Shenime – Algorithms 4: Robert Sedgewick, Kevin Wayne

Shortest paths. An edge-weighted digraph is a digraph where we associate weights or costs with each edge. A *shortest path* from vertex s to vertex t is a directed path from s to t with the property that no other such path has a lower weight.

Properties.

- Paths are directed.
- The weights are not necessarily distances.
- Not all vertices need be reachable.
- Negative weights introduce complications
- Shortest paths are normally simple
- Shortest paths are not necessarily unique
- Parallel edges and self-loops may be present

Edge-weighted digraph data type. We represent the weighted edges using the following API.

public class	DirectedEdge	
	DirectedEdge(int v, int w, double weight)	
double	weight()	*weight of this edge*
int	from()	*vertex this edge points from*
int	to()	*vertex this edge points to*
String	toString()	*string representation*

The *from( )* and *to( )* methods are useful for accessing the edge's vertices.

We represent edge-weighted digraphs using the following API:

public class	EdgeWeightedDigraph	
	EdgeWeightedDigraph(int V)	*empty V-vertex digraph*
	EdgeWeightedDigraph(In in)	*construct from in*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(DirectedEdge e)	*add e to this digraph*
Iterable<DirectedEdge>	adj(int v)	*edges pointing from v*
Iterable<DirectedEdge>	edges()	*all edges in this digraph*
String	toString()	*string representation*

Shortest paths API. We use the following API for computing the shortest paths of an edge-weighted digraph:

public class	SP	
	SP(EdgeWeightedDigraph G, int s)	*constructor*
double	distTo(int v)	*distance from s to v, ∞ if no path*
boolean	hasPathTo(int v)	*path from s to v?*
Iterable<DirectedEdge>	pathTo(int v)	*path from s to v, null if none*

Given an edge-weighted digraph and a designated vertex s, a shortest-paths tree (SPT) is a subgraph containing s and all the vertices reachable from s that forms a directed tree rooted at s such that every tree path is a shortest path in the digraph.

We represent the shortest paths with two **vertex-indexed** arrays:

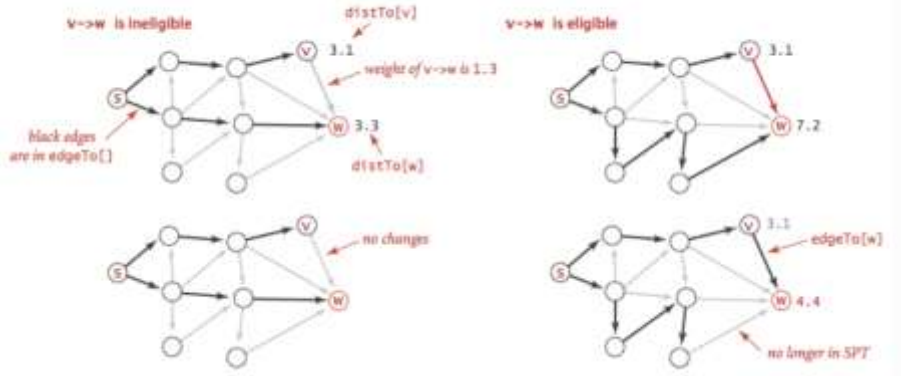
- Edges on the shortest-paths tree: edgeTo[v] is the last edge on a shortest path from s to v(mban objekte te tipit DirectedEdge);
- Distance to the source: distTo[v] is the length of the shortest path from s to v;

By convention, edgeTo[s] is *null* and distTo[s] is *0*.

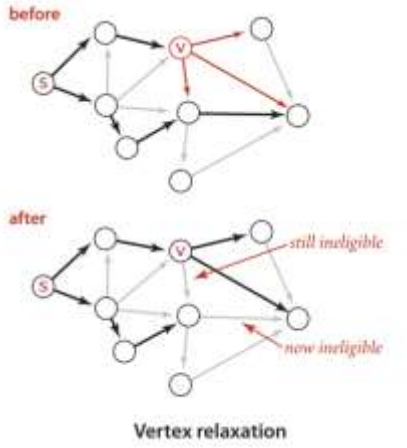
Relaxation. Our shortest-paths implementations are based on an operation known as relaxation. We initialize distTo[s] to 0 and distTo[v] to infinity for all other vertices v.

