Data Structures
Statistics 650/750
Week 3 Thursday

Christopher Genovese and Alex Reinhart

13 Sep 2017

Announcements

- Editor session followup
- Scripting Assignment: `new-homework`
- Vignette Branching Policy

Introduction to Data Structures

A **data structure** is a formal organization of data that supports a pre-specified set of operations on it, designed to allow efficient access and manipulation of the data it contains.

Most data structures are instantiated and maintained in main memory during the execution of a program, but some are designed to be work efficiently with external storage.

Data structures mesh closely with *algorithms*. Algorithms operate on data structures to produce their results, and the choices of data structure and algorithm can have a dramatic effect on the efficiency of a computation.

The most important feature that guides our choice of data structures is if a particularly representation allows us to efficiently compute the answers we need for our problem.

Aside on Computational Complexity

We often have a choice of several data structures and algorithms to solve a problem. How can we understand and compare our choices?

Computer scientists use "big O notation" for as a tool for the analysis of algorithms. The basic premises are these:

1. Every basic operation performed by a computer (e.g. addition, multiplication, comparison, etc.) takes the same amount of time.
2. An algorithm is composed of many of these operations.
3. The number of these operations executed for any given input depends on the size of the input, n.

It would be impractical to count *every* operation performed by a given algorithm – this would depend on the specific implementation, the programming language used, optimizations performed by the compiler or interpreter, and so on, and would hence be nearly impossible to find out.

Instead, big O notation captures the *order* of the relationship between n and the number of operations.

Mathematically, say g(n) is the number of operations performed by a specific algorithm on an input of size n. We say the algorithm is O(f(n)) iff a constant multiple of f(n) is an *upper bound* of g(n) for n sufficiently large.

Typically we simplify f(n) by dropping lower-order terms and constants.

There are related notations, such as "big Omega" notation, which gives a *lower* bound to the growth of the number of operations, and "big theta", which says a function is bounded above and below by a function of the same order.

The purpose here is to focus on the *algorithm itself*, not its implementation in whatever programming language we've chosen. How fast is the *algorithm* given different size inputs, regardless of whether we implemented it on a fast computer or a slow programming language or with the world's stupidest code? Different algorithms have different intrinsic performance, and choosing the right algorithm is the first step in the process of optimizing your code.

Examples:

- Computing the mean of a length n vector is an $O(n)$ operation.
- Computing the pairwise distances for a set of n points is an $O(n^2)$ operation.
- Computing all permutations of a string of length n is an $O(n!)$ operation.

Overview of Core Data Structures

See Core Data Structures below for details on structure, operations, performance.

Data Types

A **data type** (or **type** for short) is a classification of data (usually in a programming language) that specifies to a compiler or interpreter how the data is to be stored and which operations can be legally performed on it.

Programming languages support a variety of data types. These are the basic ingredients we use in the data structures we will see below.

Primitive Types

These represent data that are stored in a format that is (or is close to one) supported natively by the hardware and that have direct support as types in a programming language.

Boolean	true, false
Integer	signed or unsigned integers in various ranges
Numeric	fixed precision *floating point numbers*
Character	represents a single glyph, various representations
Pointer	an address in memory

There are others (e.g., fixed-width numbers, rational numbers, complex numbers) but these are the most commonly supported.

- Examples In C++, each variable must be *declared* to be of a certain type; primitive types have reserved words to describe them:

  In languages like Python or R, data types are not typically specified by the programmer but are inferred implicitly.

Aggregate Types

These represent simple collections of values (of specified types) stored in a *contiguous* block of memory. They are building blocks in many more complex data structures.

