search Trees: Finding the Minimum

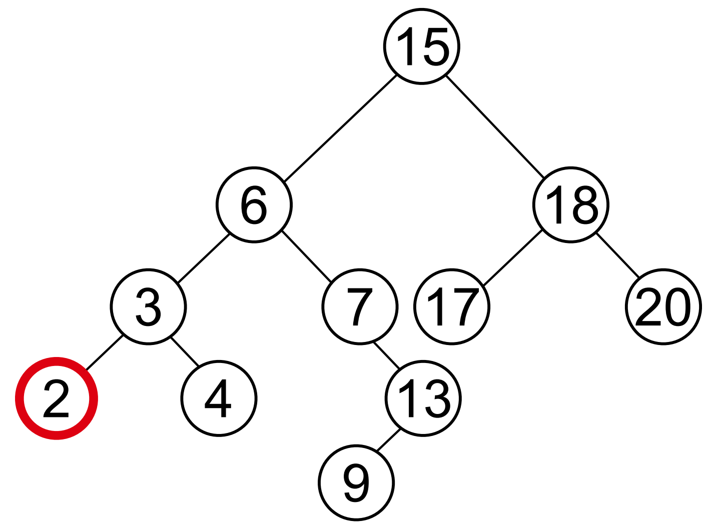
- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered

Alg: TREE-MINIMUM(x)

while left [x] ≠ NIL

  do x ← left [x]

return x



Minimum = 2

- Running time: O(h), h – height of tree

Binary Search Trees: Finding the Maximum

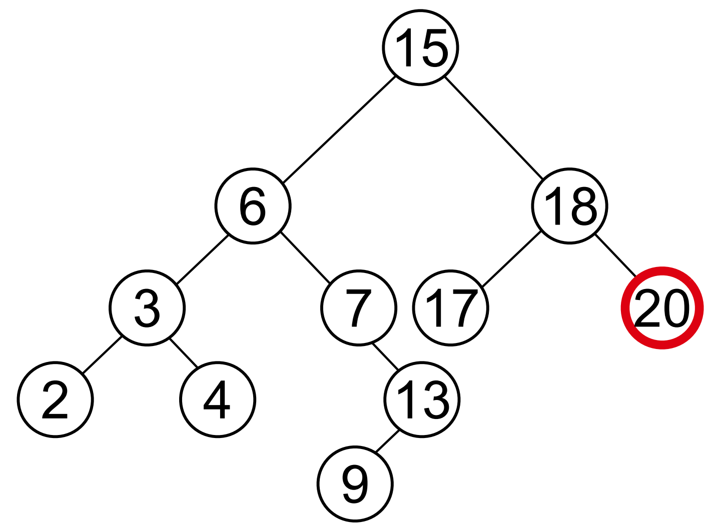
- Goal: find the maximum value in a BST
  - Following right child pointers from the root, until a NIL is encountered

Alg: TREE-MAXIMUM(x)

while right [x] $\neq$ NIL

do x ← right [x]

return x



Maximum = 20

- Running time: O(h), h – height of tree

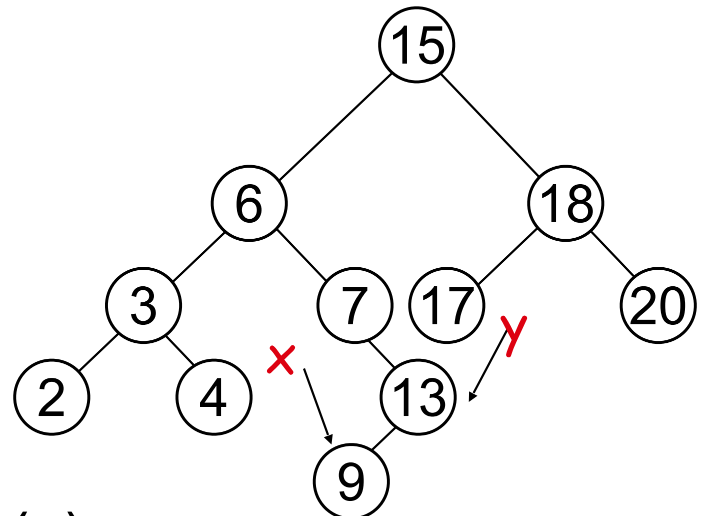
Successor

- *Def: successor* (*x*) = y, such that key [y] is the smallest key > key [x]

