class List {
    int data;
    List next;
};

List p;
A corrupted single linked list

How can you check if a list is corrupted without looping forever?
Lists are more flexible than arrays
Optimizing append

- A header node with pointers both to first and last nodes
More efficient in some situations
A circular double linked list

- Beware of infinite loops!
- Often a do-while loop is convenient
A tree node $t$

- $\text{left}(t) = \text{null}$ or $\text{key}(\text{left}(t)) < \text{key}(t)$
- $\text{right}(t) = \text{null}$ or $\text{key}(\text{right}(t)) > \text{key}(t)$

- to insert a (key,value) pair,
- to delete a node with a certain key, and
- to search for a node with a certain key.
Without balancing, the running time of insert, delete, and insert would be $O(n)$

Two Russian mathematicians, Georgy Adelson-Velsky and Evgenii Landis, discovered in 1962 the first self-balancing binary search tree with $O(\log n)$ time for insert, delete, and search: the AVL-tree.

In 1972 the German computer scientist Rudolf Bayer invented another self-balancing search tree: the red-black tree, with the same time complexity
## AVL tree balance attribute

<table>
<thead>
<tr>
<th>balance</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>left subtree is one higher than right subtree</td>
</tr>
<tr>
<td>0</td>
<td>left and right subtrees have equal heights</td>
</tr>
<tr>
<td>1</td>
<td>right subtree is one higher than left subtree</td>
</tr>
</tbody>
</table>
Insertion
Single rotations

\[ s \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow s \]

\[ s \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow s \]
Double rotations

$u$

$T_1$ $T_2$ $T_3$ $T_4$

$t$

$s$

$T_1$ $T_2$ $T_3$ $T_4$

$t$

$s$

$T_1$ $T_2$ $T_3$ $T_4$

$u$

$T_1$ $T_2$ $T_3$ $T_4$

$t$

$s$

$T_1$ $T_2$ $T_3$ $T_4$

$u$

$T_1$ $T_2$ $T_3$ $T_4$
Graphs

- Notation
- Graph traversal and connectivity
- Testing bipartiteness
- Connectivity in directed graphs
• $G = (V, E)$
• $V$ is a set of nodes or vertices
• $E$ is a set of edges or arcs
• $V = \{a, b, c, d, e, f, g, h, i, j\}$
• $E = \{a - b, b - d, \ldots, i - j\}$, or
• $n = |V|$
• $m = |E|$
Example graphs

- Cities connected by direct air flights: node = city, edge = flight
- Social networks: node = person, edge = friend
- An **undirected graph** describes friends on a social network
- When you follow somebody you have an edge from one to another, i.e. a **directed graph**
- Actually, we can view city connectivity through air flights as a directed graph but normally there is a flight back
- Chess games: node = position, edge = legal move
Graph representation: adjacency matrix

- \( n = |V| \) and \( m = |E| \)
- Number each vertex from 1..\( n \)
- Often two representations of each edge
  - If there is an edge \( i - j \) then one is stored both in \( m[i][j] \) and in \( m[j][i] \), otherwise a zero
- If \( n \) is large it can be a good idea to store only half the matrix – how?
  - \( \Theta(n^2) \) space
  - \( \Theta(1) \) time to check if there is an edge \( i - j \)
  - \( \Theta(n) \) time to find all neighbors of a node
  - \( \Theta(n^2) \) time to list all edges
Graph representation: adjacency list

- $n = |V|$ and $m = |E|$
- Every vertex has a list of neighbors
- Every edge $u - v$ is stored in both $u$ and $v$
- $\text{degree}(n)$ is the number of neighbors
- $\Theta(m)$ time to check if there is an edge $i - j$
- $\Theta(\text{degree}(n))$ time to find all neighbors of a node
- $\Theta(m)$ time to list all edges
- Store only half of the adjacency matrix for undirected graphs
- Store only the adjacency list you need in directed graph (maybe both of course)
- For a very dense graph the matrix is smaller and just as fast
- If you need both quick neighbor check and being able to quickly list all neighbors, then use both!
- Optimizing compilers use both when deciding which variable should be allocated a processor register: the variables are nodes and there is an edge \( x - y \) if \( x \) and \( y \) may be needed at the same time (and therefore cannot use the same register)
A **path** is a sequence of nodes \( p = (v_1, v_2, ..., v_k) \) such that \( v_i \) and \( v_{i+1} \) are neighbors in an undirected graph, or there is an edge from \( v_i \) to \( v_{i+1} \) in a directed graph.