Tuple	fixed-length sequence of values, with each *element* having a specified type
Array	contiguous, fixed-length collection of values all of the same type
Record/Struct	structured group of named attributes of various types in contiguous storage
Union	region of memory that can store several specified types (sometimes tagged)

In python, we have tuples that look like `(4,'Hello',True)` or `(17.4,-32.3)`.
In C++, we specify records (called structs) with explicit types:

Object Types

Most programming languages offer a facility for defining

String	A sequence of characters
Enumeration	A value that can take only one
Function	A callable object; that is, a ... function
File	A representation of a file in a file system
Stream	A stream of data elements made available sequentially
BigNum	An arbitrary-precision number
Object	A general type encapsulating specific data and behaviors

Linked Data: References and Pointers

A fundamental and oft-recurring feature of many data structures is that there are **links** from piece of data pointing to another piece of data in the same structure.

These links can take many forms from integers representing indices in some contiguous array, to hash keys, to pointers to specific memory locations (containing other records or data), and even to external data sources.

It's worth keeping in mind that in many languages, variables representing complex data types are actually references (pointers) to the data. Consider this from python. What happens?

Similarly, in a language like Java: Here `s` is an object representing data encapsulated with various behaviors/operations that are pre-specified. But the variable `s` actually stores a pointer to the data itself.

This is in contrast with `R` which has no simple reference type. This fact can make it challenging (and or inefficient in time or memory) to create complex data structures in R. Moreover, large structures like arrays in R are *copy on write*, which can get costly.

One solution is to use environments, which can associate names with persistent data, kept as references. This is the idea behind Reference Classes in R. See the `Hash` activity today for a basic application.

Core Data Structures

Data structures can be defined in the *abstract* through the set of operations they support and the requirements placed on the computational complexity of these operations.

More *concretely*, data structures can be defined through a specific implementation of these operations. The abstract representation of a data structure can have several distinct concrete implementations.

Here is a brief survey of some of the most commonly used data structures.

(Note: the abstract specification of data structures are sometimes called "abstract types".)

Lists

A **list** is linear, finite sequence of values.

Abstract specification:

- Operation `head(L)` returns first element of list `L` in O(1) time.
- Operation `tail(L)` returns rest of list `L` in O(1) time.
- Operation `cons(L,v)` adds value `v` to the head of list `L`.

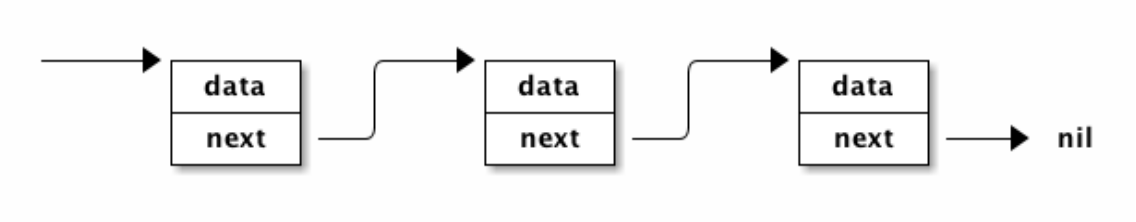
Implementations:

- Arrays:

  Store list items (or references) in array in reverse order, maintaining index to head, expand the array (by copying into a larger space) as needed.

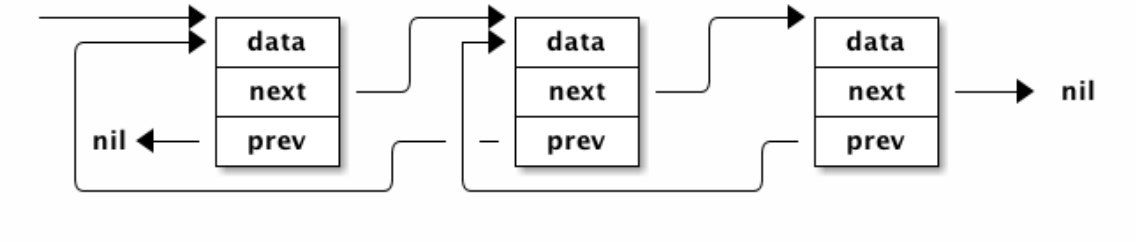
- Linked List:

  Each item is a record with data and a pointer to next item (or nil if none).



