Loop Invariants and Induction

Definition 1. A *loop invariant* is something that is true before we start and after every iteration of a loop.

We prove that a loop invariant is true by showing the following three things about it:

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Algorithm 1 add(A).

```
Input: A is an array of integers. It is indexed 1 to n.
sum = 0
for i = 1 to n do
    sum += A[i]
end for
return sum
```

Claim 1. After iteration i, the variable "sum" contains the sum of all elements in A from index 1 through i, that is, $A[1 : i]$.

Proof. We will prove this formally as a loop invariant.

Initialization: Before the first iteration of the "for" loop, $i = 0$ and sum is initialized to 0. Then $A[1 : 0]$ is empty.

Maintenance: If our statement holds before an iteration of the loop, then sum contains the sum of $A[1 : i - 1]$, and then we add $A[i]$ to it in the ith iteration, hence it now contains the sum of $A[1 : i]$.

Termination: When the loop terminates, $i = n$, and sum contains the sum of $A[1 : n]$ which is all elements in A.

□

Induction

You can argue the same thing by induction, and because you're referring to the code and loops the whole time, it'll look pretty much identical.

Base Case: this corresponds to initialization.

Inductive Hypothesis: This corresponds to the assumption before maintenance.

Inductive Step: This corresponds to maintenance.

Abstract Data Types and Depth-First Search

Let's review the main abstract data types that we might use when implementing various algorithms.

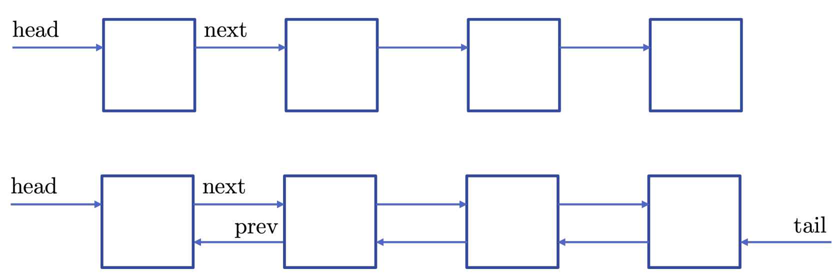
Linked Lists

Consider a list $L = [x_1, x_2, \ldots, x_n]$ where each x_i is an element in the list. In a *singly-linked list*, we keep a pointer to the first element of the list—that is, $\text{head}(L) = x_1$, and each element x_i has a pointer to the element after it, so $\text{next}(x_i) = x_{i+1}$ and $\text{next}(x_n) = \textit{null}$.

There is no reason for singly-linked lists to be used in practice. You will never see them, with the exception of perhaps a coding interview question or a puzzle.

A *doubly-linked list* also has a pointer to the last element of the list ($\text{tail}(L) = x_n$) as well as pointers form each element to the previous element ($\text{prev}(x_i) = x_{i-1}$ and $\text{prev}(x_1) = \textit{null}$).

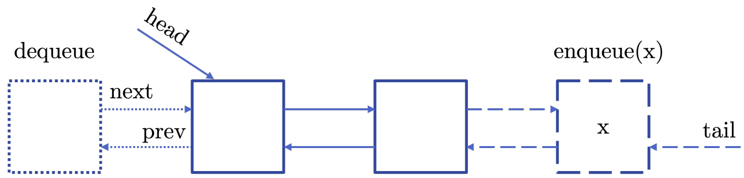
It's constant to do the actual insertion or deletion of an element, and at most linear ($O(n)$) to find an element by starting at the head or tail and moving along the list until it is found.



Queues

Queues are what's known as First-In, First-Out (FIFO) linked lists. They support the following additional operations:

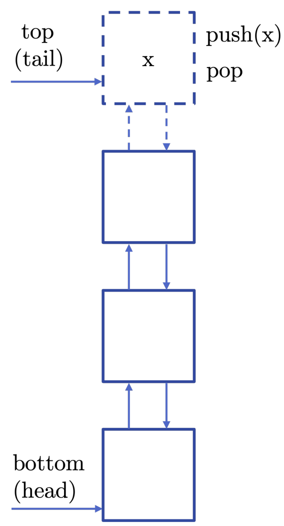
- $\text{enqueue}(q, x)$: insert element x to the back of the queue q. Formally, $q = q \circ x$.
- $\text{dequeue}(q)$: delete the element at the front of the queue q and return it. Formally, $q = [x_2, \ldots, x_n]$, return x_1.



Stacks

Stacks are what's known as Last-In, First-Out (LIFO) linked lists. They support the following additional operations:

- push(s, x): insert element x to the top (back) of the stack s. Formally, $s = s \circ x$.
- pop(s): delete the element at the top (back) of the stack s and return it. Formally, $s = [x_1, \ldots, x_{n-1}]$, return x_n.



