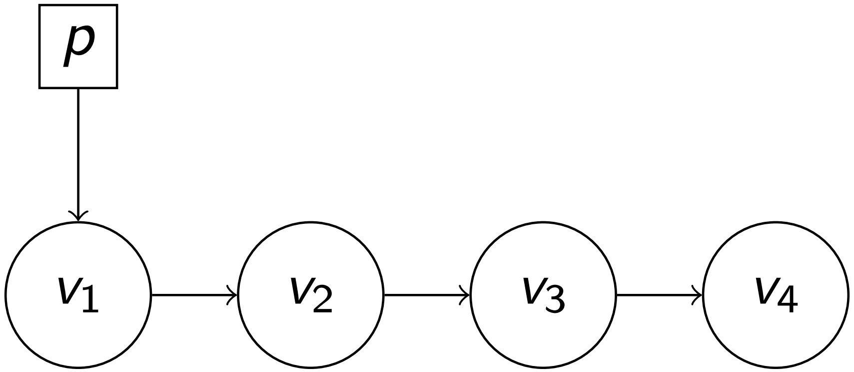
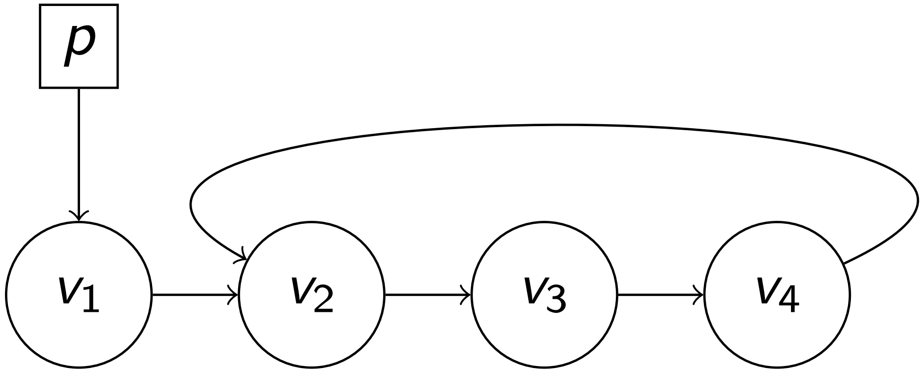
Single linked list



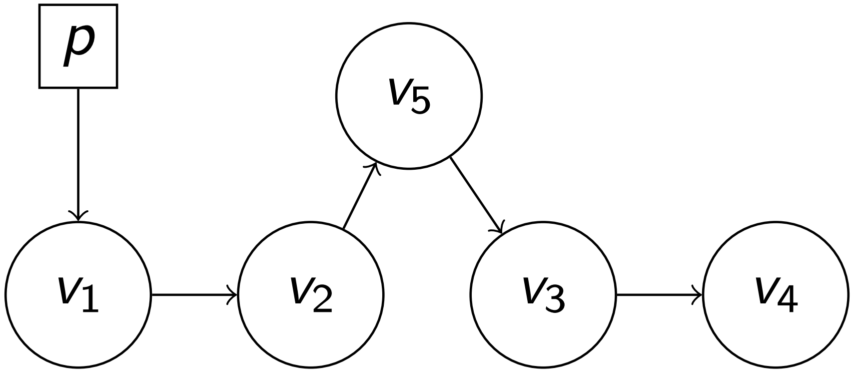

```
class List {
        int     data;
        List    next;
};

List    p;
```

A corrupted single linked list



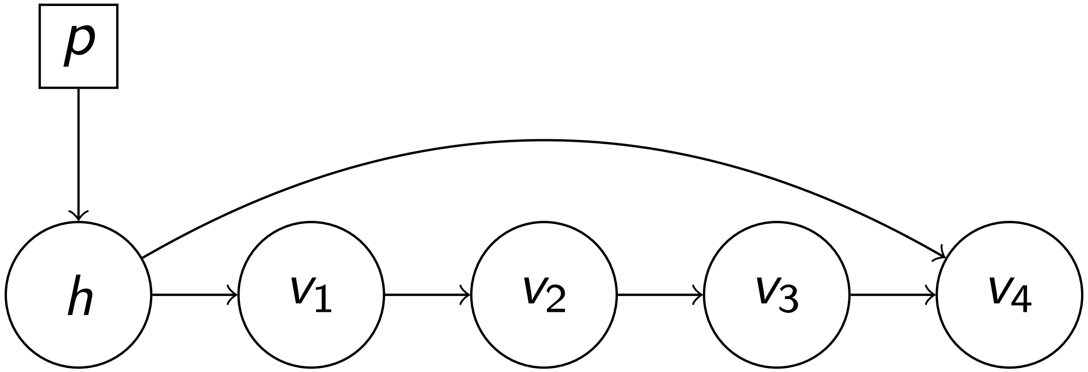
- Quiz: how can you check if a list is corrupted without looping forever?