- *Edge relaxation*. To relax an edge v->w means to test whether the best known way from s to w is to go from s to v, then take the edge from v to w, if so, update out data structure. The best known distance to w through v is the sum of distTo[v] and e.weight( ) – if that value is not smaller than distTo[w], we say the edge is ineligible, and we ignore it.
  We say that an edge e can be successfully relaxed if relax( ) would change the values of distTo[e.to( )] and edgeTo[e.to( )]. (e.to mund te jete w).

```
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;
    }
}
```



- *Vertex relaxation*. All of our implementations actually relax all the edges pointing from a given vertex.



Vertex relaxation

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

Relaksimin kjo e sheh nga pikepamja e nyjeve dhe jo e lidhjeve. Psh kjo merr ne konsiderate lidhjet e nje nyje me nyjet e tjera, kurse ajo tjetra (relaksimi lidhjeve) merrte ne konsiderate nje lidhje.

Dijkstra's algorithm. Dijkstra's algorithm solves the single-source shortest path problem.

Problem: Single-Source Shortest Paths

Input: A directed graph $G = (V, E)$, a starting vertex $s \in V$, and a nonnegative *length* ℓ_e for each edge $e \in E$.

Output: $dist(s, v)$ for every vertex $v \in V$.

(Funksionon per grafet e drejtuar dhe te padrejtuar) Dijkstra's algorithm initializing dist[s] to 0 and all other distTo[] entries to positive infinity. Then, it repeatedly relaxes and adds to the tree a non-tree vertex with the lowest distTo[] value, continuing until all vertices are on the tree or no non-tree vertex has a finite distTo[] value.

Each iteration of its main loop processes one new vertex. Which vertex to process next? The not-yet processed vertex that appears to be closest to the starting vertex.

Pseudocode:

```
Dijkstra
```

Input: directed graph $G = (V, E)$ in adjacency-list representation, a vertex $s \in V$, a length $\ell_e \geq 0$ for each $e \in E$.

Postcondition: for every vertex v, the value $len(v)$ equals the true shortest-path distance $dist(s, v)$.

```
// Initialization
1 X := {s}
2 len(s) := 0, len(v) := +∞ for every v ≠ s
// Main loop
3 while there is an edge (v, w) with v ∈ X, w ∉ X do
4     (v*, w*) := such an edge minimizing len(v) + ℓ_vw
5     add w* to X
6     len(w*) := len(v*) + ℓ_v*w*
```

You can associate the Dijkstra score for an edge (v,w) with v from X and w not from X with the hypothesis that the shortest path from s to w consists of a shortest path from s to v (which hopefully has len(v)) with the edge (v,w) (which has length l_{vw}) tacked on at the end.

Theorem: Dijkstra Running Time (Heap-based). For every directed graph G = (V,E), every starting vertex s, and every choice of nonnegative edge lengths, the heap based implementation of Dijkstra runs in O((m+n)longn) time, where m=|E| and n=|V|.

The concrete plan is to store the as-yet-unprocessed vertices (V-X in the Dijkstra pseudocode) in a heap, while maintaining the following invariant. **Invariant**: The key of a vertex w from V-X is the minimum Dijkstra score of an edge with tail v from X and head w, or +infinity if no such edge exists. The point is, as long as we maintain our invariant, we can implement each iteration of Dijkstra's algorithm with a single heap operation.

Pseudocode:

Dijkstra (Heap-Based, Part 1)

Input: directed graph $G = (V, E)$ in adjacency-list representation, a vertex $s \in V$, a length $\ell_e \geq 0$ for each $e \in E$.
Postcondition: for every vertex v, the value $len(v)$ equals the true shortest-path distance $dist(s, v)$.

```
// Initialization
1  X := empty set, H := empty heap
2  key(s) := 0
3  for every v ≠ s do
4      key(v) := +∞
5  for every v ∈ V do
6      INSERT v into H                    // or use HEAPIFY
   // Main loop
7  while H is non-empty do
8      w* := EXTRACTMIN(H)
9      add w* to X
10     len(w*) := key(w*)
       // update heap to maintain invariant
11     (to be announced)
```

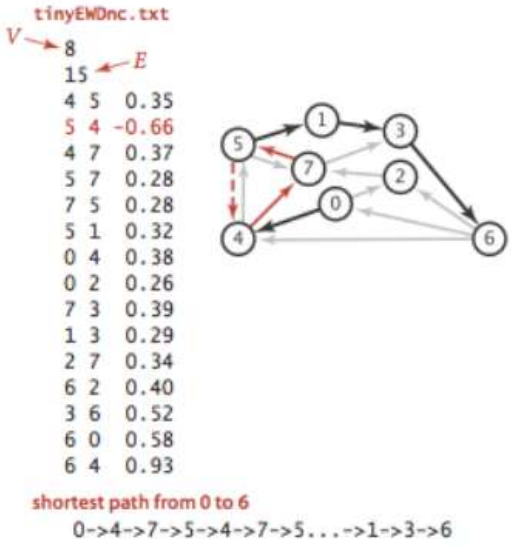
Dijkstra (Heap-Based, Part 2)

```
   // update heap to maintain invariant
12 for every edge (w*, y) do
13     DELETE y from H
14     key(y) := min{key(y), len(w*) + ℓ_{w*y}}
15     INSERT y into H
```

