Hash Tables

Dictionaries and Sets

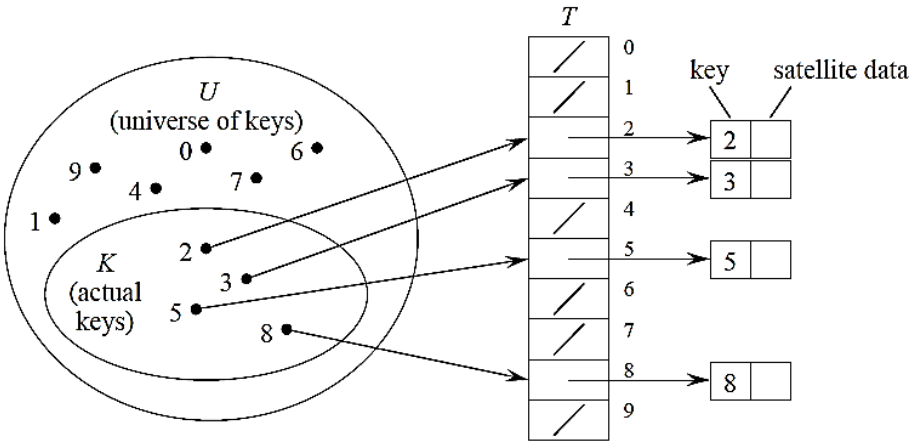
- Python dictionaries and sets are implemented using ***hashing.***
  - Sets consist of a collection of *n* keys drawn from an *immutable* universe or domain $\mathcal{U}$.
  - Dictionaries consist of a collection of (key,value) pairs, with keys drawn from an immutable domain $\mathcal{U}$.
- Members of sets and dictionaries are stored in a list of size *m,* called the ***hash table,*** by using a function $h: \mathcal{U} \rightarrow \{0, \ldots, m-1\}$ that maps objects in $\mathcal{U}$ to a position on the list.

Recall

- *Every* data object in Python is stored in binary (i.e., using a sequence of bits).
  - Integers are stored in binary (base 2) (e.g., $1101_2 = 1\times2^3 + 1\times2^2 + 1 \equiv 13$).
  - Real numbers are treated similarly (e.g., $0.125 = \frac{1}{8} = 1\times2^{-3} \equiv 0.001_2$).
  - In fact, most real numbers cannot be represented. Instead, they are approximated using rational numbers with a fixed number of bits.
  - Conversely, every binary sequence can be interpreted as an integer ⇒ when it comes to hashing it suffices to handle integer keys.

Direct Addressing

- Ideally, *h* is one to one.
- Since every bit string can be interpreted as an integer, let *h* be the *identity function.*
  - Each slot corresponds to a key in $\mathcal{U}$.
  - If there is an element *x* with key *k,* then *T*[*k*] points to *x.*




```
def Search(T,k):
     return T[k]
def Insert(T,x):
     T[key(x)] = x
def Delete(T,x):
     T[key(x)] = None
```

- However, direct addressing is rarely practical. Why?

Hash Tables

- Direct addressing is not practical when $\mathcal{U}$ is much larger than *K* (almost always the case)
- Would like space used to be small
  - Use a hash table of size $m = O(|K|)$
- Can still get $O(1)$ search time, independent of n, but on *average*, not *worst case*
- *Idea:* instead of storing *x* in slot $key(x)$, store *x* at position $h(key(x))$

  $h : \mathcal{U} \rightarrow \{0,1, \ldots, m-1\}$ is the ***hash function.***

  We say that *h* hashes *k* to $h(k)$.

