Chapter 4

Graphs

In terms of algorithms, we are not concerned with pie and line graphs, but on generic objects, and the relationship between them. Such as, traffic intersections, and the roads which connect them. Countries, and the borders between them. Actors, and the movies they've appeared in together.

This chapter provides an overview of graphs, helps us classify them, and breaks down many of the operations we can perform.

4.1 Types and Terminology

The two main components of a graph are *nodes* (sometimes called vertices) and *edges*. Nodes represent the objects in the graph, and the edges are the connections between them.

nodes	edges
intersections	roads connecting them
countries	borderts between them
actors	movies they are in together

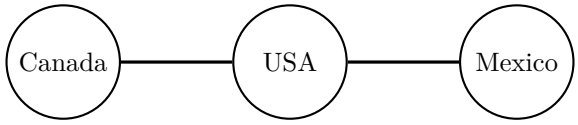


Figure 4.1: A graph of countries and borders in North America

4.1.1 Graph Classification

Within the overall frame of nodes and edges, there are several properties that allow us to classify graphs.

Directionality

Fundamentally, there are two big classes of edges.

1. Edges which connect nodes u and v, which imply that you can traverse either direction between u and v. These are equivalent to two-way streets. If edges in a graph are bidirectional, then the graph is known as *undirected*.

2. Edges which connect nodes u and v, which imply that you can traverse only from u to v, but not the other way. These are equivalent to one-way streets. If edges in a graph are directional, then the graph is known as *directed*.

In general, we only consider graphs as having all directed or undirected edges, and if we need two-way edges in a directed graph between, we create two edges, one in each direction between the two nodes in question.

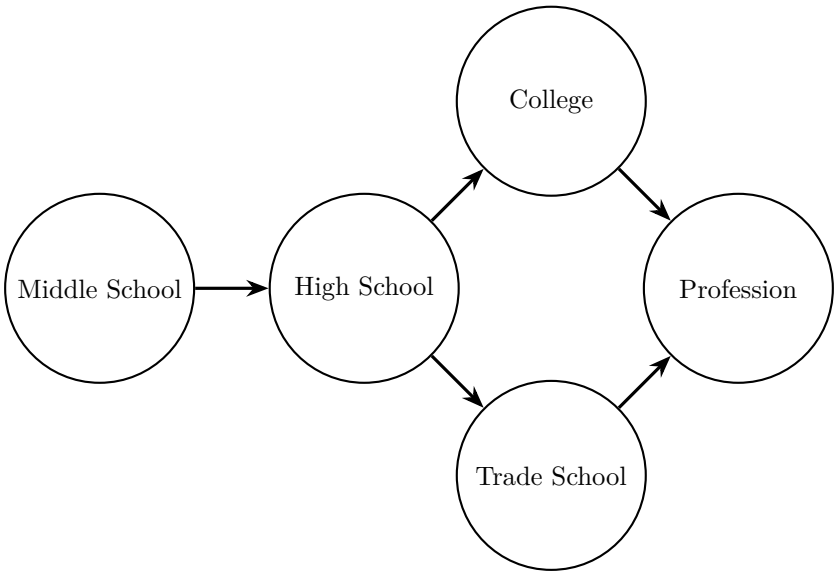


Figure 4.2: An example of a directed graph of career progression. Note the arrows.

Weighted-ness

Edges in a graph may have heterogeneous weights attached to them. We might consider this as the length of an edge, using the road analogy. In other graphs, the edges may have no weight (just implying a connection), or all equal weights (often weight 1). Depending on which type a graph falls into, it is known as *weighted* or *unweighted*.

