ALGORITHMS & ADVANCED DATA STRUCTURES (#9)

HASHING

ADAPTED FROM
CS 146 SJSU (KATERINA POTIKA)

Dynamic Sets

- data structures rather than straight algorithms
- In particular, structures for *dynamic sets*
- Elements have a *key* and *satellite data*

Dynamic Sets

- Dynamic sets support *queries* such as:
  - ***Search(S, k),***
  - ***Minimum(S),***
  - ***Maximum(S),***
  - ***Successor(S, x),***
  - ***Predecessor(S, x)***

- They may also support *modifying operations* like:
  - ***Insert(S, x),***
  - ***Delete(S, x)***

Keys

Keys as natural numbers

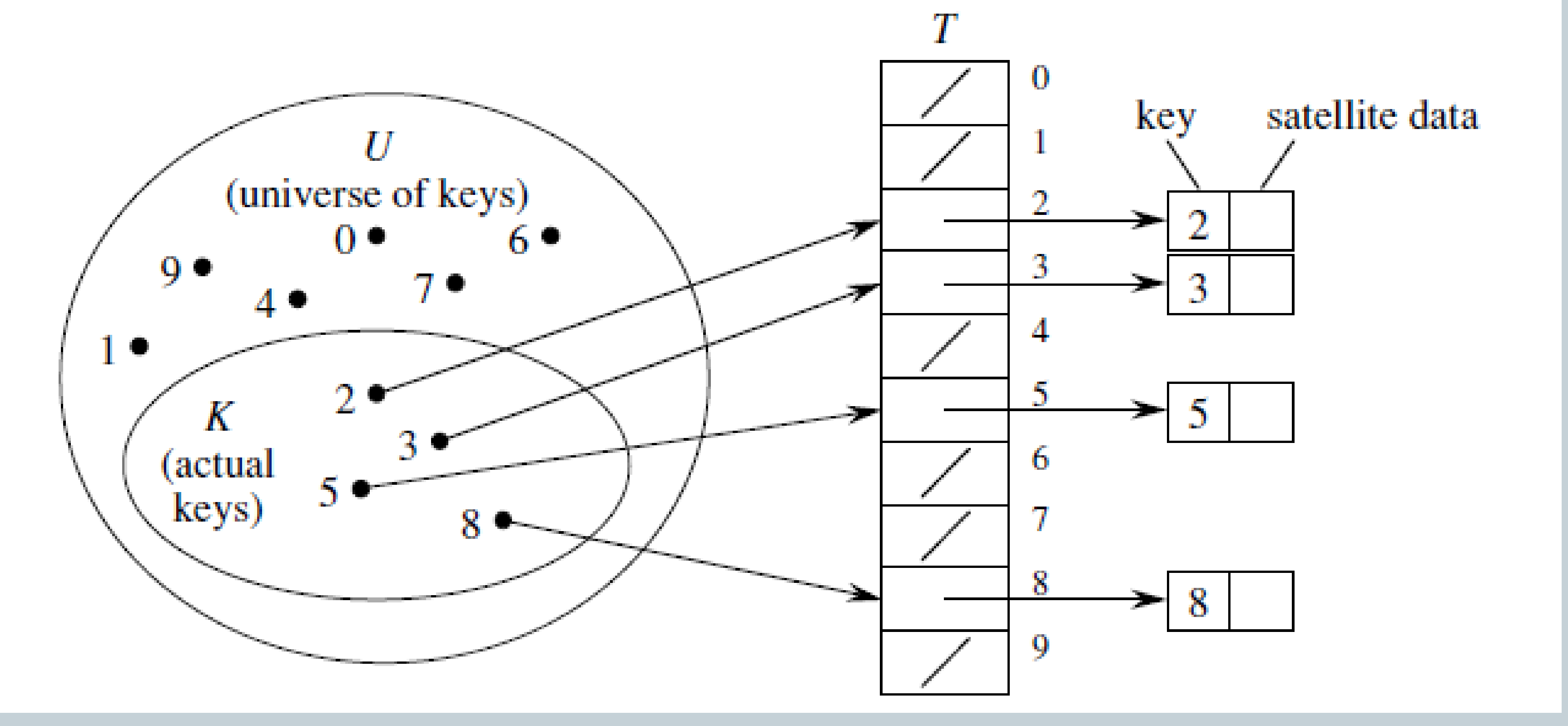
- Hash functions assume that the keys are natural numbers.
- When they're not, have to interpret them as natural numbers.
- ***Example:*** Interpret a character string as an integer expressed in some radix notation. Suppose the string is `CLRS`:
  - ASCII values: `C` $= 67$, `L` $= 76$, `R` $= 82$, `S` $= 83$.
  - There are 128 basic ASCII values.
  - So interpret `CLRS` as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141{,}764{,}947$.

In Java hashCode()

Hash Tables

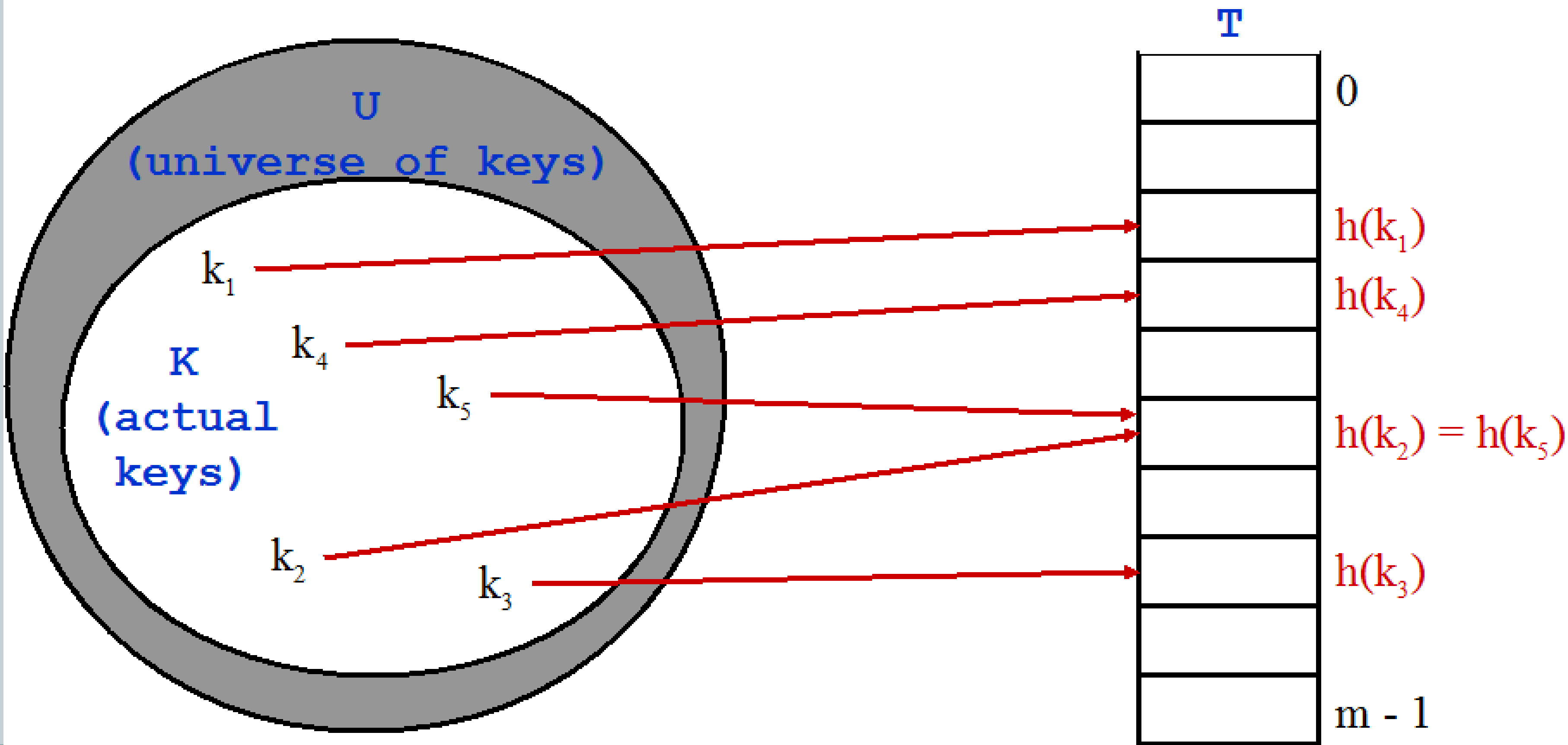
- More formally:
- Given a table T and a record x, with key (= symbol) and satellite data, we need to support:
- Insert (T, x)
- Delete (T, x)
- Search(T, x)
- We want these to be fast, but don't care about sorting the records
- The structure we will use is a *hash table*
- Supports all the above in O(1) expected time!

