CS251 Final Review

- Trees
- Hash Tables
- Graphs
- Strings

CS251 Final Review

- Trees
- Hash Tables
- Graphs
- Strings

Trees

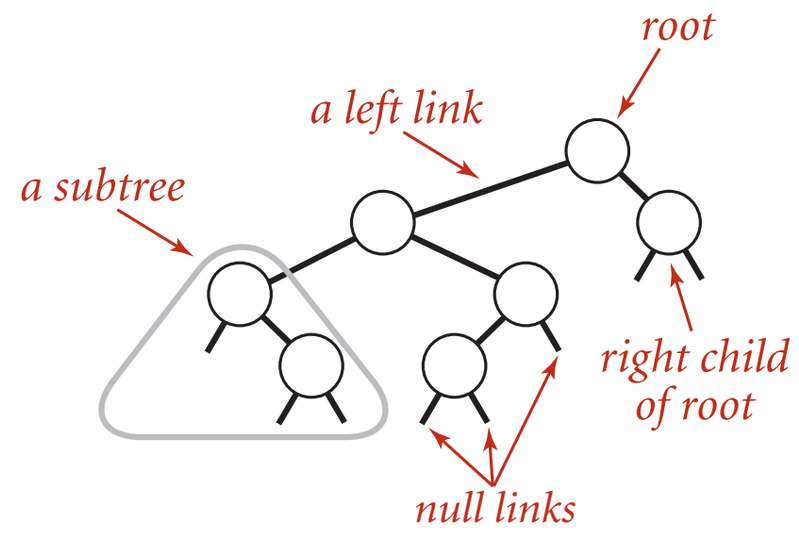
- Binary Search Trees
- Balanced Search Trees

Binary search trees

Definition. A BST is a binary tree in symmetric order.

A binary tree is either:

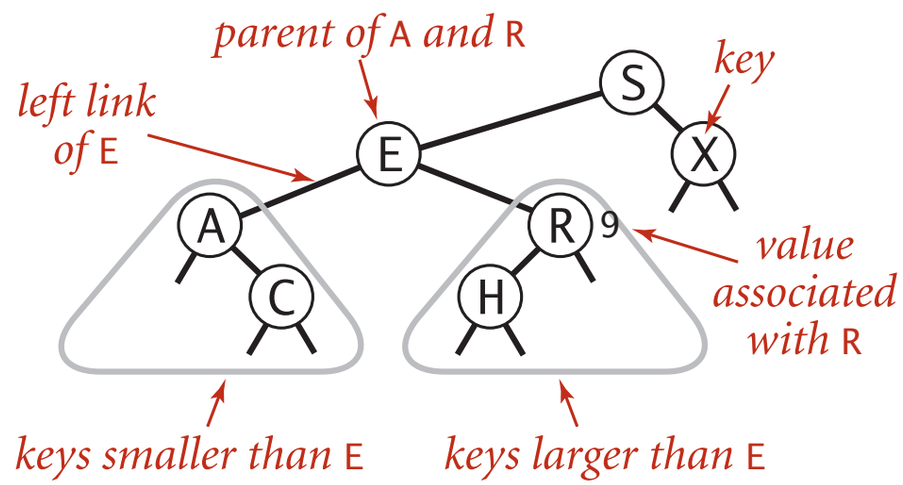
- Empty.
- Two disjoint binary trees (left and right).



Anatomy of a binary tree

Symmetric order. Each node has a key, and every node's key is:

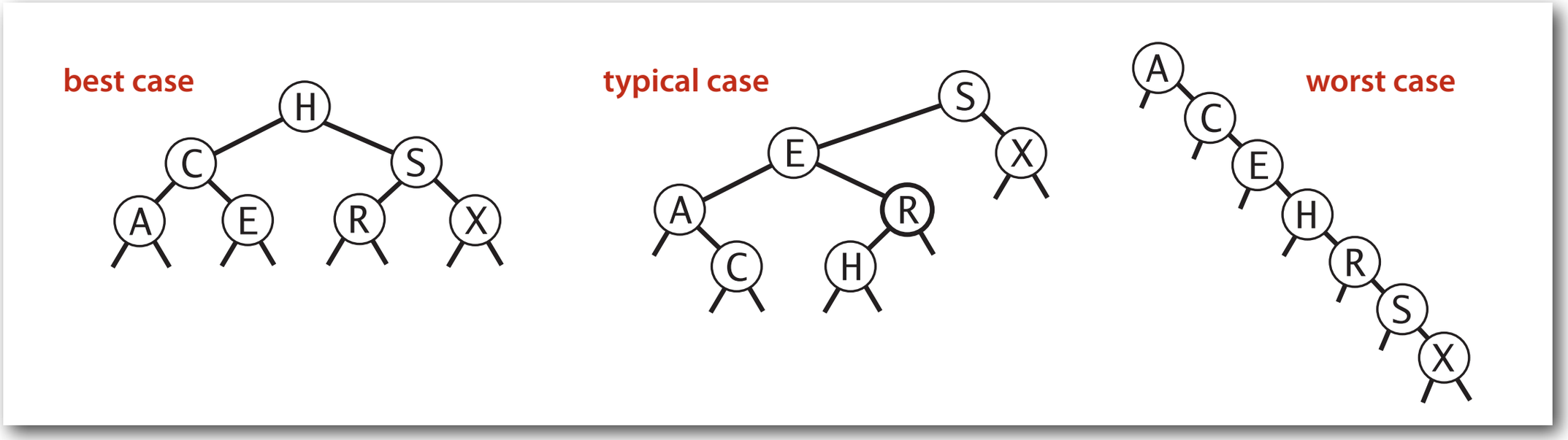
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Anatomy of a binary search tree

Tree shape

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to depth of node.



Remark. Tree shape depends on order of insertion.

ST implementations: summary

implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	`equals()`
binary search (ordered array)	lg N	N	lg N	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	?	`compareTo()`

BST: ordered symbol table operations

	sequential search	binary search	BST
search	N	lg N	h
insert	1	N	h
min / max	N	1	h
floor / ceiling	N	lg N	h
rank	N	lg N	h
select	N	1	h
ordered iteration	N log N	N	N

h = height of BST
(proportional to log N
if keys inserted in random order)

worst-case running time of ordered symbol table operations

Trees

- Binary Search Trees
- Balanced Search Trees

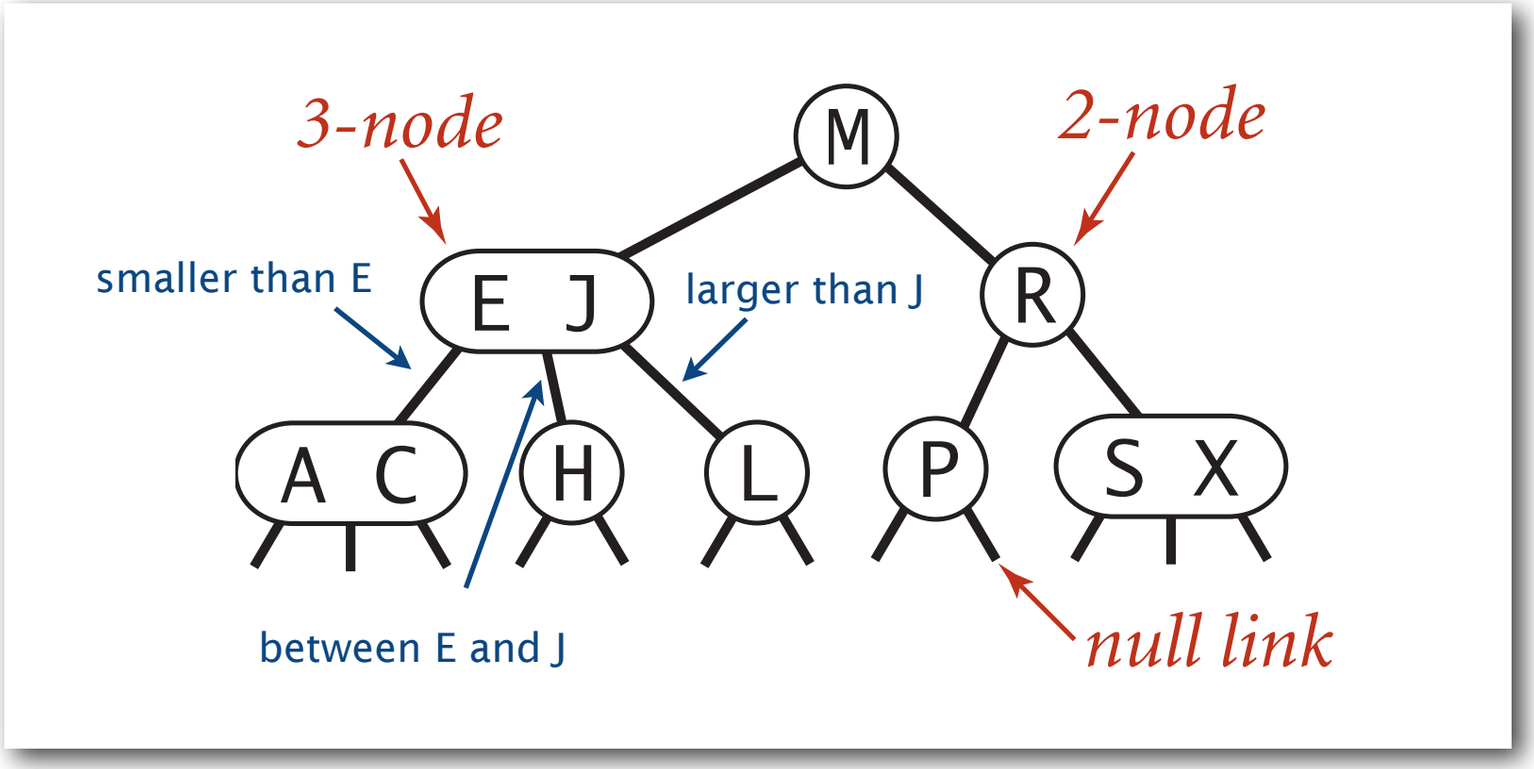
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

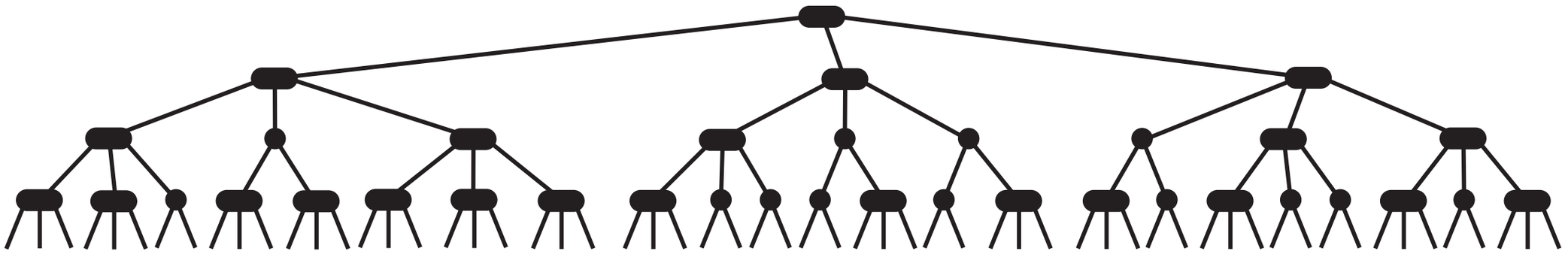
Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



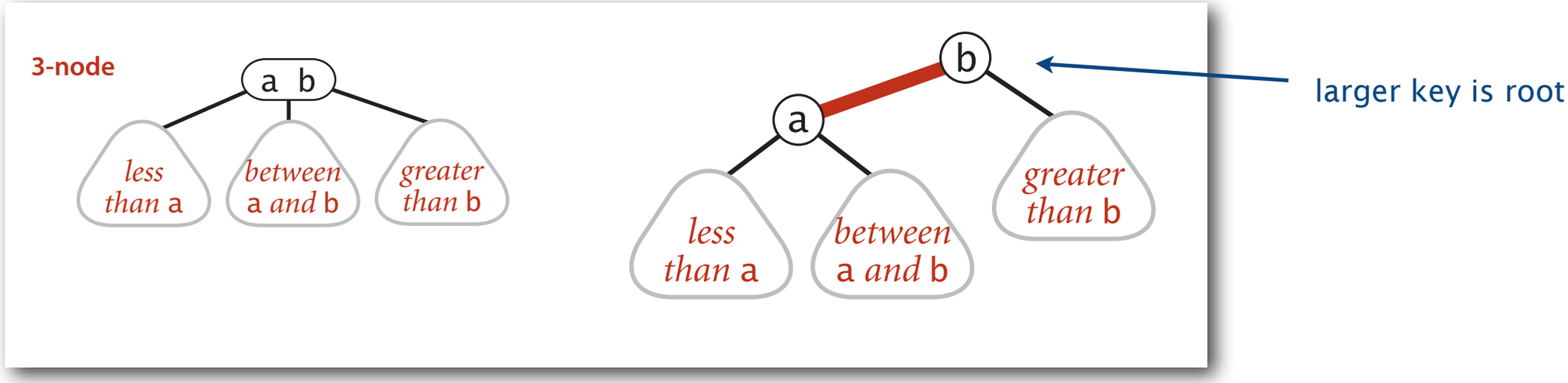
Tree height.

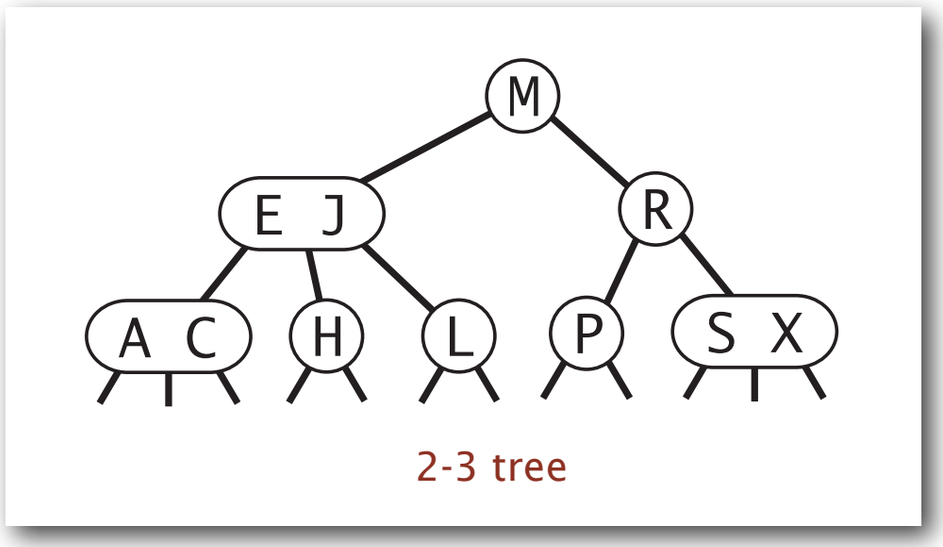
- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

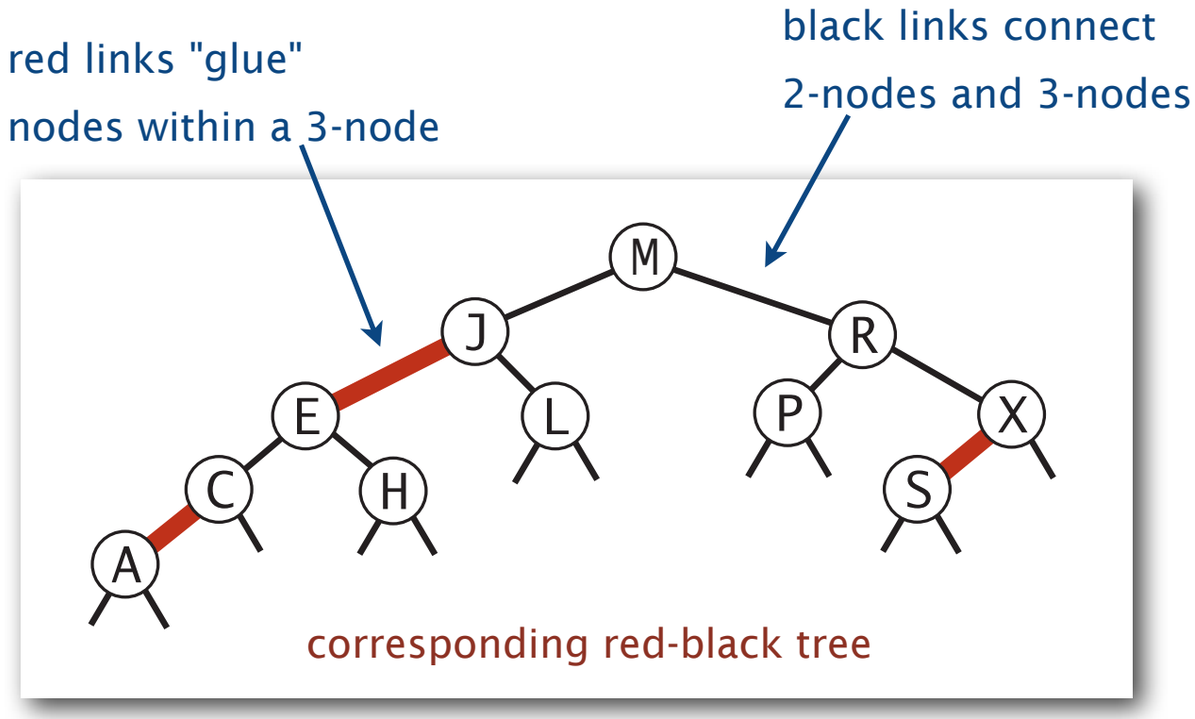
Guaranteed logarithmic performance for search and insert.

