LC 101

Data Structures

March 13, 2017

Data Structures

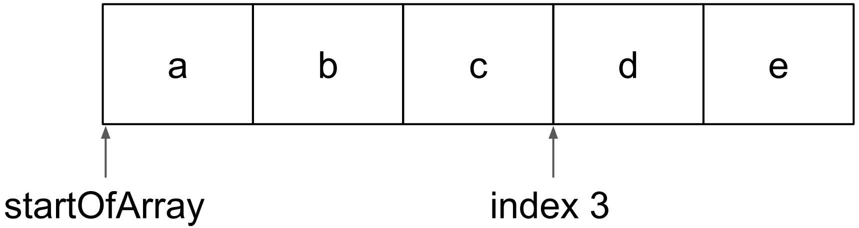
- *Data structures* are ways of organizing data
  - Both abstract ideas and concrete implementations
- Certain basic data structures occur frequently
- Choice of data structure can have a large impact on performance
  - Implementation can also have a large impact on performance

Lists

- A *list* is a linear sequence of data
    - Abstractly, think of writing down a list of things, like a shopping list
- Lists can be categorized by some features
    - Static vs dynamic
        - A *static* list has a fixed size and cannot grow or shrink
        - A *dynamic* list can change size
    - Direct access vs sequential access
        - In a *direct access* list, any element can be accessed efficiently
        - In a *sequential access* list, elements can only be accessed by starting at the beginning of the list and iterating through the elements

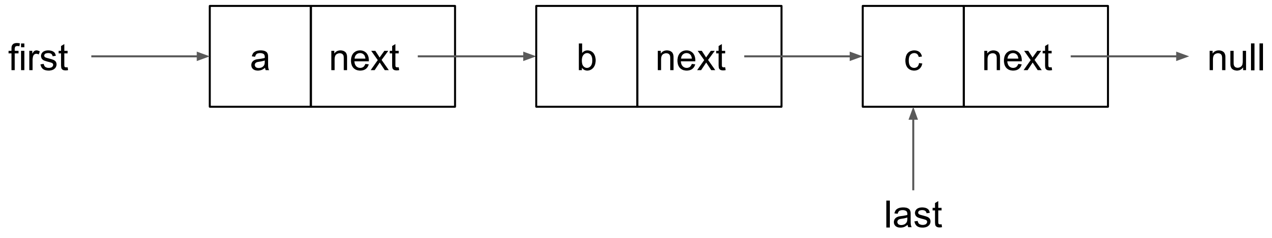
Array

- An *array* is generally defined as a *static, direct access* list
  - Note that this is different from the "array" type in JavaScript or the "list" type in Python
    - These are *dynamic*, direct access lists
      - These are actually *array lists* (more on this later)
  - Languages such as C, C#, and Java have array as a built-in type
- Must be created with a specific size
- Generally implemented as a sequential area of memory
  - Location of an element can be calculated by: **`startOfArray + index * sizeOfElement`**



Linked List

- A *linked list* is a dynamic, sequential access list
  - Most languages do not have this as a built-in type, though many have it in their libraries
- Implemented by each *node* in the list having a reference to the next node
- A node can only be accessed by starting at the first node and *walking* down the list
  - A reference to the last node if often kept for quick access to the end of the list



Doubly Linked List

- A *singly linked* list can only be access in one direction (from head to tail)
- A *doubly linked* list includes links going the other direction so it can be iterated over in either direction

Array vs Linked List

- Array
  - Access to any element is fast
    - Constant time – access time is unrelated to array size
  - Size must be specified when created and cannot change
  - We can remove an element by setting its location to null, but actually removing the hole that is left would require shifting all subsequent elements left one slot (which is expensive)
- Linked List
  - Access to any element other than the first or last is slow
    - Linear time – average time to access an element grows as the list grows
  - Can grow or shrink as needed
  - Can actually remove elements from the list, causing the list to shrink
  - Extra memory needed for the links

Array List

- An array list (also called a dynamic array) is a *dynamic, direct access* list
  - This is actually the “array” type in JavaScript and the “list” type in Python
  - Often provided in a library for other languages
- Generally implemented by
  - Starting with a fixed size array
  - If the list grows too large for the array then resize by
    - Allocating a new, larger array
    - Copying the elements from the old array to the new array
- The resize operations are expensive, but if we *amortize* the cost over all of the additions then adding an element is still constant time

Stack

- A *stack* is a linear data structure with two basic operations
  - *Push* will add an element to the top of the stack
  - *Pop* will remove an element from the top of the stack
  - A third operation, *peek*, is often included as it is cheap to implement
    - *Peek* lets you retrieve the top element of the stack without removing it
- Provides *LIFO (Last In, First Out)* access
- Think of a stack of cards where you can put a card on top of the stack or look at or remove a card from the top of the stack, but can’t access any cards further down in the stack