Hash Functions

- Many different functions used in practice
  - Interpret key *k* as a number
- Division method
  - $h(k) = k \bmod m$
  - If *k* consists of many "digits," can interpret as a an $r + 1$ digit number in base *b*:

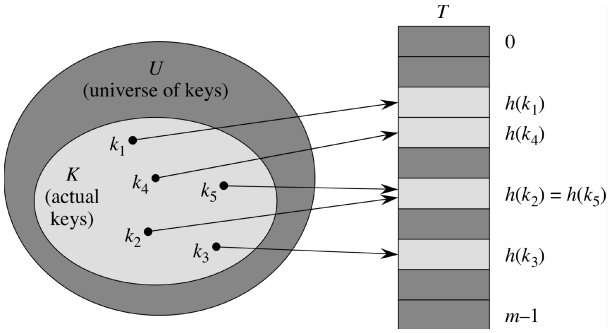
$$h(k) = \left(\sum_{i=0}^{r} k_i b^i \right) \bmod m$$

Example: 10011001110100011001101

- Multiplication method: $h(k) = \lfloor m(k \cdot A \bmod 1) \rfloor$

Collisions

- In practice, it is usually impossible to guarantee that *h* is one to one (why?)
- A **collision** occurs when two or more keys from *K* hash to the same slot
- Can happen whenever, even with well-designed hash functions
  - May or may not happen if $|K| \leq m$
  - Will definitely happen if $|K| > m$
- Can resolve by:
  - Finding a different slot (*open addressing*)
  - *Chaining* (i.e., storing all objects with same hash in another data structure—e.g., a list)
    - The hash table becomes a list of lists
    - Primary list of size *m,* secondary lists of variable size

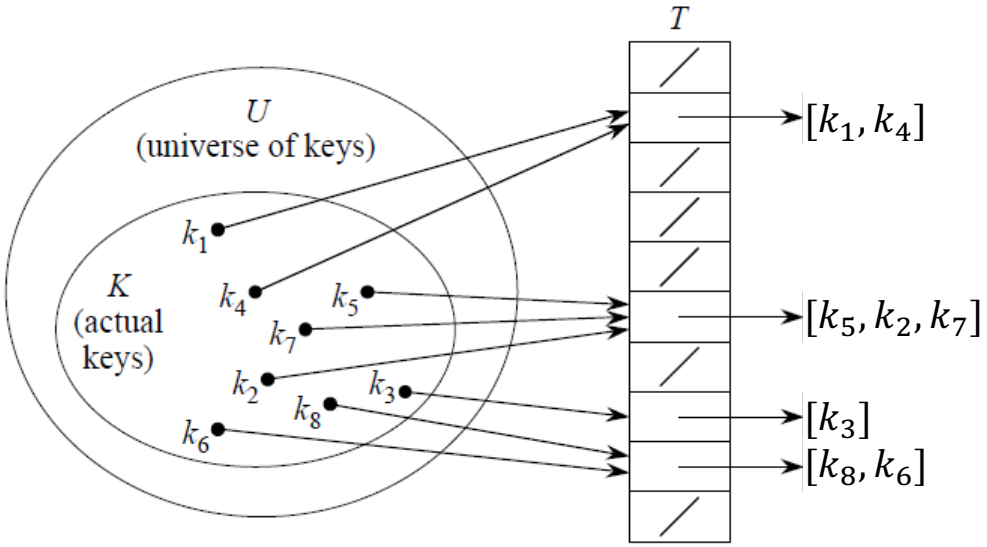


END

Chaining

Chaining

- Use a secondary list to store all items that hash to the same slot.



- n_i denotes the length of the ith list.

Operations

- Insertion: **`Insert(T,x)`**
  - Append *x* to the list $T[h(key(x))]$
  - Can be done in $O(1)$ time
- Search: **`Search(T,k)`**
  - Search for element with key *k* in list $T[h(k)]$
  - Worst case is proportional to len$(T[h(k)])$
- Deletion: **`Delete(T,x)`**
  - Given the position *i* of *x,* simply del $T[h(key(x))][i]$ or $T[h(key(x))]$.pop(i) or $T[h(key(x))]$.remove$(key(x))$
  - Worst case is proportional to len$(T[h(key(x))])$
  - With more advanced data structures, can be done in $O(1)$ time

Performance

- Good performance is guaranteed provided that secondary lists are short.
- Long lists can form if too many items hash to the same value.
- In the worst case, the hash function is 0 and everything goes into the list at the 0th position.
  - This is no different than just using a list.
- Picking a good hash function is a function of the table size as well as the specific data being put into the table.