- Doubly-Linked List

  Each item is a record with data and pointers to next and previous items (or nil if none).



Stacks, Queues, Deques

A **stack** is a collection of values where only the most recently added item is available. Items are available *Last In, First Out* (LIFO). (Think of a stack of cafeteria trays.)

Abstract specification:

- Operation `push(S,x)` adds item `x` to the top of stack `S` in O(1) time.
- Operation `pop(S)` removes and returns the top item on stack `S` in O(1) time. (Popping an empty stack is an error.)
- Operation `isEmpty(S)` indicates whether stack `S` is empty in O(1) time.
- (Optional) Operation `peek(S)` returns the top item on stack `S` without removing it, in O(1) time.

Implementations: Array or Linked List

A **queue** is a collection of values where only the *least* recently added item is available (at the front) and where items can be added only at the end. Items are available *First In, First Out* (FIFO). (Think of a line at the grocery store.)

Abstract specification:

- Operation `enqueue(Q,x)` adds item `x` to the end of queue `Q` in O(1) time.
- Operation `dequeue(Q)` removes and returns the item at the front of queue `Q` in O(1) time. (Dequeueing an empty queue is an error.)
- Operation `isEmpty(Q)` indicates whether stack `Q` is empty in O(1) time.
- (Optional) Operation `peek(Q)` returns the top item on queue `Q` without removing it, in O(1) time.

Implementations: Doubly-Linked List or Rolling Array

A **deque** (pronounced "deck") is a generalization of stacks and queues that stands for double-ended queue.

The abstract specification states that items can be added to or removed from *either end* in O(1) time, with operations `addFront(D,x)`, `popFront(D)`, `addRear(D,x)`, `popRear(D)`, and `isEmpty(D)` at least.

Implementation is often with a doubly-linked list.

Priority Queues

A **priority queue** is a collection of items with associated scores (called *priorities*) in which it is efficient to remove (or view) the item with highest priority. (Think of the line at a hospital emergency room, where the most serious cases are taken first and among those equally serious those who have waited longest.)

Abstract specification:

- `insert(Q, x, p)` adds item `x` with priority `p` to priority queue `Q`.
- `removeHighest(Q)` removes and returns the highest priority item from priority queue `Q`. (It is an error to remove from an empty queue.)
- `findHighest(Q)` returns highest priority item from priority queue `Q` without removing it.
- `isEmpty(Q)` indicates whether `Q` is empty.

`insert` and `findHighest` can be implemented in O(1) time, though for naive implementations the latter can be O(n).

Implementations: arrays, heaps

Language implementations:

- `collections.deque` in Python
- `rstackdeque` or `dequer` packages in R
- `std::stack` and `std::queue` in C++
- `DataStructures.jl` for Julia

Hash Tables

A **hash table** (aka *dictionary*, *associative array*, *hash map*, or sometimes *map*) that map (typically sparse) values of one type to values of another. We look up **values** stored in the table by their **keys**. Think of a dictionary: we look up the definition of a word using the word as a key into the dictionary.

Abstract specification (basic):

- `insert(H, k, v)` – Insert value `v` into hash table `H` associated with key `k`
- `lookup(H, k)` – Find the value, if any, in `H` associated with key `k`
- `remove(H, k)` – Remove a key and its associated value from the table
- ...

The idea is that these operations are fast, ideally O(1), even though the set of possible values may be large and sparse

The trick to making this work is using a good **hash function**. A *hash function* is a function which takes an object (integer, tuple, string, anything) and returns a number. Hash functions are designed to be very fast, and so that slightly different inputs give very different outputs. The key property:

Two unequal objects are unlikely to have the same hash value

We will talk more about hash functions and their many uses later in the course.