  E.g.: successor (15) = 17

  successor (13) = 15

  successor (9) = 13

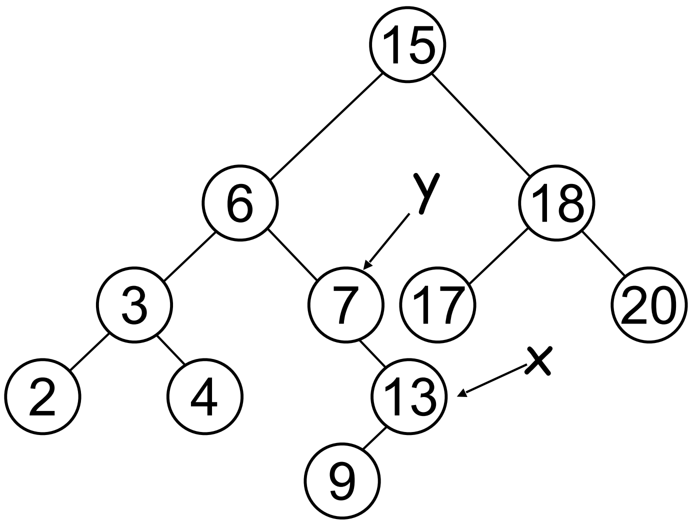


- Case 1: right (x) is non empty
  - *successor* (*x*) = the minimum in right (x)
- Case 2: right (x) is empty
  - go up the tree until the current node is a left child: *successor* (*x*) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element

Finding the Successor

Alg: TREE-SUCCESSOR*(x)*

1. **if** right [x] ≠ NIL
2. **then return** TREE-MINIMUM(right [x])
3. y ← p[x]
4. **while** y ≠ NIL and x = right [y]
5. **do** x ← y
6. y ← p[y]
7. **return** y



Running time: O (h), h – height of the tree

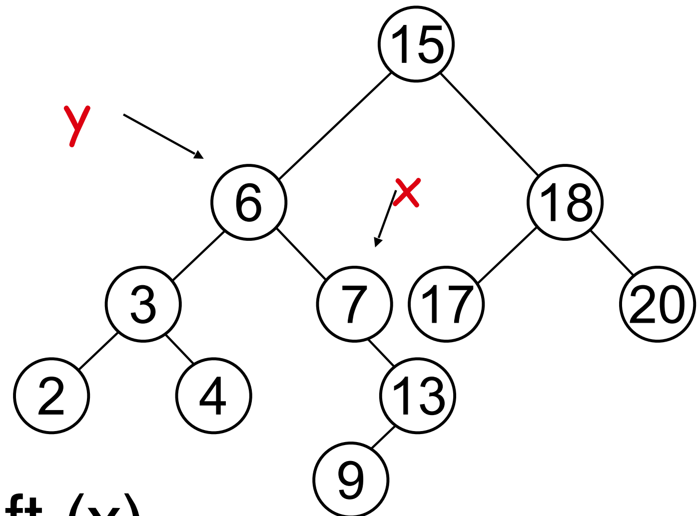
Predecessor

Def: predecessor (*x*) *= y*, such that key [y] is the biggest key < key [x]

E.g.: predecessor (15) = 13

predecessor (9) = 7

predecessor (7) = 6



Case 1: left (x) is non empty

predecessor (*x*) = the maximum in left (x)

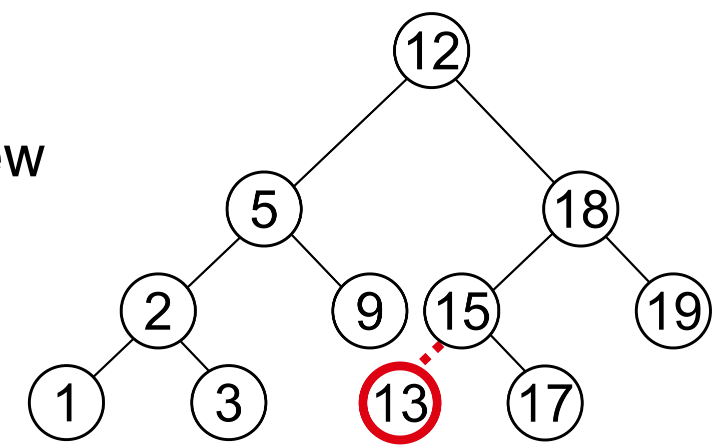
Case 2: left (x) is empty

- go up the tree until the current node is a right child: *predecessor* (*x*) is the parent of the current node
- if you cannot go further (and you reached the root): x is the smallest element