END

Open Addressing

Open Addressing

- Store all keys in the hash table itself
- Each slot contains either a key or **`None`**
- To search for key k, probe slot $h(k)$
  - If slot $h(k)$ contains key k, the search is successful.
  - If this slot contains **`None`**, the search is unsuccessful.
  - If it contains a different key, we need to look elsewhere: We compute the index of some other slot and continue probing until we either find k or **`None`**.
- Need the sequence of slots probed to be a permutation of $\langle 0,1, \dots, m-1 \rangle$ so that we probe all slots if necessary and never probe a slot more than once
- Hash function becomes $h : \mathcal{U} \times \underbrace{\{0, \dots, m-1\}}_{\textit{Probe number}} \to \underbrace{\{0, \dots, m-1\}}_{\textit{Slot number}}$

Operations

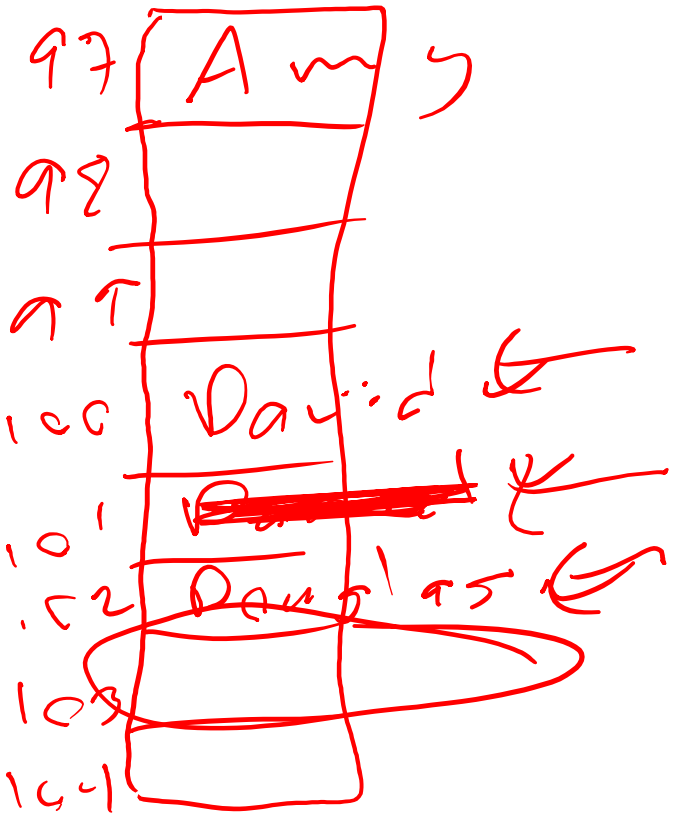
```
HASH-SEARCH(T, k)
  i = 0
  repeat
      j = h(k, i)
      if T[j] == k
          return j
      i = i + 1
  until T[j] == NIL or i = m
  return NIL
```

```
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"
```

How about deletion?

Example

- Hash function is ascii value of first letter of word
- Insert: David (hash: 100)
- Insert: Amy (hash: 97)
- Insert: Daniel (hash: 100)
- Insert: Douglas (hash: 100)
- Lookup: Douglas (hash: 100)
- Lookup: Devin (hash: 100)
- Delete: Daniel (hash: 100)
- Lookup: Douglas (hash: 100)