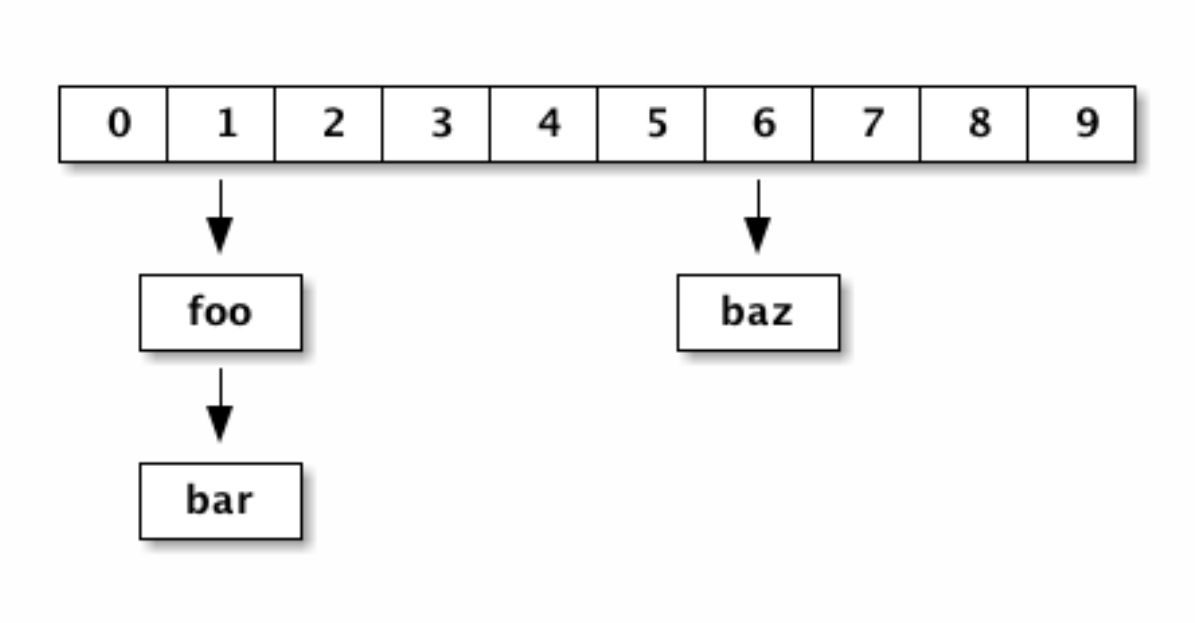
Language Implementations:

- Native: `dict` in python, `map` in clojure, tables in `lua`, Julia, ...
- `std::unordered_map` in C++
- `hash` package in R (only string keys supported). R's lists are *not* hash tables, and have O(n) lookup or worse. R does use hash tables internally ("environments") to store connect variable names to their values.

Here is a basic implementation of a hash table. Suppose a hash function gives output in the range [0, N], where N is a suitably large number. We pick a smaller number M and allocate an array of size M.

To add an item to the set, we calculate i = hash(item) mod M. Then look up the ith element in the array.

- If it is empty, add the item as the first element of a linked list.
- If it already contains an entry, search the linked list there. If the item is not already in the list, append it. This is called *chaining*.



When two separate items end up in the same *bucket* – the same array element – we call it a **collision**.

To determine if an item is already in the set, calculate i = hash(item) mod M again. Search the linked list at that index to see if the item is there.

If M is suitably large – much larger than the number of items in the set – there will be few collisions and looking up any element will be fast.

Hash sets have a statistic called a *load factor*: the average number of entries per bucket. If the load factor is high, there are many collisions, and looking up entries will require searching lists. Many hash set implementations automatically grow their backing array when the load factor gets too high.

Note: The hash set or table implementation in your chosen programming language, like Python or R, will automatically deal with collisions – you don't have to write this chaining logic.

(There are other schemes besides chaining to handle collisions, like linear or quadratic *probing* and *Robin Hood hashing*.)