Proposition. Dijkstra's algorithm solves the single-source shortest-paths problem in edge-weighted digraphs with nonnegative weights using extra space proportional to V and time proportional to ElogV (in the worst case). Proportional to E and time proportional to ElogE (in the worst case).

Shortest paths in general edge-weighted digraphs. We can solve shortest path problems if all weights are nonnegative or there are no cycles.

- Negative cycles. A negative cycle is a directed cycle whose total weight (sum of the weights of its edges) is negative. The concept of a shortest path is meaningless if there is a negative cycle. Accordingly, we consider edge-weighted digraphs with no negative cycles.



- Bellman-Ford algorithm. Initialize distTo[s] to 0 and all other distTo[] values to infinity. Then, considering the digraph's edges in any order, and relax all edges. Make V such passes.

```
for (int pass = 0; pass < G.V(); pass++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
          relax(e);
```

We do not consider this version in detail because it always relaxes VE edges.
Queue-based Bellman-Ford algorithm. The only edges that could lead to a change in distTo[] are those leaving a vertex whose distTo[] value changed in the privous pass. To keep track of such vertices, we use a FIFO queue. Bellman-Ford implements this approach by maintaining two additional data structures: A queue of vertices to be relaxed; A vertex-index Boolean array onQ[] that indicates which vertices are on the queue, to avoid duplicates.
We need to ensure that the algorithm terminates after V passes.

- Negative cycle detection. In many applications, our goal is to check for and to check for and extract negative cycles. Accordingly, we add the following methods to the API:

`boolean hasNegativeCycle()`	*has a negative cycle?*
`Iterable<DirectedEdge> negativeCycle()`	*a negative cycle* (`null` *if no negative cycles*)

There is a negative cycle reachable from the source if and only if the queue is nonempty after the Vth pass through all the edges. Moreover, the subgraph of edges in our edgeTo[] array must contain a negative cycle. To implement negativeCycle(), BellmanFordSP.java builds an edge-weighted digraph from the edges in edgeTo[] and looks for a cycle in that

digraph. To find the cycle, it used EdgeWeightedDirectedCycle.java. We amortize the cost of this check by performing this check only after every Vth edge relaxation.

Minimum spanning tree. An edge-weighted graph is a graph where we associate weights or costs with each edge. A minimum spanning tree (MST) of an edge-weighted graph is a spanning tree whose weight is no larger than the weight of any other spanning tree.

Assumptions.

- The graph is connected.
- The edge weights are all different.

Edge-weighted graph data type. We represent the weighted edges using the following API:

public class Edge implements Comparable<Edge>		
	Edge(int v, int w, double weight)	*initializing constructor*
double	weight()	*weight of this edge*
int	either()	*either of this edge's vertices*
int	other(int v)	*the other vertex*
int	compareTo(Edge that)	*compare this edge to* e
String	toString()	*string representation*

The either( ) and other( ) methods are useful for accessing the edge's vertices, the compareTo( ) method compares edges by weight.

We represent edge-weighted graphs using the following API:

public class EdgeWeightedGraph		
	EdgeWeightedGraph(int V)	*create an empty* V*-vertex graph*
	EdgeWeightedGraph(In in)	*read graph from input stream*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(Edge e)	*add edge* e *to this graph*
Iterable<Edge>	adj(int v)	*edges incident to* v
Iterable<Edge>	edges()	*all of this graph's edges*
String	toString()	*string representation*