Graphs

Definition 2. A (directed) *graph* $G = (V, E)$ is defined by a set of vertices V and a set of (ordered) edges $E \subseteq V \times V$.

Definition 3. A *directed edge* is an ordered pair of vertices (u, v) and is usually indicated by drawing a line between u and v, with an arrow pointing towards v.

Definition 4. An *undirected edge* is an unordered pair of vertices $\{u, v\}$ and is usually indicated by drawing a line between u and v. It indicates the existence of ordered edges (u, v) and (v, u).

Typically undirected edges will also be notated (u, v) out of sloppiness.

Some conventions:

- We will refer to the number of vertices (or the *size* of the vertex set $|V|$) as n.
- We will refer to the number of edges (or the *size* of the edge set $|E|$) as m.
- Often we will simply name the vertices $V = \{1, \ldots, n\}$ so an edge (i, j) is an edge from the i^{th} vertex to the j^{th} vertex.
- You may also hear vertices referred to as "nodes" or edges referred to as "arcs."

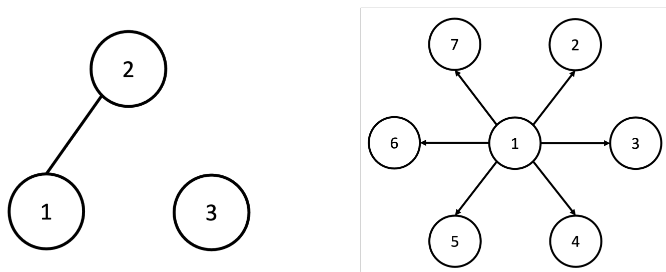


Figure 1: Left: An example undirected graph. $V = \{1, 2, 3\}$. $E = \{(1, 2)\}$. Right: An example directed graph. $V = \{1, 2, 3, 4, 5, 6, 7\}$. $E = \{(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7)\}$.

Definition 5. We call vertices i and j *adjacent* or *neighbors* if there is an edge $(i, j) \in E$. In directed graphs, we may explicitly refer to *out-neighbors* ($\{j : (i, j) \in E\}$) or *in-neighbors* ($\{j : (j, i) \in E\}$).

Definition 6. The *degree* of a vertex v is the number of neighbors it has. That is, $d_v = |\{u : (v, u) \in E\}|$. For directed graphs, we may refer to a vertex's *in-degree* or *out-degree*, and its *degree* is the sum of these.

Definition 7. A *path* from u to w is a sequence of edges $e_1, e_2, \ldots, e_k$ such that $e_1 = (u, v_1), e_i = (v_{i-1}, v_i)$, and $e_k = (v_{k-1}, w)$. That is, the first edge starts at u, the last edge ends at w, and each proceeding edge ends where the previous edge starts.

Definition 8. We say that a pair of vertices are *connected* if there exists a path between them.

We see graphs all over; networks are an entire field of study! What can you represent with graphs?

- Transportation networks (roads, airlines)
- Communication networks (Bitcoin peer-to-peer network)
- Information network (internet with links)
- Social networks
- Dependency network (course prerequisites, food chain)

What graph problems do you know?

- Shortest path
- Traveling salesman
- Scheduling

Abstract Data Types for Graphs

There are two primary ways that we represent graphs in the computer.

Exercise: Ask yourself the following questions for both adjacency matrices and adjacency lists to fill out the pros and cons (below) for each graph ADT below:

- What is the worst-case runtime to look up a specific edge (i, j)?
- What is the worst-case space needed to store the graph?
- What is the runtime to list all edges adjacent to i? On average, per edge adjacent to i?

Definition 9. An *adjacency matrix* for $G = (V, E)$ is an $n \times n$ binary matrix A where $A_{ij} = 1$ if and only if $(i, j) \in E$. We use a 2-dimensional array.

Pros of using an adjacency matrix:

- Look-up of a specific (i, j) edge is $O(1)$.

Cons of using an adjacency matrix:

- Space is $\Omega(n^2)$, independent of m. This can be very wasteful for sparse graphs where m is small.
- Listing all of i's edges is $\Omega(n)$ time, which can again be wasteful if i has small degree.

Definition 10. An *adjacency list* for $G = (V, E)$ is an array A of length n where the i^{th} entry contains a linked list of i's neighbors. That is, j is in the list $A[i]$ if and only if $(i, j) \in E$.

Pros of using an adjacency list:

- Listing all of i's edges is $O(d_i)$ time, hence $O(1)$ *per* neighbor.
- Space is $O(n + m)$.

Cons of using an adjacency list:

- Look-up of a specific (i, j) edge is $O(d_i) = O(n)$.