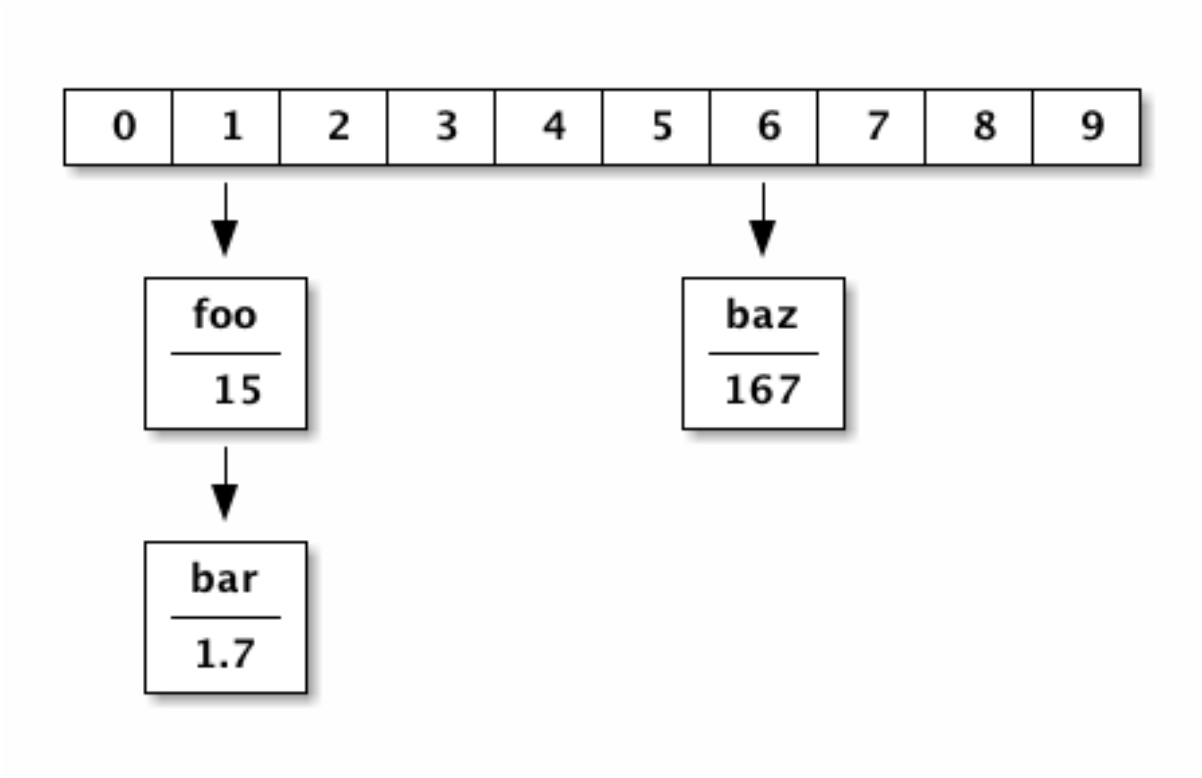
Exercise

Assume you have a function `hash()` which calculates hash values of strings and returns them as integers. Write simple versions of `lookup()`, `insert()`, and `remove()` in a language of your choice for a hash set of strings.

Hash maps/tables

Hashing can also be used to implement *associative arrays*: arrays that support lookup by a *key* instead of by index. An associative array stores (key, value) pairs, and one built with a hash table has O(1) lookup of keys.

To insert an element into an associative array, we hash its key. We follow the same steps as a hash set, except we insert the (key, value) pair into the linked list instead of just the key:



Sets

A **set** is a container that behaves like a mathematical set: they are unordered and contain only one of each object.

Specification:

- `elementOf(S, x)`
- `insert(S,x)`
- `remove(S,x)`
- `subsetOf(S1,S2)`, `union(S1,S2)`, ...
- many more...

A *hash set* is a fast (O(1) for access and insertion) implementation of a sort based on hashing.

Languages like `python` and `clojure` provide native set types. Languages like C++ and Java have extensive libraries supporting a variety of sets.

- `std::set, =std::unordered_set`, and `boost` libraries in C++
- `SortedSet` and `SortedDict` in `DataStructures.jl` in Julia
- `sets` package in R. Actually stores sets as lists when created, and converts them to hash tables for set operations that require it.

