Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2018

# Topic Notes: Fundamental Data Structures

Before we get into algorithms, we will review and/or introduce some of the data structures we will be using all semester.

A good reference for these, along with reference implementations, is the Bailey text. It includes the structure package, some of which replicates features of standard Java API structures, others of which are unique to this package.

---

## Basic Linear Structures

The basic *linear structures* are your standard "one-dimensional" *list* structures: *arrays*, *linked lists*, and *strings*.

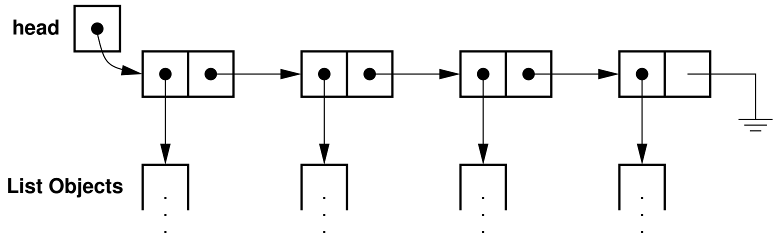
Some characteristics of arrays:

- allow efficient contiguous storage of a collection of data
- efficient direct access to an arbitrary element by *index*
- cost of add/remove depends on index

Strings are usually built using arrays, and normally consist of bits or characters.

Important operations on strings include finding the length (whose efficiency depends on whether the strings is *counted* or *null-terminated*), comparing, and concatenating.

You have seen the idea of a *linked list*:



This structure is made up of a pointer to the first list element and a collection of list elements.

Let's consider the complexity of our operations on a singly linked list.

- `add(0)` : 1 step

- `add(i)` : $i$ steps
- `add(n)` : $n$ steps
- `get/set(0)` : 1 step
- `get/set(i)` : $i$ steps
- `get/set(n-1)` : $n$ steps
- `remove(0)` : 1 step
- `remove(i)` : $i$ steps
- `remove(n-1)` : $n$ steps
- `get` all values in sequence : about $\frac{n^2}{2}$ steps (hey, we need an `Iterator`!)

How do these compare to similar operations on `Vectors`/`ArrayLists`?

- adding at the front is easier.
- adding at the end is harder.
- adding in the middle, well it depends where.
- the cost is consistent, though, since there is no reallocation and copying to grow the structure.
- removing at the front is easier.
- removing at the end is harder.
- removing in the middle is probably similar.
- getting/setting an arbitrary value is harder.

---

## Iterators

How do we "visit" each item in a collection? With a `Vector`/`ArrayList`, or an array, it's easy. We can write a `for` loop:

```
public <T> void traverse(ArrayList<T> v) {
  int i;

  for (i=0; i<v.size(); i++) {
    T visitme = v.get(i);
    // do something with visitme
  }
}
```

But imagine if someone has changed the implementation of `ArrayList`. It no longer has an array, but a linked structure.

Notice that to get access to the $n^{th}$ element, we need to visit the first $n - 1$ elements. If our `ArrayList` contained one of these linked structures instead of an array, our traverse method suddenly becomes very inefficient.

This is not good. What is the complexity of `get()`? In order to get the item at position $i$, we have to start at the beginning and we have to follow links until we find the right element.

What we want to do is to use the previous value returned, and take the one pointed to by the list element we just used to get that previous value. But how? We don't have that information!

We often need a way of cycling through all of the elements of a data structure. Java provides exactly what we need: `java.util.Iterator<E>`

A data structure can create an object of type `Iterator`, which can be used to cycle through the elements. For example, built-in Java classes `Vector` and `ArrayList` have methods:

```
  public Iterator<E> iterator()
```

that we can print out the elements of `Vector<E>/ArrayList<E> v` as follows:

```
    for (Iterator<E> iter=v.iterator(); iter.hasNext(); )
        System.out.println(iter.next());
```

Or in Java 5 and up, if our class implements the `Iterable interface` (which simply requires the method `iterator`) we can use the enhanced `for` loop (sometimes called a "for each" loop):

```
    for (E item: v) {
      System.out.println(item);
    }
```

## Stacks and Queues

These basic structures are used for many purposes, including as building blocks for more restrictive linear structures: *stacks* and *queues*.

For a stack, additions (*pushes*) and removals (*pops*) are allowed only at one end (the *top*), meaning those operations can be made to be very efficient. A stack is a *last-in first-out (LIFO)* structure.

For a queue, additions (*enqueues*) are made to one end (the *rear* of the queue) and removals (*dequeues*) are made to the other end (the *front* of the queue). Again, this allows those operations to be made efficient. A queue is a *first-in first-out (FIFO)* structure.

A variation on a queue is that of a *priority queue*, where each element is given a "ranking" and the highest-ranked item is the only one allowed to be removed, regardless of the order of insertion. A clever implementation using another structure called a *heap* can make both the insert and remove operations on a priority queue efficient.