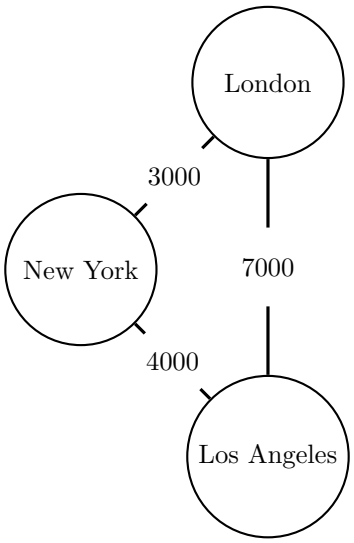


Figure 4.3: An example of a weighted graph indicating the miles between pairs of cities

Cycles

A *cycle* in a graph means we can start at one node, traverse through some sequence of edges, and end up back at the same node. Graphs which contain cycles are known as *cyclic* whereas graphs that contain no cycles are known as *acyclic*. Acyclic is an important property that allows us to apply several algorithms that we may not otherwise be able to. There are two tytpes of acyclic graphs:

1. An undirected, acyclic graph is known as a *tree*
2. A directed, acyclic graph is known as a *DAG*

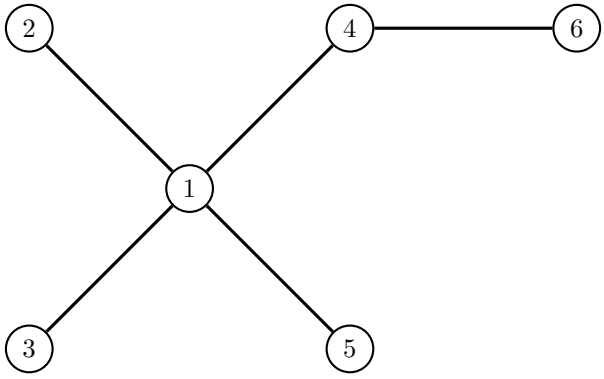


Figure 4.4: A tree

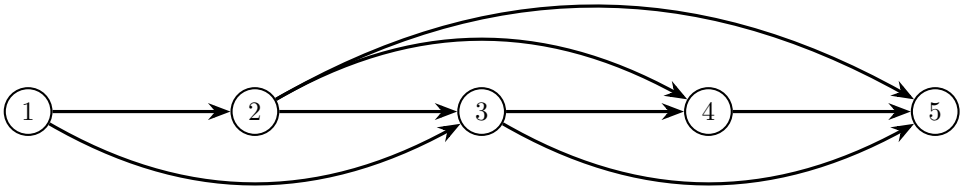


Figure 4.5: A DAG

Density

The ratio of the number of edges to the number of nodes often determines how efficiently we can apply algorithms to the graph. In a graph of n nodes, assuming at most one edge from a given node u to v, the maximum number of edges we will have is $O(n^2)$. This is known as a *dense* graph.

Conversely, if the number of edges is much smaller, closer to $O(n)$, the graph is known as *sparse*.

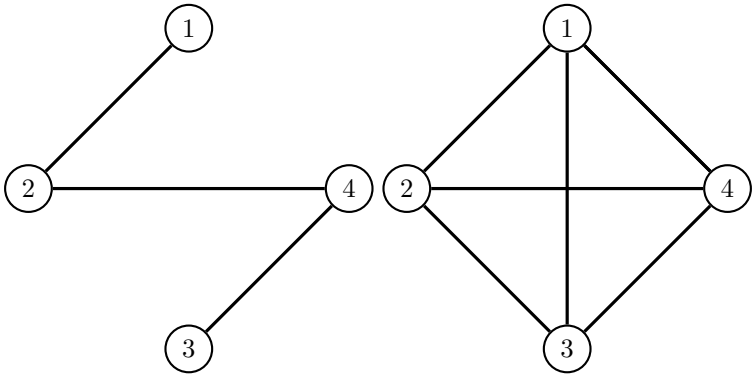


Figure 4.6: A sparse and a desnse graph

Connectivity

A graph is known as *connected* if there is some sequence of edges (going in either direction, in the case of a directed graph), between any two nodes in the graph. Otherwise, the graph is known as *disconnected*. There are two special cases of connectivity:

1. If you can reach every node from every other node, when considering edge direction, a graph is *strongly connected*. Note that all connected and undirected graphs are trivially strongly connected.

2. If each node is connected to every other node via two separate paths which do not have any nodes in common (i.e. are *node-disjoint*), the graph is *biconnected*. The implication is that if you remove any single node in the graph, the remaining graph is connected.

It is often useful to look at the connectivity on specific parts of a graph, rather than the entire thing, leading to connected, strongly connected, and biconnected *components*.