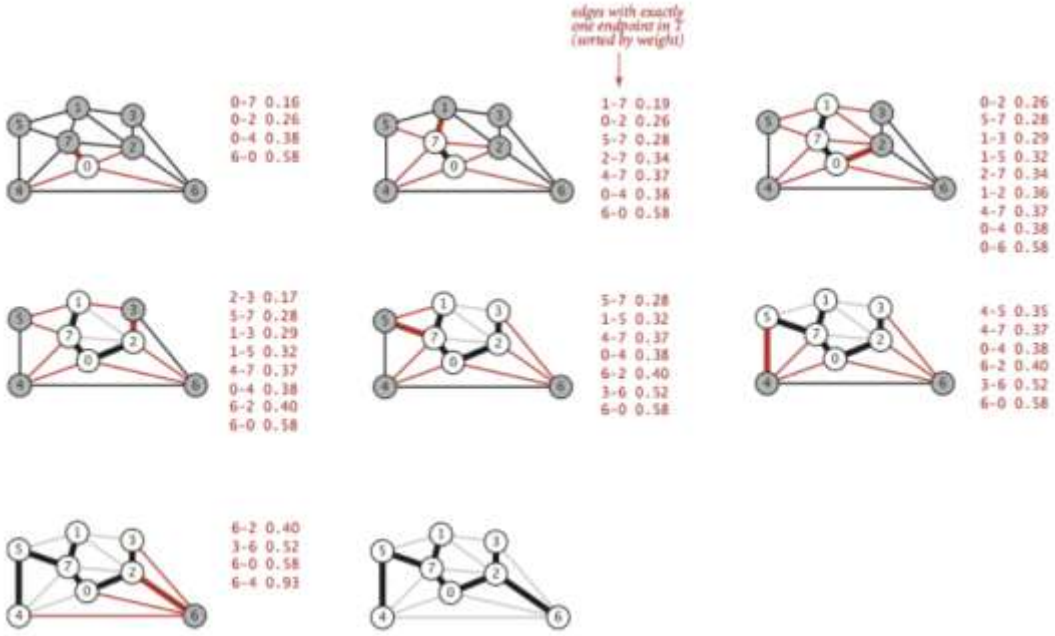
We use the following API for computing an MST of an edge-weighted graph:

public class	MST	
	MST(EdgeWeightedGraph G)	*constructor*
Iterable<Edge>	edges()	*iterator for MST edges*
double	weight()	*weight of MST*

API for MST implementations

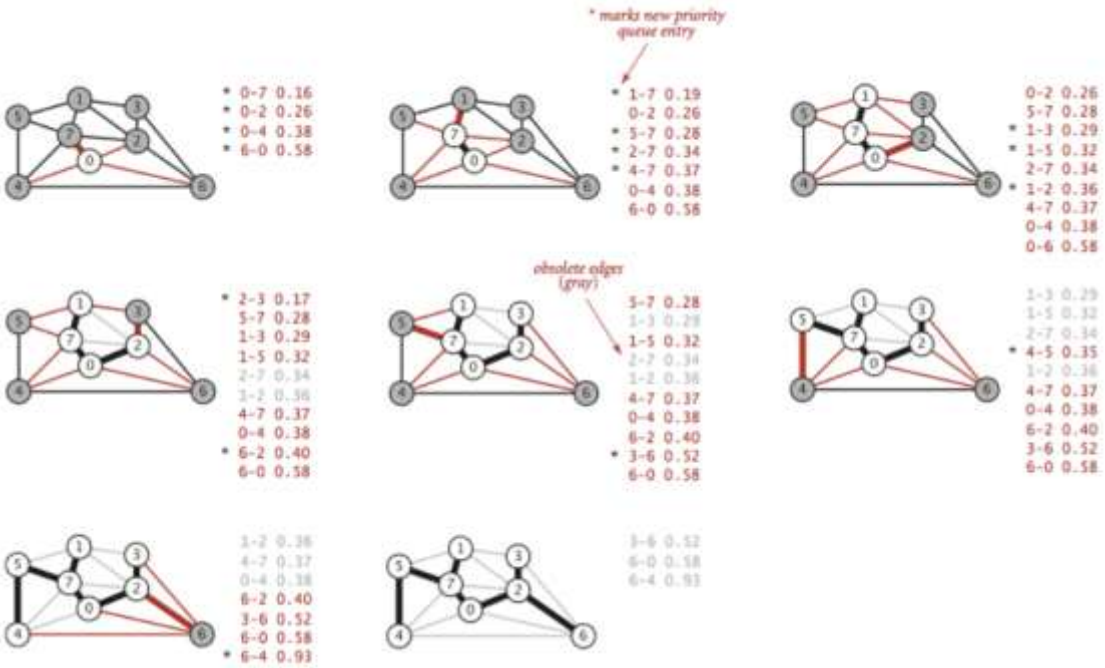
Prim's algorithm. Prim's algorithm works by attaching a new edge to a single growing tree at each step: Start with any vertex as a single-vertex tree; then add V-1 edges to it, always taking next the minimum-weight edge that connects a vertex on the tree to a vertex not yet on the tree.