# Trees

In a linear structure, every element has unique successor.

In a trees, an element may have many successors.

We usually draw trees upside-down in computer science.

You won't see trees in nature that grow with their roots at the top (but you can see some at Mass MoCA over in North Adams).

## Examples of Trees

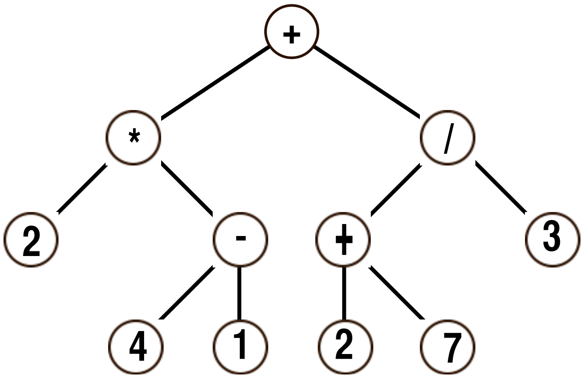
### Expression trees

One example of a tree is an *expression tree*:

The expression

```
(2*(4-1))+((2+7)/3)
```

can be represented as

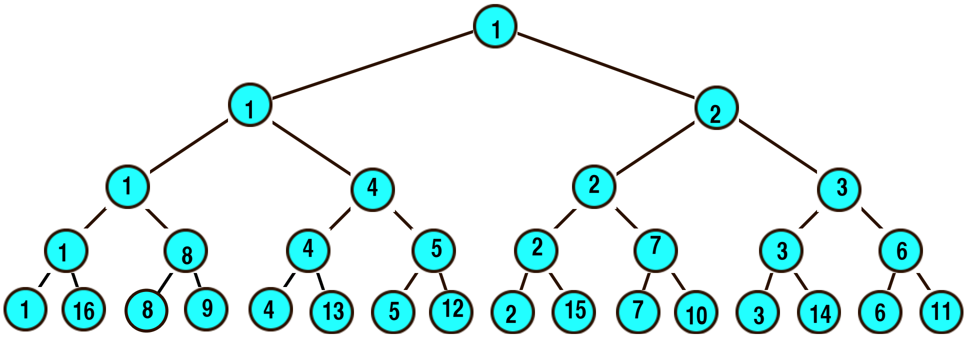


Once we have an expression tree, how can we evaluate it?

We evaluate left subtree, then evaluate right subtree, then perform the operation at root. The evaluation of subtrees is recursive.

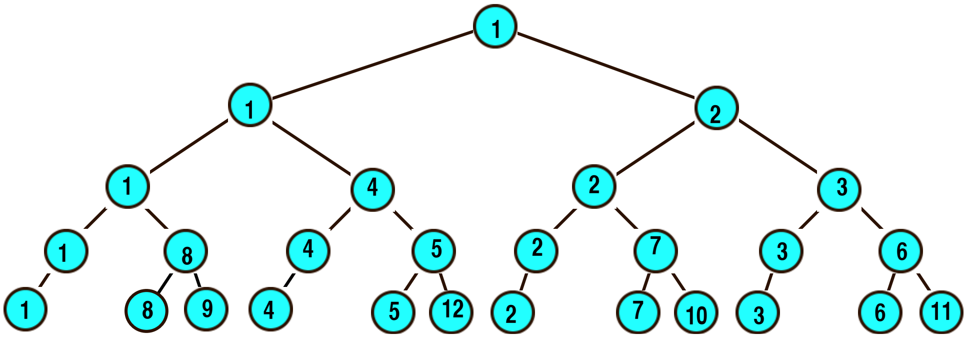
### Tournament Brackets

Another example is a tree representing a tournament bracket:



(a *complete* and *full* tree)

or



(neither complete nor full)

**Tree of Descendants**

One popular use of a tree is a pedigree chart – looking at a person's ancestors. Instead, let's look at a person's descendants. (Example drawn in class).

---

## Tree Definitions and Terminology

There are a lot of terms we will likely encounter when dealing with tree structures:

A tree is either *empty* or consists of a *node*, called the *root node*, together with a collection of (disjoint) trees, called its *subtrees*.

- An *edge* connects a node to its subtrees
- The roots of the subtrees of a node are said to be the *children* of the node.
- There may be many nodes without any successors: These are called *leaves* or *leaf nodes*. The others are called *interior nodes*.
- All nodes except root have unique predecessor, or *parent*.
- A collection of trees is called a *forest*.