4.2 Representation

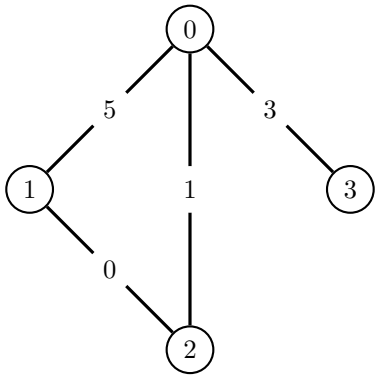
Now that we understand some of the idioms which might describe a graph, we can see how to represent graphs in code. In this section, we will see how to represent graphs which are given via the following description

> The first line of input contains 2 integers, n, the number of nodes, and m, the number of edges. $0 < n < 10^4$. Following this are m lines, each containing a triplet of integers, i, j, and k, indicating there is a directed edge for i to j with weight k. It is guaranteed that $0 <= k < 10^9$ and $0 <= i, j < n$, and that there will be at most one edge from i to j.

4.2.1 Adjacency Matrix

An adjacency matrix represents a graph of N nodes as an NxN matrix M, where M_i, j details some information about an edge originating from node i and ending at node j. In a weighted graph, this will typically be a value indicating the edge weight, and in an unweighted graph, simply a boolean. In an undirected graph, the adjacency matrix will be diagonally symmetric, whereas in a directed graph, the values will be unique. The values on the diagnoal itself will be -1 unless there are edges which connect a node to itself.

We will use the following graph as our example.



For this graph, the described input would be:

```
4 4
0 1 5
0 2 1
0 3 3
1 2 0
```

and the adjacency matrix is:

	0	1	2	3
0	-1	5	1	3
1	5	-1	0	-1
2	1	0	-1	-1
3	3	-1	-1	-1

Let's see input this code:

Listing 4.1: C++

```
// easier to statically allocate
// large-enough array than
// to use vector<vector<int>>
int am[10000][10000];
int n,m;
cin>>n>> m;
for(int i=0;i<10000;i++)
   for(int j=0;j<10000;j++)
      am[i][j]=-1;
for(int c=0;c<m;c++){
   int i,j,k;
   cin>>i>>j>>k;
   am[i][j]=k;
}
```

Listing 4.2: Java

```
int n=in.nextInt(),m=in.nextInt();
int[][] am = new int[n][n];
// Be sure to distinguish between
// no edge and zero-weight edge
for(int[] i:am)Arrays.fill(i,-1);
for(int i=0;i<m;i++)
   am[in.nextInt()][in.nextInt()]=
      in.nextInt();
```

Iterating over each edge from a node is easy.

```
void iterate(int i){
   for{int j=0;j<n;j++)if(am[i][j]!=-1){
      // do something with am[i][j]
   }
}
```

4.2.2 Adjacency List

One may notice that for an adjacency matrix, we used $O(n^2)$ space. This also means if we needed to, iterate over all edges emenating from a node, we would take $O(n)$ time. While these are natural complexities in a desnse graph, it is not very practical in a sparse one. In the latter case, not only is it a waste of memory to store a whole bunch of -1's, but a waste of time to iterate over all those potential edges which don't exist. It would be more efficient to only store edges which actually do exist.

To cope with this problem, we use a more efficient structure, an adjacency list. An adjacency list stores only actual edges, instead of every potential edge, as an adjacency matrix does. It does this by taking each node in the graph, and storing a pair of integers for each edge which originates at that node, indicating destination node and weight. It typically uses a map to perform this lookup, so while the efficiency for each individual edge lookup is slightly raised from traversing the map, it is more than made up for by not having to iterate through non-existent edges.

Let's see how we would input the same data as the previous section. Note that we must store the edge in both directions as the graph is undirected. Also note that in cases where the graph is unweighted, we may be able to use an unordered map or HashSet instead of the map equivalents.