Inserting a new node



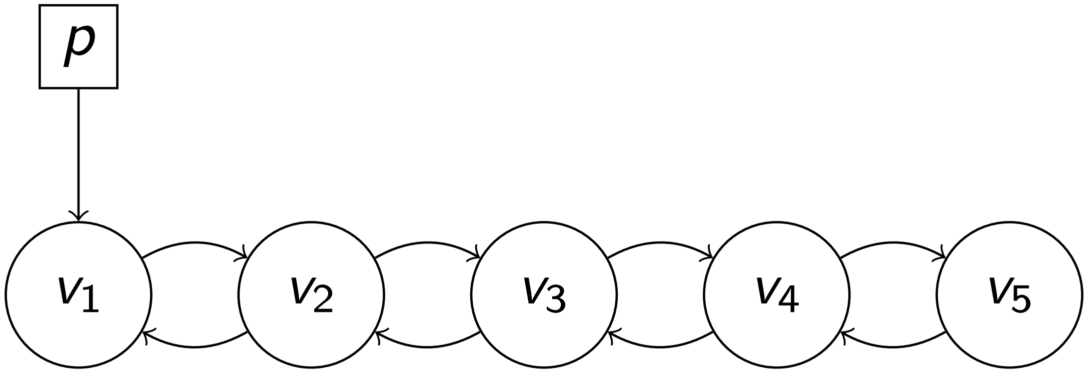
- Lists are more flexible than arrays

Optimizing append



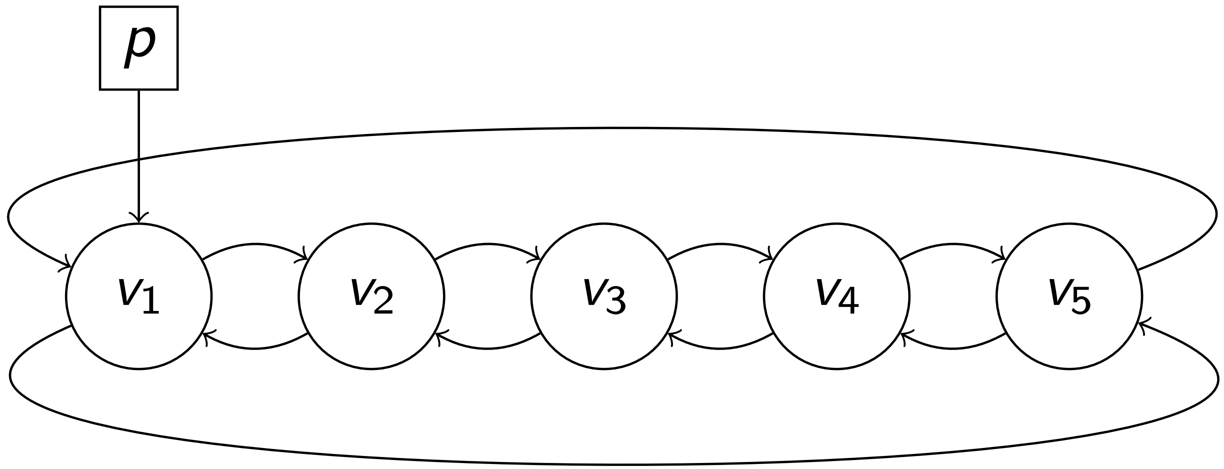
- A header node with pointers both to first and last nodes

Double linked list



- More efficient in some situations

A circular double linked list



- Beware of infinite loops!
- Often a do-while loop is convenient

Binary search tree

- A tree node t
- $\mathit{left}(t) = \mathit{null}$ or $\mathit{key}(\mathit{left}(t)) < \mathit{key}(t)$
- $\mathit{right}(t) = \mathit{null}$ or $\mathit{key}(\mathit{right}(t)) > \mathit{key}(t)$
- to insert a (key,value) pair,
- to delete a node with a certain key, and
- to search for a node with a certain key.

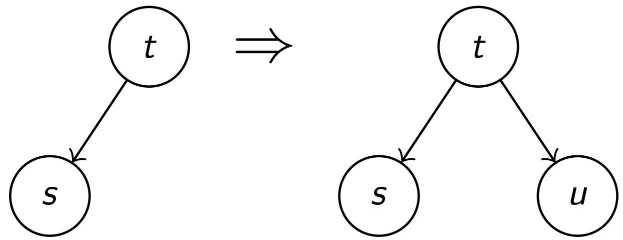
Balanced binary search trees

- Without balancing, the running time of insert, search, and delete 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