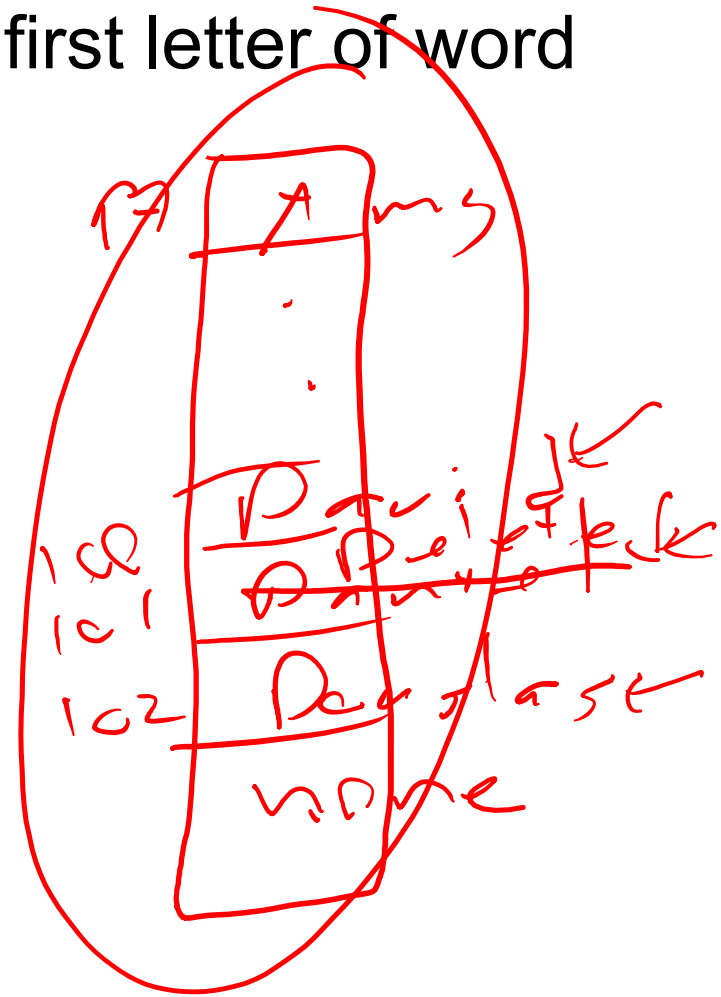


Deletion

- Suppose that you want to delete key k
- Cannot just store None in the slot containing k—why?
- Instead, store a special value **`Deleted`** in the slot where k used to be
- Search treats **`Deleted`** as though the slot holds a key that does not match the one being searched for
- Problem is that performance no longer depends on the current size of K (the load factor)
- Insertion treats **`Deleted`** as though the slot were empty so that it can be reused

Example

- Hash function is ascii value of first letter of word
- Insert: David (hash: 100)
- Insert: Amy (hash: 97)
- Insert: Daniel (hash: 100)
- Insert: Douglas (hash: 100)
- Lookup: Douglas (hash: 100)
- Lookup: Devin (hash: 100)
- Delete: Daniel (hash: 100)
- Lookup: Douglas (hash: 100)



Computing Probe Sequence

- The goal is *simple uniform hashing:* Each key is equally likely to have any of $m!$ permutations of $\langle 0,1,\ldots,m-1\rangle$ as its probe sequence.
- It's hard to realize, so use approximations via auxiliary hash functions: h', h'', …
  - Linear probing: $h(k,i) = (h'(k) + i) \bmod m$
  - Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
    - Must constrain m, c_1, c_2 to guarantee a full probe sequence
  - Double hashing: $h(k,i) = (h'(k) + i \cdot h''(k)) \bmod m$
    - Must have $\gcd\big(m, h''(k)\big) = 1$—why?