Source Node	(Destination Node:Weight)
0	(1:5), (2:1), (3:3)
1	(0:5), (2:0)
2	(0:1), (1:0)
3	(0:3)

Listing 4.3: C++

```
int n,m;
cin>>n>>m;
unordered_map<int,int> al[m];
for(int c=0;c<m;c++){
   int i,j,k;
   cin>>i>>j>>k;
   al[i].insert(make_pair(j,k));
   al[j].insert(make_pair(i,k));
}
```

Listing 4.4: Java

```
int n=in.nextInt(),m=in.nextInt();
ArrayList<HashMap<Integer,Integer>>
   al=new ArrayList<>();
for(int i=0;i<n;i++)
   al.add(new
      HashMap<Integer,Integer>());
for(int c=0;c<m;c++){
   int i=in.nextInt(),
      j=in.nextInt(),
      k=in.nextInt();
   al.get(i).put(j,k);
   al.get(j).put(i,k);
}
```

Iterating over each edge is still straightforward:

Listing 4.5: C++

```
void iterate(int i){
   for (auto& j:al[i]){
      // do something with j
   }
}
```

Listing 4.6: Java

```
void iterate(int i){
   for(int j:al.get(i).keySet()){
      // do something with
      // al.get(i).get(j)
   }
}
```

Except in very specific circumstances, we will find that the adjacency list representation is the more convenient to work with.

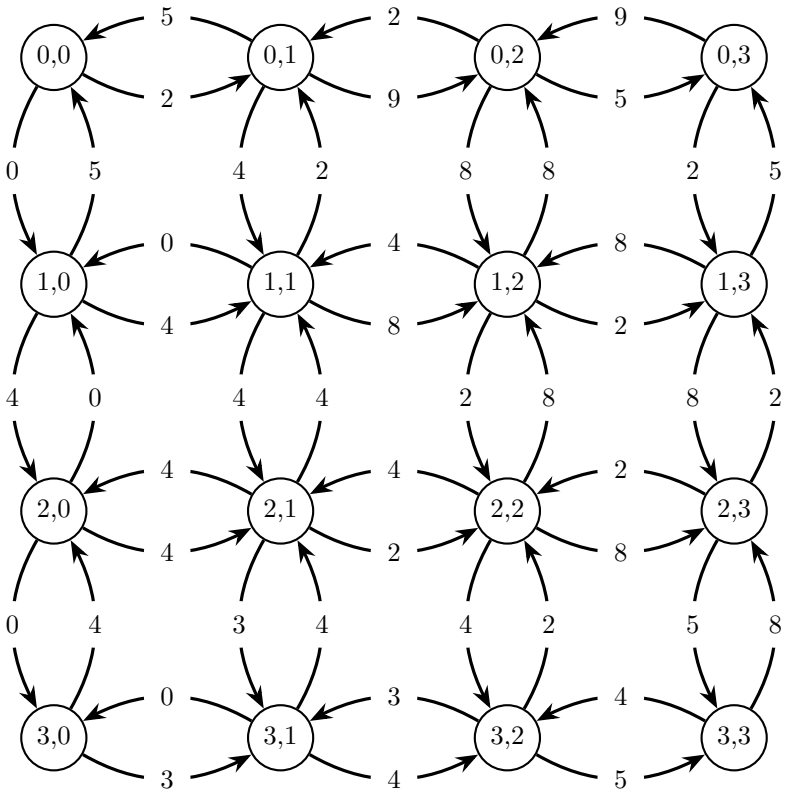
4.2.3 Implied Graphs

While the above are the canonical graph representations, there are comomn cases where it does not make sense to store the graph at all. This is common when the nodes given exist as x, y coordinates. Consider:

> The first line of input contains 2 integers, r and c, the number of colums and rows in the graph. $0 < c, r < 10^3$. Following this are r lines, each containing $c$integers d, where $0 <= d < 100$ and the i-th number in the j-column represents the cost to enter the node at the i-th row and j-th column. One may enter a node d fron any of the 4 cardinal directions.

The input and graph would look as follows:

```
4 4
5 2 9 5
0 4 8 2
4 4 2 8
0 3 4 5
```



While we can feasibly represent the graph, it is a bit cumbersome, and may become impractical in larger and denser graphs. Instead, we notice that the graph is exceptional formulaic, we can take advantage of the fact that the connectivity between two nodes is implied by the description itself. Given a node, say, $2, 3$, we do not need either of the adjacency structures to determine which nodes we have edges going to as it's simply the four cardinal direction. Given that, all we have to do is have a way to look up the weight of that edge, which is easy enough to do with an array.