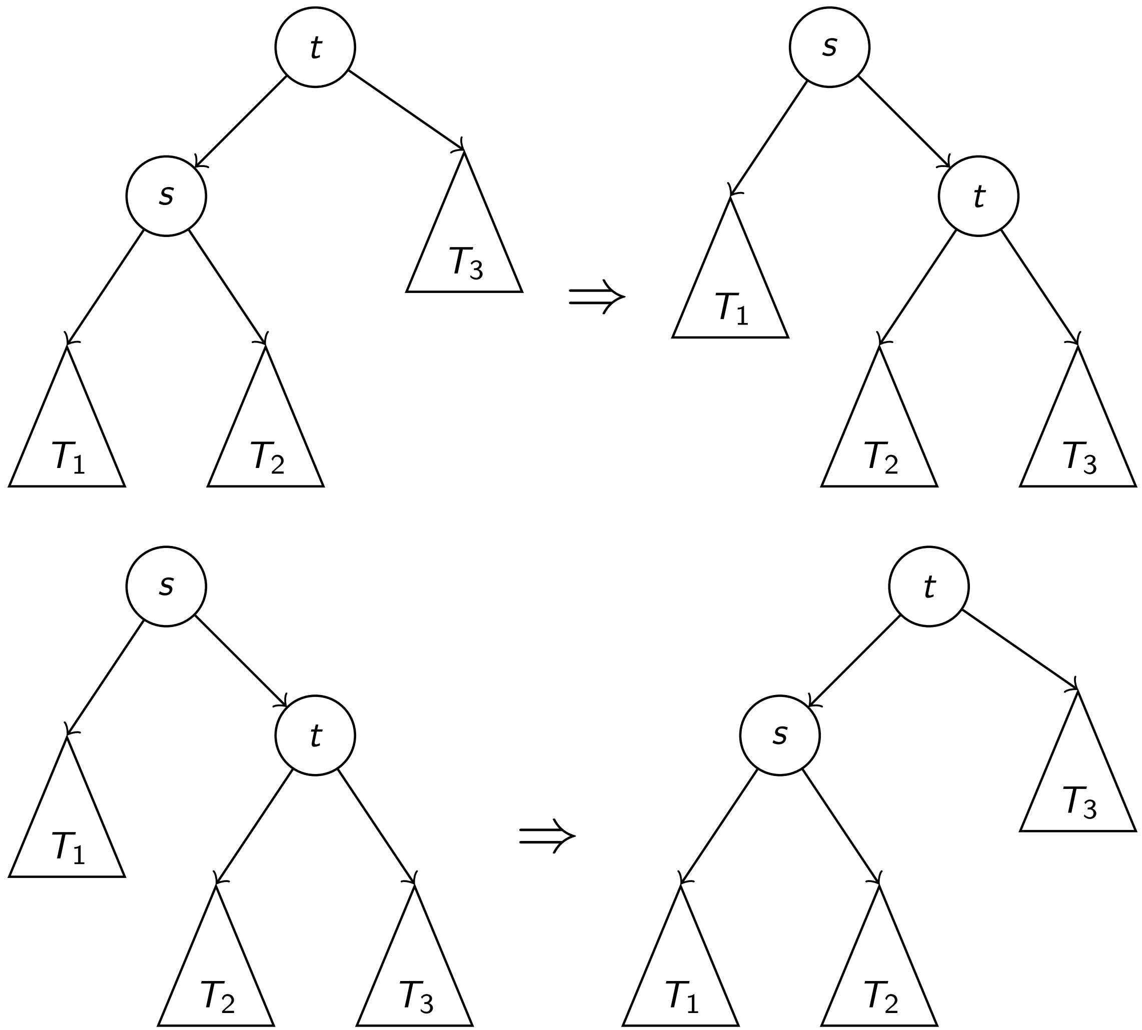
AVL tree balance attribute

balance	meaning
-1	left subtree is one higher than right subtree
0	left and right subtrees have equal heights
1	right subtree is one higher than left subtree