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.





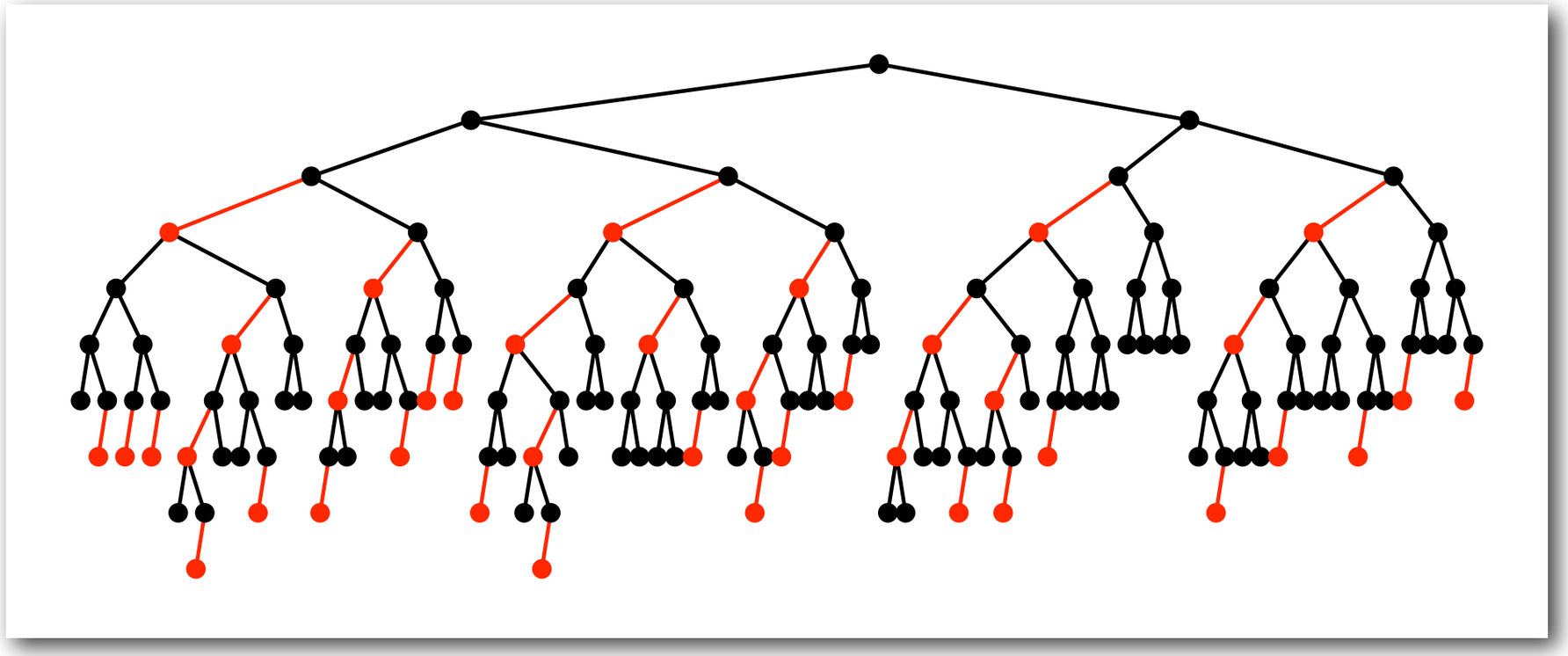


Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \lg N$ in the worst case.

Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.



Property. Height of tree is $\sim 1.00 \lg N$ in typical applications.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N *	1.00 lg N *	1.00 lg N *	yes	`compareTo()`

* exact value of coefficient unknown but extremely close to 1

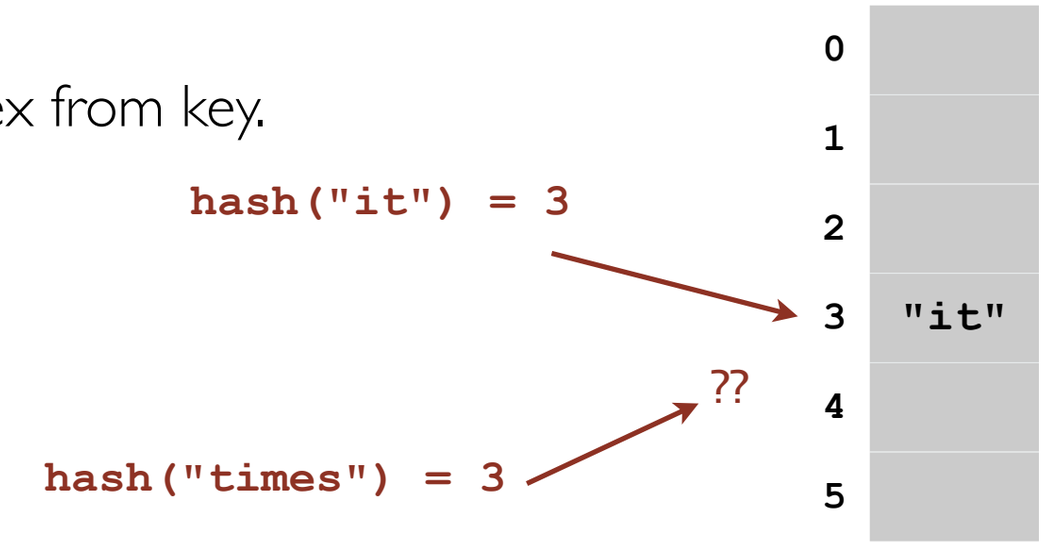
CS251 Final Review

- Trees
- Hash Tables
- Graphs
- Strings

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure
  to handle two keys that hash to the same array index.

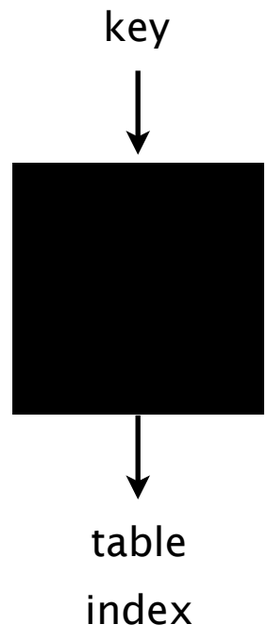
Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.



thoroughly researched problem,
still problematic in practical applications

Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

Practical challenge. Need different approach for each key type.

Modular hashing

Hash code. An **`int`** between **-2^{31}** and **$2^{31}-1$**.

Hash function. An **`int`** between **`0`** and **`M-1`** (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)
{  return key.hashCode() % M;  }
```

bug

```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

1-in-a-billion bug

hashCode() of "polygenelubricants" is -2^{31}

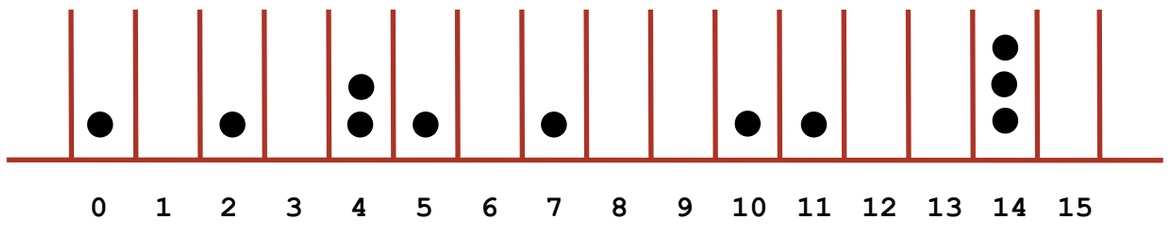
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

correct

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.

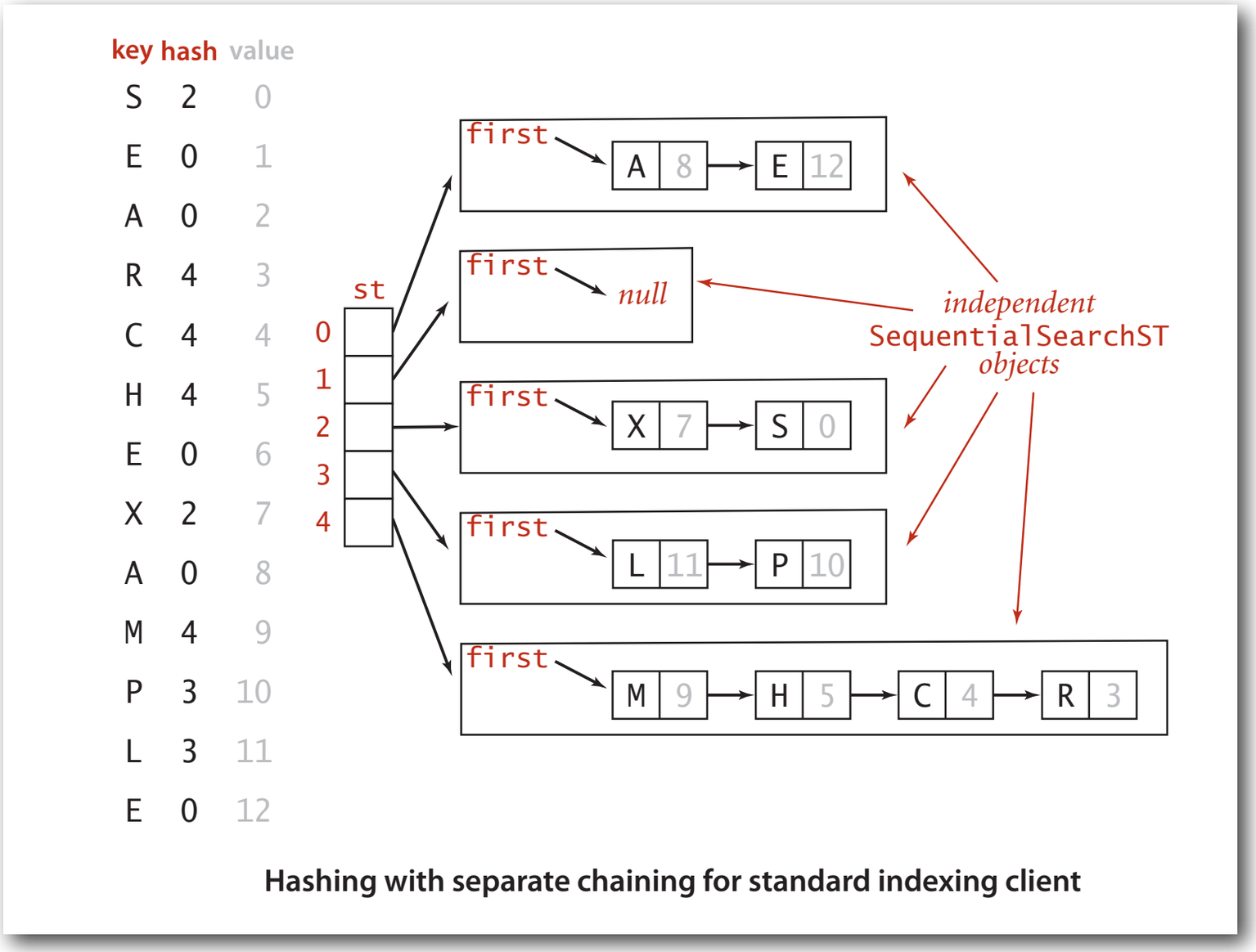


Java's `String` data uniformly distribute the keys of Tale of Two Cities

Separate chaining ST

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: only need to search i^{th} chain.



Hashing with separate chaining for standard indexing client

Linear probing

Use an array of size $M > N$.

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at table index i if free; if not try $i + 1$, $i + 2$, etc.
- Search: search table index i; if occupied but no match, try $i + 1$, $i + 2$, etc.

-	-	-	S	H	-	-	A	C	E	R	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

-	-	-	S	H	-	-	A	C	E	R	I	-
0	1	2	3	4	5	6	7	8	9	10	11	12

insert I
hash(I) = 11

-	-	-	S	H	-	-	A	C	E	R	I	N
0	1	2	3	4	5	6	7	8	9	10	11	12

insert N
hash(N) = 8

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`

* under uniform hashing assumption

Hashing vs. balanced search trees

Hashing.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement **`compareTo()`** correctly than **`equals()`** and **`hashCode()`**.

Java system includes both.

- Red-black trees: **`java.util.TreeMap`**, **`java.util.TreeSet`**.
- Hashing: **`java.util.HashMap`**, **`java.util.IdentityHashMap`**.

CS251 Final Review

- Trees
- Hash Tables
- Graphs
- Strings

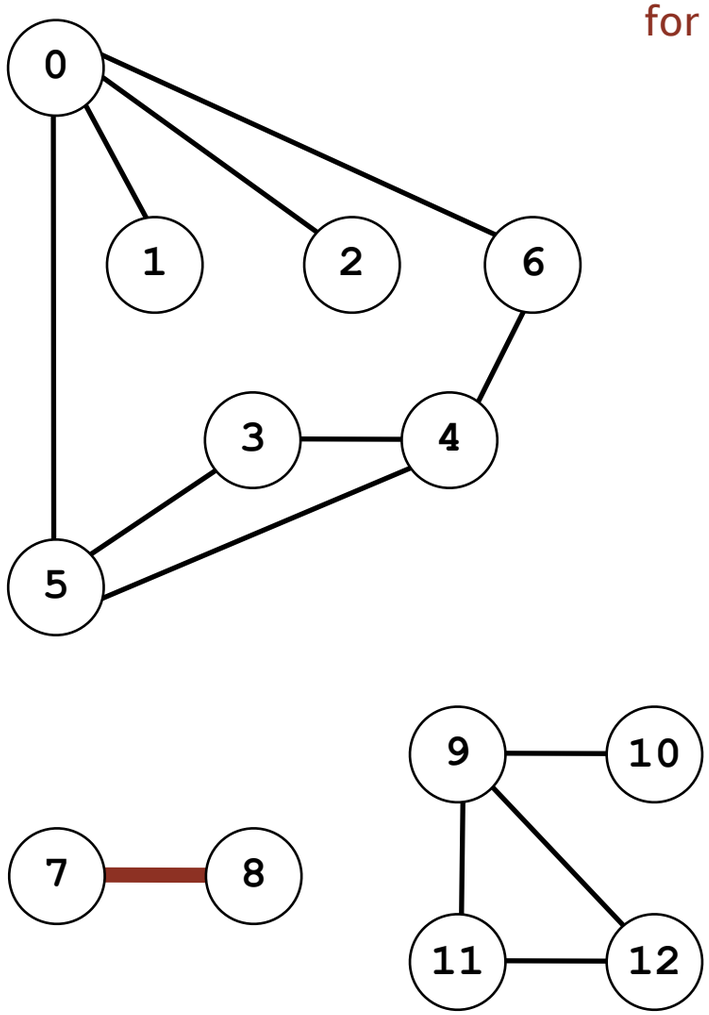
Graphs

- Undirected Graphs
- Directed Graphs
- Minimum Spanning Trees
- Shortest Paths

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;

for each edge v-w in graph: **`adj[v][w] = adj[w][v] = true`**.