END

Data Structure Selection

Data Structure Selection

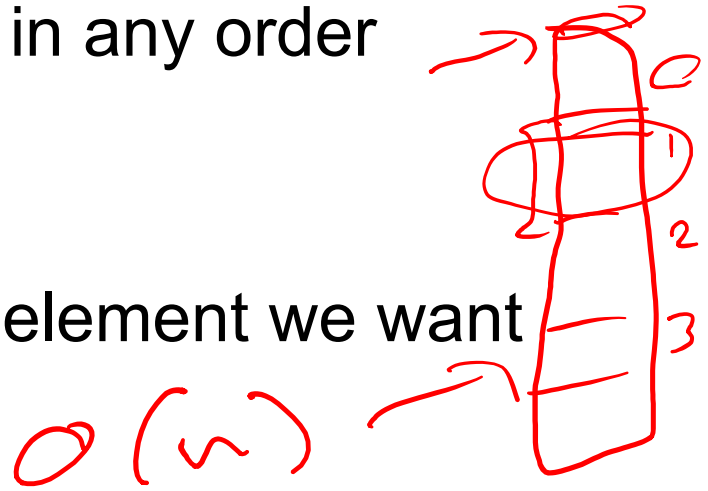
- We have now seen the several data structures.
  - List
  - Stack/queue
  - Heap
  - Hash table
- How do we decide which one to use for a particular problem?

Data Structure Selection

- Each one offers a different set of operations.
- Each one has a different runtime efficiency.
- As a general rule, the more restrictive the set of operations, the faster the internal implementation.

Lists

- Lists are the most general in terms of operations
- Every element can be accessed in any order
- Can do anything with a list
  - Just not always fast
- O(1) if we know the index of the element we want
- Slower runtimes if we need to
  - Maintain a particular ordering
  - Rearrange elements
  - Search for a particular target element
  - Search for an element meeting a property (maximum)



Stacks/Queues

- Stacks and queues restrict which elements you can access
  - The first and last elements
- Input ordering is maintained when output
- O(1) to insert or remove elements
- Not possible to:
  - Rearrange elements
  - Search for a particular target element
  - Search for an element meeting a property (maximum)

Heaps

- Heaps only allow you to remove the elements in order
  - Biggest first (maxHeap) or smallest first (minHeap)
- No indexed access
- Input order is not maintained
- Only useful for keeping a sorted list
- O(log n) to remove the top element or add element
  - Runtime to maintain a sorted list with an actual list is slower

Hash Tables

- Hash tables give fast access for lookup operations
  - Example: Find the phone number for a given name.
- No indexed access
- Input order is not maintained
- Sorted order is not maintained
- O(1) lookup and insertion
  - Runtime to do lookup with an actual list is slower.

Hybrid Structures

- Many basic data structures are often provided by the programming language libraries.
  - List, dequeue, dictionary, and so on
- But sometimes a hybrid structure needs to be implemented for a particular application.
  - We need fast lookup of names but also want to maintain the order in which the names were entered into the system
    - Linked list running through a hash table