- Lazy Implementation: We use a priority queue to hold the crossing edges and find one of minimal weight. Each time that we add an edge to the tree, we also add a vertex to the tree. To maintain the set of crossing edges, we need to add to the priority queue all edges from that vertex to any non-tree vertex. But we must do more: any edge connecting the vertex just added to a tree vertex that is already on the priority queue now becomes ineligible (it is no longer a crossing edge because it connects two tree vertices). The lazy implementation leaves such edges on the priority queue. Time Complexity: ElogE; Space Complexity: E



- Eager Implementation: To improve the lazy implementation of Prim's algorithm, we might try to delete ineligible edges from the priority queue, so that the priority queue contains only the crossing edges. But we can eliminate even more edges. When we add a vertex v to the tree, the only possible change with respect to each non-tree vertex w is that adding v brings w closer than before to the tree. In short, we do not need to keep on the priority queue all of the edges from w to vertices tree – we just need to keep track of the minimum-weight edge and check whether the addition of v to the tree necessitates that we update the minimum, which we can do as we process each edge in s adjacency

list. In other words, we maintain on the priority queue just one edge for each non-tree vertex: the shortest edge that connects it to the tree. Time Complexity: ElogV; Space Complexity: V



In general, Prim's algorithm grows a spanning tree from a starting vertex one edge at a time, with each iteration extending the reach of the tree-so-far by one additional vertex. As a greedy algorithm, the algorithm always chooses the cheapest edge that does the job.
Pseudocode:

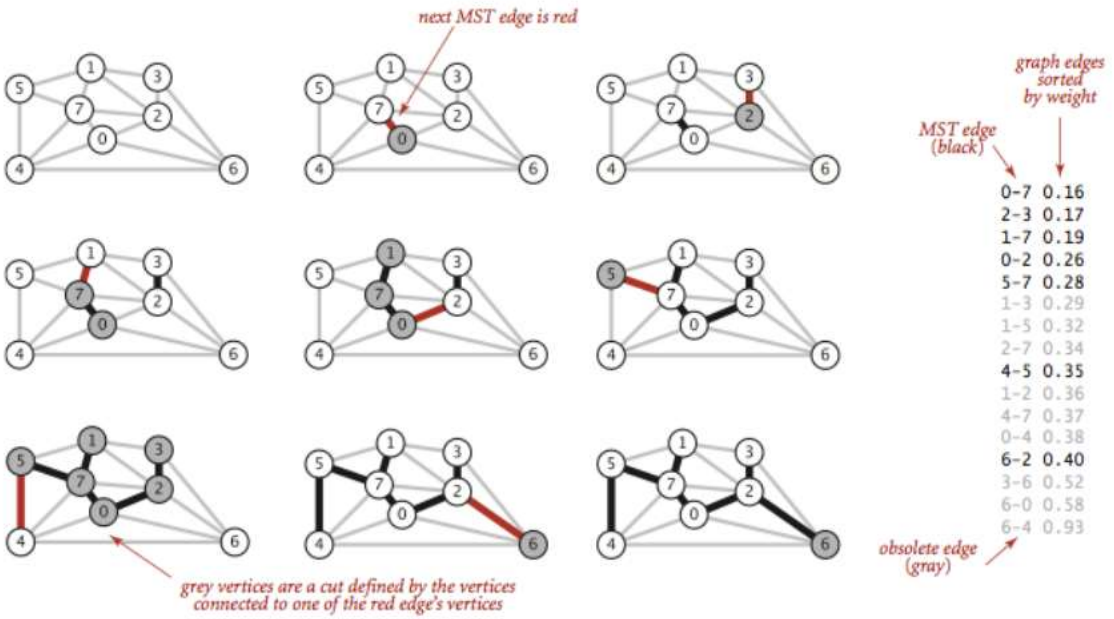
`Prim` (Heap-Based)

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.

Output: the edges of a minimum spanning tree of G.

```
   // Initialization
 1 X := {s}, T = ∅, H := empty heap
 2 for every v ≠ s do
 3     if there is an edge (s, v) ∈ E then
 4         key(v) := c_sv, winner(v) := (s, v)
 5     else      // v has no crossing incident edges
 6         key(v) := +∞, winner(v) := NULL
 7     INSERT v into H
   // Main loop
 8 while H is non-empty do
 9     w* := EXTRACTMIN(H)
10     add w* to X
11     add winner(w*) to T
       // update keys to maintain invariant
12     for every edge (w*, y) with y ∈ V − X do
13         if c_w*y < key(y) then
14             DELETE y from H
15             key(y) := c_w*y, winner(y) := (w*, y)
16             INSERT y into H
17 return T
```

Kruskal's algorithm. Kruskal's algorithm processes the edges in order of their weight values taking for the MST each edge that does not form a cycle with edges previously added, stopping after adding V-1 edges. The black edges form a forest of trees that evolves gradually into a single tree, the MST.