Insertion

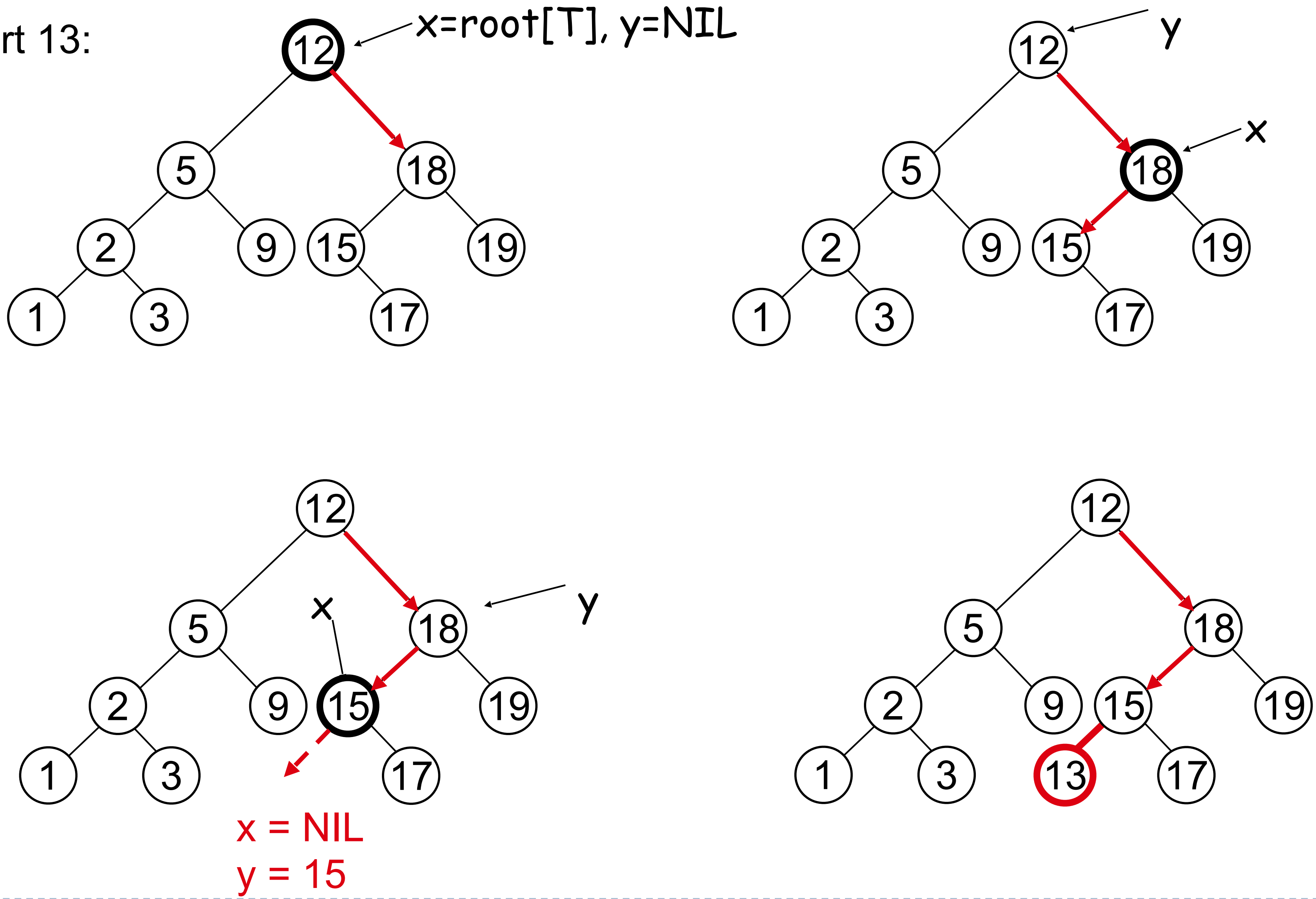
- Goal:
  - Insert value v into a binary search tree
- Idea:
  - If key [x] < v move to the right child of x, else move to the left child of x
  - When x is NIL, we found the correct position
  - let y be parent of x, if v < key [y] insert the new node as y's left child else insert it as y's right child
  - Beginning at the root, go down the tree and maintain:
    - Pointer x : traces the downward path (current node)
    - Pointer y : parent of x ("trailing pointer")