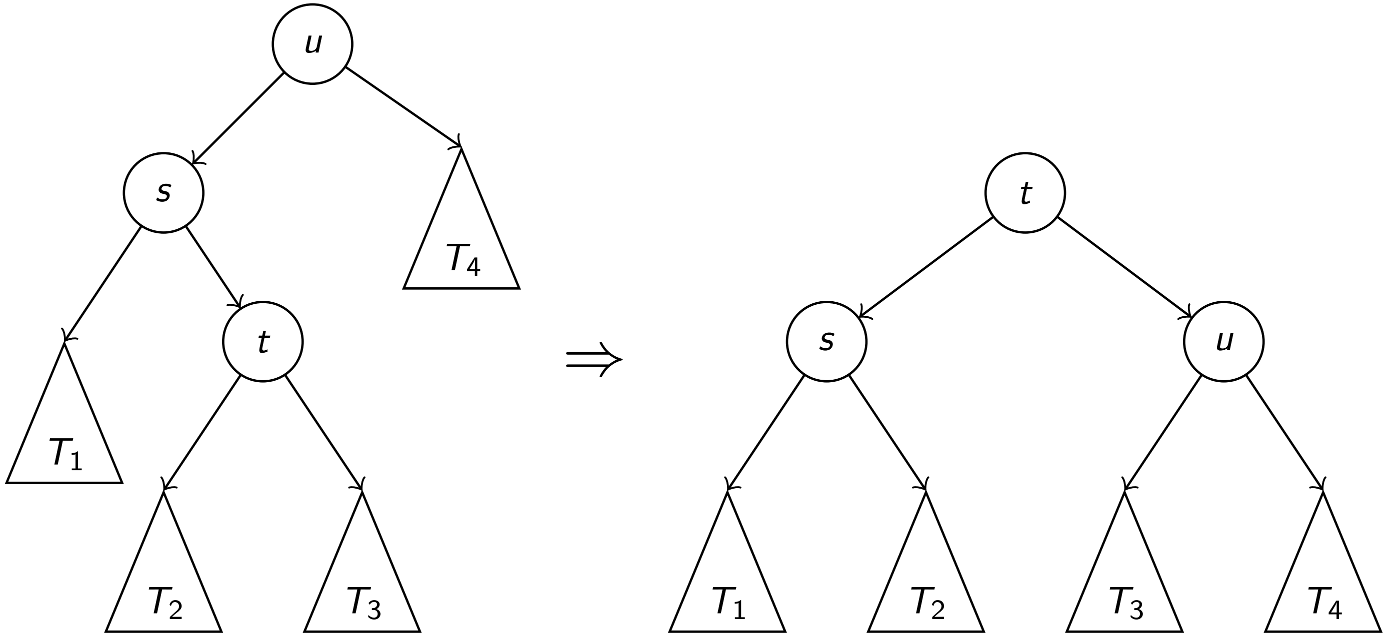
Insertion

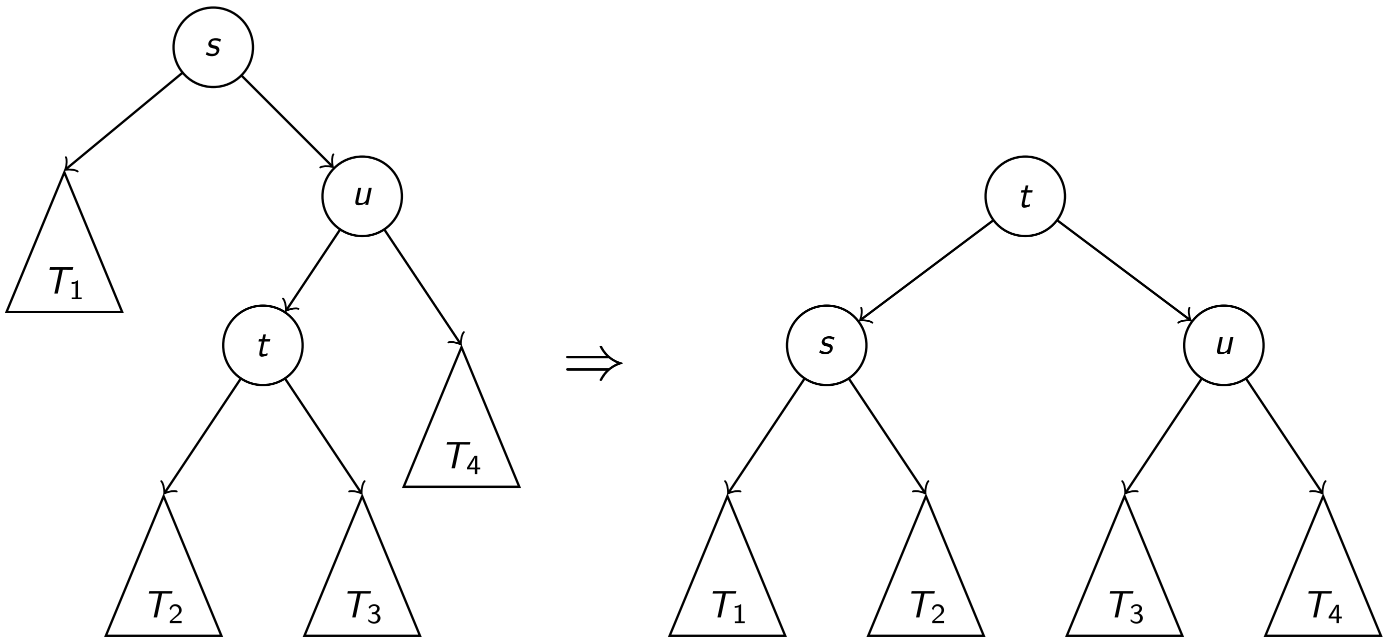


Single rotations



Double rotations





Hash maps or hash tables

- Store (key,value) pairs using an array of size m
- Insert, search and delete operations
- Compute an index from the key (modulo m) using a hash function
- When two keys are mapped to the same index there is a collision
- Two main approaches to handle collisions:
  - with separate chaining (öppen hashtabell)
  - with open addressing (sluten hashtabell)
- With n pairs, $\alpha = \frac{n}{m}$ is the load factor
- Separate chaining uses linked lists for the pairs
- Open addressing stores the pairs in the array

Separate chaining

- The table is an array of linked lists
- Compared with only one list, this will likely be m times faster
- Some alternatives:
  - Always insert at the beginning (no search needed if you know it is a new pair)
  - Keep each list sorted
  - Move frequently used pairs to the beginning of the list
- Advantage: simple to implement
- Disadvantage: less simple to allocate memory for the list nodes efficiently (allocation and freeing/garbage collecting nodes takes time)
- Usually a good choice
- If α gets too big, the operations will be slower but still work
- Resize array if needed

Open addressing

- Invented by Gene Amdahl
- $\alpha < 1$ (but see below — significantly less than one is better)
- Three ways to handle collisions
  - Linear probing
  - Quadratic probing
  - Double hashing probing

Linear probing

- An array element either contains a pair or a value "empty" (e.g. null)
- Sometimes it is simpler to have one array for keys and another for values, with the key and value of a pair stored at index i in the two arrays
- First compute an index $i \leftarrow f(key) \mod m$
- If $a[i]$ is not empty, set $i \leftarrow (i+1) \mod m$ and check again
- Otherwise insert new pair at i
- Similar for search
- Quiz: can we do the same for delete plus storing "empty" in the array?