If all nodes in \( p \) are distinct then it is a **simple path**.

An undirected graph is **connected** if there is a path between every pair of nodes.

A **cycle** is a path which consists of a simple path followed by the first node such as \((u, v, w, u)\).
A connected undirected graph is a tree if it has no cycle.

A tree has $n - 1$ edges.

In a rooted tree one node, $r$ is called the node.
Depth first search: DFS

```c
int dfnum; /* Depth-first search number. */

procedure dfs(v)
begin
  dfn(v) ← dfnum
  visited(v) ← true
  dfnum ← dfnum + 1

  for each w ∈ succ(v) do
    if (not visited(w))
      dfs(w)
  end
end

procedure depth_first_search(V)
begin
  dfnum ← 0
  for each v ∈ V do
    visited(v) ← false
    dfs(v)
  end
end
```
Properties of depth-first search have been studied extensively by Robert Tarjan.

DFS is used a lot in compilers.

His algorithms tend to be faster than others’ and more beautiful than art.

They are art actually.
DFS example
The s-t connectivity problem

- The problem is to find a path from \( s \) to \( t \).
- Often we want to find the shortest path from \( s \) to \( t \).
- The **distance** between two nodes \( u \) and \( v \) is the number of edges on a shortest path from \( u \) to \( v \).
- How to solve the connectivity problem?
  - Check all nodes \( v \) at a distance \( k \) from \( s \) until either \( v = t \) or there are no more nodes to check, in which case \( s \) and \( t \) are not connected.
  - Let \( k = 1, 2, 3, ..., \infty \)
  - This is called **breadth first search**, or simply **BFS**
  - Think of an onion. You are in the center and explore one layer at a time outwards.
Breadth first search

- Is there a path from \( a \) to \( j \)?
- A node \( v \) is added to a layer only the first time \( v \) is seen
- Check one layer at a time.
  - \( L_0 = \{ a \} \)
  - \( L_1 = \{ b \} \)
  - \( L_2 = \{ d, h \} \)
  - \( L_3 = \{ c, e, g \} \)
  - \( L_4 = \{ f, j \} \)
- We don’t need the layers. A list is sufficient.
BFS implementation to find path $s - t$

procedure BFS($G, s, t$)

$q \leftarrow$ new list containing $s$

for $v \in V$ visited($v$) $\leftarrow 0$

visited($s$) $\leftarrow 1$

while $q \neq null$

$v \leftarrow$ take out the first element from $q$

for $w \in \text{neighbor}(v)$

if not visited($w$) then

visited($w$) $\leftarrow 1$

add $w$ to end of $q$

pred($w$) $\leftarrow v$

if $w = t$ then

print ”found path $s - t$”

return

print ”found no path $s - t$”
Finding the actual path $s - t$

- We want to find the path $a - j$
- It is $p = (a, b, d, g, j)$
- For each node $w$ except the first, the attribute $\text{pred}(w)$ is the previous node in $p$.
- $\text{pred}(j) = g$, $\text{pred}(g) = d$, etc
- What is the running time of BFS?
- The while loop has up to $n$ iterations with $|V| = n$
- Each node has at most $n$ neighbors, so $O(n^2)$?
- What do you say?
BFS time complexity

- But in total $m$ edges so $2m = \sum_{v \in V} \text{degree}(v)$ edges to process.
- $2m$ since each edge is in two adjacency lists
- Thus BFS can be implemented in $O(n + m)$ with adjacency lists