To implement Kruskal's algorithm, we use a priority queue to consider the edges in order by weight, a union-find data structure to identify those that cause cycles, and a queue to collect the MST edges.

Kruskal's algorithm computes the MST of any connected edge-weighted graph with E edges and V vertices using extra space proportional to E and time proportional to ElogE.

Pseudocode.

Kruskal (Union-Find-Based)

Input: connected undirected graph $G = (V, E)$ in adjacency-list representation and a cost c_e for each edge $e \in E$.
Output: the edges of a minimum spanning tree of G.

```
// Initialization
T := ∅
U := INITIALIZE(V)   // union-find data structure
sort edges of E by cost   // e.g., using MergeSort
// Main loop
for each (v, w) ∈ E, in nondecreasing order of cost do
    if FIND(U, v) ≠ FIND(U, w) then
        // no v-w path in T, so OK to add (v, w)
        T := T ∪ {(v, w)}
        // update due to component fusion
        UNION(U, v, w)
return T
```

Initializing the union-find data structure takes O(n) time. The sorting step requires O(mlongn) time. There are m iterations of the main loop and each uses two Find operations. There is one Union operation for each edge added to the output which, as an acyclic graph, has at most n-1 edges. Provided the Find and Union operations run in O(logn) time, the total running time is:

	preprocessing	$O(n) + O(m \log n)$
	$2m$ FIND operations	$O(m \log n)$
	$n - 1$ UNION operations	$O(n \log n)$
+	remaining bookkeeping	$O(m)$
	total	$O((m + n) \log n)$.

Algoritmi Prim	**Algoritmi Kruskal**
Pema qe po krijohet qendron gjithmone e lidhur.	Pema qe po krijohet zakonisht qendron e palidhur.
Zgjidhet një nyje (vertex) random, dhe vazhdohet duke zgjedhur nyjen e rradhës me peshë më të vogël.	Zgjidhet lidhja (edge) me peshën më të vogël e me rradhë.
Algoritmi i Prim është më i shpejtë në dense graphs (grafet ku numri i edge është afër numrit maksimal të mundshëm).	Algoritmi Kruskal është më i shpejtë për grafet sparse (numri i edge është afër minimumit të mundshëm).

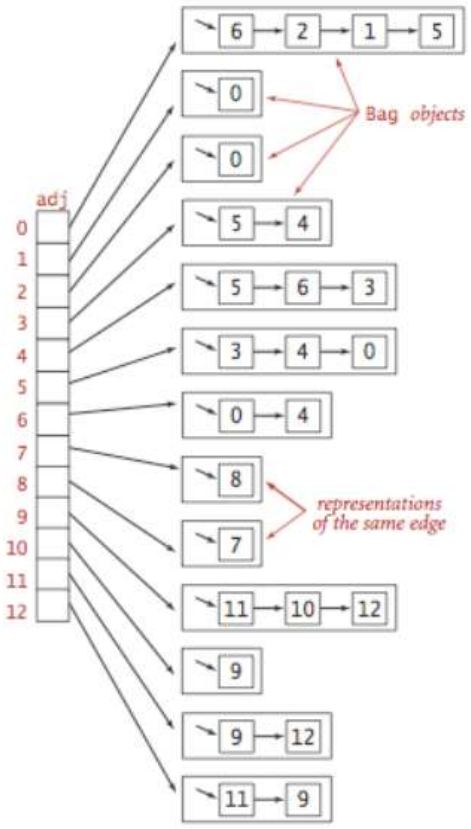
Graphs. A graph is a set of vertices and a collection of edges that each connect a pair of vertices. We use the names 0 through V-1 for the vertices in a V-vertex graph.