END

Graph Search

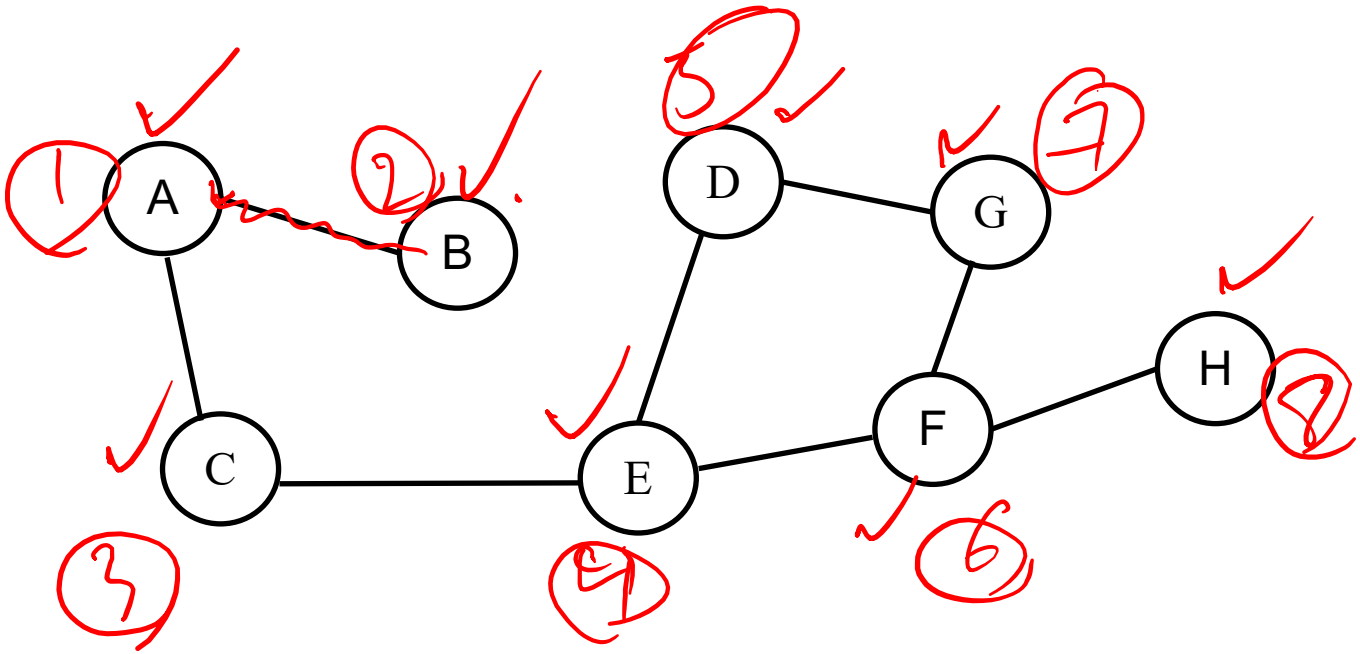
Systematically Exploring a Graph

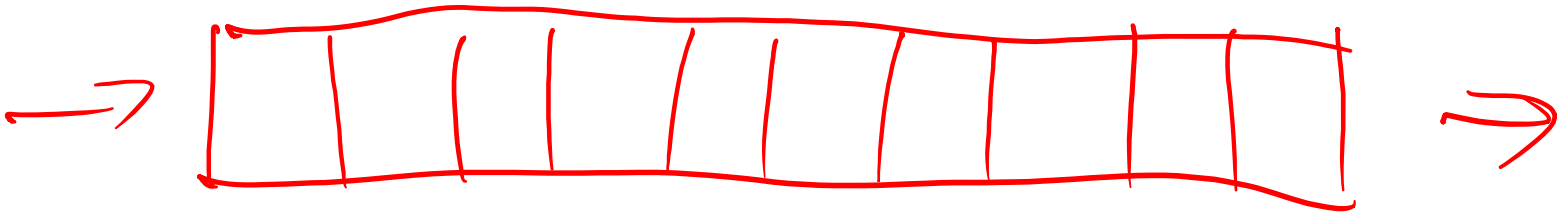
- Two procedures allow you to explore a graph in linear time and compile information about the graph: *breadth-first search* (BFS) *and depth-first search* (DFS).
- DFS and BFS have similar structure.
  - Both keep a data structure *D* of pending edges to explore and repeatedly choose an edge from *D* to explore.
  - *D* is initially loaded with the edges adjacent to an arbitrary vertex *v.*
  - Both procedures run in linear time.
- DFS and BFS differ only on the nature of *D*.
  - DFS uses a stack.
  - BFS uses a queue.