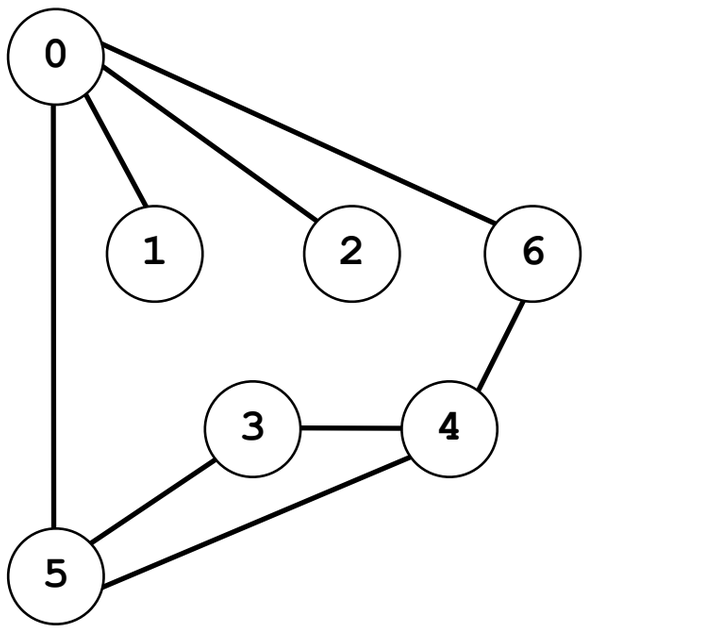
two entries for each edge

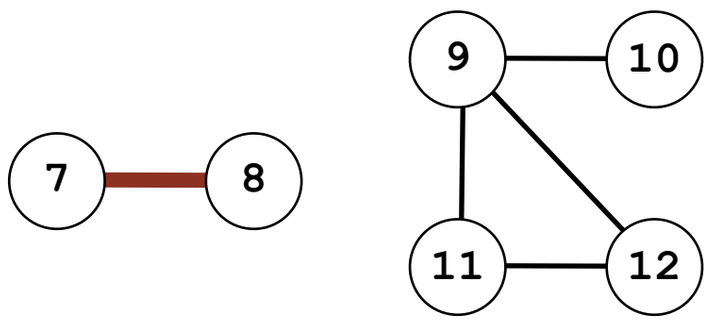
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

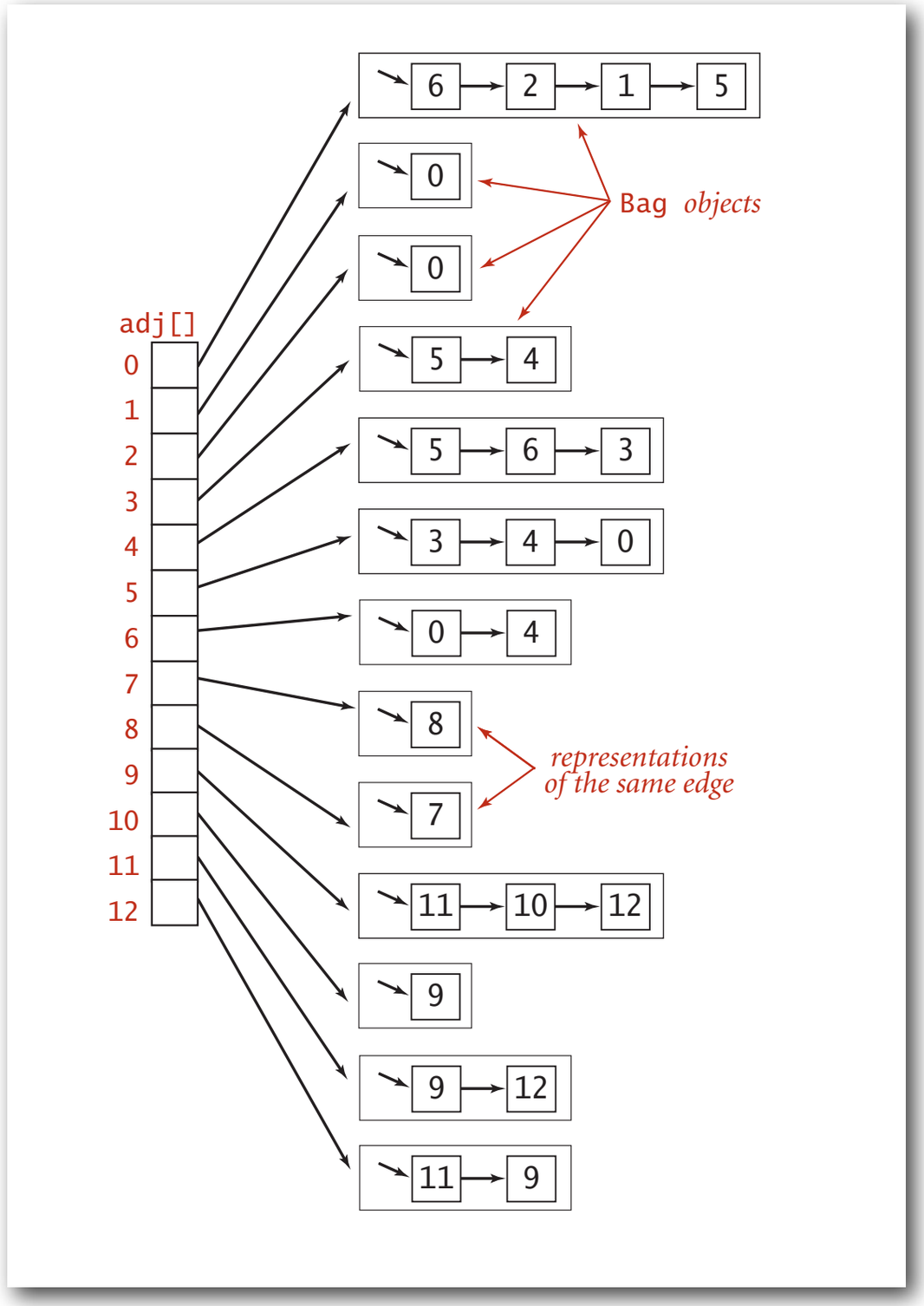
Adjacency-list graph representation

Maintain vertex-indexed array of lists.

(use **`Bag`** abstraction)







Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be "sparse."

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

* disallows parallel edges

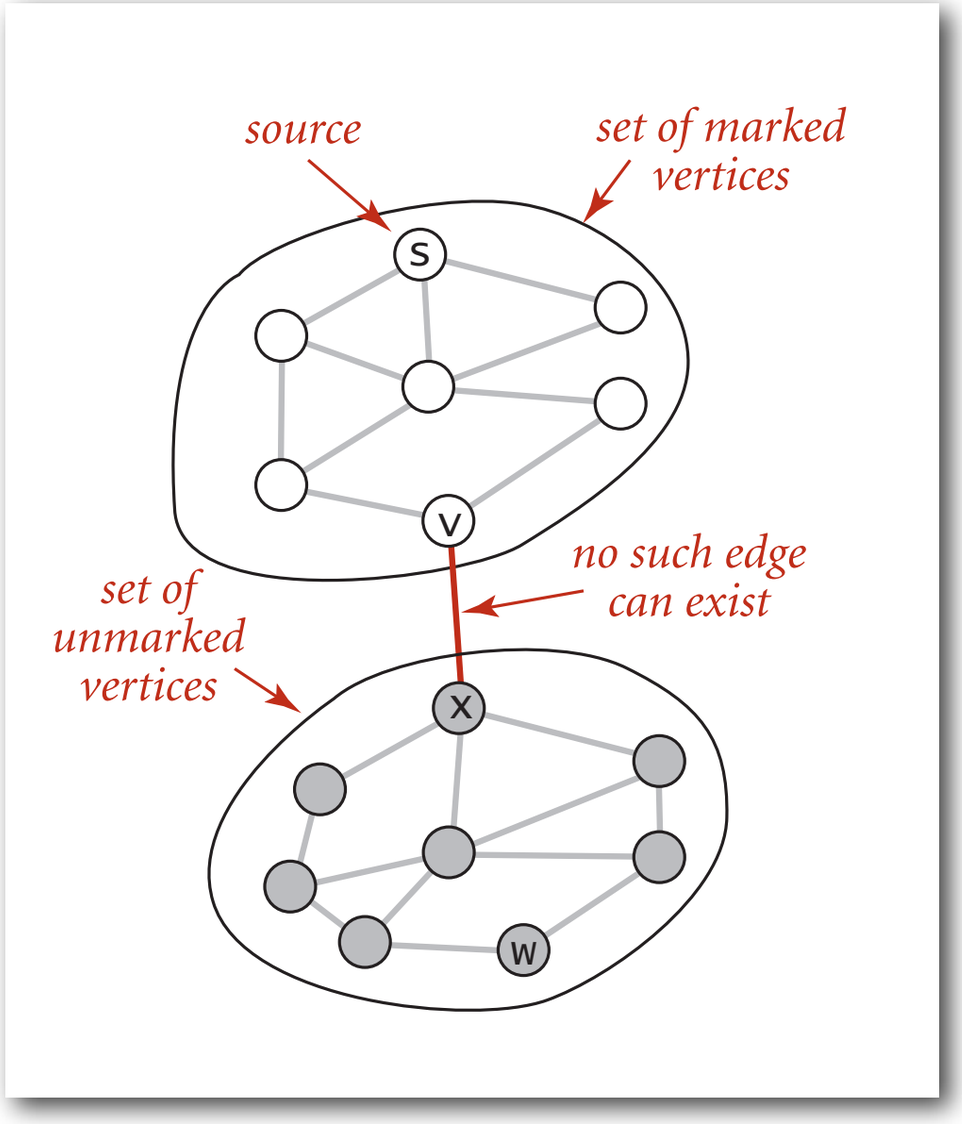
Depth-first search properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees.

Pf.

- Correctness:
  - if w marked, then w connected to s (why?)
  - if w connected to s, then w marked
    (if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one)

- Running time: each vertex connected to s is visited once.



Breadth-first search

Depth-first search. Put unvisited vertices on a stack.

Breadth-first search. Put unvisited vertices on a queue.

Shortest path. Find path from s to t that uses fewest number of edges.

BFS (from source vertex s)

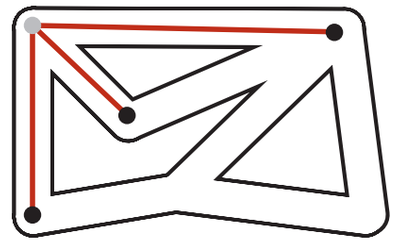
Put s onto a FIFO queue, and mark s as visited.

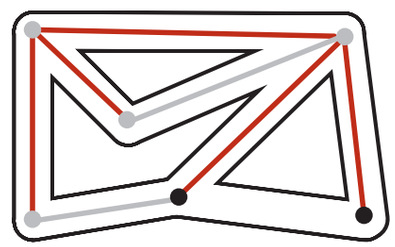
Repeat until the queue is empty:

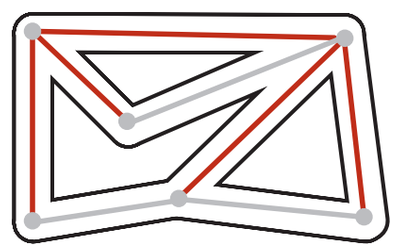
- remove the least recently added vertex v

- add each of v's unvisited neighbors to the queue,

and mark them as visited.







Intuition. BFS examines vertices in increasing distance from s.

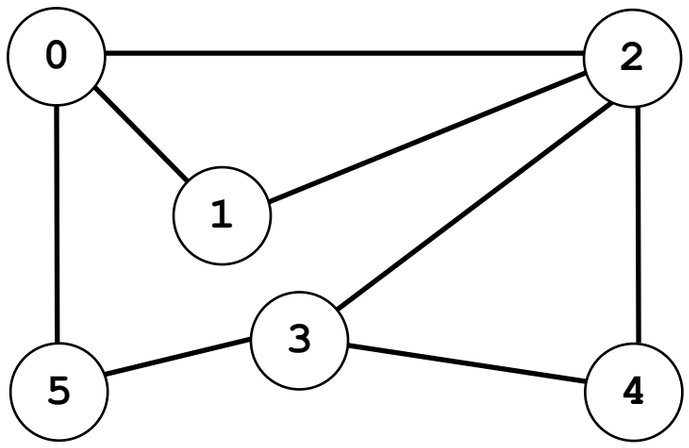
Breadth-first search properties

Proposition. BFS computes shortest path (number of edges) from s
in a connected graph in time proportional to $E + V$.

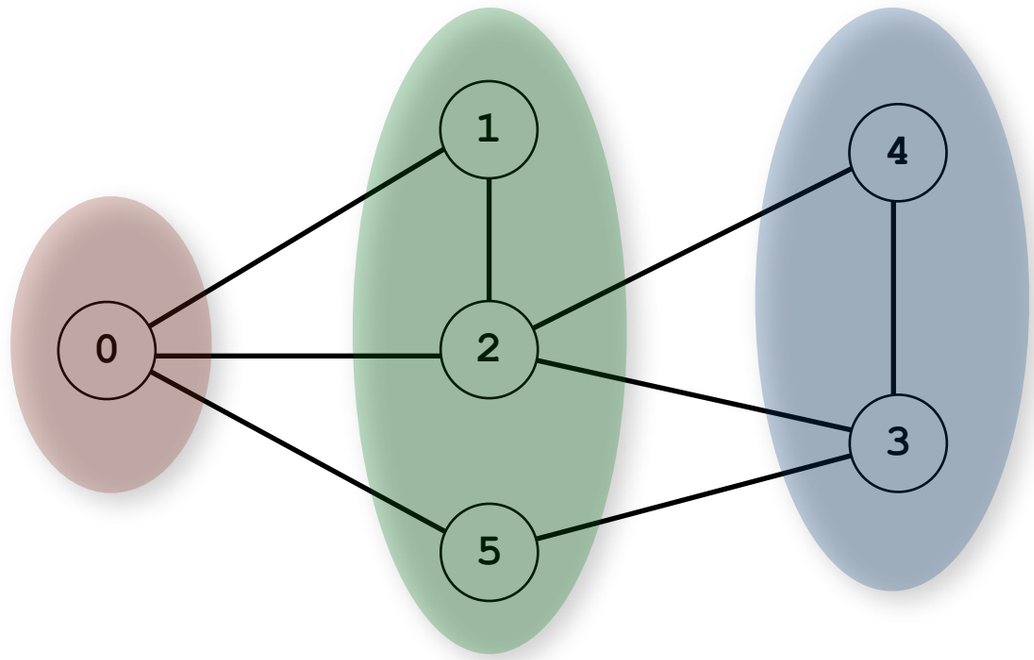
Pf.

- Correctness: queue always consists of zero or more vertices of distance k from s, followed by zero or more vertices of distance $k + 1$.

- Running time: each vertex connected to s is visited once.



standard drawing



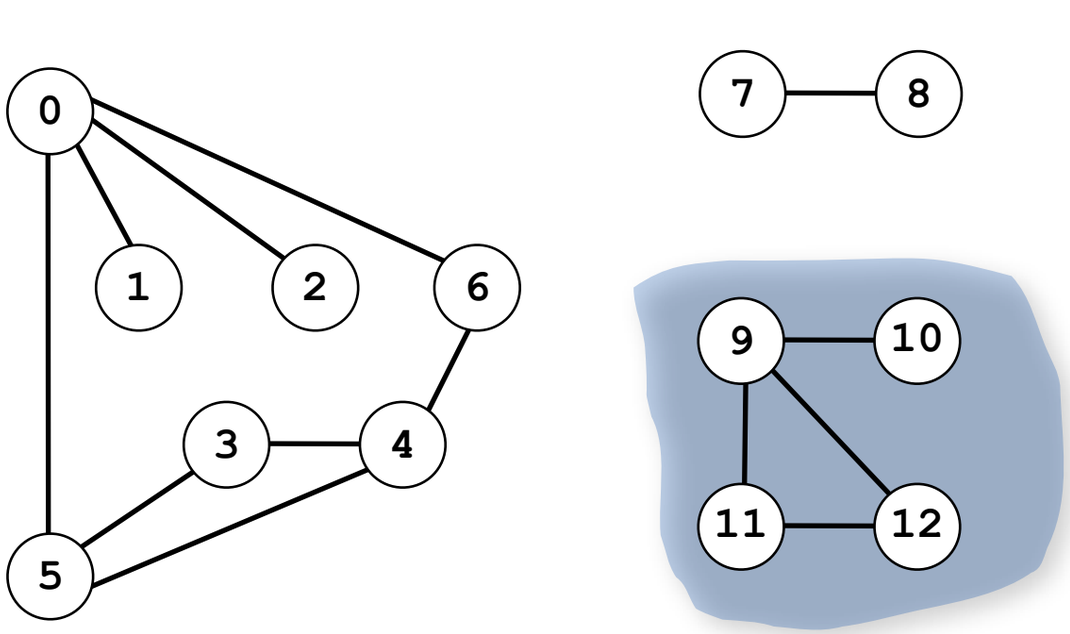
dist = 0 dist = 1 dist = 2

Connected components

The relation "is connected to" is an equivalence relation:

- Reflexive: v is connected to v.
- Symmetric: if v is connected to w, then w is connected to v.
- Transitive: if v connected to w and w connected to x, then v connected to x.