Stack Implementation

- A linked list can be easily used to implement a stack as it is efficient to add or remove elements from the head of the list (or the tail if a `last` reference is used)
- If an array list (or array) is used to implement a stack then the additions and deletions should be done at the *end* of the array list, not the front
  - Adding or removing an item at the front of the array list would require shifting all of the other elements (a very expensive operation)
  - An array should only be used if the stack has a *maximum depth*

Queue

- A queue is a linear data structure with two basic operations
  - *Enqueue* will add an element to the end of the queue
  - *Dequeue* will remove an element from the front of the queue
- Provides *FIFO (First In, First Out)* access
- Think of the checkout line at a store where customers are checked out in the order in which they join the line

Queue Implementation

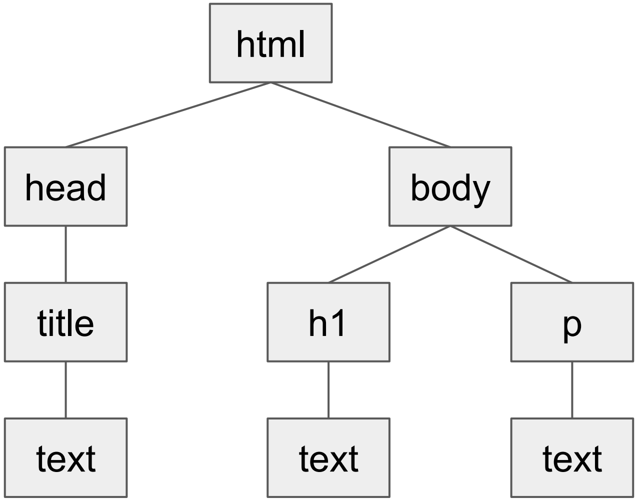
- A linked list with a `last` reference can be used to efficiently implement a queue
  - Enqueue could add an element to the head of the list and dequeue could remove it from the tail (or vice versa)
- An array or array list cannot easily be used to efficiently implement a queue
  - Either the enqueue or dequeue operation will require shifting all of the elements in the queue
    - (Actually, it *is* possible to efficiently implement a queue with a maximum length using an array with a technique called a *circular buffer*)

Deque

- A deque is a double-ended queue
  - Basically just a queue where elements can be enqueued or dequeued at either end of the queue

Tree

- A *tree* is a hierarchical data structure where every node (except the *root* node) has a single *parent* node
    - Most often represented with the root node at the top and the descendant nodes in layers below with links between parent and child



Tree Terminology

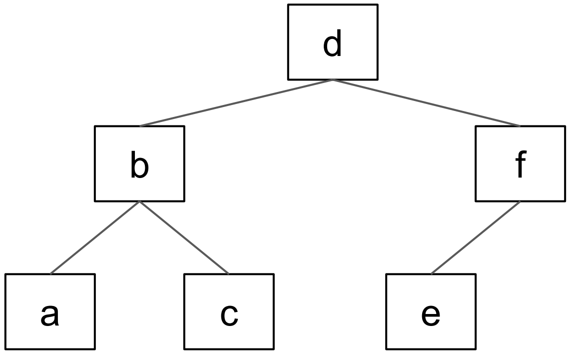
- *Root* – the top node of a tree (the only node without a parent)
- *Parent* – the node directly connected to another node when moving towards the root
- *Child* – a node directly connected to another node when moving away from the root
- *Siblings* – a group of nodes with the same parent
- *Descendant* – a child, or grandchild, or great grandchild, etc., of a node
- *Ancestor* – a parent, or grandparent, or great grandparent, etc., of a node
- *Leaf* – a node without children
- *Internal node* – a node with at least one child
- *Vertex* – another term for node (more abstract term)
- *Edge* – the connection between nodes
- *Path* – a sequence of nodes and edges between two nodes
- *Depth* of a node – the distance (number of edges) between a node and the root
- *Level* of a node – one plus the depth
- *Height* of a node – the length of the longest path from that node to a descendant leaf
- *Height* of a tree – the height of the root

Tree Implementation

- A tree **could** be implemented with just one reference per node by giving each node just a parent reference, but this is generally not very useful
  - No way to get from a parent to its children, which is a common operation
- List of children
  - Each node could have a list of references to its children
- First child, next sibling
  - Each node could have a link to its first child and its next sibling
- Order of the children is often important