Example: Direct Addressing



Hash Functions

Next problem: *collision*

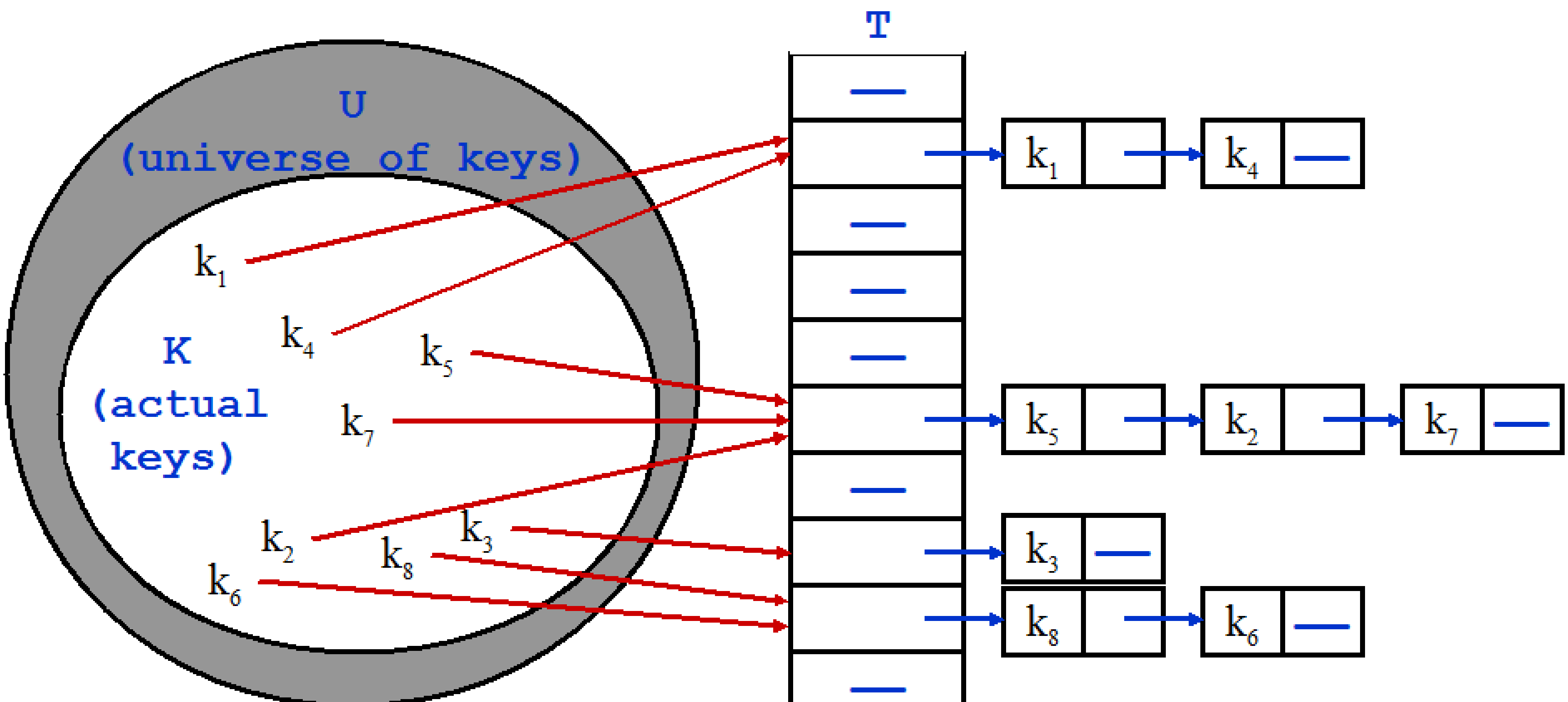


Resolving Collisions

- *How can we solve the problem of collisions?*
- Solution 1: *chaining*
- Solution 2: *open addressing*

Chaining

Chaining puts elements that hash to the same slot in a linked list:



Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table: the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*

Analysis (average-case) of chaining

- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$
- *What will be the average cost of a successful search?* A: $O(1 + \alpha/2) = O(1 + \alpha)$

Choosing A Hash Function

- Clearly choosing the hash function well is crucial
- *What will a worst-case hash function do?*
- *What will be the time to search in this case?*
- *What are desirable features of the hash function?*
- Should distribute keys uniformly into slots
- Should not depend on patterns in the data

  - if we know the keys in advance then we can always derive a **perfect hash function** (each key hashes to its own unique location).

Hash Functions: The Division Method

- $h(k) = k \bmod m$
- In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- *What happens to elements with adjacent values of k?*
- *What happens if m is a power of 2 (say 2^P)?*
- *What if m is a power of 10?*
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

example

- let m = 7
- let k = 8, 14, 12, 2, 4
- let h(k) = k % m

- remind the students that 2 % 7 = 2 … NOT 5.
- Can you give numbers that will and will not give collisions?