Def. A connected component is a maximal set of connected vertices.



3 connected components

v	id[v]
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	2
10	2
11	2
12	2

Remark. Given connected components, can answer queries in constant time.

Graphs

- Undirected Graphs
- Directed Graphs
- Minimum Spanning Trees
- Shortest Paths

Depth-first search in digraphs

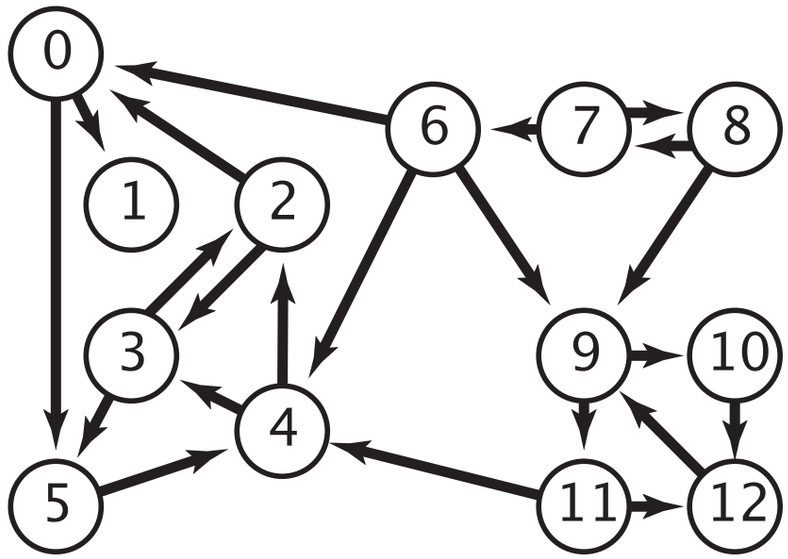
Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked vertices w adjacent from v.



Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

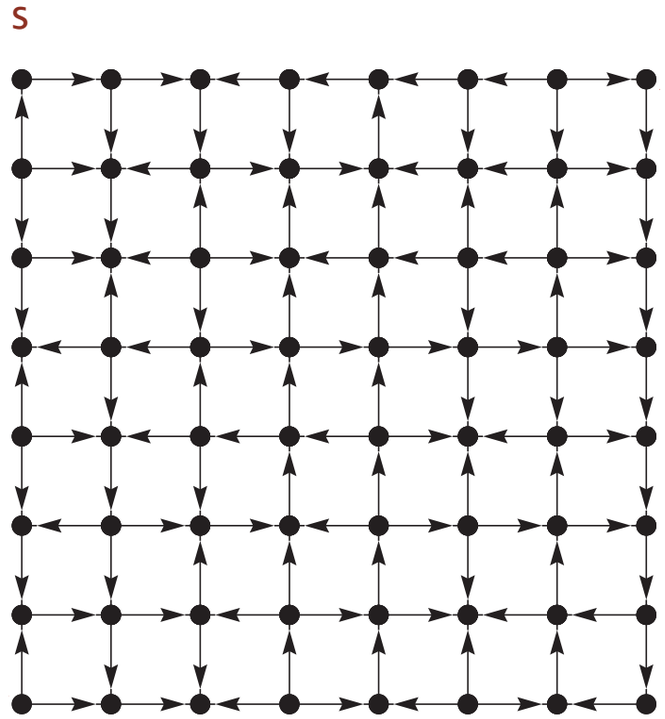
BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- **remove the least recently added vertex v**
- **for each unmarked vertex adjacent from v:**

  add to queue and mark as visited..

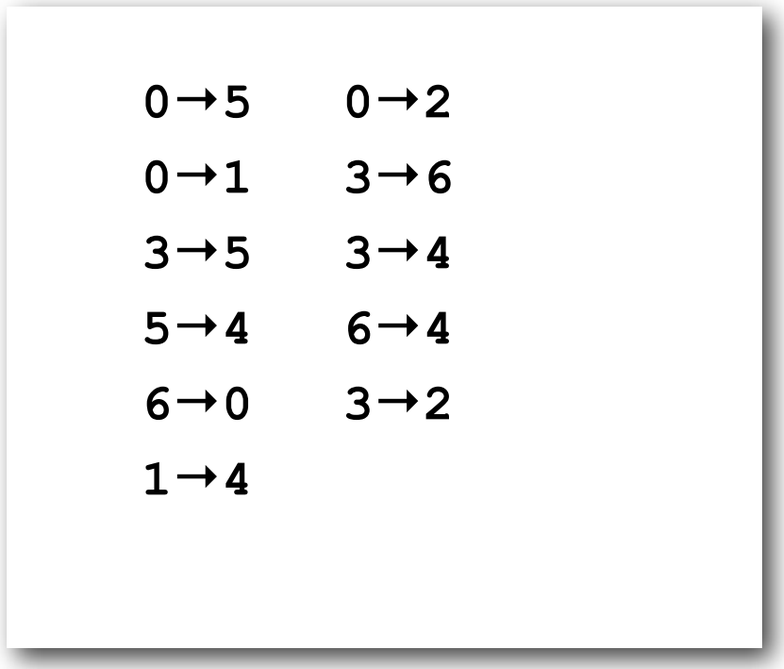


Proposition. BFS computes shortest paths (fewest number of edges).

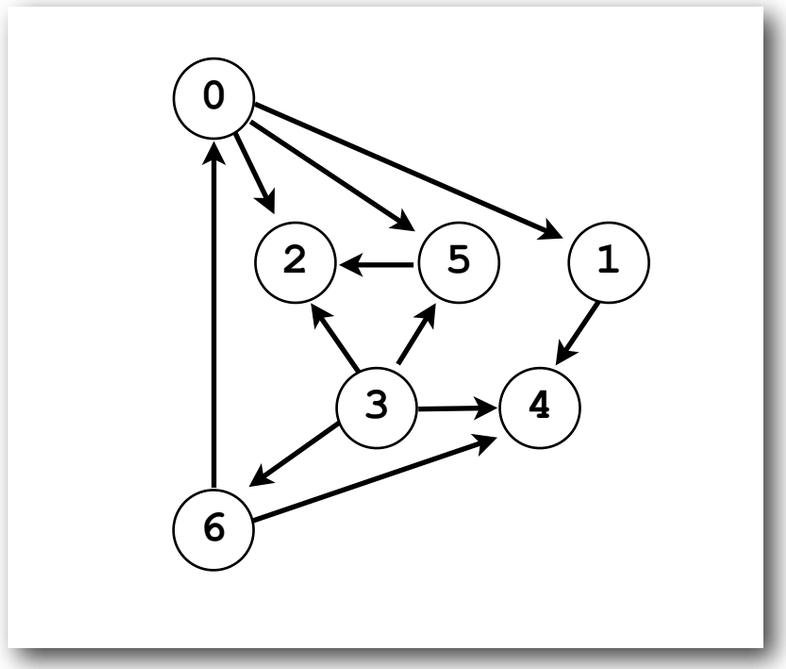
Topological sort

DAG. Directed acyclic graph.

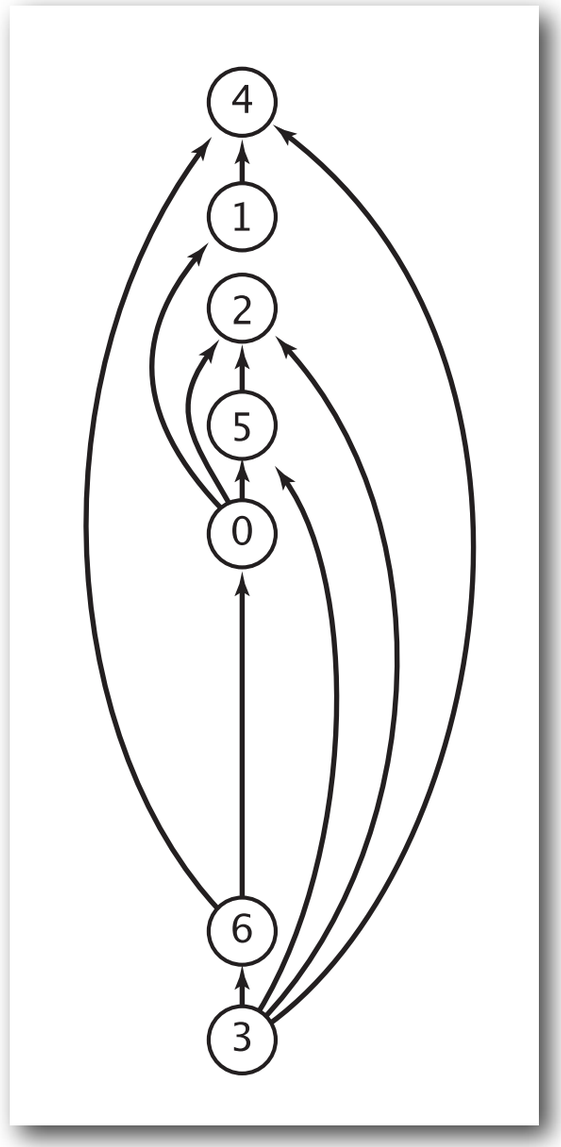
Topological sort. Redraw DAG so all edges point up.



directed edges



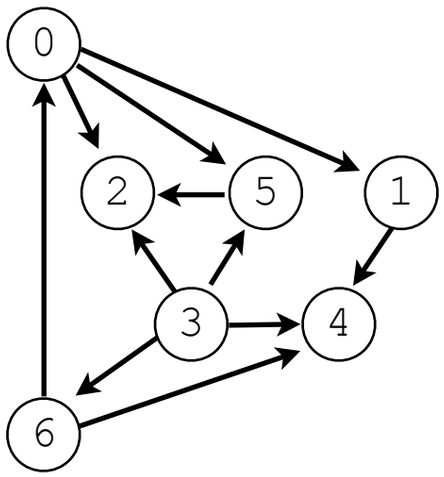
DAG



topological order

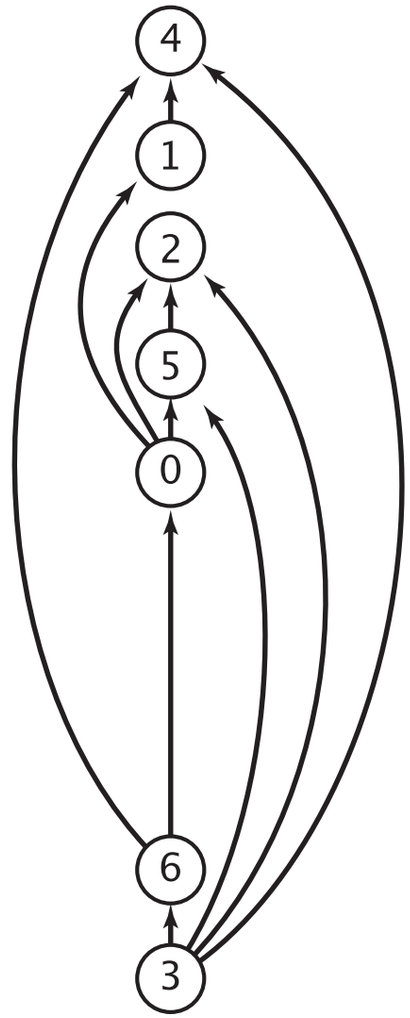
Solution. DFS. What else?

Reverse DFS postorder in a DAG



0→5
0→2
0→1
3→6
3→5
3→4
5→4
6→4
6→0
3→2
1→4

	marked[]	reversePost
dfs(0)	1 0 0 0 0 0 0	-
dfs(1)	1 1 0 0 0 0 0	-
dfs(4)	1 1 0 0 1 0 0	-
4 done	1 1 0 0 1 0 0	4
1 done	1 1 0 0 1 0 0	4 1
dfs(2)	1 1 1 0 1 0 0	4 1
2 done	1 1 1 0 1 0 0	4 1 2
dfs(5)	1 1 1 0 1 1 0	4 1 2
check 2	1 1 1 0 1 1 0	4 1 2
5 done	1 1 1 0 1 1 0	4 1 2 5
0 done	1 1 1 0 1 1 0	4 1 2 5 0
check 1	1 1 1 0 1 1 0	4 1 2 5 0
check 2	1 1 1 0 1 1 0	4 1 2 5 0
dfs(3)	1 1 1 1 1 1 0	4 1 2 5 0
check 2	1 1 1 1 1 1 0	4 1 2 5 0
check 4	1 1 1 1 1 1 0	4 1 2 5 0
check 5	1 1 1 1 1 1 0	4 1 2 5 0
dfs(6)	1 1 1 1 1 1 1	4 1 2 5 0
6 done	1 1 1 1 1 1 1	4 1 2 5 0 6
3 done	1 1 1 1 1 1 1	4 1 2 5 0 6 3
check 4	1 1 1 1 1 1 0	4 1 2 5 0 6 3
check 5	1 1 1 1 1 1 0	4 1 2 5 0 6 3
check 6	1 1 1 1 1 1 0	4 1 2 5 0 6 3
done	1 1 1 1 1 1 1	4 1 2 5 0 6 3



reverse DFS postorder is a topological order!

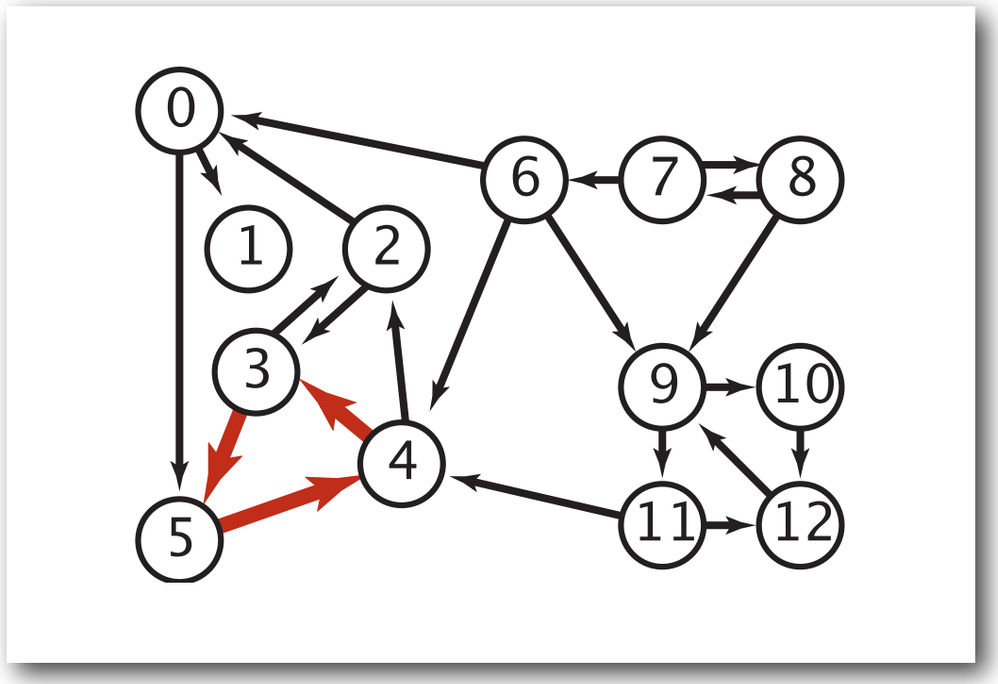
Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.