We would read and iterate over this data structure as follows.

```
void input(){
  int d[1001][1001];
```

```
   int r,c;
   cin>>r>>c;
   for(int i=0;i<r;i++)for(int j=0;j<c;j++)cin>>d[i][j];
}

void iterate(int i_0, int j_0){
   // iterate over all cardinal positions which are in bounds
   for(int i=-1;i<2;i++)for(int
        j=-1;j<2;j++)if(i_0+i>=0&&i_0+i<r&&j+0+j>=0&&j+0+j<c&&i*j==0&&i!=j){
      // do something on d[i_0+i][j_0+j]
   }
}
```

We can easily extend the iterate function to operate on all directions including diagonals:

```
void iterate(int i_0, int j_0){
   for(int i=-1;i<2;i++)for(in
        j=-1;j<2;j++)if(i_0+i>=0&&i_0+i<r&&j_0+j>=0&&j_0+j<c&&xi!=0||j!=0){
      // do something on d[i_0+i][j_0+j]
   }
}
```

ASCII Grids

A common use of this type of graph involves inputting an ASCII grid which defines a "map" of sorts, where the character at a given position in the input defines some property of the node at that position. Consider the following input, which might describe a room bounded by walls (X), with a single door (D), and treasure located at various locations (T). The problem might ask us, say, to find the shortest path from the door that visits all the treasure locations.

```
XXXXXXXXXDXXX
X   T         X
X           T X
X      T      X
X   T         X
XXXXXXXXXXXXX
```

We can input the data as follows, and then use it to process the data as an implied graph, which in this case is usually unweighted.

Listing 4.7: C++

```
string d[rows];
for(int i=0;i<rows;i++)
   cin>>d[i];
// do whatever on d[x][y]
```

Listing 4.8: Java

```
char[][] d=new char[rows][];
for(int i=0;i<rows;i++)
   d[i]=in.nextLine().toCharArray();
// do whatever on d[x][y]
```

4.2.4 Linked Data Structures

Consider a situation where a graph we seek to encode forms a *rooted* tree. This is a tree which has a hierarchy such that one node is the "entrypoint" to the tree, and all edges and nodes arise therefrom. This is what is often colloquially called a tree, and seen in, say, binary search trees and heaps.

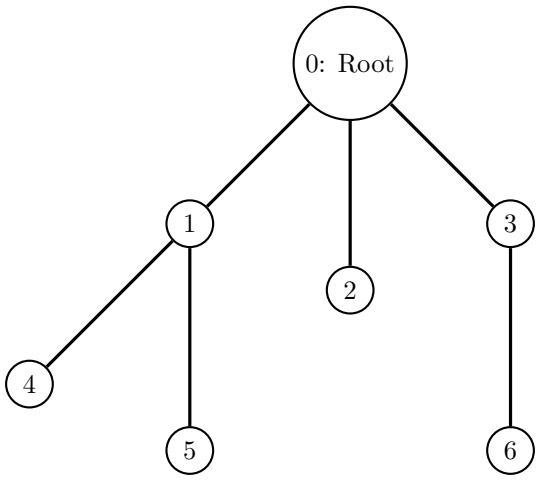


Figure 4.7: A rooted tree

In this graph, we can see that the root has *children*, in this case 1, 2, and 3. And nodes 2 and 3 also have children of their own. Nodes which have no children are known as *leaf* nodes.

This hierarchical nature provides the structure we need to encode this graph in code. We will consider two cases.

Binary Trees

In a binary tree, we have generally three pieces we might want to encode for a node, parent, left child, and right child. While in introuctory classes we might create a struct such as this:

```
class node{
   node ancestor;
   node left_child;
   node right_child:
}
```