- **Self-loop**: An edge that connects a vertex to itself
- **Parallel edges**: two edges are parallel if they connect the same pair of vertices
- When an edge connects two vertices, we say that the vertices are **adjacent** to one another and that the edge is incident on both vertices.
- The **degree** of a vertex is the number of edges incident on it
- **Subgraph**: a subset of a graph's edges (and associated vertices) that constitutes a graph
- **Path**: a sequence of vertices connected by edges, with no repeated edges
- **Simple path**: a path with no repeated vertices
- **Cycle**: a path whose first and last vertices are the same
- **Simple cycle**: A cycle with no repeated vertices
- The **length** of a path or a cycle is its number of edges
- We say that one vertex is connected to another if there exists a path that contains both of them
- A graph is **connected** if there is a path from every vertex to every other vertex
- A graph that is not connected consists of a set of **connected components**, which are maximal connected subgraphs
- An **acyclic** graph is a graph with no cycles
- A **tree** is an acyclic connected graph
- A **forest** is a disjoint set of trees
- A **spanning tree**: a subgraph that contains all of that graph's vertices and is a single tree. A **spanning forest** of a graph is the union of the spanning tree of its connected components.
- A **bipartite** graph is a graph whose vertices we can divide into two sets such that all edges connect a vertex in one set with a vertex in the other set.

public class	Graph	
	Graph(int V)	*create a V-vertex graph with no edges*
	Graph(In in)	*read a graph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v-w *to this graph*
Iterable<Integer>	adj(int v)	*vertices adjacent to* v
String	toString()	*string representation*

API for an undirected graph

Graph representation. We use the adjacency-lists representation, where we maintain a vertex-indexed array of lists of the vertices connected by an edge to each vertex.



Adjacency-lists representation (undirected graph)

There are two graph search strategies - breadth-first search and depth-first search. The generic algorithm systematically finds all the reachable vertices, taking care to avoid exploring anything twice. It maintains an extra variable with each vertex that keeps track of whether or not it has already been explored, planting a flag the first time that vertex is reached. The main loop's responsibility is to reach a new unexplored vertex in each direction.

`GenericSearch`

Input: graph $G = (V, E)$ and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

---

mark s as explored, all other vertices as unexplored
while there is an edge $(v, w) \in E$ with v explored and w unexplored **do**
  choose some such edge (v, w) `// underspecified`
  mark w as explored

Breadth-first search and depth-first search correspond to two specific decisions about which edge to explore next. No matter how this choice is made, the GenericSearch algorithm is guaranteed to be correct.

It should be possible to implement the algorithm in linear time, as long as we can quickly identify an eligible edge (v,w) in each iteration of the while loop.

Breadth-first search (BFS). The high-level idea of bfs is to explore the vertices of a graph in "layers". Layer 0 consists only of the starting vertex s. Layer 1 contains the vertices that neighbor s, meaning the vertices v such that (s,v) is an edge of the graph. Layer 2 comprises the neighbors of layer1 vertices that do not already belong to layer0 or 1, and so on.

To find a shortest path from s to v, we start at s and check for v among all the vertices that we can reach by following one edge, then we check for v among all the vertices that we can reach from s by following two edges, and so forth.

To implement this strategy, we maintain a queue of all vertices that have been marked but whose adjacency lists have not been checked. We put the source vertex on the queue, then perform the following steps until the queue is empty:

- Remove the next vertex v from the queue.
- Put onto the queue all unmarked vertices that are adjacent to v and mark them

Pseudocode:

BFS

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

```
1 mark s as explored, all other vertices as unexplored
2 Q := a queue data structure, initialized with s
3 while Q is not empty do
4     remove the vertex from the front of Q, call it v
5     for each edge (v, w) in v's adjacency list do
6         if w is unexplored then
7             mark w as explored
8             add w to the end of Q
```

Implementing bfs in linear time requires a simple FIFO data structure known as a queue. BFS uses a queue to keep track of which vertices to explore next. Each iteration of the while loop explores one new vertex. In line 5, bfs iterates through all the edges incident to the vertex v or through all the outgoing edges from v. Unexplored neighbors of v are added to the end of the

queue and are marked as explored; they will eventually be processed in later iterations of the algorithm.

BFS discovers all the vertices reachable from the starting vertex, and it runs in linear time. The running time of BFS is O(m+n), where m=|E| and n=|V|.

What is unique about bfs is that, with just a couple extra lines of code, it efficiently computes shortest-path distances.

Depth-first search (DFS). DFS employs a more aggressive strategy for exploring a graph, very much in the spirit of how you might explore a maze, going as deeply as you can and backtracking only when absolutely necessary.