Goal. Given a digraph, find a directed cycle.



Solution. DFS. What else? See textbook.

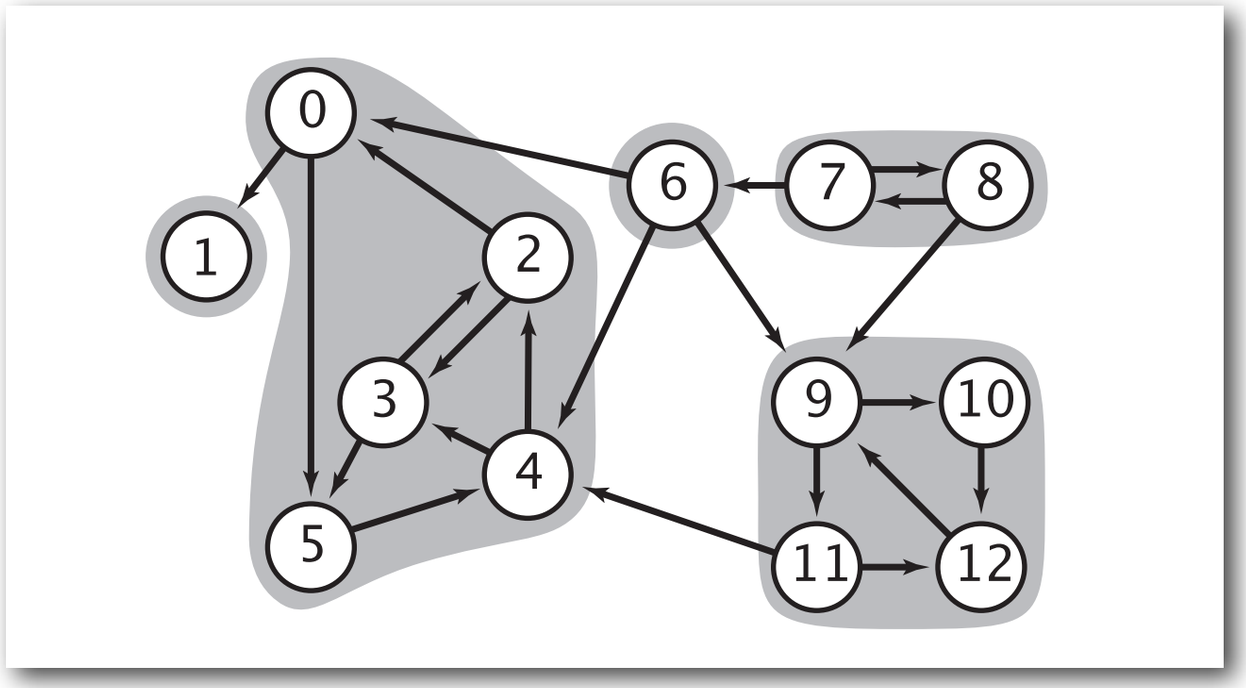
Strongly-connected components

Def. Vertices v and w are strongly connected if there is a directed path from v to w and a directed path from w to v.

Key property. Strong connectivity is an equivalence relation:

- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is strongly connected to x.

Def. A strong component is a maximal subset of strongly-connected vertices.



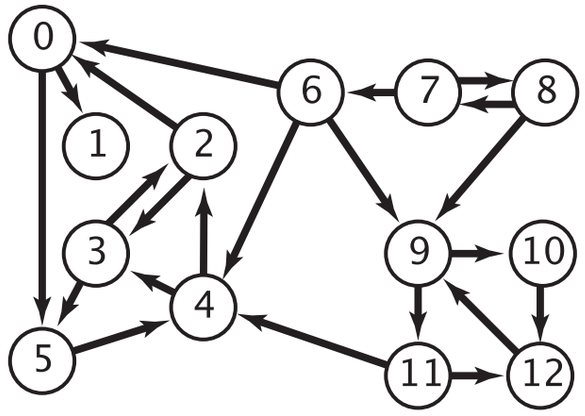
Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

- Run DFS on G^R to compute reverse postorder.
- Run DFS on G, considering vertices in order given by first DFS.

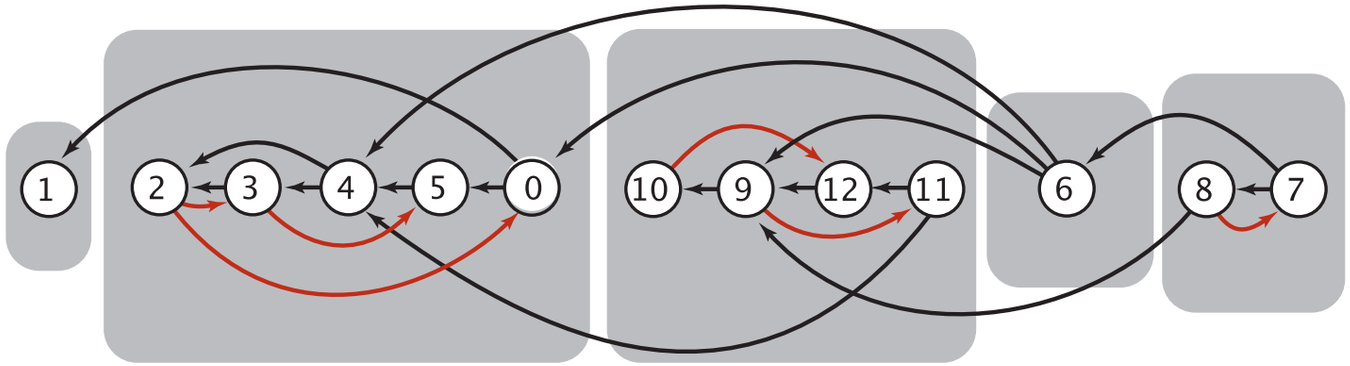
DFS in original digraph

G



check unmarked vertices in the order

1 0 2 4 5 3 11 9 12 10 6 7 8




```
dfs(1)
1 done
```

```
dfs(0)
  dfs(5)
    dfs(4)
      dfs(3)
        check 5
        dfs(2)
          check 0
          check 3
        2 done
      3 done
      check 2
    4 done
  5 done
  check 1
0 done
check 2
check 4
check 5
check 3
```

```
dfs(11)
  check 4
  dfs(12)
    dfs(9)
      check 11
      dfs(10)
        check 12
      10 done
    9 done
  12 done
11 done
check 9
check 12
check 10
```

```
dfs(6)
  check 9
  check 4
  check 0
6 done
```

```
dfs(7)
  check 6
  dfs(8)
    check 7
    check 9
  8 done
7 done
check 8
```

Proposition. Second DFS gives strong components. (!!)

Graphs

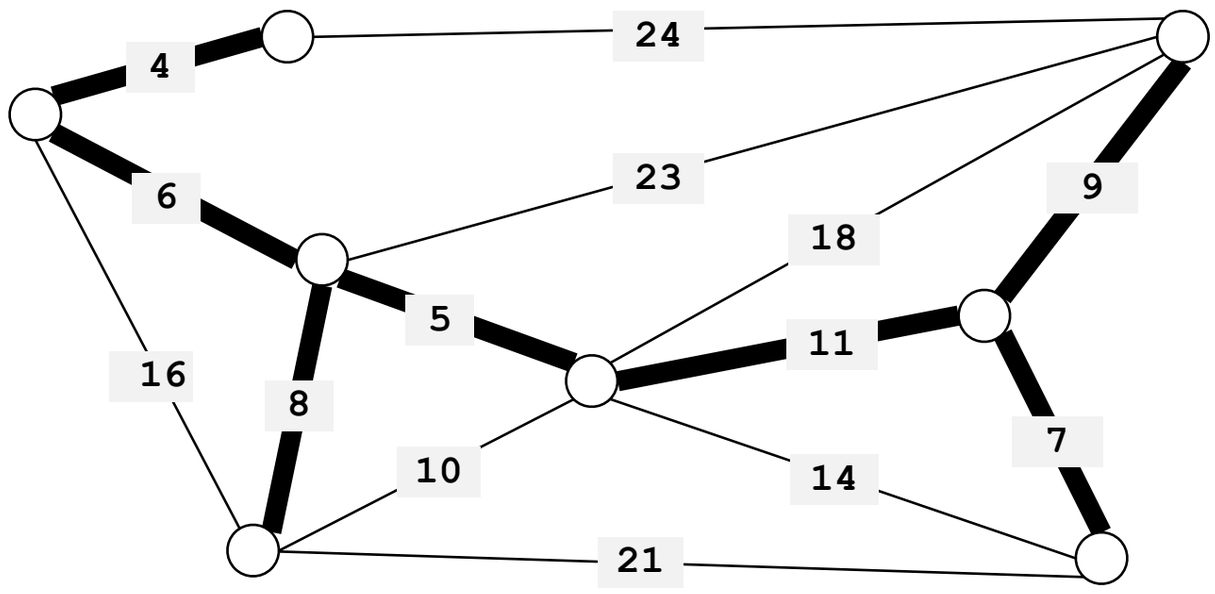
- Undirected Graphs
- Directed Graphs
- Minimum Spanning Trees
- Shortest Paths

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A spanning tree of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



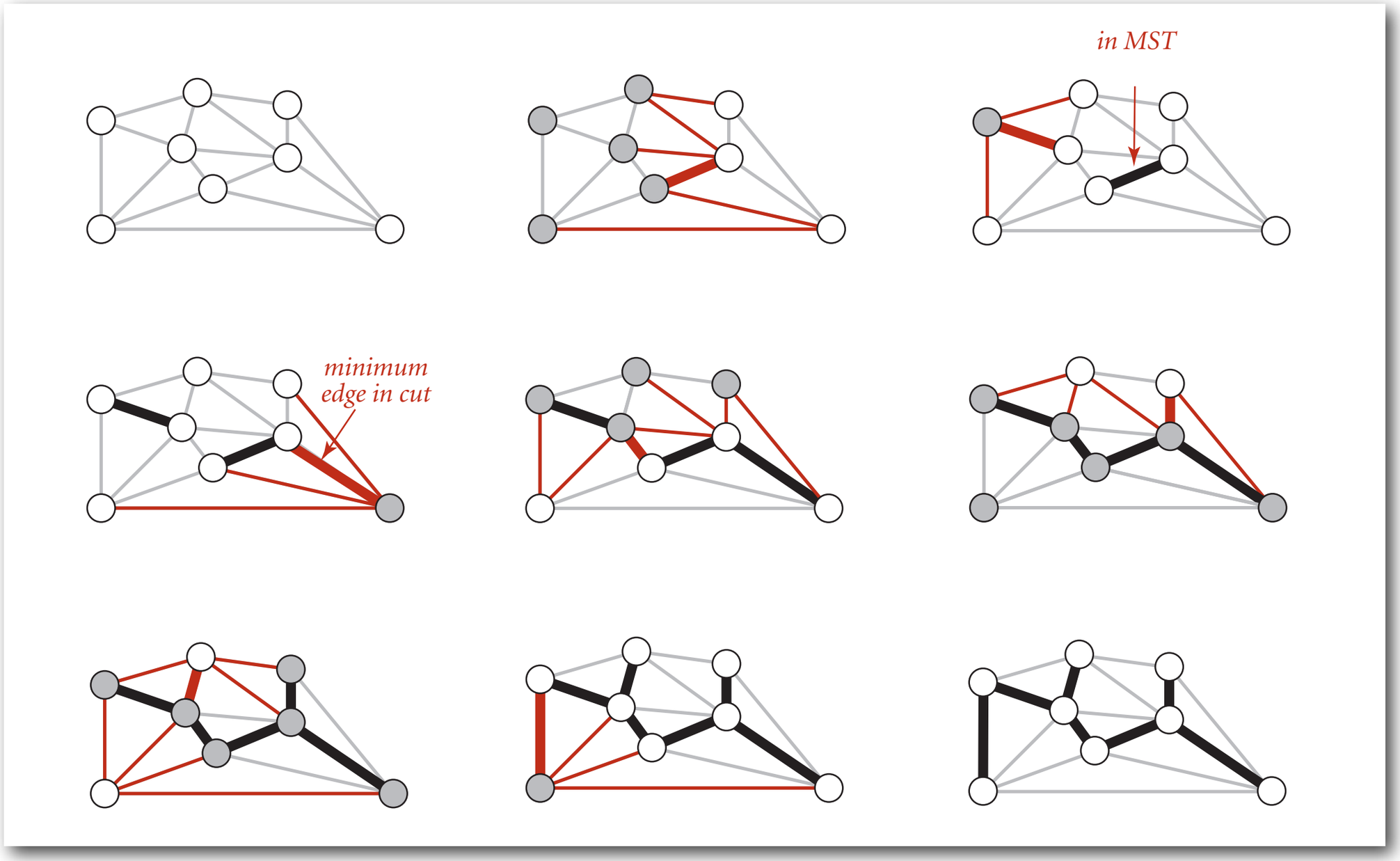
spanning tree T: cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Brute force. Try all spanning trees?

Greedy MST algorithm

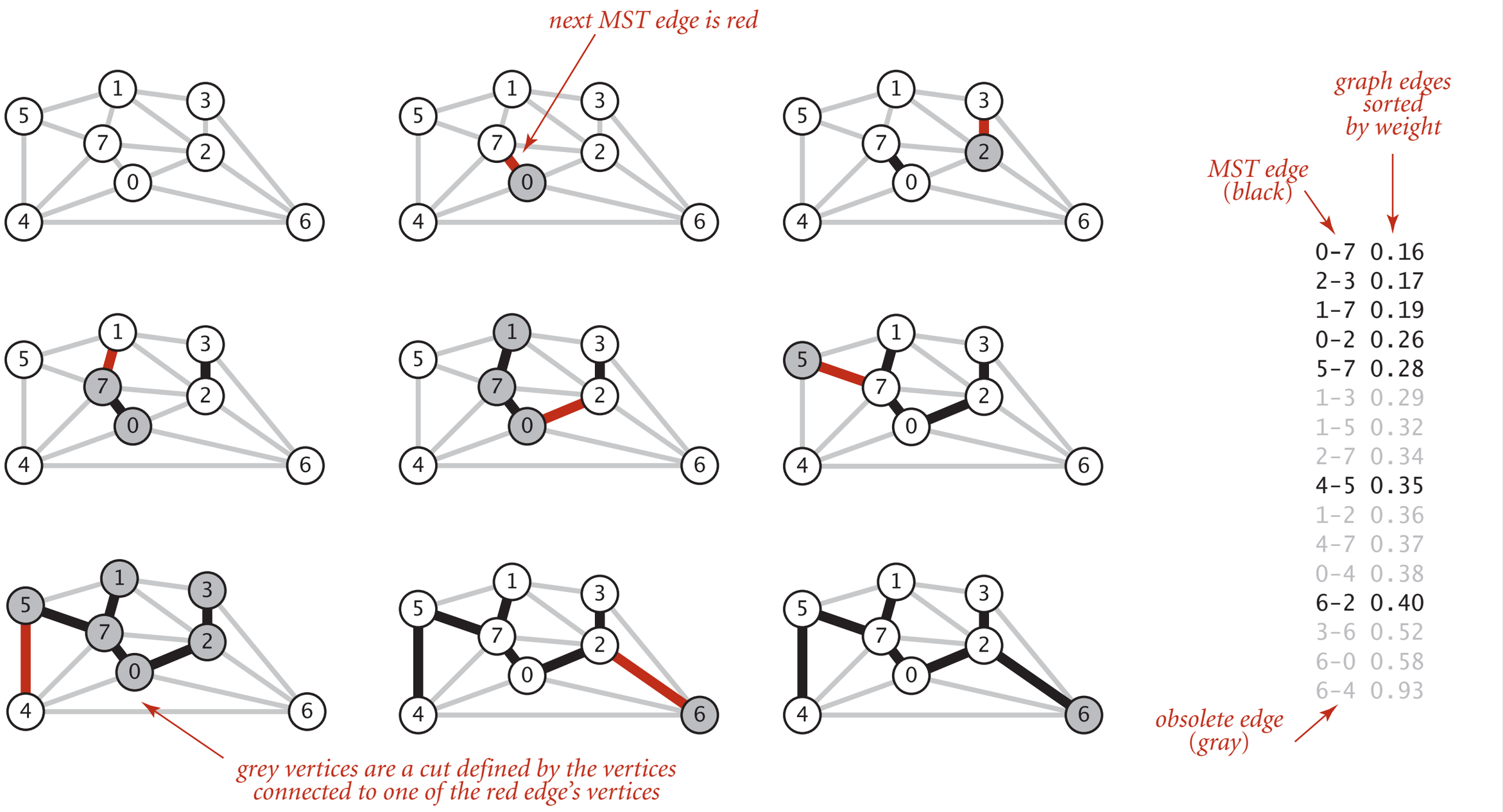
Proposition. The following algorithm computes the MST:

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Continue until $V - 1$ edges are colored black.