Insert value 13



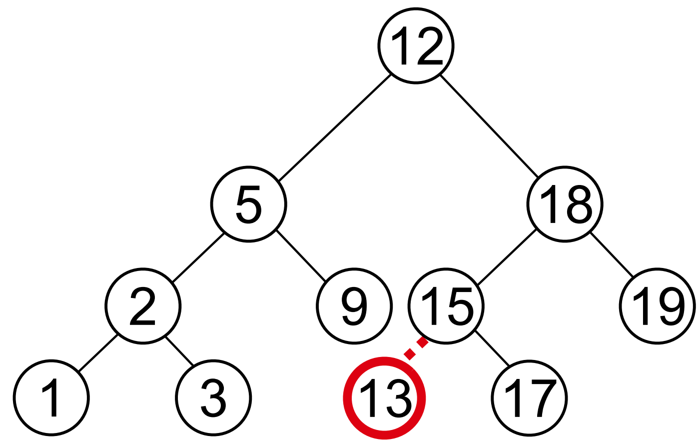
Insertion: Example

Insert 13:



Tree Insertion

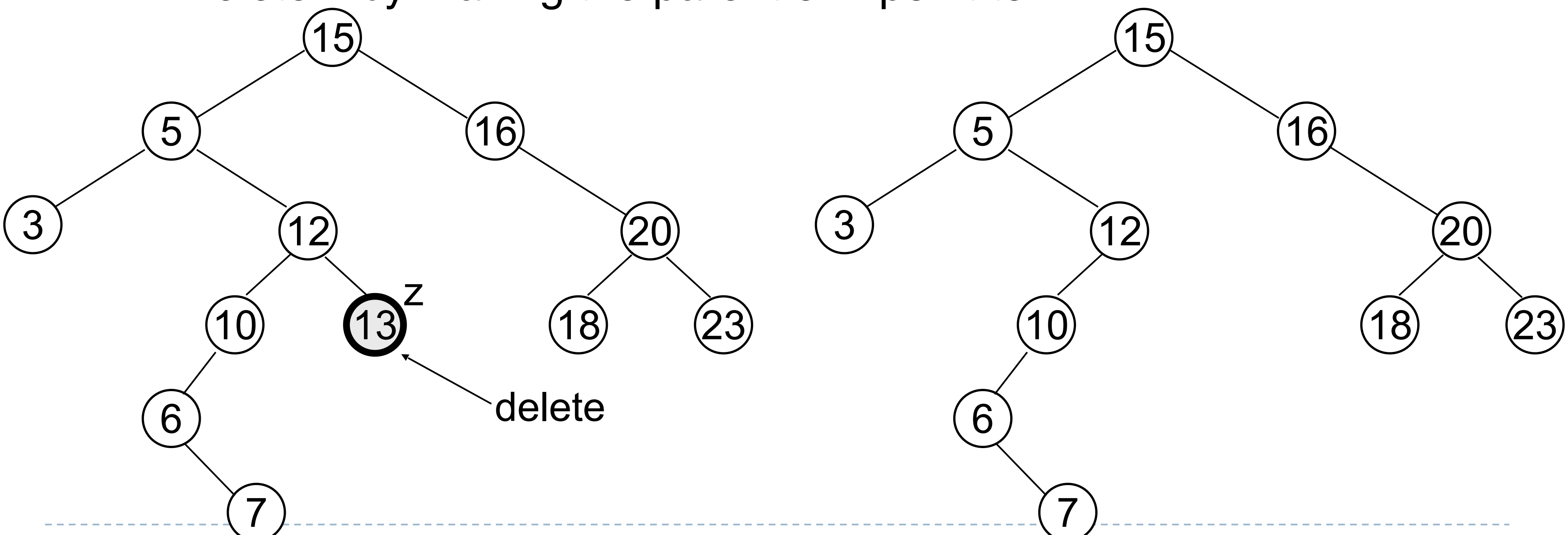
```
y ← NIL
x ← root [T]
while x ≠ NIL
    do y ← x
        if key [z] < key [x]
            then x ← left [x]
            else x ← right [x]
p[z] ← y
if y = NIL
  then root [T] ← z    // Tree T was empty
  else if key [z] < key [y]
          then left [y] ← z
          else right [y] ← z
```