Linear probing: delete

- Answer: no, since we then might not find some of the keys.
- Empty hash table:

empty	empty	empty	empty	empty	empty	empty

- Insert two pairs with $f(k_1) \mod m = f(k_2) \mod m$

empty	k_1	k_2	empty	empty	empty	empty

- Delete the pair k_1

empty	empty	k_2	empty	empty	empty	empty

- Search for k_2

empty	empty	k_2	empty	empty	empty	empty

will give up at first probe since it sees "empty"
- What can we do? Quiz: why not store a value "deleted" and skip such when searching?

Linear probing: delete

- Answer: yes, that works but gives other problems
- Empty hash table:

empty	empty	empty	empty	empty	empty	empty

- Insert two pairs with $f(k_1) \mod m = f(k_2) \mod m$

empty	k_1	k_2	empty	empty	empty	empty

- Delete the pair k_1

empty	deleted	k_2	empty	empty	empty	empty

- Search for k_2

empty	deleted	k_2	empty	empty	empty	empty

will skip ”deleted” and find k_2

- Quiz: when is this bad?

Storing ''deleted''

- Answer: we may store many ''deleted'' that must be skipped
- But if we insert and come to a ''deleted'' then we can use that position.
- I use both position and index for the same place in the array but it is a slight abuse of English.
- If we have too many ''deleted'' we can clean the hash table to remove them by reinserting everything
- Quiz: instead of storing ''deleted'', can we not move items ''to the left''?

Linear probing: delete with move

- Answer: yes, if we are careful
- Empty hash table:

empty	empty	empty	empty	empty	empty	empty

- Insert two pairs with $f(k_1) \mod m = f(k_2) \mod m$

empty	k_1	k_2	empty	empty	empty	empty

- Delete the pair k_1 and move k_2

empty	k_2	empty	empty	empty	empty	empty

- Search for k_2

empty	k_2	empty	empty	empty	empty	empty

  will find k_2
- Quiz: what could go wrong?

Linear probing: delete with bad move

- Answer: moving a key to the left of its originally computed index
- Insert two pairs with $f(k_1) \mod m = f(k_2) \mod m$

empty	k_1	k_2	empty	empty	empty	empty

- Insert a pair with $f(k_0) \mod m = 0$

k_0	k_1	k_2	empty	empty	empty	empty

- Delete k_1 and move k_2

k_0	k_2	empty	empty	empty	empty	empty

- Delete k_0 and move k_2

k_2	empty	empty	empty	empty	empty	empty

will never find k_2

Linear probing: correct delete with move

```
procedure delete(k, h)
begin
    i ← h(k) mod m
    while true do {
        a[i] ← empty
        j ← i
        while true do {
            i ← (i + 1) mod m
            if a[i] = empty then
                return
            k ← h(a[i]) mod m
            if not (j ≤ k < i  or i ≤ j < k  or k < i < j) then
                a[j] ← a[i]
                break
    }
end
```

- Three conditions needed due to modulo m.

A simple model of unsuccessful search in linear probing

- We ignore clustering and instead assume all positions are equally likely to be occupied.
- If the random variable X is the number of probes in an unsuccessful search, what is the expected value of X, $\mathbb{E}[X]$?
- $\mathbb{P}(X \geq k)$ is the probability that the first $k-1$ positions are occupied, and the last is empty.
  - The probability the first probed is occupied is: $\frac{n}{m}$,
  - the two first: $\frac{n}{m} \cdot \frac{n-1}{m-1}$,
- For $k > 1$ we can write:

$$\mathbb{P}(X \geq k) \quad = \quad \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-k+2}{m-k+2} \leq \left(\frac{n}{m}\right)^{k-1} = \alpha^{k-1},$$

- with the last expression is valid also for $k = 1$.