Kruskal's algorithm

Kruskal's algorithm. [Kruskal 1956] Consider edges in ascending order of weight. Add the next edge to the tree T unless doing so would create a cycle.

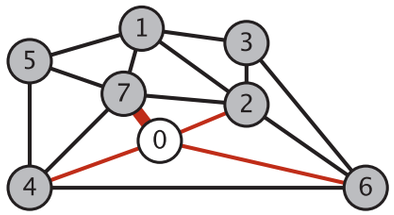


Prim's algorithm example

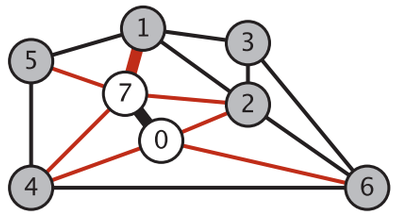
Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

Start with vertex 0 and greedily grow tree T. At each step,

add to T the min weight edge with exactly one endpoint in T.

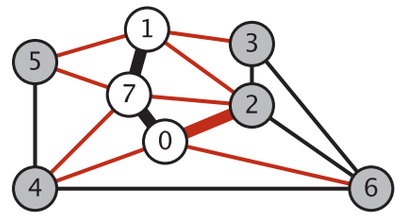


```
0-7 0.16
0-2 0.26
0-4 0.38
6-0 0.58
```

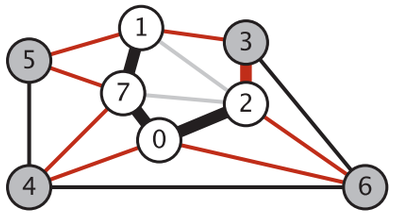


edges with exactly one endpoint in T (sorted by weight)

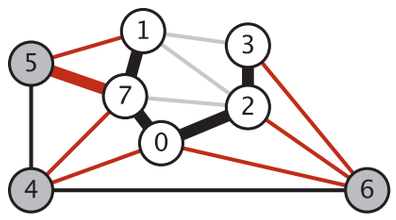
```
1-7 0.19
0-2 0.26
5-7 0.28
2-7 0.34
4-7 0.37
0-4 0.38
6-0 0.58
```



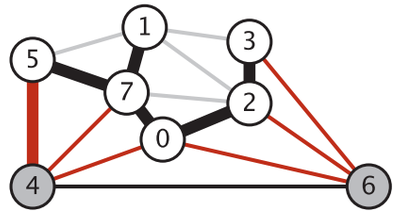
```
0-2 0.26
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
0-6 0.58
```



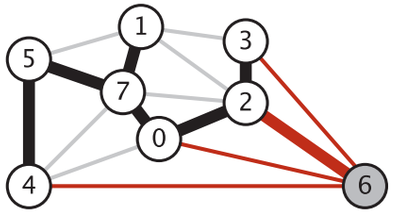
```
2-3 0.17
5-7 0.28
1-3 0.29
1-5 0.32
4-7 0.37
0-4 0.38
6-2 0.40
6-0 0.58
```



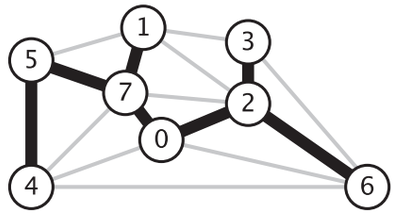
```
5-7 0.28
1-5 0.32
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58
```



```
4-5 0.35
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58
```



```
6-2 0.40
3-6 0.52
6-0 0.58
6-4 0.93
```

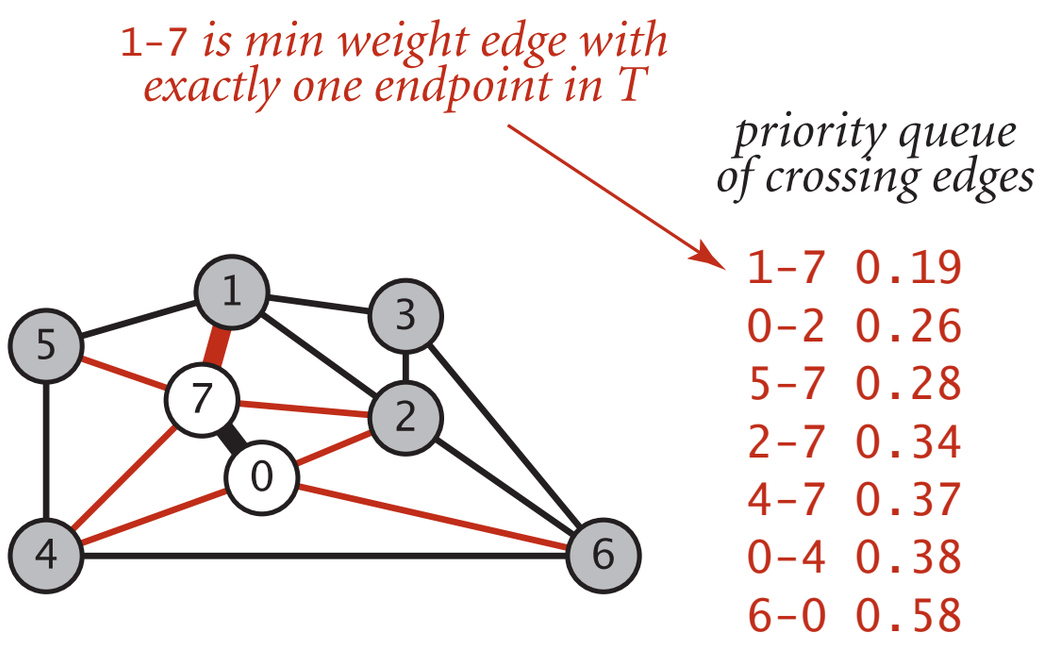


Prim's algorithm: lazy implementation

Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.

- Delete min to determine next edge $e = v$–w to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise, let v be vertex not in T:
  - add to PQ any edge incident to v (assuming other endpoint not in T)
  - add v to T



Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ in the worst case.

Pf.

operation	frequency	binary heap
delete min	E	log E
insert	E	log E

Graphs

- Undirected Graphs
- Directed Graphs
- Minimum Spanning Trees
- Shortest Paths

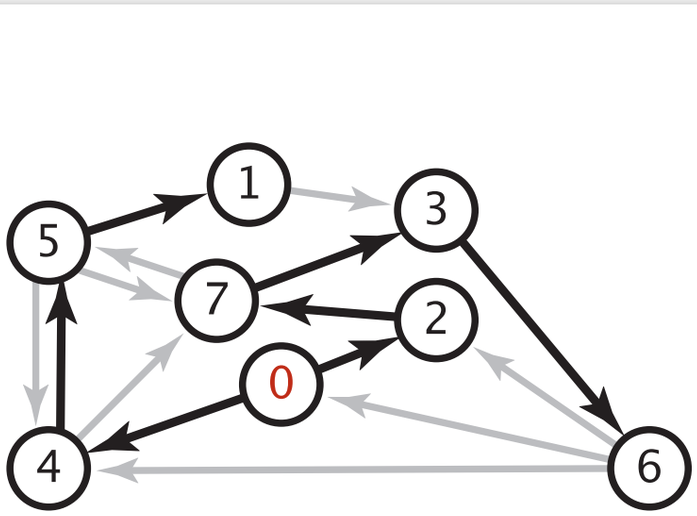
Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A shortest path tree (SPT) solution exists. Why?

Consequence. Can represent the SPT with two vertex-indexed arrays:

- **`distTo[v]`** is length of shortest path from s to v.
- **`edgeTo[v]`** is last edge on shortest path from s to v.



	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

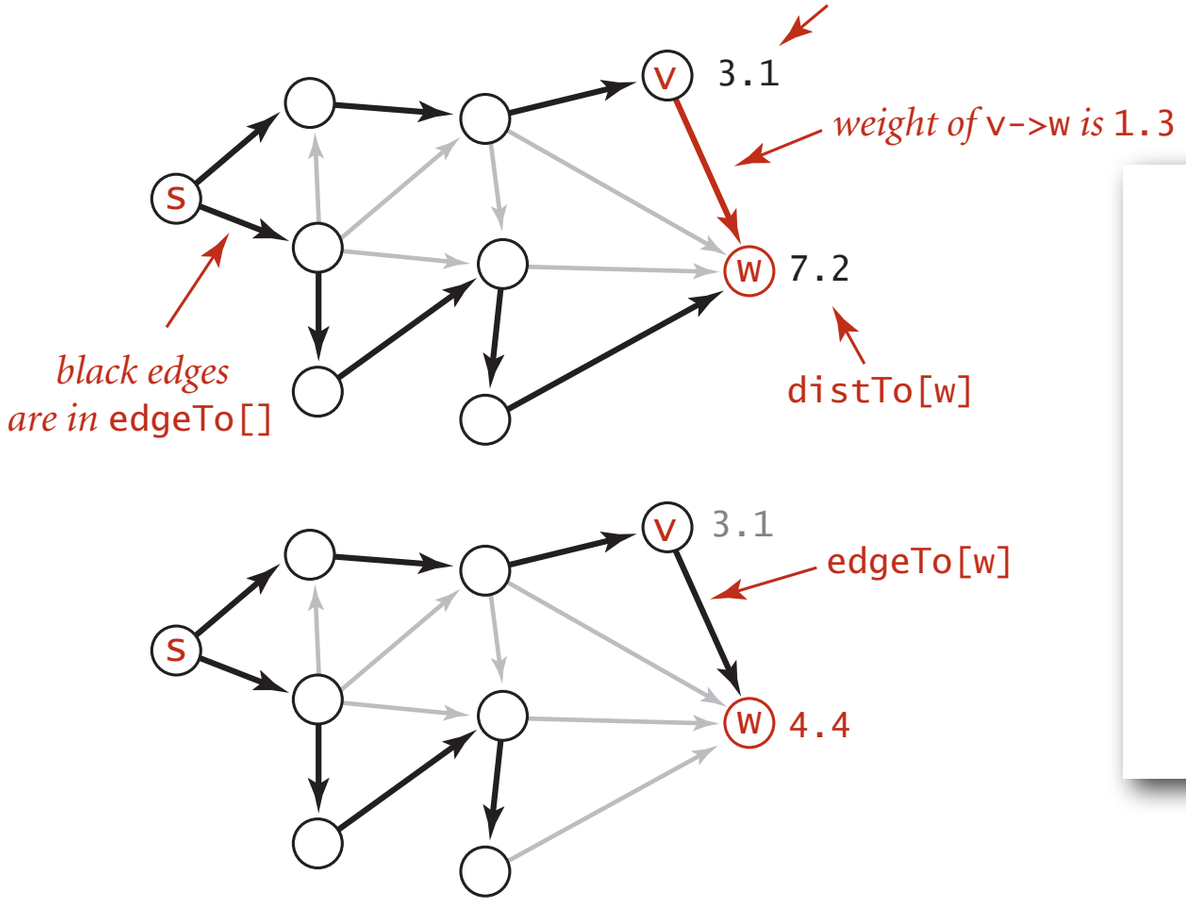
shortest path tree from 0

Edge relaxation

Relax edge $e = v \rightarrow w$.

- **`distTo[v]`** is length of shortest known path from s to v.
- **`distTo[w]`** is length of shortest known path from s to w.
- **`edgeTo[w]`** is last edge on shortest known path from s to w.
- If $e = v \rightarrow w$ gives shorter path to w through v, update **`distTo[w]`** and **`edgeTo[w]`**.

`v->w` successfully relaxes




```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
   }
}
```

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Efficient implementations. How to choose which edge to relax?

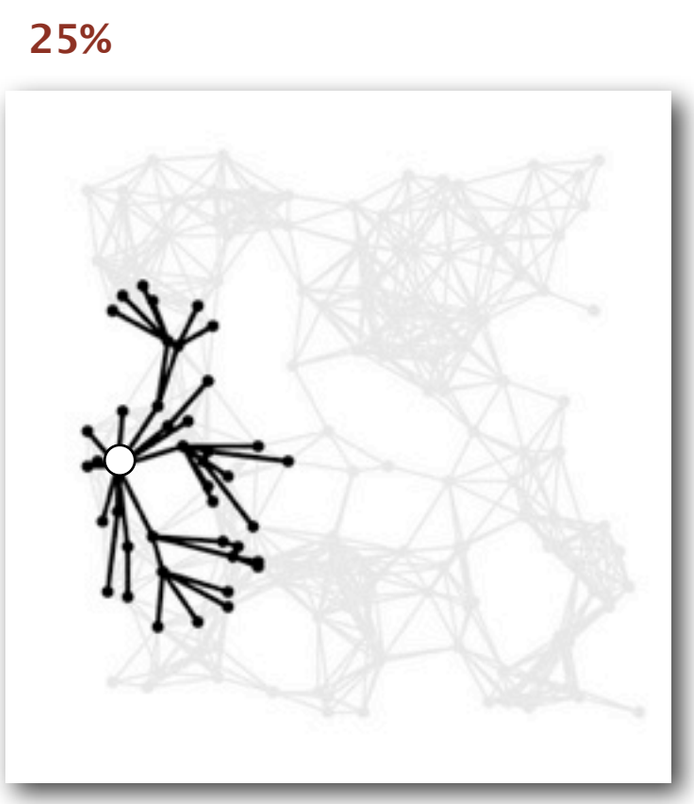
Ex 1. Dijkstra's algorithm (nonnegative weights).

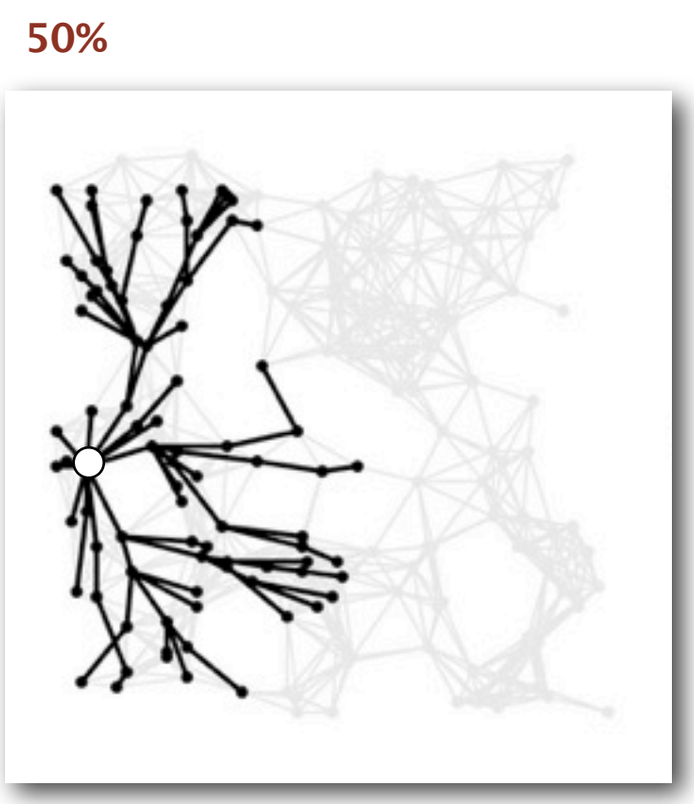
Ex 2. Topological sort algorithm (no directed cycles).