Running time: O(h)

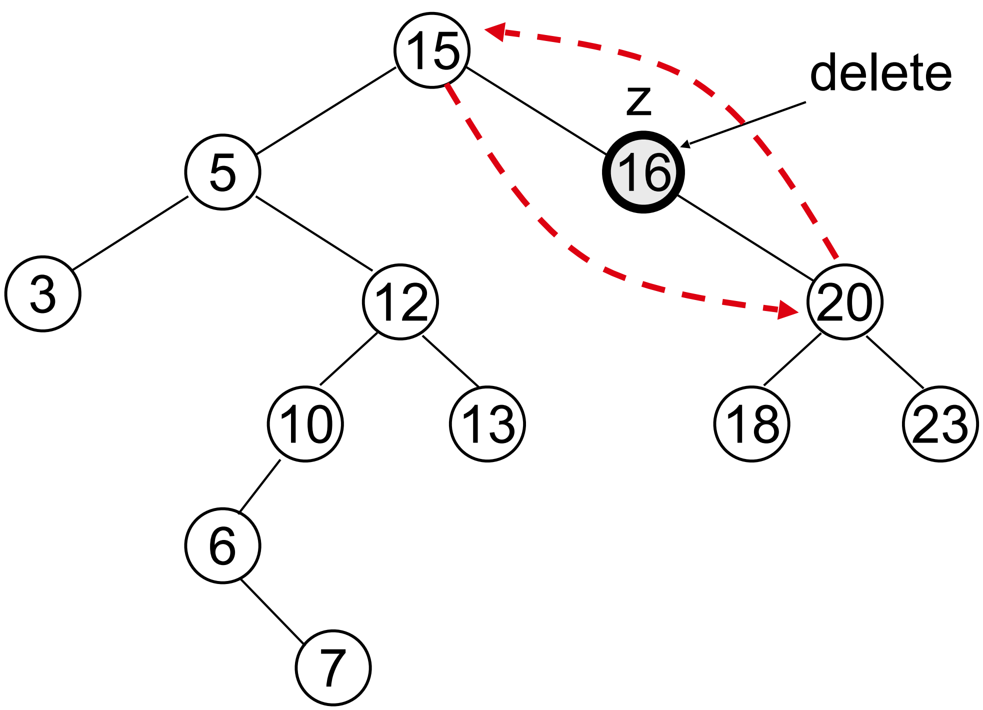
Deletion

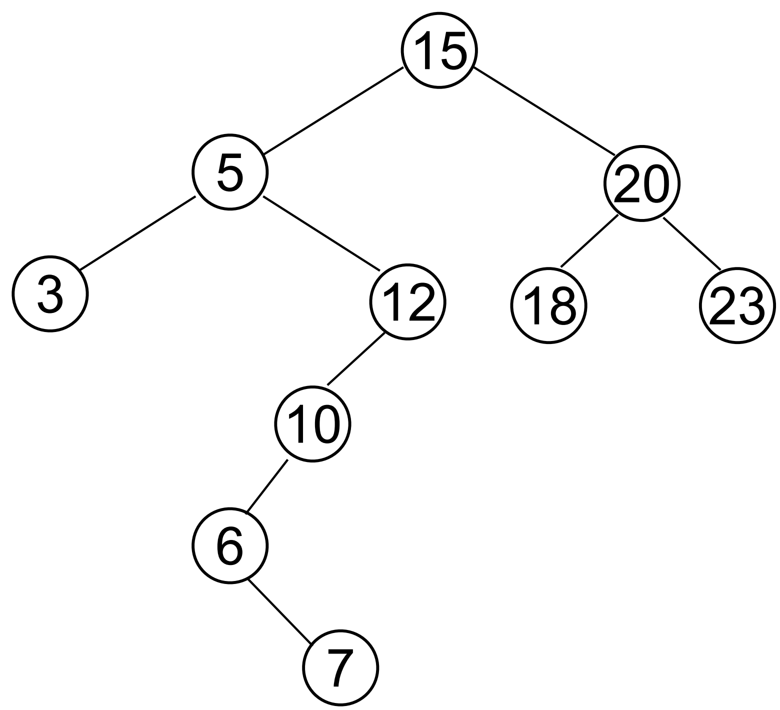
- Goal:
  - Delete a given node z from a binary search tree
- Idea:
  - Case 1: z has no children
    - Delete z by making the parent of z point to NIL