DFS/BFS Steps

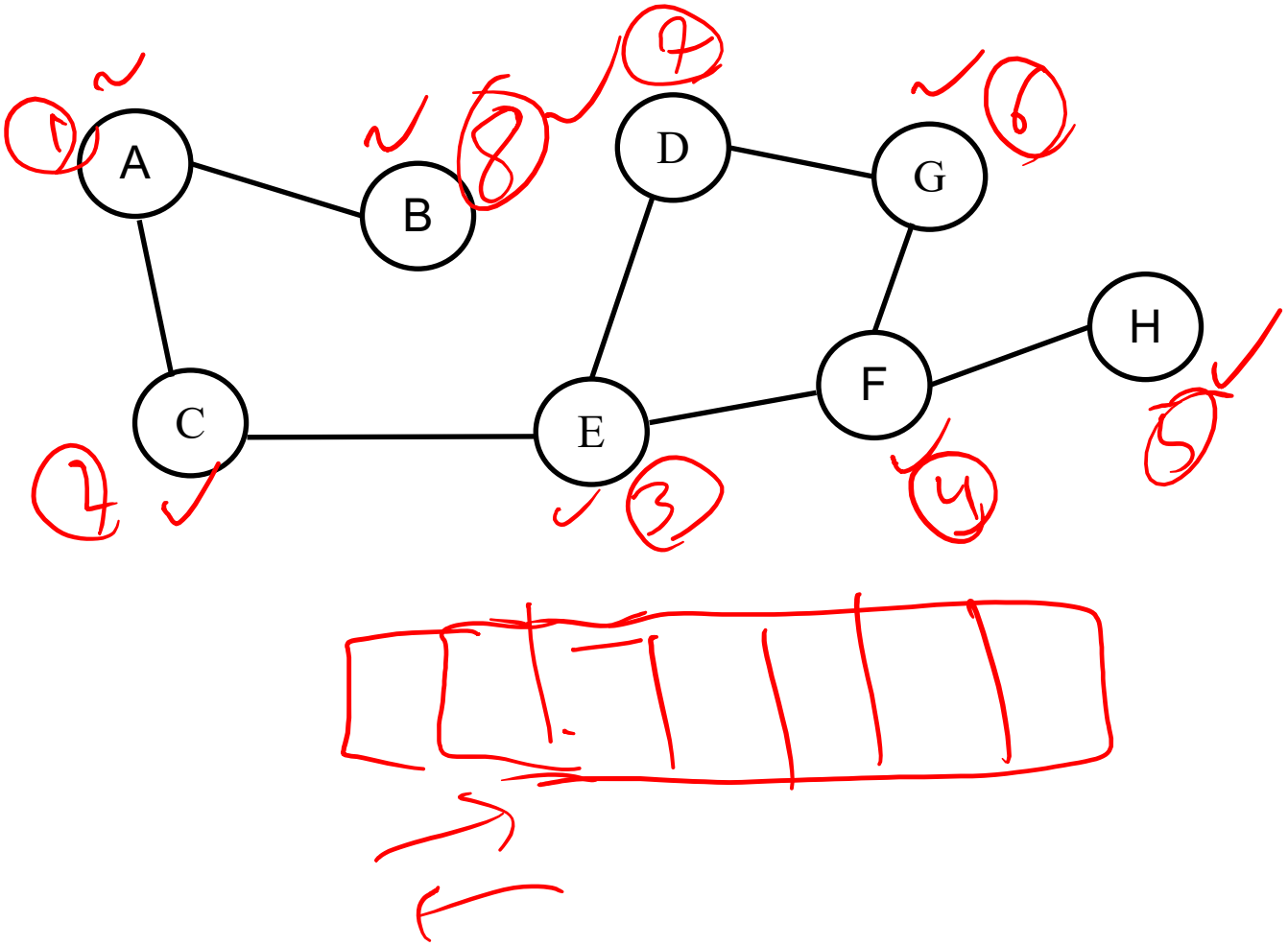
- Given a graph G, containing vertices (V) and edges (E)
- Pick the starting vertex, V1
- Mark V1 as processed
- Push V1 and all its edges to neighboring nodes into the data structure (stack or queue)
- While the data structure is not empty:
  - Pop top/front element off data structure
  - Mark vertex as processed (if not already)
    - Record edge as part of the spanning tree
  - Push all neighbors of vertex that have not already been processed onto data structure
    - Along with the edges to those neighbors