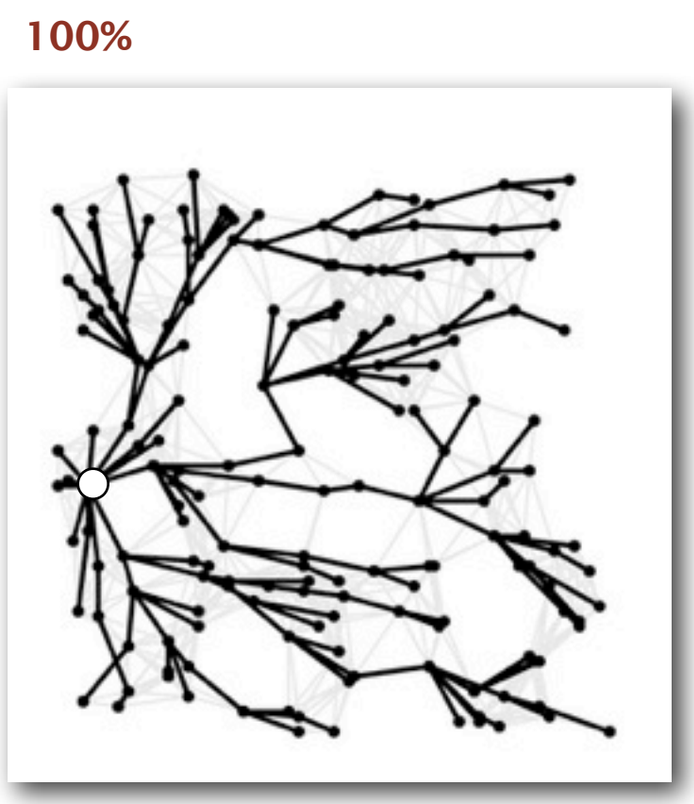
Ex 3. Bellman-Ford algorithm (no negative cycles).

Shortest path trees

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest **`distTo[]`** value).
- Add vertex to tree and relax all edges incident from that vertex.







Priority-first search

Insight. Four of our graph-search methods are the same algorithm!

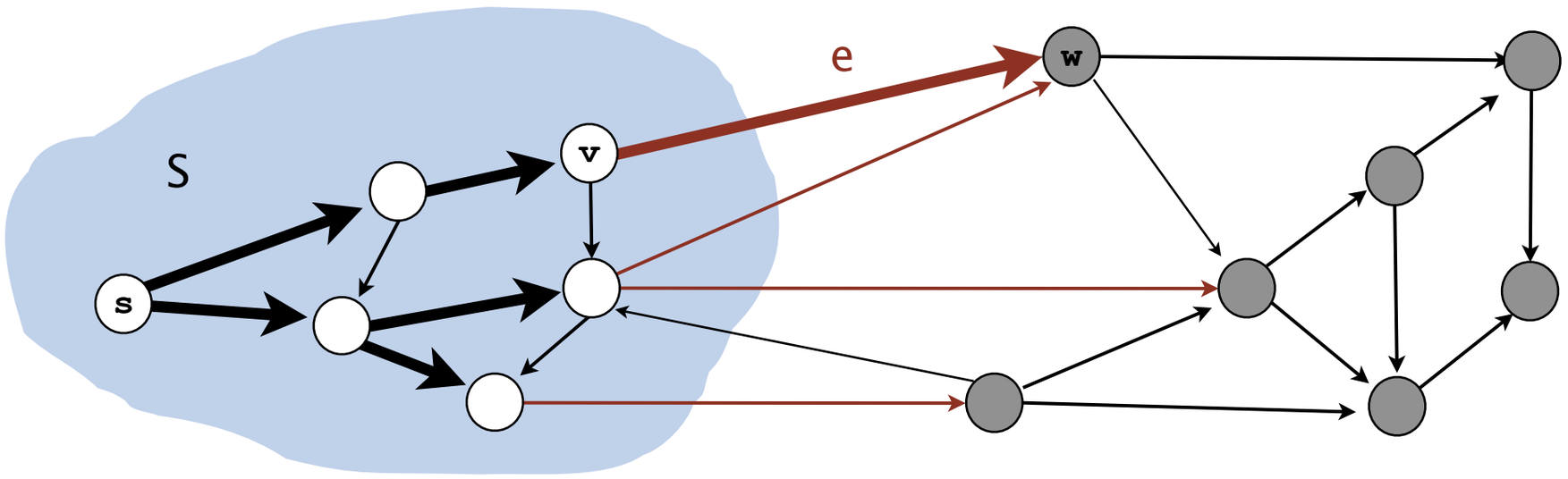
- Maintain a set of explored vertices S.
- Grow S by exploring edges with exactly one endpoint leaving S.

DFS. Take edge from vertex which was discovered most recently.

BFS. Take edge from vertex which was discovered least recently.

Prim. Take edge of minimum weight.

Dijkstra. Take edge to vertex that is closest to S.

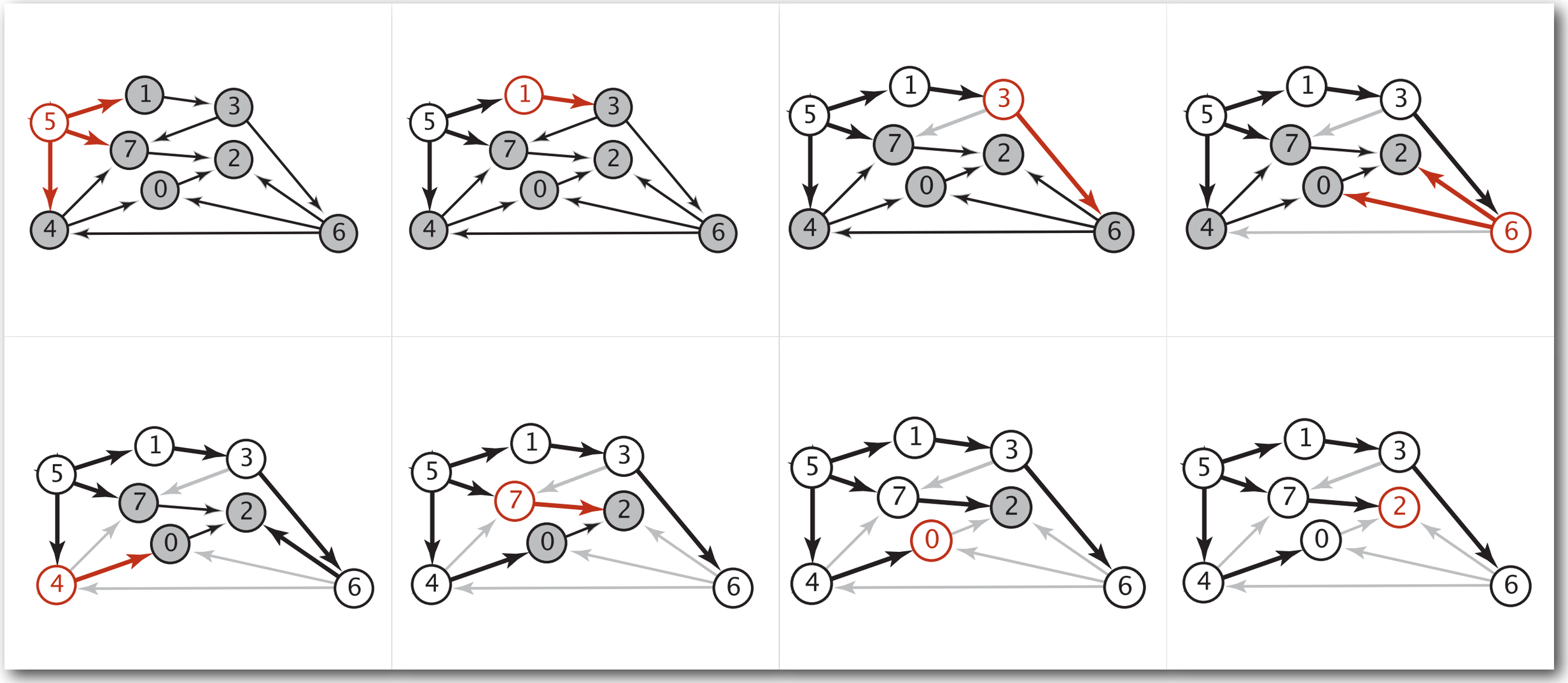


Challenge. Express this insight in reusable Java code.

Shortest paths in edge-weighted DAGs

Topological sort algorithm.

- Consider vertices in topologically order.
- Relax all edges incident from vertex.



topological order: 5 1 3 6 4 7 0 2

Bellman-Ford algorithm

Observation. If **`distTo[v]`** does not change during phase i,

no need to relax any edge incident from v in phase $i+1$.

FIFO implementation. Maintain queue of vertices whose **`distTo[]`** changed.

be careful to keep at most one copy of each vertex on queue (why?)

Overall effect.

- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

CS251 Final Review

- Trees
- Hash Tables
- Graphs
- Strings

Strings

- String Sorts
- Tries
- Compression

Key-indexed counting

Goal. Sort an array `a[]` of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move records.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

for (int r = 0; r < R; r++)
   count[r+1] += count[r];

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

count frequencies

compute cumulates

move records

copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

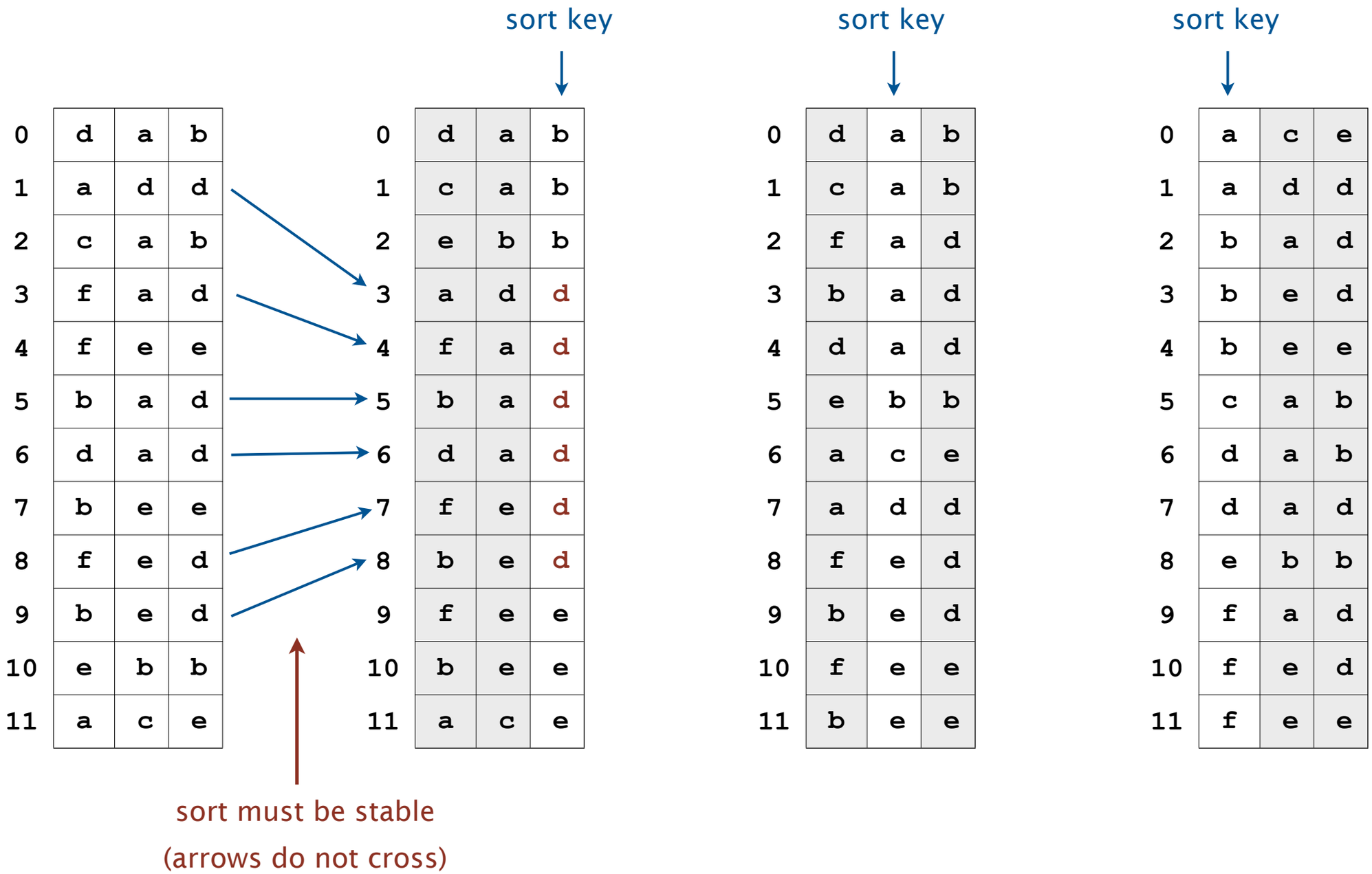
r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Least-significant-digit-first string sort

LSD string sort.

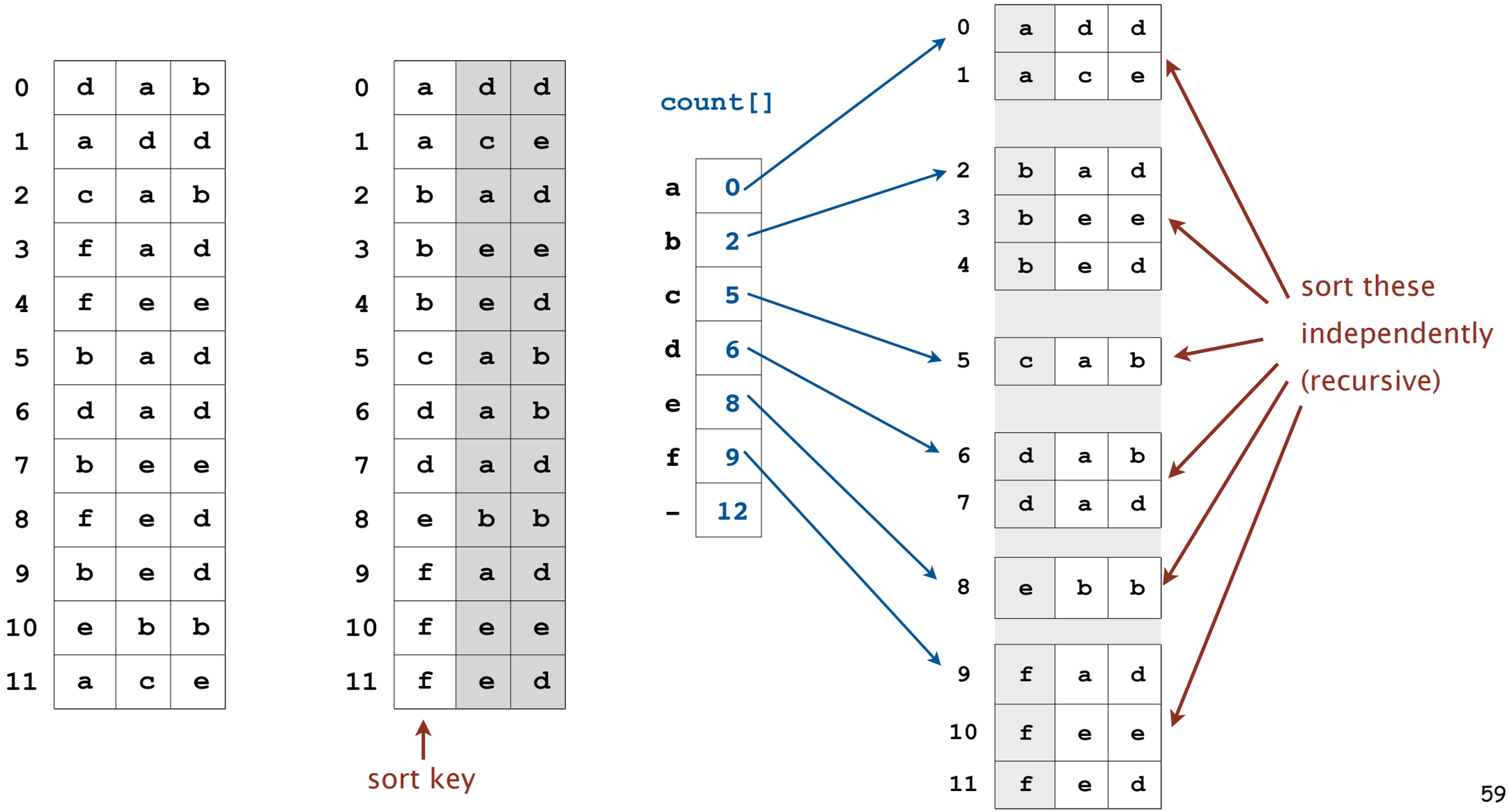
- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).