Trees

Trees represent hierarchical organization of data. Data is arranged in *nodes* which have *children* (and sometimes *parents*). We follow from the *root* of the tree out to the leaves. These are important both as data structures and as mathematical objects. We will see many varieties of trees, but for now, we look at a simple case where each node can have at most two children. These are called **binary trees**, and they are an important case.

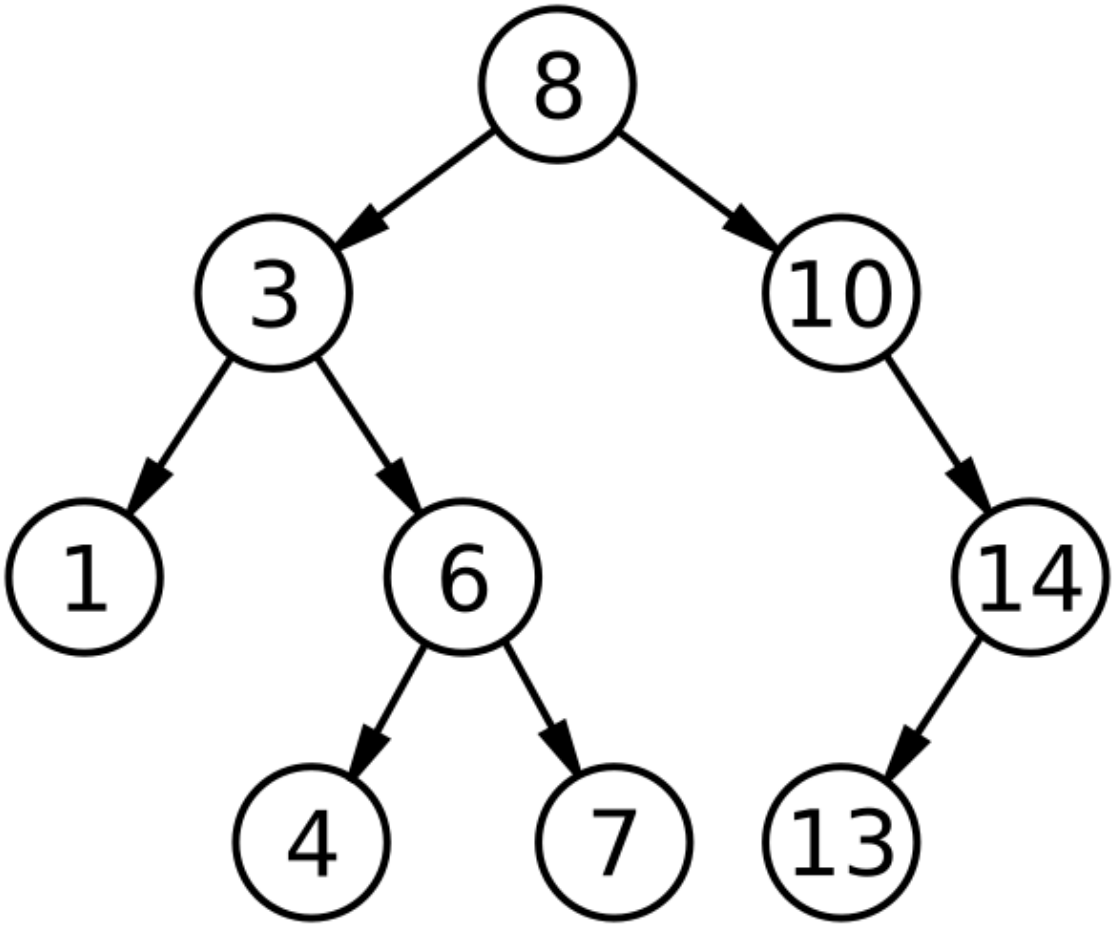
- Binary trees A *binary tree* is an efficient way to store ordered data so that searching for specific entries is fast. By maintaining a sorted tree of entries, we can find any entry by traversing the tree.