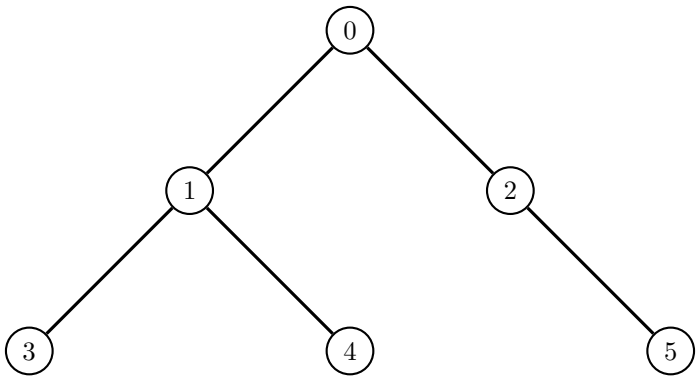
We can instead simply store the data in an array. Consider:

> Input begins with a line containg N, the number of nodes. Following are N lines, each containing 3 integers, the ancestor, left and right

child of node i. If an ancestor or child does not exist, -1 is input. It is guaranteed that the input forms a rooted binary tree and $0 < n < 1000$.

The input, tree, and array structure would appear as follows.

```
6
-1 1 2
0 3 4
0 -1 6
1 -1 -1
1 -1 -1
2 -1 -1
```



index	parent	left child	right child
0	-1	1	2
1	0	3	4
2	0	-1	6
3	2	-1	-1

We can input this data as follows:

```
int d[1000][3]; // parent, left child, right child
int n;
cin>>n;
for(int i=0;i<n;i++)for(int j=0;j<3;j++)cin>>d[i][j];
```

General Trees

In general trees, we do not have a fixed number of children, but may have multiple children. Consider:

> Input begins with a line containg N, the number of nodes. Following are N lines, each describing a node. Each line starts with an integer

M, indicating the number of descendencts of this node, followed by M integers, indicating a child. It is guaranteed that the input forms a rooted tree and $0 < N < 1000$.

The input is similar to the binary case, however we have to store an arbitrary number of children. We can do this by replacing the fixed array columns which held the two children with a list (or set if we need $O(1)$ lookup).

```
6
3 1 2 3
2 4 5
0
1 6
0
0
0
```

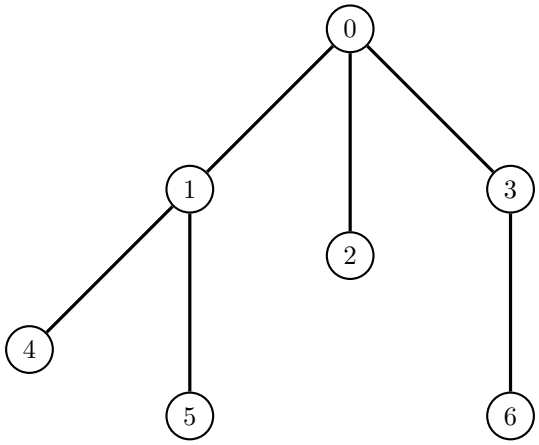


Figure 4.8: A rooted tree

index	parent	children
0	-1	1, 2, 3
1	0	4, 5
2	0	
3	0	6
4	1	
5	1	
6	3	

We can input this data as follows:

```
int n;
```

```
cin>>n;
int p[1000];//parents
for(int i=0;i<1000;i++)p[i]=-1;
vector<int> c[1000];//children
for(int i=0;i<n;i++){
   int m;
   cin>>m;
   for(int j=0;j<m;j++){
      int l;
      cin>>l;
      c[i].push_back(l); //set child
      p[l]=i; //set parent
   }
}
```

This method of using two separate arrays/lists is common and can be used to store arbitrary data without the overhead of explicitly creating classes or other custom structures.

4.3 Algorithms

4.3.1 Search

DFS

BFS

Dijkstra

MST

Others

- All Pairs
- negative weights
- reverse distances
- dense graphs
- path reconstruction

4.3.2 Flow and Match

In-Out Nodes

graph-split transformation

4.3.3 Componentization

SCC

BCC

2-sat

4.4 State Explosion

4.5 FSMs

4.6 DAGs

4.7 Trees

4.7.1 HLD

4.7.2 Binary Lifting

Suppose we were posed the problem:

> We are given a tree. Each node is assigned an arbitrary value, such that the value of every node is guaranteed to be strictly less than the value of its direct ancestor. We wish to respond to a set of queries where each query is a node, and we are to respond with the highest ancestor node whose value is at most 10x our own, or -1 if none such exists.