A simple formula for $\mathbb{E}[X]$

$$
\begin{aligned}
\mathbb{E}[X] &= \sum_{k=1}^{\infty} k \cdot \mathbb{P}(X = k) \\
&= \sum_{k=1}^{\infty} k \cdot (\mathbb{P}(X \geq k) - \mathbb{P}(X \geq k+1)) \\
&= 1 \cdot \mathbb{P}(X \geq 1) - 1 \cdot \mathbb{P}(X \geq 2) \\
&\quad + 2 \cdot \mathbb{P}(X \geq 2) - 2 \cdot \mathbb{P}(X \geq 3) \\
&\quad + 3 \cdot \mathbb{P}(X \geq 3) - 3 \cdot \mathbb{P}(X \geq 4) \\
&\quad \dots \\
&= 1 \cdot \mathbb{P}(X \geq 1) \\
&\quad + 2 \cdot \mathbb{P}(X \geq 2) \\
&\quad + 3 \cdot \mathbb{P}(X \geq 3) \\
&\quad \dots \\
&= \sum_{k=1}^{\infty} \mathbb{P}(X \geq k) \\
&\leq \sum_{k=1}^{\infty} \alpha^{k-1} \\
&= \sum_{k=0}^{\infty} \alpha^{k} = \frac{1}{1-\alpha}, \text{ since } \alpha < 1.
\end{aligned}
$$

Expected number of probes in an unsuccessful search

α	$\mathbb{E}[X]$
0.2	1.25
0.3	1.43
0.4	1.67
0.5	2.00
0.6	2.50
0.7	3.33
0.8	5.00
0.9	10.00
0.95	20.00
0.98	50.00

- Recall this is an optimistic estimation since clustering is ignored
- In reality, long sequences of occupied positions tend to grow longer
- See Knuth TAOCP Volume 3 for a more detailed analysis
- This analysis is sufficient to convince us to avoid large α

Quadratic probing

- The purpose of quadratic probing is to reduce the risk of clustering by adding i^2 instead of only i to the initial hash value.
- The intent is to leave a cluster quickly.
- Below h' is the original hash function

$$h(k, i) = (h'(k) + i^2) \mod m$$

- Clustering is reduced but if two different keys have the same hash value, there can be secondary clustering since the positions probed for these keys will be the same.
- Quiz: can we now know we will find an empty position if there is one?

Quadratic probing

- Answer: no
- Assume $m = 3$ and $h'(k) = 0$
- This is a very bad hash function but for illustration only, but during debugging it can be useful
- The sequence of visited positions would be: $(0, 1, 1)$ from $(0, (0 + 1^2) \mod 3 = 1, (0 + 2^2) \mod 3 = 1)$
- We miss postion 2
- Note it did not help that m is a prime number (which might have been useful...)
- Quiz: can we add a constraint to make this work?

Making quadratic probing work better

- Answer: yes.
- If m is prime and we also require that $\alpha = n/m < \frac{1}{2}$, it will work.
- Let i and j be the probe numbers made for two different searches or insertions.
- Assume both operations resulted in the same hash value so they start searching at the same positions.
- i and j will start at one, be incremented, and probe until an empty position is found.
- Assume the operation using i inserted something first, somewhere.
- The operation using j will initially use the same positions, i.e. same values as i.
- We want to show that when i and j have *different values* they would not map to the same positions.
- That means j does not "return to" a position in the sequence used by i.

A lemma

Lemma

If m is prime and $\alpha = \frac{n}{m} < \frac{1}{2}$, and $i \neq j$, then quadratic probing will find an empty position in less than $\frac{m}{2}$ probes

A proof

Proof.

Let $0 \leq i, j < \lceil \frac{m}{2} \rceil$, and $h'(k_1) = h'(k_2)$. Assume incorrectly that two different probe numbers, i and j, are mapped to the same positions.

$$\begin{array}{rcl} (h'(k_1) + i^2) \mod m & = & (h'(k_2) + j^2) \mod m \\ (h'(k_1) + i^2) & \equiv & (h'(k_2) + j^2) \mod m \\ i^2 & \equiv & j^2 \mod m \\ i^2 - j^2 & \equiv & 0 \mod m \\ (i - j)(i + j) & \equiv & 0 \mod m \end{array}$$

Since m is prime and $i \neq j$, either $i - j$ or $i + j$ is divisible by m, but since both $i - j$ and $i + j$ are less than m, none of them can be divisible by m. A contradiction. Therefore the first $\lceil \frac{m}{2} \rceil$ probes are to different positions and since $\alpha < \frac{1}{2}$, an empty position will be found. $\square$

Double hashing

- Another alternative is to use an additional hash function:

$$h(k, i) \quad = \quad (h_1(k) + i \cdot h_2(k)) \mod m$$

- Assume two different keys had the same hash value in quadratic probing, i.e., with $h_1(k)$.
- Then it is hoped that the risk that they have the same value also for $h_2(k)$ is much less.
- In practice this removes most clustering.
- By guaranteeing that $h_2(k)$ is relatively prime to m, all positions will be probed. Two simple ways to achieve this are:
  - let $m = 2^n$ and ensure that $h_2(k)$ always is odd, or
  - let m be prime and $0 < h_2(k) < m$.