Hash Functions: Universal Hashing

- As before, when attempting to foil an malicious adversary: randomize the algorithm
- *Universal hashing*: pick a hash function randomly in a way that is independent of the keys that are actually going to be stored
- Guarantees good performance on average, no matter what keys adversary chooses

Open Addressing

- Basic idea:
- To insert: if slot is full, try another slot, …, until an open slot is found (*probing*)
- To search, follow same sequence of probes as would be used when inserting the element
  - If reach element with correct key, return it
  - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)

Example: spell checking

- Table needn't be much bigger than n

Linear Probing

- We can think of our hash function as having an extra parameter h(k, i), where i is the **probe number** that starts at zero for each key (i.e., it's the number of the try).
- Simplest type of probing
  - $h(k,i) = (h(k) + i) \% m$
  - This is called **linear probing**

- With open addressing, we require that for every key k, the ***probe sequence***

$$h(k,0), h(k, 1),\ldots ,h(k,m-1)$$

Note: m size of hash table

Insert

- If i=0, then h(k,i) = h(k), i.e., the “home” location of the key. If that is occupied, we use the next location to the right, wrapping around to the front of the array.

```
HASH-INSERT(T, k)
i = 0
do
        j=h(k,i)
        if T[j] == NULL
                T[j]=k
                return j
        else i++
while i!= m-1
error “hash table overflow”
```

Linear Probing (cont'd)

- Another example
  - m = 9
  - k = 0, 9, 18, 27, 36, 45, 54
- Can you do it fast?
- If all keys map to same location -> BAD!
- Talk them through how it is O(n)

Linear Probing (search cont'd)

- We use the same algorithm for searching as we did for insertion. We probe. If we come to an empty spot, we can stop and say the key is not in the table.
- So what's the search cost? O(n) … why? (explain worst case)

Search


```
HASH-SEARCH(T,k)
i =0
do
    j =h(k, i)
    if T[j] == k
        return j
    i ++
while T[j] == NULL or i != m
return NULL
```

Linear Probing (cont'd)

- Assuming simple uniform hashing, what is the expected insertion cost?
- Let α = n/m (the **load factor** of the hash table)
  - 0 <= α <= 1
- Percent of time you'll have a collision? α
- 1 + α + α^2 + α^3 …
  - i.e., no collisions, 1 collision, 2 collisions …
- This is a geometric series and converges to:
  - 1 / (1-α)
  - Show what happens for half-full table, 90% full table
  - 100-slot table or 100000-slot table with 90% full … it still takes on average 10 attempts

Expected cost (con’d)

- So expected behavior depends on load factor, not on size of table … so it’s O(1)
- What’s the catch? Well, expected is good for many applications, but not, e.g., missile defense … the worst case is still O(n) with linear probing.
- If $\alpha = 0.5$, we expect it to take 2 tries to insert a key. This does not (directly) depend on the number of keys present.

Linear Probing (problems)

- How good is linear probing?
- Well, consider two hash tables that are 50% full.
  - First: full on one side
  - Other: sprinkled evenly
- The first exhibits a large **cluster** of filled slots. We don't want large clusters.

Linear Probing (problems)

- Linear probing is prone to **primary clustering** (long run of filled locations).
- it takes a long time to get to the end of a cluster, and then you wind up just adding to its length

Hashing (cont'd)

- Linear probing is simple but has the disadvantage of **primary clustering**.
- We can try **quadratic probing**.
  - h(k,i) = (h(k)+i^2) % m
  - In general:
    - h(k,i) = (h(k)+c1*i + c2* i^2)%m

Quadratic Probing

- Ex:
  - let m=13
  - let h(k,i) = (h(k) + i^2)%m
  - let k = 3, 4, 26, 2, 66, 0, 22, 99
- show what happens when collides
- how it does not result in primary clustering?

Quadratic Probing

- Quadratic probing avoids the problem of primary clustering, but instead has **secondary clustering**.
- **Secondary clustering** is a long "run" of filled locations <u>along a probe sequence.</u>
- If many keys hash to the same value, those collisions will all fill the same probe sequence (right?).
- Still better..

Quadratic Probing

- Ex: m = 4
  - h(k) = (k+i^2)%m
  - k = 0, 1, 4
  - 4 will keep colliding with 0, then 1

Quadratic Probing

- A bigger problem with quadratic probing is that you are not guaranteed to find an empty slot, even though one exists.
  - make your hash table prime

Double Hashing

- $h(k,i) = (h1(k) + h2(k)*i)\%m$
- The probe increment is not fixed, but is determined by the key itself.
- Why use double hashing?
  - It avoids secondary clustering.
- Again, there is no guarantee we will find an empty slot, unless we pick our constants very carefully.

Hashing - DELETION

- how do you find the smallest value in sorted, unsorted, or hash table?
- Answer: for the hash table, it isn’t 1 or n, it’s m

Example

- Let’s consider deleting a value from a table:
  - let m = 7
  - let h(k,i) = (k+i)%m
  - insert 4 <draw it>
  - insert 7 <draw it>
  - delete 7 <draw it>
  - insert 11 <draw it>
  - find 11 <draw & trace it>
  - del 4 <draw it>
  - find 11 <draw & trace it … oops!>

Hashing - DELETION (cont'd)

- We need to leave a marker when a key is deleted, so that when we search for a key, we know that we should not stop probing.
  - This marker is called a **tombstone**.
- For that matter, how do we know if a key is actually present in a slot to begin with? We need a flag!

- draw two parallel arrays: data & flags
  - flags: E = empty, F = filled, D = deleted

Hashing (cont'd)

- Searching using flags:
  - E => we stop! key is <u>not</u> there
  - F => we compare
    - if True => stop! found it!
    - if False => keep going
  - D => keep going! <DON'T compare>

Hashing (cont'd)

- So deletions can slow you down. If you are doing a lot of deletions, you're better of using a different style of hash table-> CHAINING
- When you insert, you want to remember the first deleted spot you found (so that you can reclaim the D spots and avoid excessive probing).

Hashing (cont'd) - insertion

- When inserting, we still probe as usual, but we remember the first deleted slot we saw.
- example (to capture first location):
  - avail = -1
  - if (avail == -1) avail = locn.
- If a deleted spot was found, use it over the empty slot.
- Why? 1. the deleted slot is closer to the **home location** of that key. 2. We get to reclaim a deleted slot.