BFS Example





DFS Example

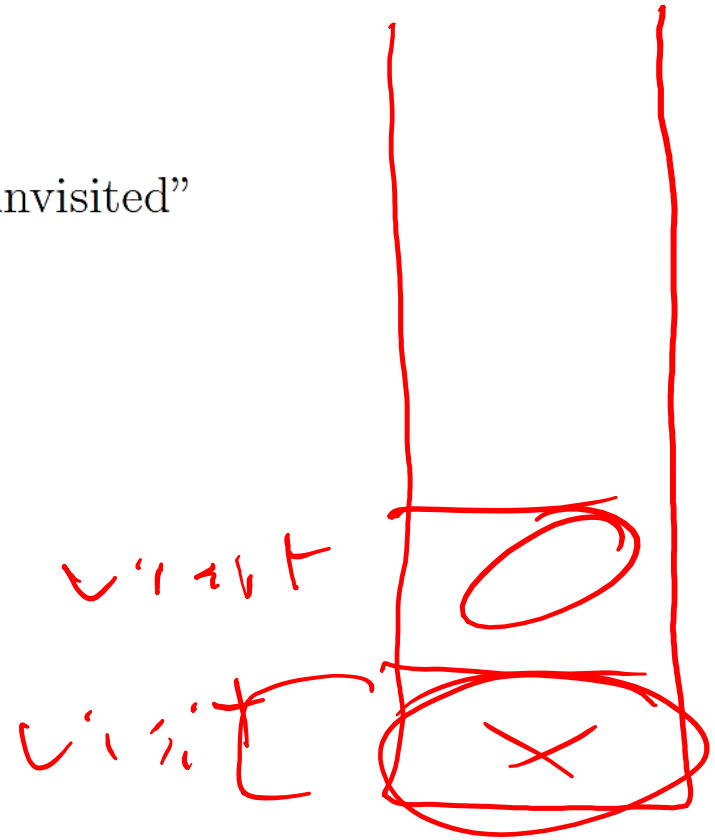


Recursive Depth-First Search

When implemented recursively, DFS makes use of the system stack.

```
DFS(V)
1  T ← ∅
2  for i ← 1 to n
3        do DF#[v_i] ← 0        ▷ mark "unvisited"
4  count ← 1
5  for i ← 1 to n
6        do if DF#[v_i] = 0
7              then VISIT(v_i)

VISIT(v)
1  DF#[v] ← count
2  count ← count + 1
3  for each vertex w in v's adjacency list
4        do if DF#[w] = 0
5              then add (v, w) to T
6                   VISIT(w)
```



Connected Components

- Consider a wireless sensor network consisting of n sensors, each of which can transmit within distance r. Pairs of sensors that can directly communicate are linked with an edge. This information can be used to partition the sensors into sets of sensors that can communicate directly or indirectly.
- Consider a pool of customers using different accounts. You wish to identify related sets of people (e.g., families). You create edges between customer IDs based on the same credit card usage, or address, and so on. The information can be used, for example, to estimate the possibility of fraud.
- Both of these problems are conveniently modeled using undirected graphs whose vertices need to be partitioned into sets of related notes, the connected components of the graph.

Connected Components

Can modify DFS to find the number of connected components

```
CC(G)
// G = (V, E) is an undirected graph with vertices v_1, ..., v_n.
1  for i = 1 to n
2        DF#[v_i] = 0              // mark "unvisited"
3  count = 1
4  numCC = 0
5  for i = 1 to n
6        if DF#[v_i] == 0
7              numCC = numCC + 1
8              VISIT(v_i)
9  return numCC

VISIT(v)
1  DF#[v] = count
2  count = count + 1
3  for each vertex w in v's adjacency list
4        if DF#[w] == 0
5              VISIT(w)
```

END

Acknowledgements

- Introduction to Algorithms, 3rd edition, by T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein; MIT Press, 2009