  A node in a tree has three parts:

  - value
  - pointer to left child
  - pointer to right child

  The tree is called "binary" because it only has two children. There are other variations, like B-trees, where nodes can have multiple children. PostgreSQL uses B-trees for indexing, as we discussed last week.

  Suppose we have the entries {8, 3, 10, 14, 6, 1, 4, 7, 13}.



(`https://commons.wikimedia.org/wiki/File:Binary_search_tree.svg`)

To search if a node is in the tree takes only O(log n) time, as does insertion and deletion, since the data structure is effectively already sorted – traversing the tree is like doing a binary search of a sorted array. We can also find all nodes greater than a certain value, less than a certain value, in a certain range, and so on, by carefully traversing the tree. This is how Postgres accelerates queries with indices.

Question: What happens if we accidentally insert all the entries in sorted order?

- Binary tree traversal A common operation given a tree is to "visit" all the nodes in the tree (presumably doing something with the information stored in those nodes). This operation is called *traversing* the tree.

  There are many different orders in which one can traverse a tree, but three are particularly valuable in practice:

  Preorder Visit root, Visit left subtree, Visit right subtree

  Inorder Visit left subtree, Visit root, Visit sight subtree

  Postorder Visit left subtree, Visit right subtree, Visit root

  As you can see, the name refers to when the root is visited relative to its subtree.

- Exercise to think about
  - If I have a binary tree, how can I produce an array or list containing its elements in sorted order?
    Consider an R implementation where each node is a list with entries `value`, `left`, and `right`. Start at the root node. Leaf nodes have no `left` or `right` entries.
  - Are there assertions that would be useful to use in an implementation of a binary tree?
    - Assert that left value is less than right value
    - Assert that node's value is greater than the left child's, and similarly for the right
- Applications
  - Sets and maps
  - Radix trees (routing tables)
  - Heaps (which lead to priority queues, which lead to graph routing algorithms like A*)
  - R-trees, k-d trees, and spatial indexing. Very useful for nearest-neighbor and other spatial algorithms
- Language Implementations
  - None in R :(
  - `bintrees` package in Python

Graphs

A **graph** is a collection of nodes and edges that represent *pairwise relationships* between various entities.

Graphs are important both as data structures and as mathematical objects, and we will cover them in depth soon.

Activity

Pull into your local version of the `documents` repository. Under `ClassFiles/week3` are several sub-directories: `Stack`, `Queue`, `Hash`, and `Tree`. Within each subdirectories is a `TASK` file and several source files in different languages. The `TASK` specifies a task to perform.

You can find a copy of these notes (as an Org and a PDF file) in `LectureNotes/week3/week3R.*` in the `documents` repository. The section Core Data Structures at the end of the notes has sections on the core data structures we just discussed.

Do the following:

1. Briefly review the `TASK` files in the sub-directories and select *one* of the tasks. (If you are already familiar with, say, stacks and queues, choose one of the other tasks.)
2. Create a branch `week3R` off of master in your `assignments` repository and switch to it. (You need not push this branch to github, though you are welcome to.)
3. Copy the `TASK` file and the source files in your chosen language (as specified by the file extension, for instance) into your `assignments` repository (in the `week3R` branch).
4. Read the section in the Core Data Structures corresponding to the task you chose (Stack, Queue, Hash Table/Dictionary, Tree).
5. Tackle the task in your editor or IDE, making commits along the way.

Activities Summary:

- `Stack`: Simple postfix command language Implement a new (stack intensive) command in the language
- `Queue`: Simulate performance of different queueing schemes Implement processing path for a new "customer"
- `Hash`: Document comparison Implement function to compute how many times do words (or pairs or triples or ...) in one document appear in another (sorting n most common).
- `Tree`: Union-Find Algorithm Implement FindComponent and InSameComponent operations

Updates: `Queue` and `Tree` are not ready... sorry.