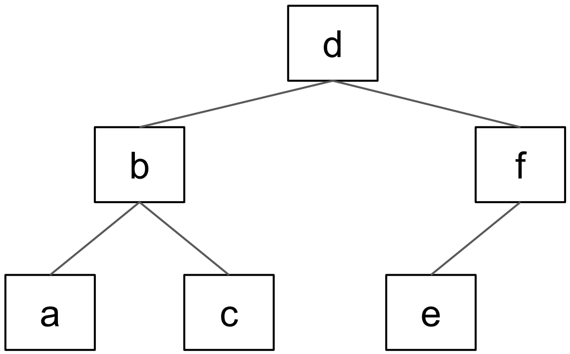
Binary Tree

- A *binary tree* is a tree where each node has at most two children



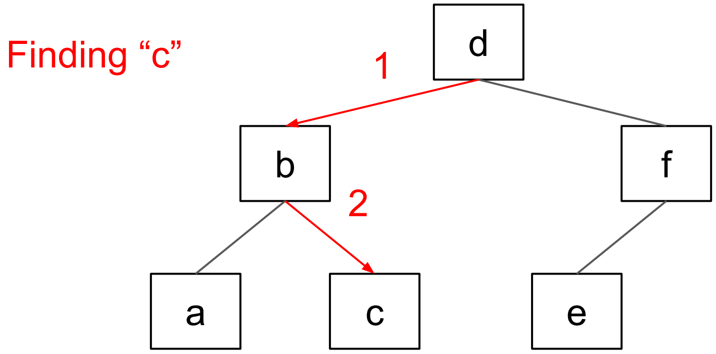
Binary Search Tree

- A binary search tree is an ordered tree where
  - All the left descendants of a node are less than (or equal to) that node's element
  - All the right descendants of a node are greater than (or equal to) that node's element



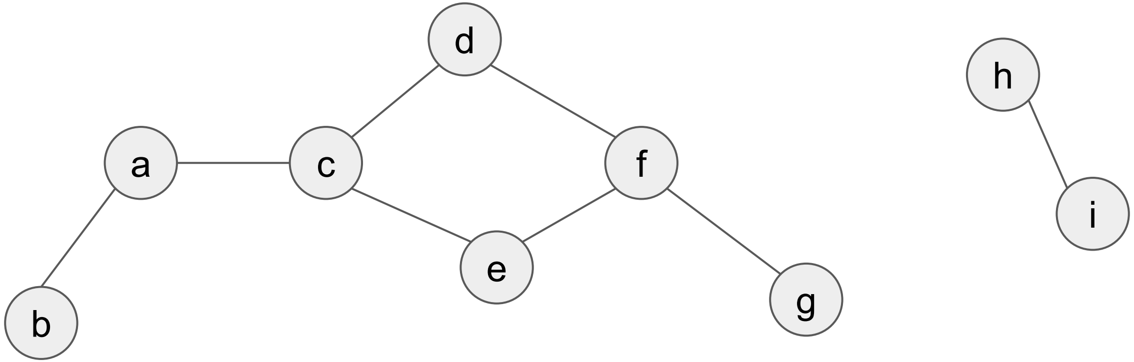
Binary Search Tree

- We can find a particular value in the tree by
  - Starting at the root node
  - Following either the left or right branch depending on whether the value is less than or greater than the current node



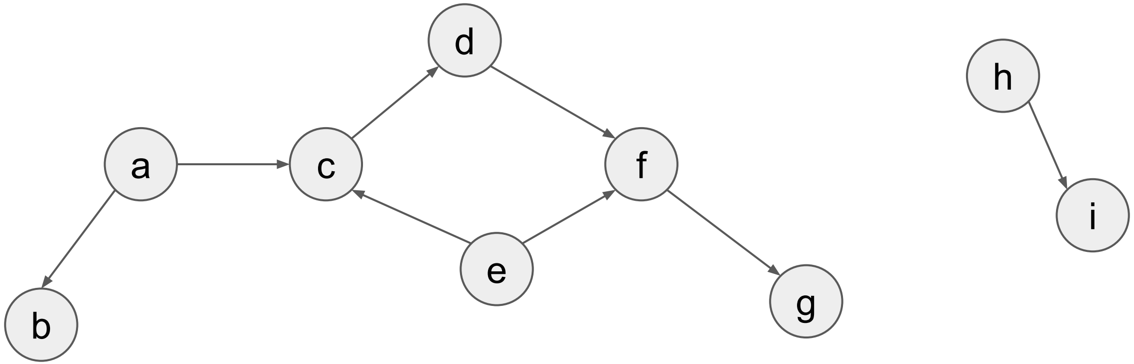
Graph

- A graph is a collection of nodes (a.k.a. vertices) and edges connecting pairs of vertices