Other terms are borrowed from the family tree analogy:

- *sibling* – nodes sharing the same parent
- *ancestor* – a node's parent, parent's parent, *etc.*
- *descendant* – a node's child, child's child, *etc.*

Some other terms we'll use:

- A *simple path* is series of distinct nodes such that there is an edge between each pair of successive nodes.
- The *path length* is the number of edges traversed in a path (equal to the number of nodes on the path - 1)
- The *height of a node* is length of the longest path from that node to a leaf.
- The *height of the tree* is the height of its root node.
- The *depth of a node* is the length of the path from the root to that node.
- The *degree of a node* is number of its direct descendents.
- The idea of the *level of a node* is defined recursively:
  - The root is at level 0.
  - The level of any other node is one greater than the level of its parent.

  Equivalently, the level of a node is the length of a path from the root to that node.

We often encounter *binary trees* – trees whose nodes are all have degree $\leq 2$.

We will also orient the trees: each subtree of a node is defined as being either the *left* or *right*.

Iterating over all values in linear structures is usually fairly easy. Moreover, one or two orderings of the elements are the obvious choices for our iterations. Some structures, like an array, allow us to traverse from the start to the end or from the end back to the start very easily. A singly linked list however, is most efficiently traversed only from the start to the end.

For trees, there is no single obvious ordering. Do we visit the root first, then go down through the subtrees to the leaves? Do we visit one or both subtrees before visiting the root?

There are four standard *tree traversals*, considered here in terms of binary trees (though most can be generalized):

1. *preorder*: visit the root, then visit the left subtree, then visit the right subtree.

2. *in-order* visit the left subtree, then visit the root, then visit the right subtree.
3. *postorder*: visit the left subtree, then visit the right subtree, then visit the root.
4. *level-order*: visit the node at level 0 (the root), then visit all nodes at level 1, then all nodes at level 2, etc.

For example, consider the preorder, in-order, and postorder traversals of the expression tree

```
         /
   *           2
 +   -
4 3 10 5
```

- preorder leads to prefix notation:
  / * + 4 3 - 10 5 2
- in-order leads to infix notation:
  4 + 3 * 10 - 5 / 2
- postorder leads to postfix notation:
  4 3 + 10 5 - * 2 /

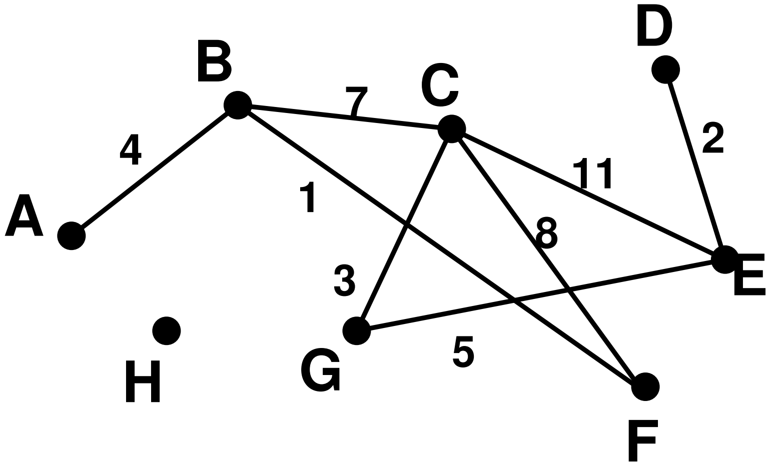
---

# Graphs

A *graph* $G$ is a collection of *nodes* or *vertices*, in a set $V$, joined by *edges* in a set $E$. Vertices have labels. Edges can also have labels (which often represent *weights*). Such a graph would be called a *weighted graph*.

The graph structure represents relationships (the edges) among the objects stored (the vertices).

For a tree, we might think of the tree nodes as vertices and edges labeled "parent" and "child" to represent nodes that have those relationships.



- Two vertices are *adjacent* if there exists an edge between them.

  e.g., A is adjacent to B, G is adjacent to E, but A is not adjacent to C.

- A *path* is a sequence of adjacent vertices.

  e.g., A-B-C-F-B is a path.

- A *simple path* has no vertices repeated (except that the first and last may be the same).

  e.g., A-B-C-E is a simple path.

- A simple path is a *cycle* if the first and last vertex in the path are same.

  e.g., B-C-F-B is a cycle.

- *Directed graphs* (or *digraphs*) differ from *undirected graphs* in that each edge is given a direction.

- The *degree* of a vertex is the number of edges incident on that vertex.

  e.g., the degree of C is 4, the degree of D is 1, the degree of H is 0.

  For a directed graph, we have more specific *out-degree* and *in-degree*.

- Two vertices $u$ and $v$ are *connected* if a simple path exists between them.

- A *subgraph* $S$ is a *connected component* iff there exists a path between every pair of vertices in $S$.

  e.g., {A,B,C,D,E,F,G} and {H} are the connected components of our example.

- A graph is *acyclic* if it contains no cycles.

- A graph is *complete* if every pair of vertices is connected by an edge.

There are two principal ways that a graph is usually represented:

1. an *adjacency matrix*, or
2. *adjacency lists*.