Example

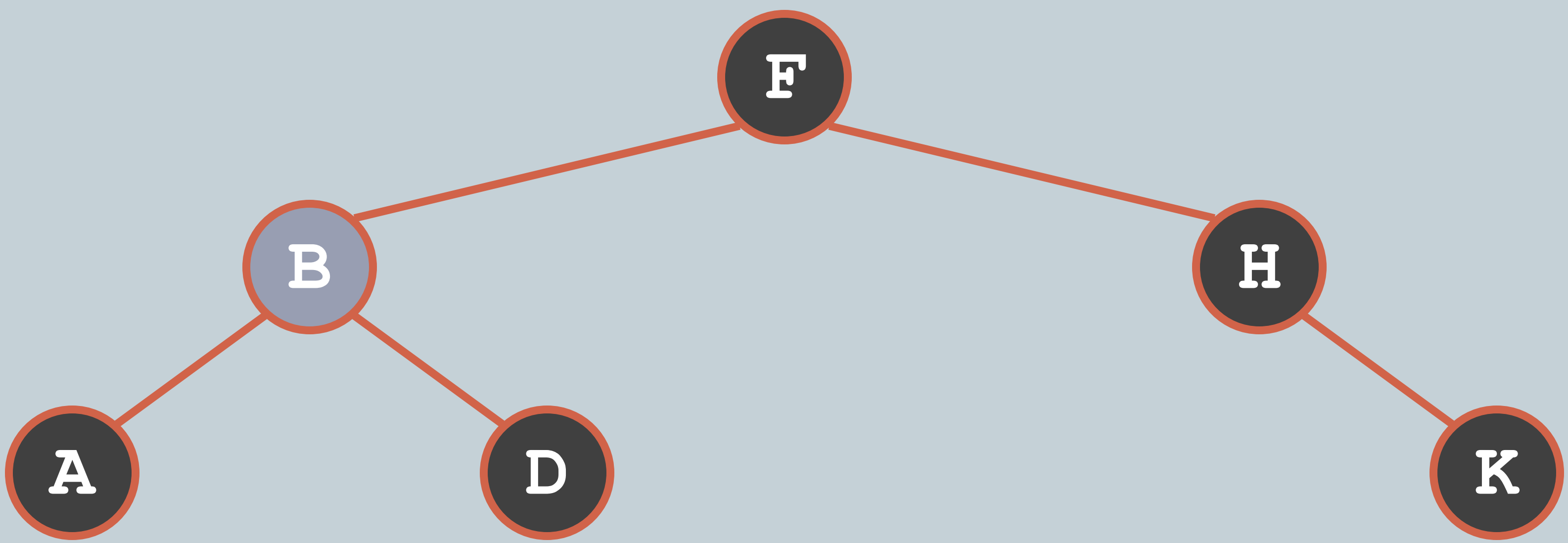
- Given the following numbers {5, 29, 20, 0, 27, 18} and a hash function h(x) = (x % 9): Insert these numbers into an initially empty hash table with collisions
  - resolved by chaining.
  - resolved by open addressing with linear probing
- Show the populated hash tables. Assume the array that stores this hash table has 9 slots.

Binary Search Trees (Ch 12)

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

Binary Search Trees

- BST property:
  x *[leftSubtree].key ≤ x.key ≤ x[rightSubtree].key*
- Example:



Inorder Tree Walk

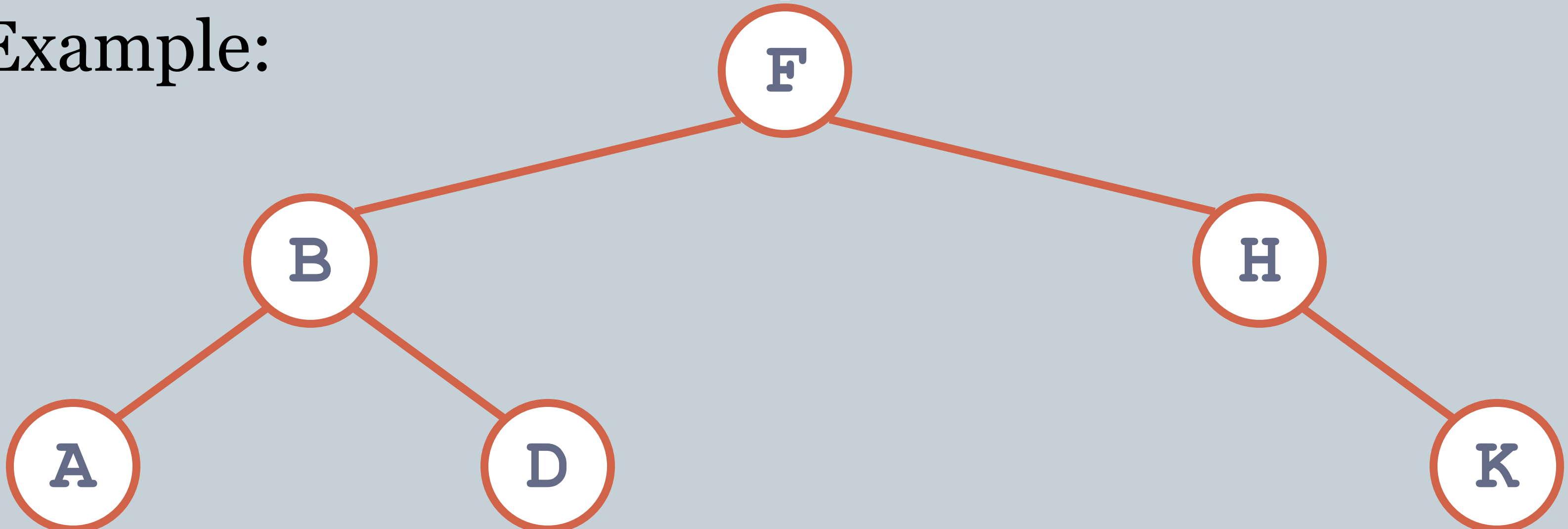
- *What does the following code do?*

```
TreeWalk(x)
    TreeWalk(x.left);
    print(x);
    TreeWalk(x.right);
```

- A: prints elements in sorted (increasing) order
- This is called an *inorder tree walk*
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Inorder Tree Walk

- Example:



- *How long for a tree walk?*
- *Can your prove that inorder walk prints in monotonically increasing order? (induction)*

Time complexity of Inorder Walk

- $\Theta(n)$

Querying a BST

- Search
- Minimum
- Maximum
- Successor
- Predecessor

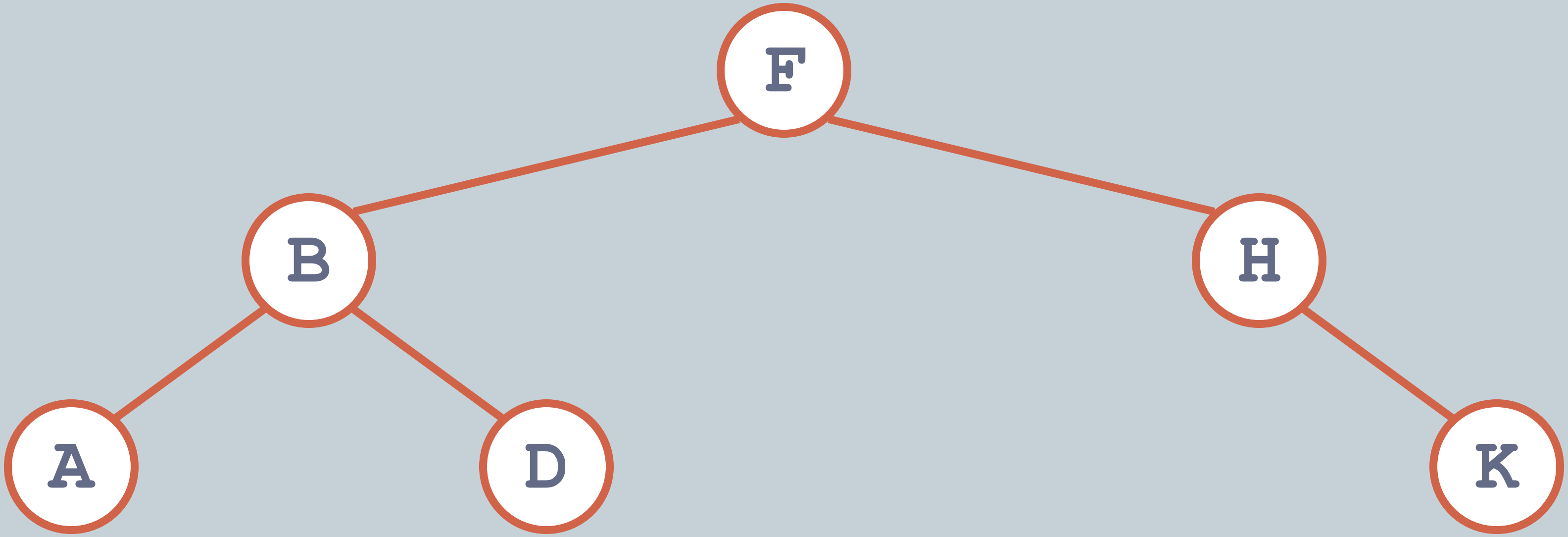
Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
Tree-Search(x, k)
     if (x = NULL  or  k = x.key)
         return x;
     if (k < x.key)
         return Tree-Search(x.left, k);
     else
         return Tree-Search(x.right, k);
```

BST Search: Example

- Search for *D* and *C*:



Operations on BSTs: Search

- Here's another function that does the same:

```
Tree-Search(x, k)
    while (x != NULL  and  k != x.key)
        if (k < x.key)
            x = x.left;
        else
            x = x.right;
    return x;
```

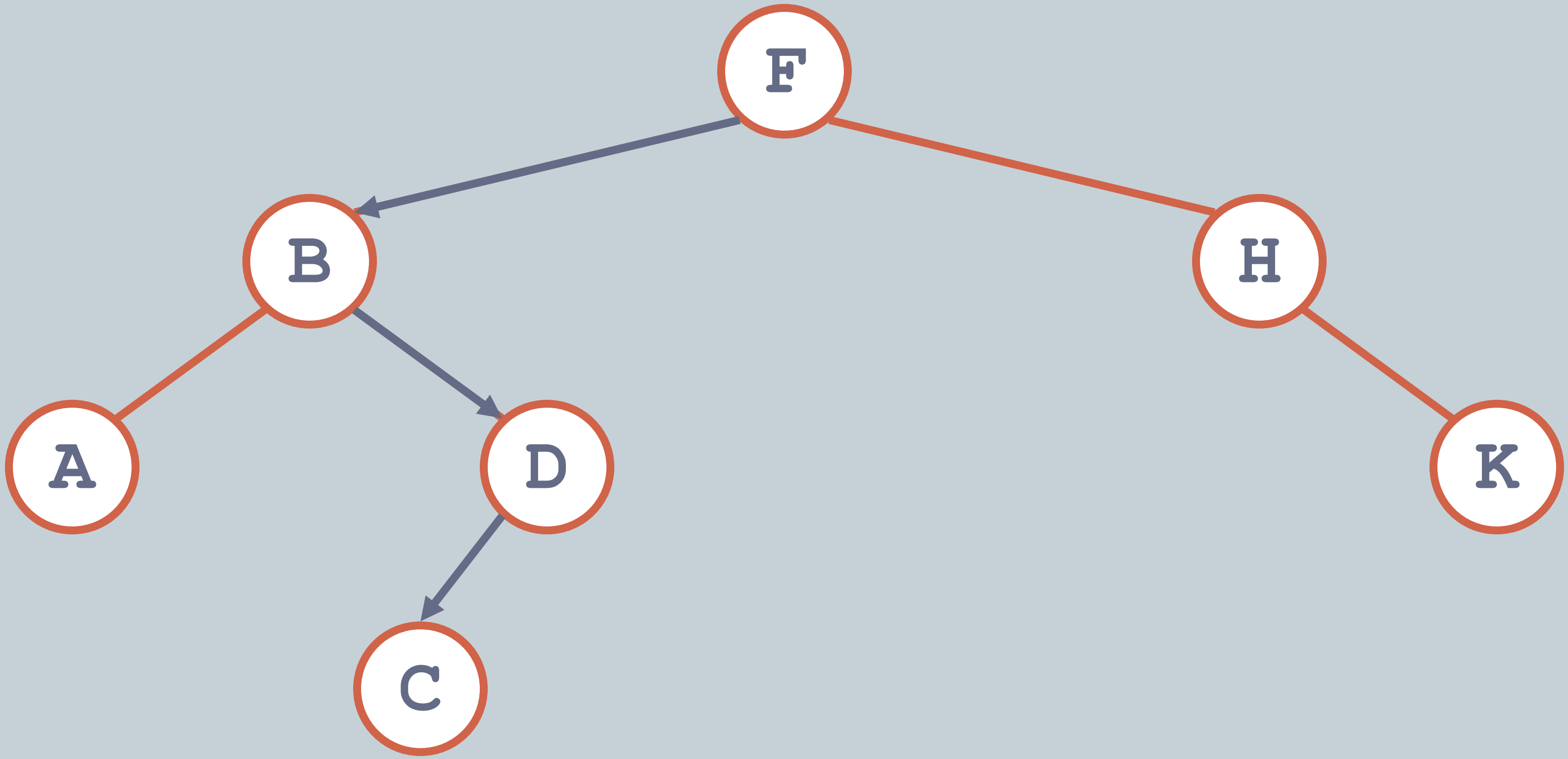
- *Which of these two functions is more efficient?*

Operations of BSTs: Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a "trailing pointer" to keep track of where you came from (like inserting into singly linked list)

BST Insert: Example

- Example: Insert *C*



BST Search/Insert: Running Time

- *What is the running time of Tree-Search() or Tree-Insert()?*
- A: O(h), where h = height of tree
- *What is the height of a binary search tree?*
- A: worst case: $h = O(n)$ when tree is just a linear string of left or right children
  - We'll keep all analysis in terms of h for now
  - Later we'll see how to maintain $h = O(\lg n)$

Sorting With Binary Search Trees

- Informal pseudo code for sorting array A of length *n*:

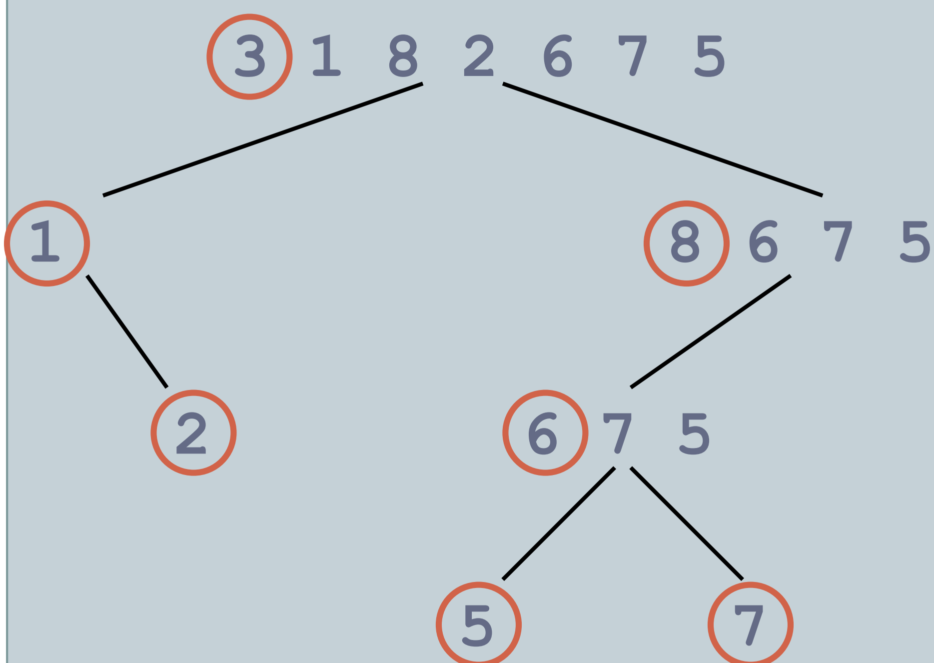
```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