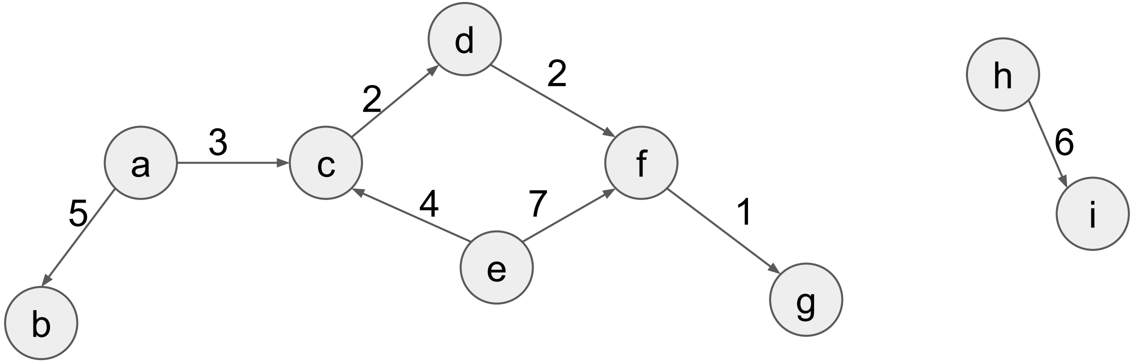
Directed Graphs

- Graphs can be *undirected* or *directed*
  - In a directed graph each edge has a direction



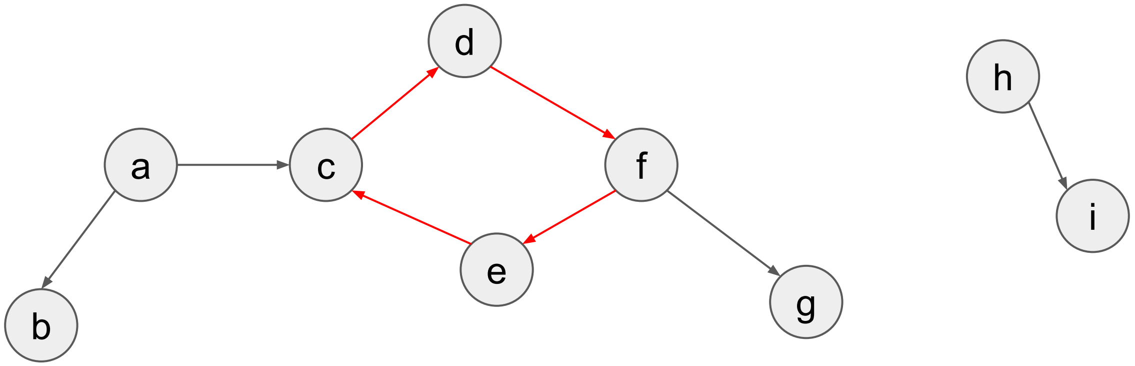
Weighted Graphs

- Graphs can be *unweighted* or *weighted*
  - In a weighted graph each edge has a weight
    - The weight is just some value associated with the edge



DAG

- A DAG is a directed acyclic graph
  - A directed graph has a cycle if there is path leading from any node back to itself



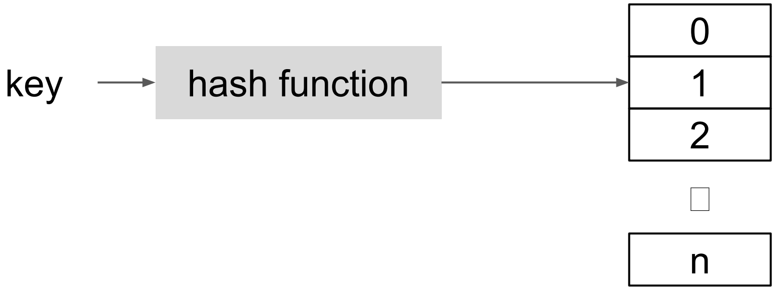
Cycle in red (so graph is **not** a DAG)

Map

- A *map* (a.k.a. *dictionary*, *associative array*) maps a collection of keys to values
  - Python has a built-in dictionary type. JavaScript objects are basically maps. Other languages often implement maps in libraries
- Basic operations:
  - Add a key:value pair
  - Remove a key:value pair
  - Look up the value for a key
  - Update the value for a key

Map Implementation

- A map is often implemented as a *hash map* (a.k.a. *hash table*)
  - A *hash function* is used to convert a key into an index
    - Note that though the name is the same, this hash function is much simpler and much faster than the cryptographic hashes we talked about in unit 2
    - Needs to be extremely fast
    - Should generate an index value between 0 and some maximum value *n*
  - The value returned from the hash function is then used as an index into an array



Collision Resolution

- Since multiple keys can (and often will) hash to the same index, we need a way of resolving *collisions*
- One way of resolving collisions is *chaining*
    - Each element of the array is itself a list (usually a linked list) where each element of the list is a key:value pair