If the tree were balanced, we could walk from the node representing each query to the root, until we found the first admissable node. The tree, however, is not guaranteed to be balanced, and therefore, for N nodes andd Q queries, our runtime might be $O(N * Q)$.

Binary lifting is a technique which allows us to solve such queries in $O(log(N))$ time after $O(N * log(N))$ preprocessing, or more generally, if we have a function which is monotonic on the path from a given node to the root, it allows us to binary search the specific ancestor which fulfills some criteria.

From a high level, the idea is to preprocess the tree, such that each node maintains a pointer to it's immediate ancestor, its ancestor 2 above, its ancestor 4 above, etc. With these links, we can both traverse to the n-th ancestor of any node in $log(N)$ time, but it also provides us the structure to perform the binary search we need.

The links in a tree might look as follows:

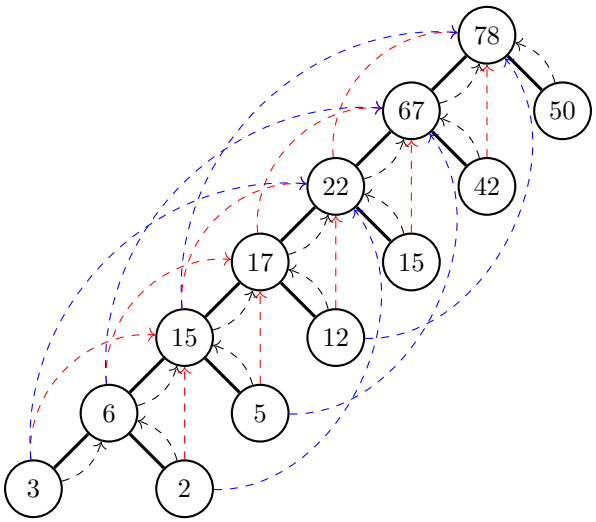


Figure 4.9: A preprocessed tree with uplinks. The 1-links are in black, the 2-links are in red, and the 4-links are in blue.

In order to perform the requested lookup, at say node 3, we would simply perform a binary search. We start with a pointer at the 3 node, and check the largest link, the 4 link, and see that the value 22 is fewer than the necessary 30, so we move our pointer to that node. We now check the 2 link at that node, see that it is too big at 78, so we do not move our pointer. We check the 1 link, and see that it is 67, too large. So 22 is our answer, correctly, and was queried in $O(log(N))$ time.

To perform the preprocessing, we create an array for each node which stores the 1, 2, 4, etc. links. This array is sized at most the heigh of the tree. The 1 link is simply our parent. We can find the other links easily as well. the 1 link or our 1 link is our 2 link. The 2 link of our 2 link is our 4 link, etc.

Listing 4.9: C++

```
void preprocess(int node){
   // assume links initialized to -1
   links[node][0] = parent[node];
   // we look up the i-1'th link of our i-1'th link to get our i'th link
   for(int i=1;i<=max_link;i++)
      links[node][i]=links[links[node][i-1]][i-1];

   preprocess(child[node][0], child[node][1]);
}

int search(int node){
   for(int i=max_link;i>=0;i--)
      if(links[node][i]!=-1&&my_condition(links[node][i]))
```

```
        node=links[node][i];
    return node;
}
```

4.7.3 Loweset Common Ancestor

The lowest common ancestor of two nodes (LCA) is the lowest node in a tree which is an ancestor of two specified nodes. Or thought about differently, if one started at the root and was attempting to traverse to a pair of nodes, the LCA is the point where those two paths diverge.

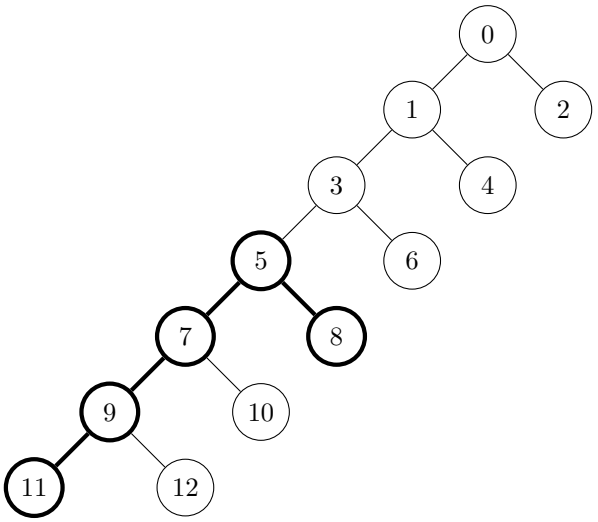


Figure 4.10: The LCA of nodes 11 and 8 is 5.