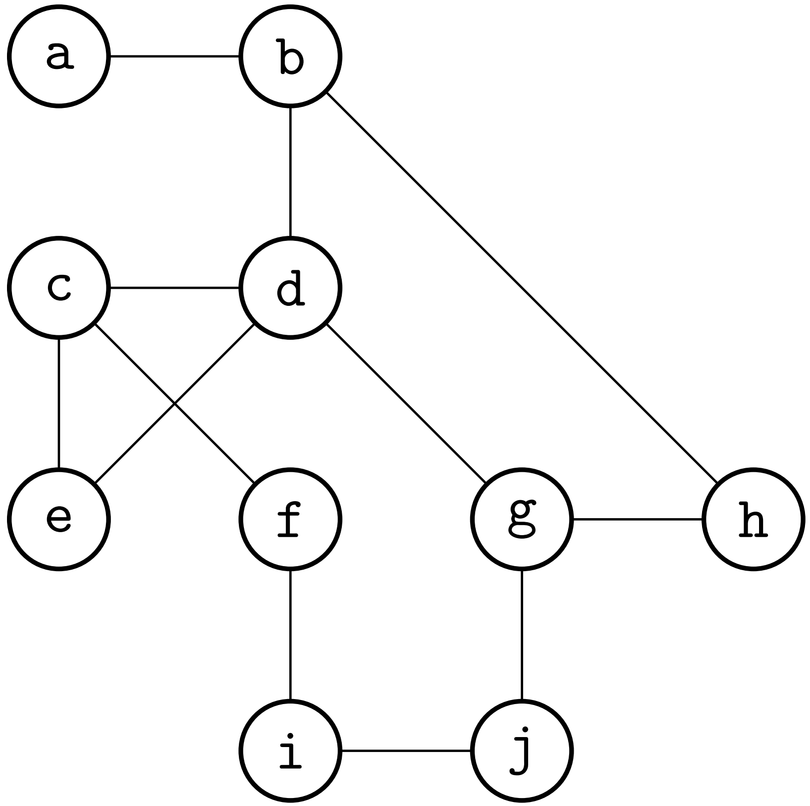
Summary of hash tables

- No obvious "best" choice
- Luckily both alternatives are easy to implement.
- Best is to make performance measurements before changing anything, obviously.
- Especially the number of cache misses can explain performance differences.
- See EDAG01 for profilers for C (and C++).
- In EDAG01 you will even use a simulator from IBM which explains what happens each clock cycle in a modern CPU (POWER8).

Graphs

- Notation
- Graph traversal and connectivity
- Testing bipartiteness
- Connectivity in directed graphs

Graphs



- $G = (V, E)$
- V is a set of nodes or vertices
- E is a set of edges or arcs
- $V = \{a, b, c, d, e, f, g, h, i, j\}$
- $E = \{a - b, b - d, ..., i - j\}$, or
- $E = \{(a, b), (b, d), ..., (i, j)\}$
- $n = |V|$
- $m = |E|$

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
- $\Theta(n^2)$ 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
- $degree(n)$ is the number of neighbors
- $\Theta(degree(n))$ time to find all neighbors of a node
- $\Theta(m)$ time to list all edges

Optimizations

- Store only half of the adjacency matrix for undirected graphs
- 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)

Paths and connectivity

- 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).

Trees

- 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

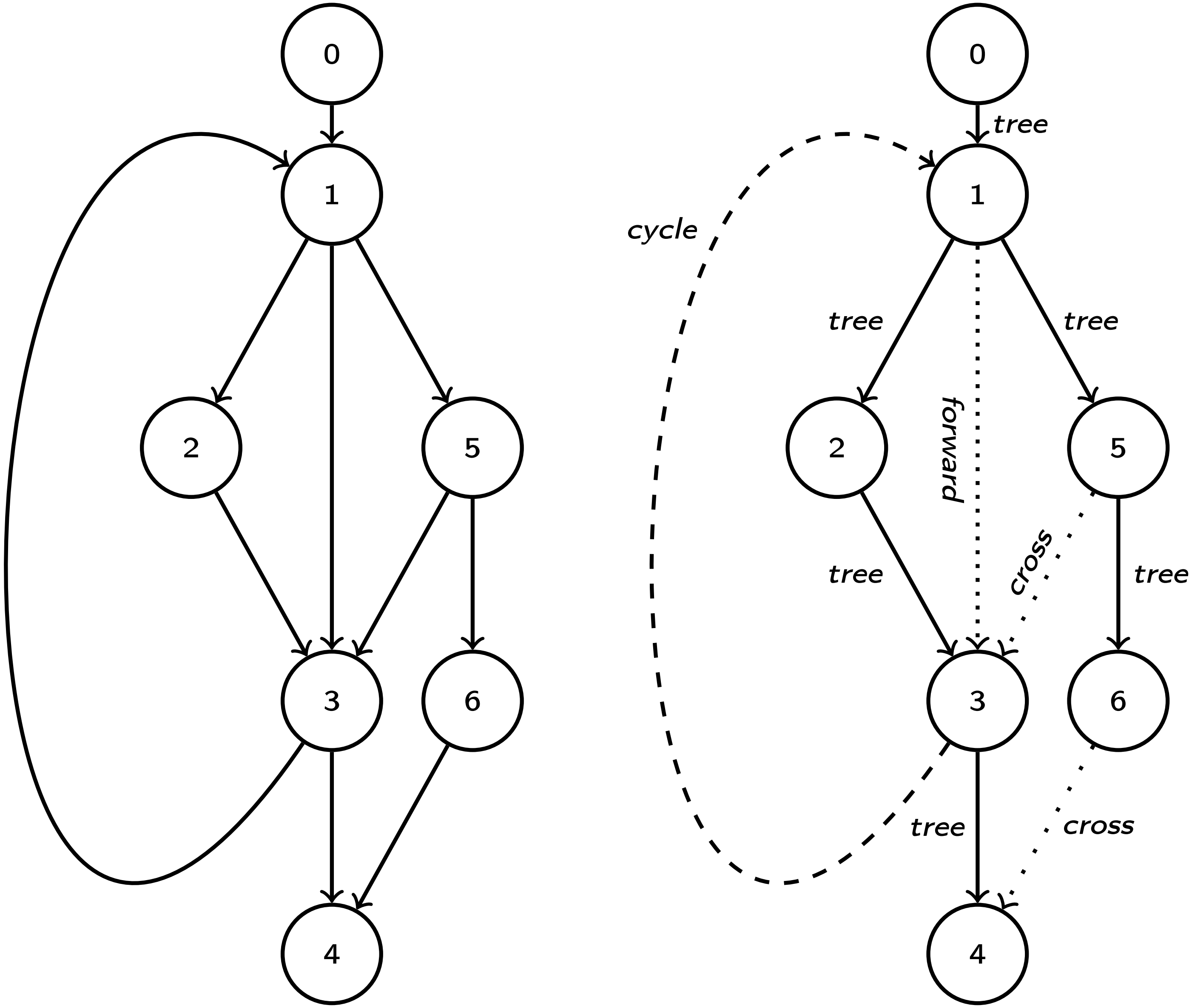
int *dfnum* ; /* Depth-first search number. */
procedure $dfs(v)$
begin
  $dfn(v) \leftarrow dfnum$
  $visited(v) \leftarrow$ true
  $dfnum \leftarrow dfnum + 1$