As a running example, we will consider an undirected graph where the vertices represent the states in the northeastern U.S.: NY, VT, NH, ME, MA, CT, and RI. An edge exists between two states if they share a common border, and we assign edge weights to represent the length of their border.

We will represent this graph as both an adjacency matrix and an adjacency list.

In an adjacency matrix, we have a two-dimensional array, indexed by the graph vertices. Entries in this array give information about the existence or non-existence of edges.

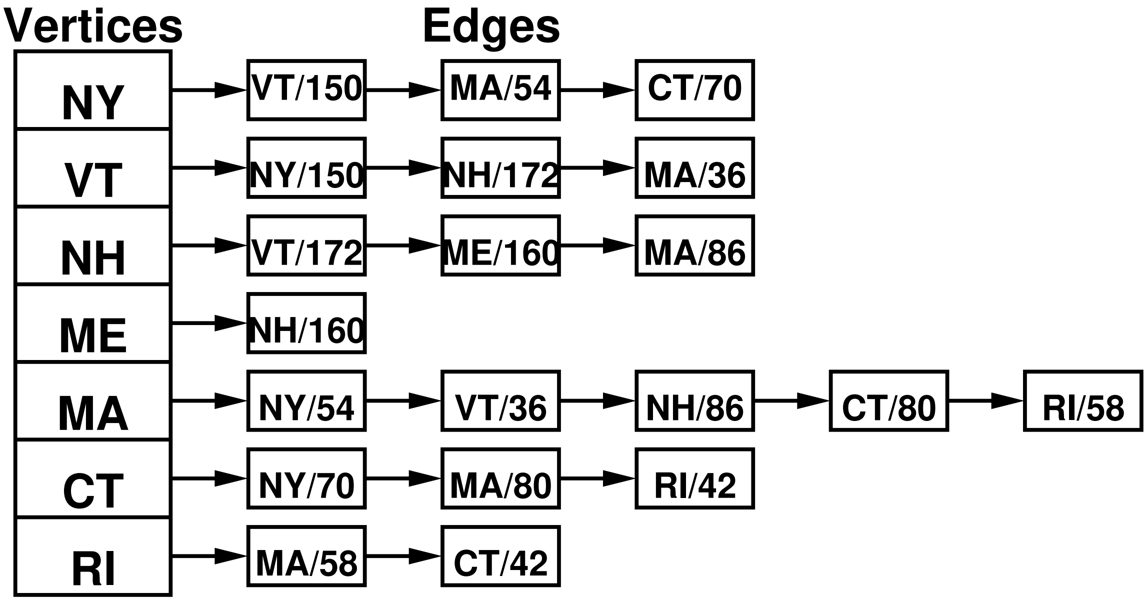
We represent a missing edge with `null` and the existence of an edge with a label (often a positive number) representing the edge label (often representing a weight).

Adjacency matrix representation of NE graph

| | NY | VT | NH | ME | MA | CT | RI |
|---|---|---|---|---|---|---|---|
| NY | null | 150 | null | null | 54 | 70 | null |
| VT | 150 | null | 172 | null | 36 | null | null |
| NH | null | 172 | null | 160 | 86 | null | null |
| ME | null | null | 160 | null | null | null | null |
| MA | 54 | 36 | 86 | null | null | 80 | 58 |
| CT | 70 | null | null | null | 80 | null | 42 |
| RI | null | null | null | null | 58 | 42 | null |

If the graph is undirected, then we could store only the lower (or upper) triangular part, since the matrix is symmetric.

An adjacency list is composed of a list of vertices. Associated with each each vertex is a linked list of the edges adjacent to that vertex.



In some cases, a matrix representation is more desirable. In other cases, it is the list representation. It depends on the density of the graph and which graph operations need to be most efficient for the task at hand.

In class, we will work through the implementation of a few methods of a simple directed graph (one with no vertex or edge labels) for both an adjacency matrix and an adjacency list representation.

In the example implementations, we will also consider the efficiencies in terms of both time and space.

---

## Sets and Dictionaries

A *set*, just like in mathematics, is a collection of distinct *elements*.

There are two main ways we might implement a set.

If there is a limited, known group of possible elements (a *universal set*) $U$, we can represent any subset $S$ by using a *bit vector* with the bit at a position representing whether the element at that position in $U$ is in the subset $S$.

If there is no universal set, or the universal set is too large (meaning the bit vector would also be large, even for small subsets), a linear structure such as a linked list of the elements of the set can be used.

A *dictionary* is a set (or *multiset*, if we allow multiple copies of the same element) which is designed for efficient addition, deletion, and search operations. The specific underlying implementation (array, list, sorted array, tree structure) depends on the expected frequency of the operations.

We will consider many of these data structures more carefully, and will see several more advanced data structures later in the couese.