Standard Algorithm

In the standard algorithm, we in turn, start at each node and traverse back to the root. We push the nodes visited on a stack. Then, we successively compare the nodes on the head of each stack, The last pair of nodes which are euqal is the LCA.

Path from 8 to root	8,5,3,1,0
Path from 11 to root	11,9,7,5,3,1,0

If we look at successive nodes in the two paths from the end (as we would were they on a stack), we would see 0, 1, 3, and 5 all match. As 5 is the last match, it is the LCA.

This algorithm wowrks great in balanced trees ($O(log(N))$) or if we only have a single query ($O(N)$), however it does not do well if we have many queries in an unbalanced tree.

Unbalanced Trees

With a bit of preprocessing, we can get efficient LCA queries even in unbalanced trees using the binary lifting technique. The overall technique is as follows:

1. Assume we have a constant-time function that allows us to check whether some node is a direct ancestor of another.
2. If one node is a direct ancestor of the other, we are finished, as that node is the LCA.
3. Choose one node A, and binary lift from that node with a condition of "is the candidate an ancestor of B". We assumed we can evaluate this condition in constant time, and the condition is monotonic as we go from node A towards the root (going from "false" to "true"), so binary lifitng applies.

The only missing piece is the ability to determine in constant-time whether some node is a direct ancestor of another. It turns out this is straightforward. If we perform a DFS traversal, labeling nodes in the order that we visit them (pre-order, though it turns out not to matter), and save the value of the first descendent we visit, and the last, then we can check if a node is our descendent by simply checking whether the node's value is between these first and last values. Here is our tree labeled after the DFS:

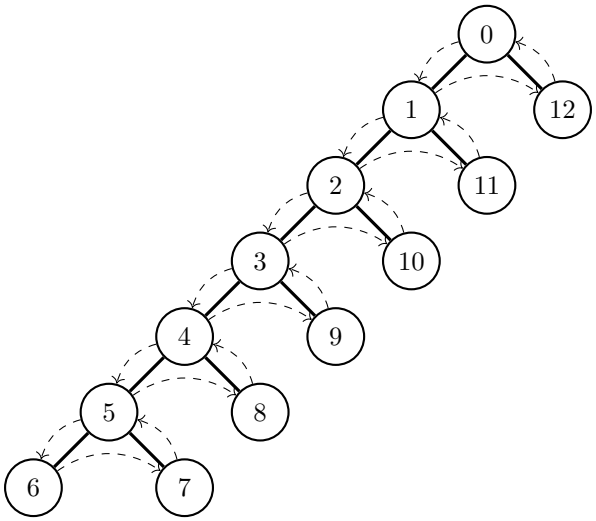


Figure 4.11: Example: Node 2 would cache 3 and 10, as 3 is the first node in 2's subtree, and 10 is the last.

Listing 4.10: C++

```
int index=0;
void dfs(int node){
   order[node]=index++;
   smallest_descendent[node]=index+1;
   dfs(child[node][1]);
   dfs(child[node][2]);
   largest_descendent[node]=index;
}

bool is_ancestor(int child, int node){
   return
        smallest_descendent[node]<=order[child]&&largest_descendent[node]>=order[child];
}
```

Now that we have established a constant-time query for is ancestor, we simply perform the binary lift from one node to find the lowest node which is an ancestor of some other node. This algorithm takes linear preprocessing time for the DFS, but queries only take logarithmic time, making this technique applicable for even multiple queries in unbalanced trees.

Constant Time

Number nodes in order as if it were a complete binary tree. xor two nodes, take highest 1-bit. That's the number of bits to mask out to get the LCA.

Map it to non-binary trees if necessary.

4.8 counting loops

4.9 union find