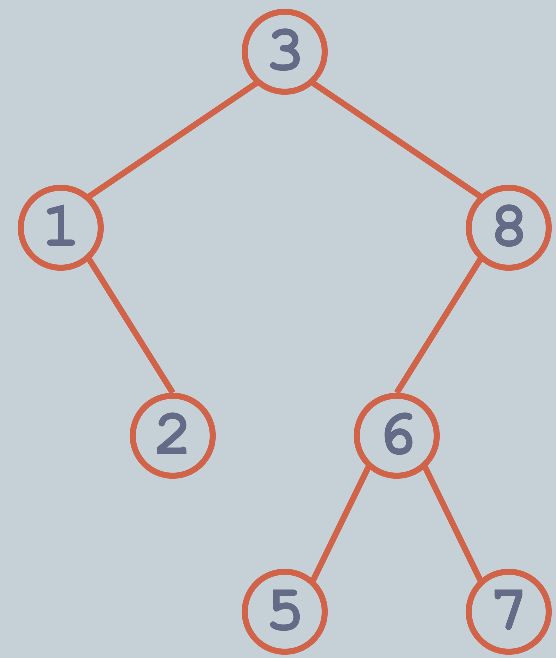
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
  - *Worst case?*
  - *Average case? (hint: remind you of anything?)*

Sorting With BSTs

- Average case analysis
  - It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```





Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as quicksort, different order!
    - Example: consider inserting 5

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTsort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTSort? Why?*
- A: quicksort
  - Better constants
  - Sorts in place
  - Doesn't need to build data structure

More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*
  - Insert
  - Minimum
  - Extract-Min

BST Operations: Minimum

- *How can we implement a Minimum() query?*
- *Where can the minimum be?*
- *What is the running time?*

- *What about the Maximum query?*

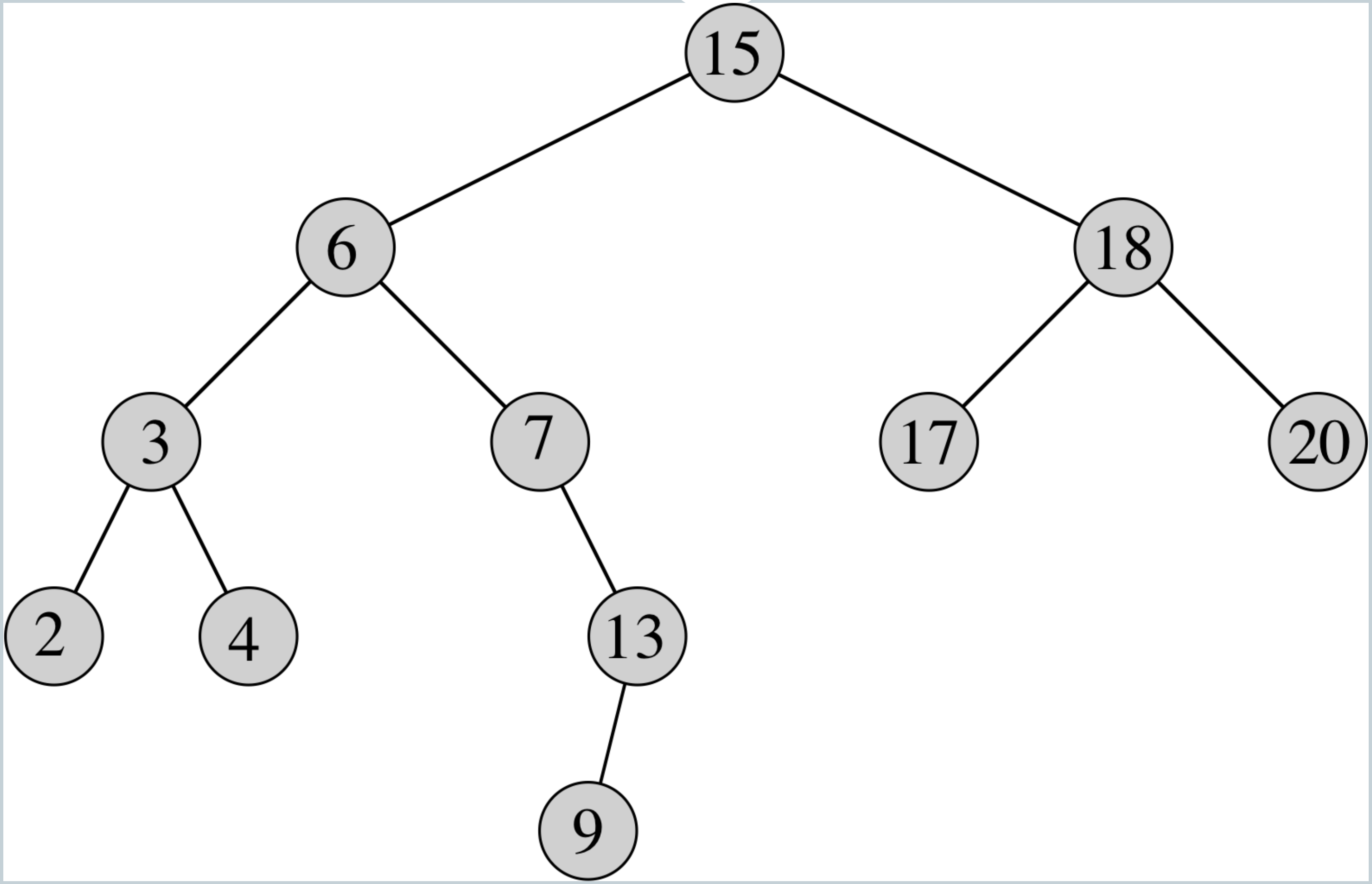
BST Operations: Successor

- For deletion, we will need a Successor() operation
- *What is the successor of node 3? Node 15? Node 13?*
- *What are the general rules for finding the successor of node x? (hint: two cases)*