  for each $w \in succ(v)$ **do**
    if (**not** $visited(w)$)
      $dfs(w)$
end
procedure $depth\_first\_search(V)$
    $dfnum \leftarrow 0$
    for each $v \in V$ **do**
      $visited(v) \leftarrow$ false
    $dfs(s)$
end

DFS

- 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 often more beautiful than art

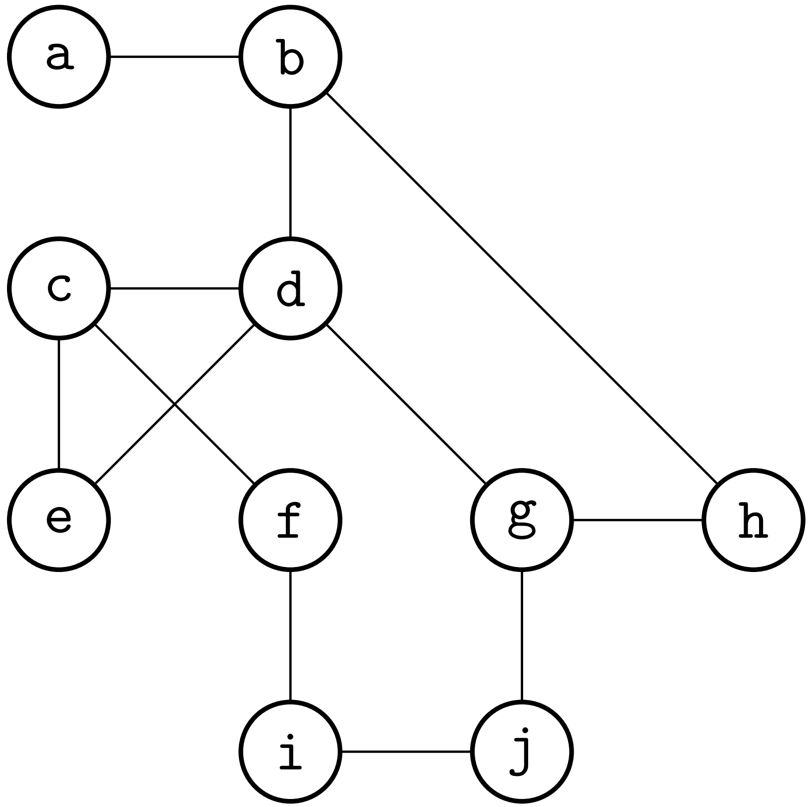
DFS example


0
1
2
5
3
6
4
0
tree
1
cycle
tree
tree
2
forward
5
cross
tree
tree
3
6
tree
cross
4

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 start 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 $w \in neighbor(v)$

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 a path $a - j$
- One is $p = (a, b, d, g, j)$
- For each node w except the first, the attribute $pred(w)$ is the previous node in p.
- $pred(j) = g$, $pred(g) = d$, 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 $2m = \sum_{v \in V} degree(v)$ edges to process.
- $2m$ since each edge is in two adjacency lists
- Thus BFS can be implemented in $O(n + m)$ with adjacency lists