Deletion

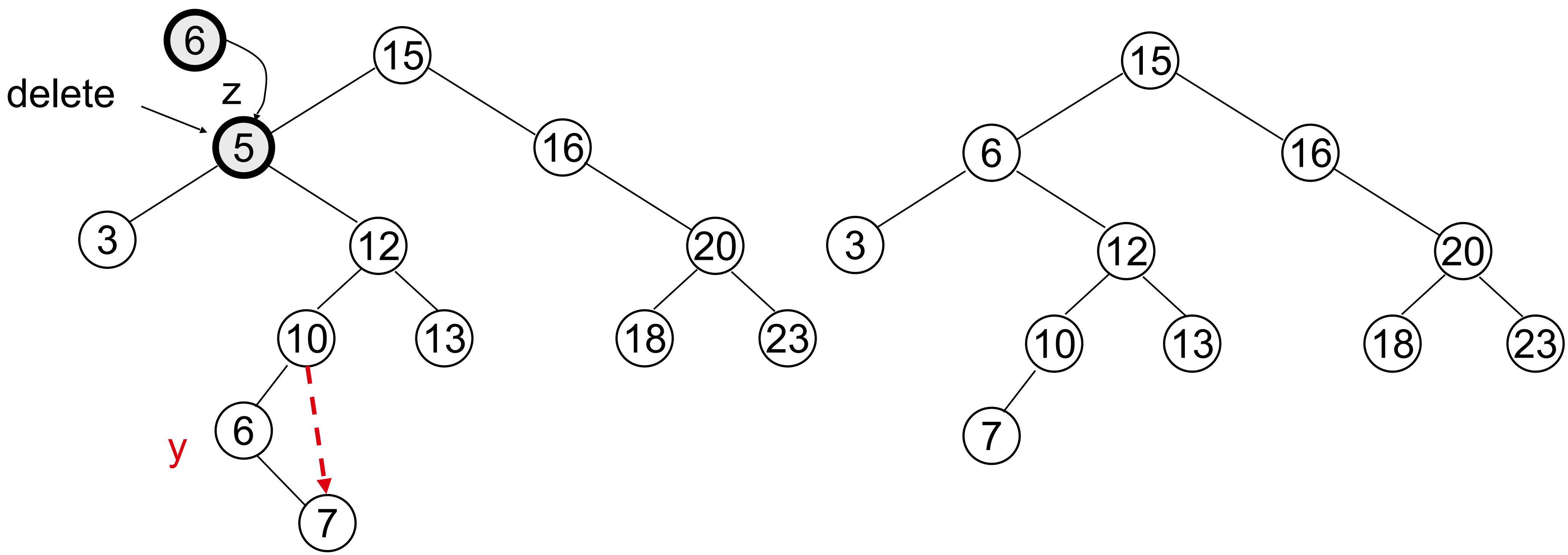
- Case 2: z has one child
  - Delete z by making the parent of z point to z's child, instead of to z





Deletion

- Case 3: z has two children
  - z's successor (y) is the minimum node in z's right subtree
  - y has either no children or one right child (but no left child)
  - Delete y from the tree (via Case 1 or 2)
  - Replace z's key and satellite data with y's.



Binary Search Trees: Summary

- Operations on binary search trees:
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)
- These operations are fast if the height of the tree is small

What's next...

- Binary Search Trees (Cont.d)
- Midterm Review