BST Operations: Successor

- Two cases:
  - Case 1: x has a right subtree: successor is minimum node in right subtree (leftmost node in right subtree)
  - Case 2: x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- Predecessor: similar algorithm

Successor of 7(case 1) and 13(case2)

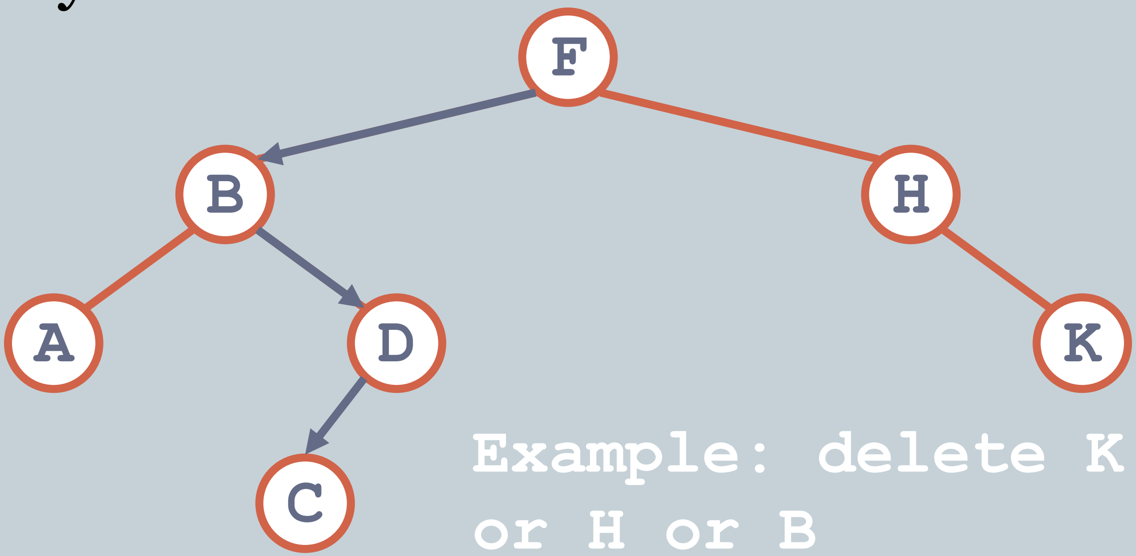


Successor

- TREE-SUCCESSOR(x)
- **if** x.*right* !=NIL
- **return** TREE-MINIMUM(x.*right*)
- y=x.*p*
- **while** y != NIL and x == y.*right*
- x = y
- y = y.*p*
- **return** y

BST Operations: Delete

- Deletion is a bit tricky
- 3 cases:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it

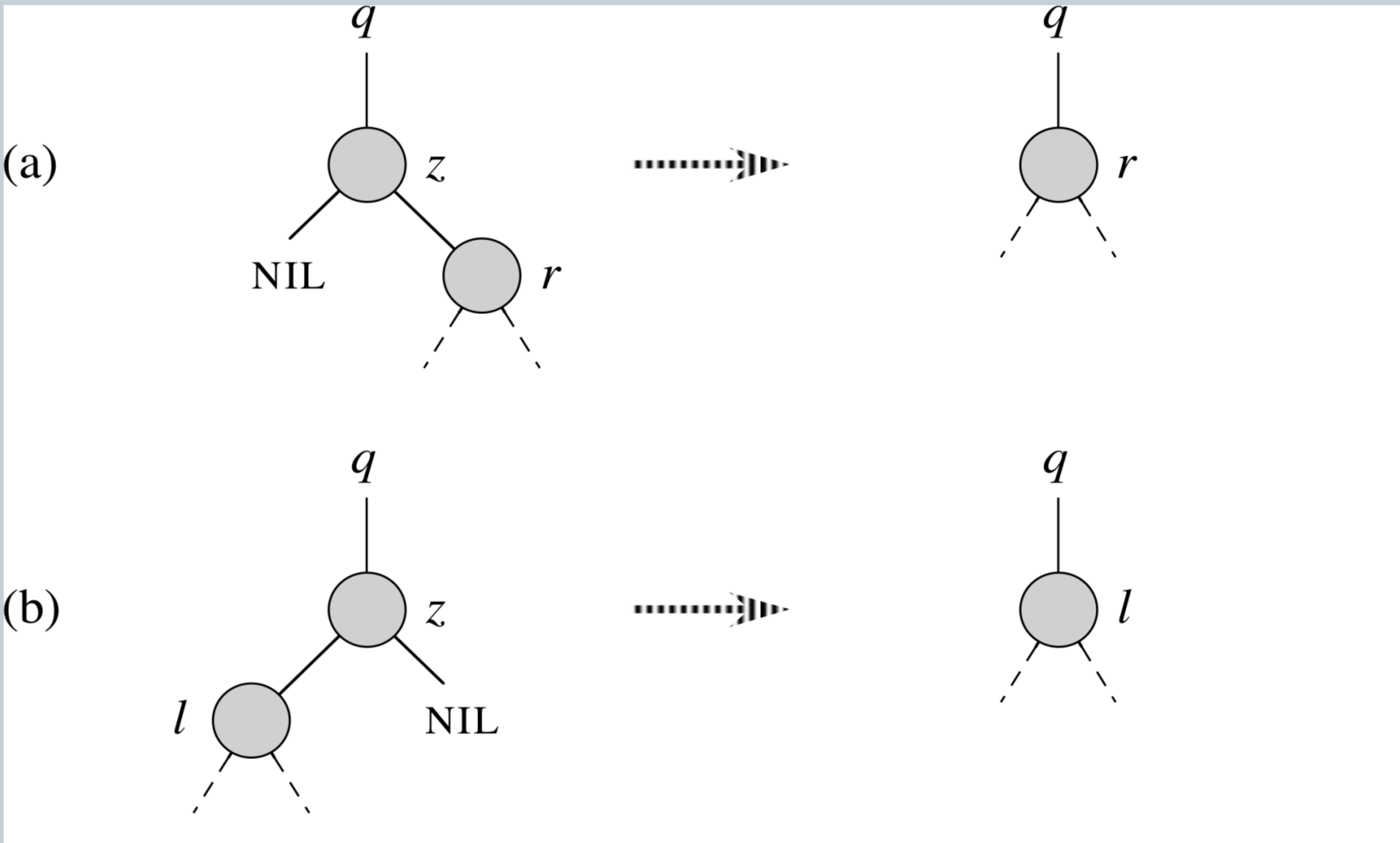


Example: delete K or H or B

BST Operations: Delete

- *Why will case 2 always go to case 0 or case 1?*
- A: because when x has 2 children, its successor is the minimum in its right subtree
- *Could we swap x with predecessor instead of successor?*
- A: yes. *Would it be a good idea?*
- A: might be good to alternate