Most-significant-digit-first string sort

MSD string sort.

- Partition file into R pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).



Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	`compareTo()`
mergesort	$N \lg N$	$N \lg N$	N	yes	`compareTo()`
quicksort	$1.39\ N \lg N$ *	$1.39\ N \lg N$	$c \lg N$	no	`compareTo()`
heapsort	$2\ N \lg N$	$2\ N \lg N$	1	no	`compareTo()`
LSD †	$2\ N\ W$	$2\ N\ W$	$N + R$	yes	`charAt()`
MSD ‡	$2\ N\ W$	$N \log_R N$	$N + D\ R$	yes	`charAt()`

stack depth D = length of longest prefix match

* probabilistic

† fixed-length W keys

‡ average-length W keys

Strings

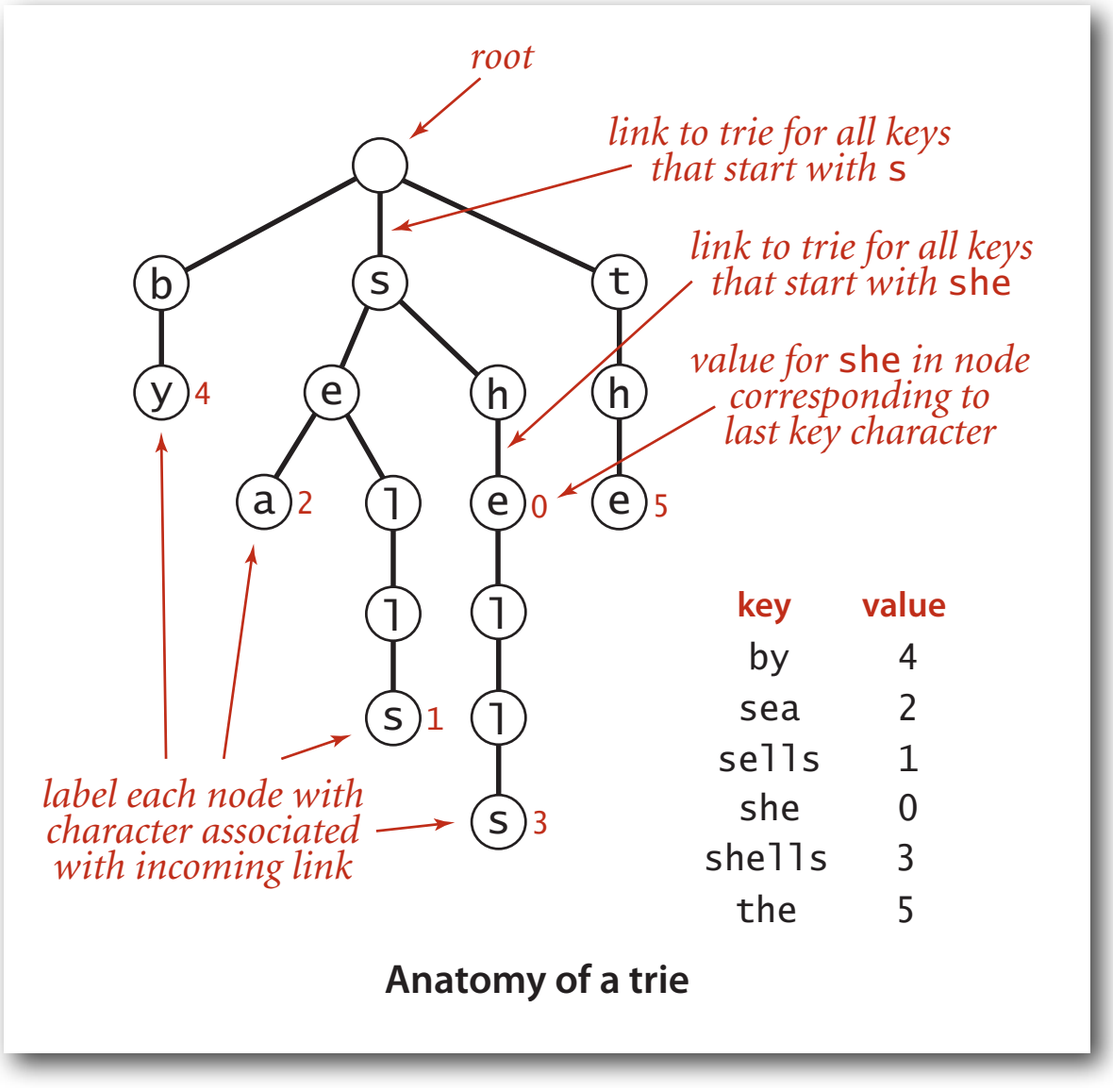
- String Sorts
- Tries
- Compression

Tries

Tries. [from retrieval, but pronounced "try"]

- Store characters and values in nodes (not keys).
- Each node has R children, one for each possible character.
- For now, we do not draw null links.

Ex. **she sells sea shells by the**



Anatomy of a trie

Trie performance

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

Search hit. Need to examine all L characters for equality.

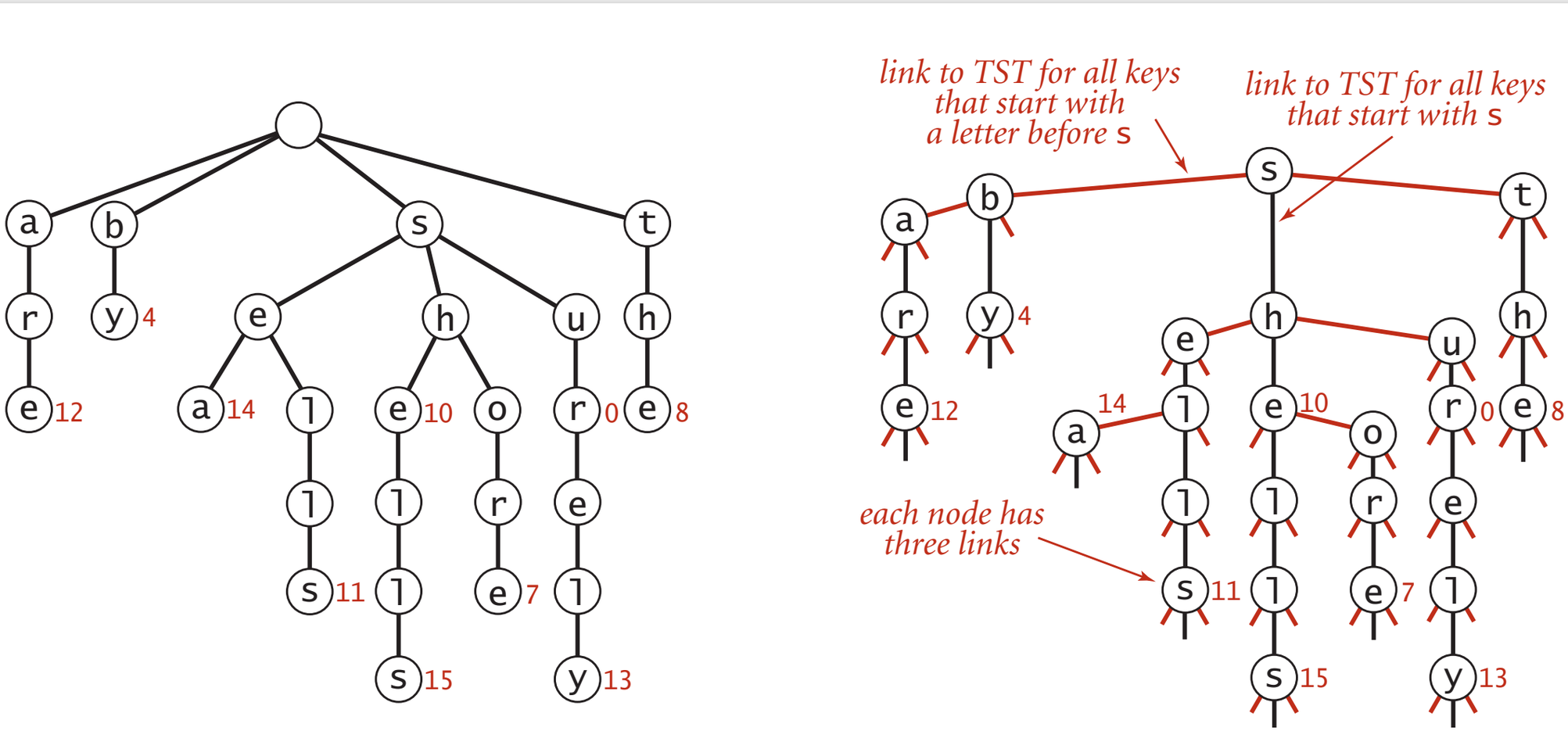
Space. R null links at each leaf.
(but sublinear space possible if many short strings share common prefixes)

Bottom line. Fast search hit and even faster search miss, but wastes space.

Ternary search tries

TST. [Bentley-Sedgewick, 1997]

- Store characters and values in nodes (not keys).
- Each node has three children: smaller (left), equal (middle), larger (right).



TST representation of a trie

String symbol table implementation cost summary

	character accesses (typical case)				dedup	
implementation	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	4 N	1.40	97.4
hashing	L	L	L	4 N to 16 N	0.76	40.6
R-way trie	L	$\log_R N$	L	(R + 1) N	1.12	out of memory
TST	L + ln N	ln N	L + ln N	4 N	0.72	38.7

Remark. Can build balanced TSTs via rotations to achieve $L + \log N$ worst-case guarantees.

Bottom line. TST is as fast as hashing (for string keys), space efficient.

TST vs. hashing

Hashing.

- Need to examine entire key.
- Search hits and misses cost about the same.
- Need good hash function for every key type.
- No help for ordered symbol table operations.

TSTs.

- Works only for strings (or digital keys).
- Only examines just enough key characters.
- Search miss may only involve a few characters.
- Can handle ordered symbol table operations (plus others!).

Bottom line. TSTs are:

- Faster than hashing (especially for search misses).
- More flexible than red-black trees (next).

Strings

- String Sorts
- Tries
- Compression

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
```

Representation. Use 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

```
1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1
```

15 7 7 11

← 16 bits (instead of 40)

Q. How many bits to store the counts?

A. We'll use 8.

Q. What to do when run length exceeds max count?

A. If longer than 255, intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

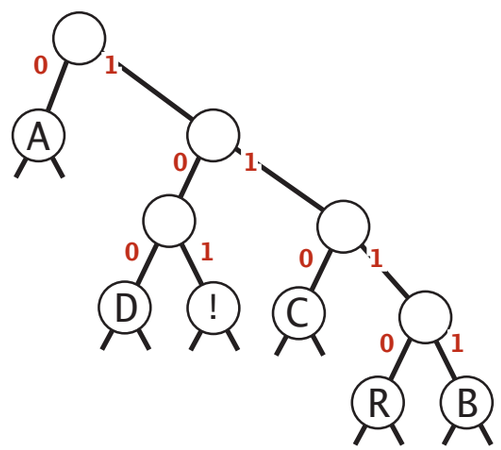
A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.

Codeword table

key	*value*
!	101
A	0
B	1111
C	110
D	100
R	1110

Trie representation



Compressed bitstring

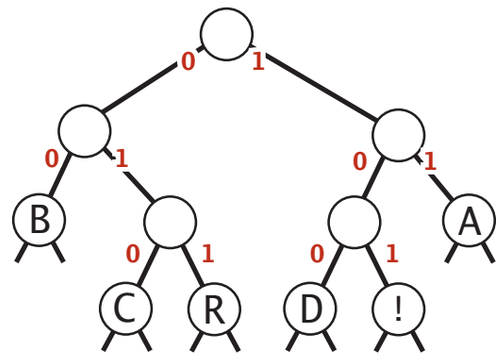
01111111001100100011111100101 ← *30 bits*

A B RA CA DA B RA !

Codeword table

key	*value*
!	101
A	11
B	00
C	010
D	100
R	011

Trie representation



Compressed bitstring

11000111101011001100011111101 ← *29 bits*

A B R A C A D A B R A !

Huffman codes

Q. How to find best prefix-free code?

David Huffman

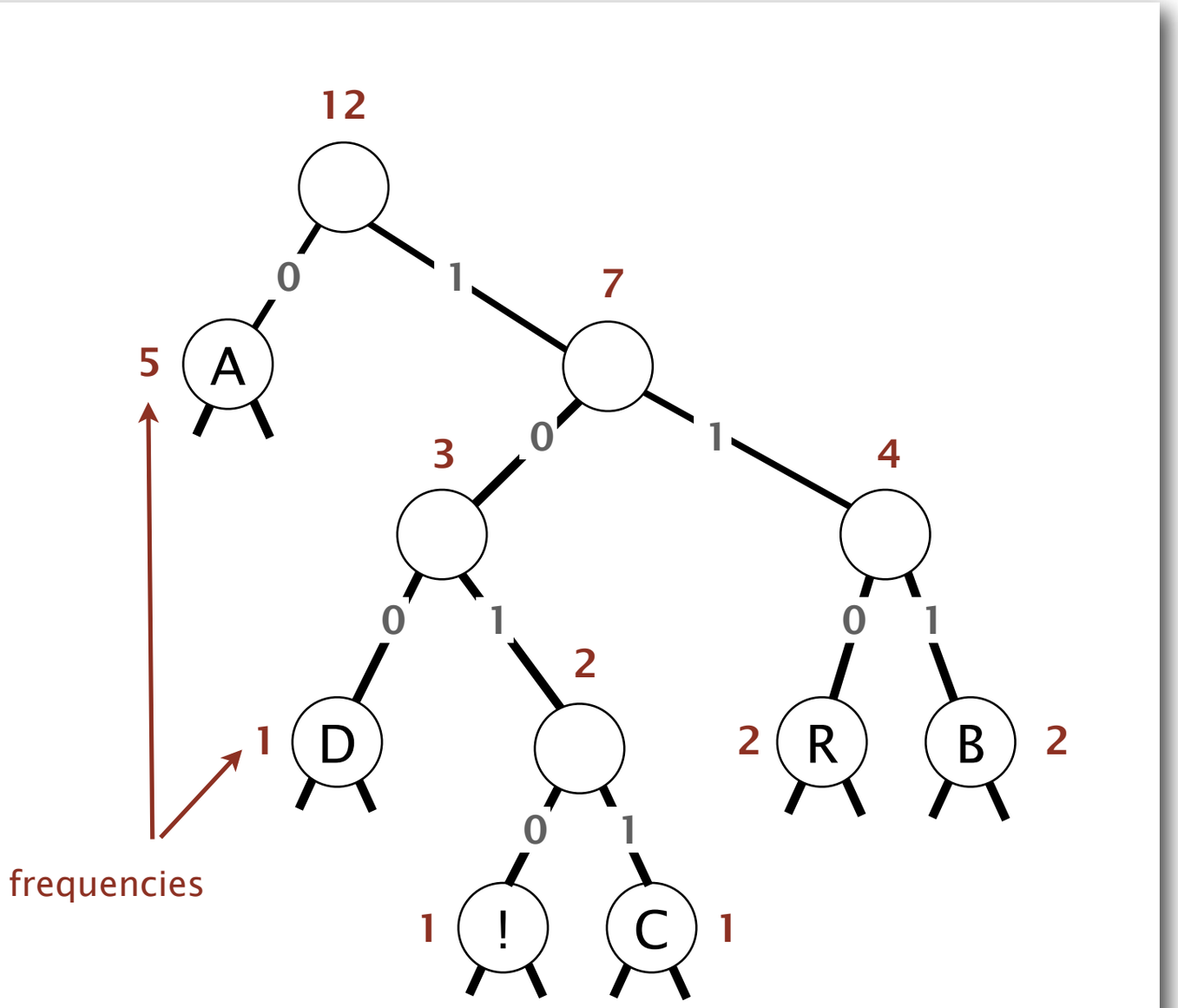
Huffman algorithm:

- Count frequency **`freq[i]`** for each char **`i`** in input.
- Start with one node corresponding to each char **`i`** (with weight **`freq[i]`**).
- Repeat until single trie formed:
  - select two tries with min weight **`freq[i]`** and **`freq[j]`**
  - merge into single trie with weight **`freq[i] + freq[j]`**

Applications. JPEG, MP3, MPEG, PKZIP, GZIP, PDF, . . .

Constructing a Huffman encoding trie

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Huffman code construction for A B R A C A D A B R A !