Delete z



Delete z

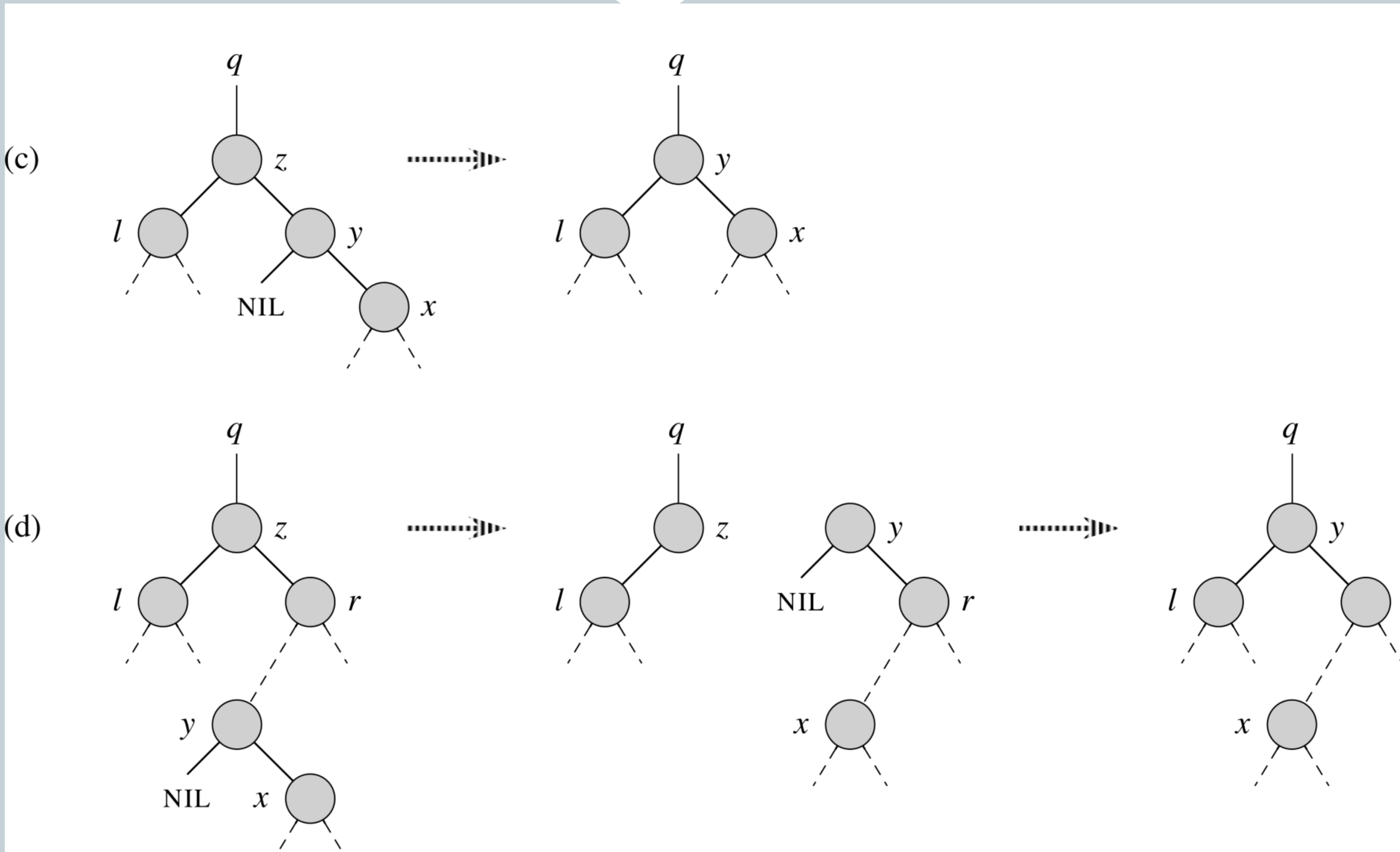


Fig1: 4, 1 & deletemin- Fig2:5, 8,-7, -11

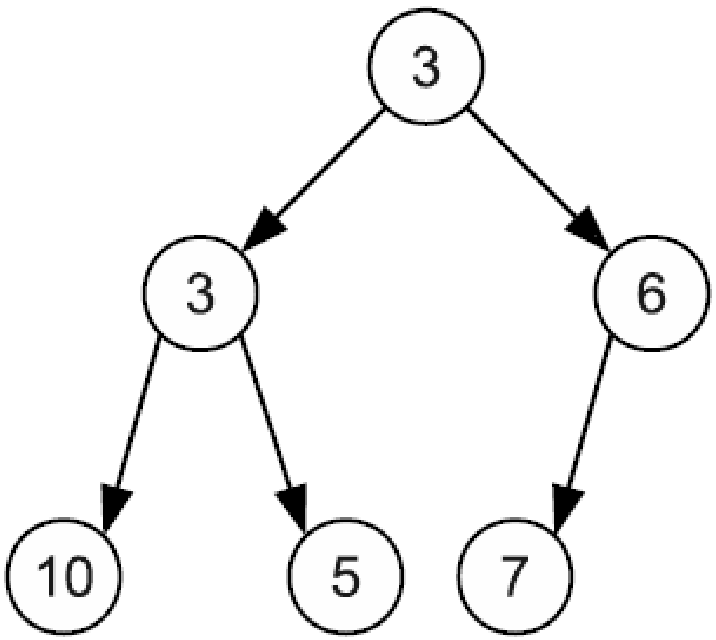


Figure 1: A heap

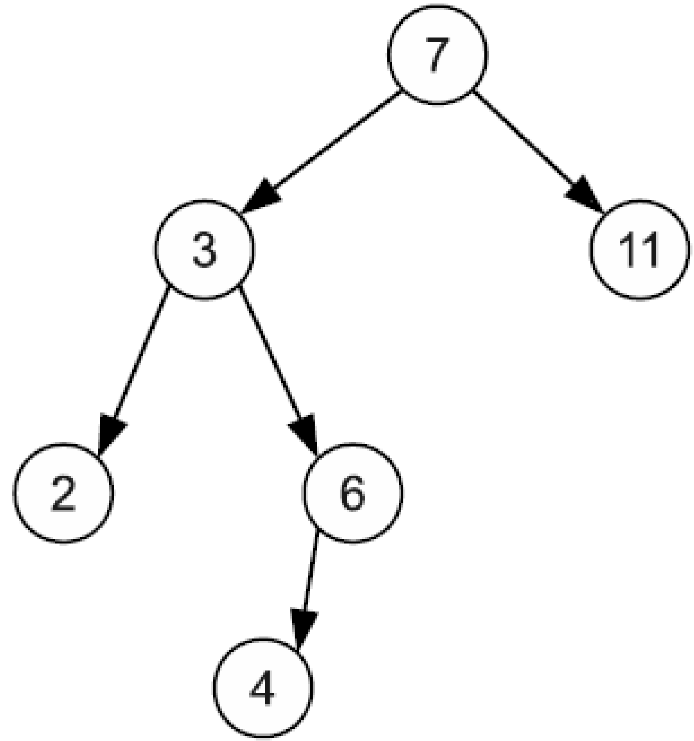


Figure 2: A binary search tree