A classic recursive method for systematically examining each of the vertices and edges in a graph. To visit a vertex

- Mark it as having been visited
- Visit recursively all the vertices that are adjacent to it and that have not yet been marked

public class	Search	
	Search(Graph G, int s)	*find vertices connected to a source vertex s*
boolean	marked(int v)	*is v connected to s?*
int	count()	*how many vertices are connected to s?*

It is easy to modify depth-first search to not only determine whether there exists a path between two given vertices but to find such a path. We seek to implement the following API:

public class	Paths	
	Paths(Graph G, int s)	*find paths in G from source s*
boolean	hasPathTo(int v)	*is there a path from s to v?*
Iterable<Integer>	pathTo(int v)	*path from s to v; null if no such path*

To accomplish this, we remember the edge v-w that takes us to each vertex w for the first time by setting edgeTo[w] to v. In other words, v-w is the last edge on the known path from s to w. The result of the search is a tree rooted at the source; edgeTo[ ] is a parent-link representation of that tree.

One way to think about and implement DFS is to start from the code for BFS and make two changes: swap in a stack data structure and postpone checking whether a vertex has already been explored until after removing it from the data structure.

Pseudocode:

DFS (Iterative Version)

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

```
mark all vertices as unexplored
S := a stack data structure, initialized with s
while S is not empty do
    remove ("pop") the vertex v from the front of S
    if v is unexplored then
        mark v as explored
        for each edge (v, w) in v's adjacency list do
            add ("push") w to the front of S
```

As usual, the edges processed in the for loop are the edges incident to v or the edges outgoing from v.

DFS also has a recursive implementation:

DFS (Recursive Version)

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.
Postcondition: a vertex is reachable from s if and only if it is marked as "explored."

```
// all vertices unexplored before outer call
mark s as explored
for each edge (s, v) in s's adjacency list do
    if v is unexplored then
        DFS (G, v)
```

The aggressive nature of DFS is perhaps more obvious in this implementation – the algorithm immediately recurses on the first unexplored neighbor that it finds, before considering the remaining neighbors. In effect, the explicit stack data structure in the iterative implementation of DFS is being simulated by the program stack of recursive calls in the recursive implementation.

Connected components. A direct application of dfs is to find the connected components of a graph. “Is connected to” is an equivalence relation that divides the vertices into equivalence classes (connected components). For this task, we define the following API:

public class	CC	
	CC(Graph G)	*preprocessing constructor*
boolean	connected(int v, int w)	*are* v *and* w *connected?*
int	count()	*number of connected components*
int	id(int v)	*component identifier for* v (*between* 0 *and* count()-1)

DFS marks all the vertices connected to a given source in time proportional to the sum of their degrees and provides clients with a path from given source to any marked vertex in time proportional to its length.

Digraphs. A directed graph (or digraph) is a set of vertices and a collection of directed edges that each connects an ordered pair of vertices. We say that a directed edge points from the first vertex in the pair and points to the second vertex in the pair. We use the names 0 through V-1 for the vertices in a V-vertex graph.

Digraph data type:

public class	Digraph	
	Digraph(int V)	*create a* V*-vertex digraph with no edges*
	Digraph(In in)	*read a digraph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v->w *to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to* v *by edges pointing from* v
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

Graph representation. We use the adjacency-lists representation, where we maintain a vertex-indexed array of lists of the vertices connected by an edge to each vertex.

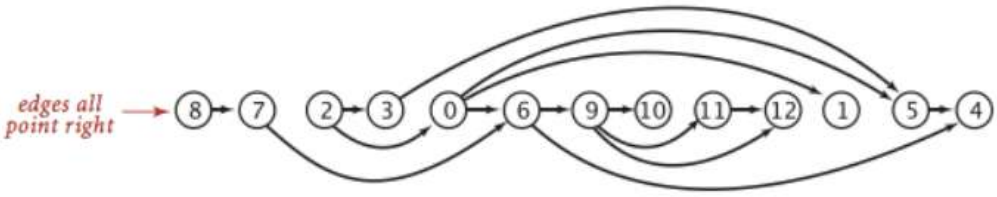
DFS and BFS are fundamentally digraph-processing algorithms.

Cycles and DAGs. Directed cycles are of particular importance in applications that involve processing digraphs.

DFS visits each vertex exactly once. Three vertex orderings are of interest in typical applications:

- Preorder: Put the vertex on a queue before the recursive calls.
- Postorder: Put the vertex on a queue after the recursive calls.
- Reverse postorder: Put the vertex on a stack after the recursive calls.

Topological sort. Given a digraph, put the vertices in order such that all its directed edges point from a vertex earlier in the order to a vertex later in the order. This problem is solved using dfs. Remarkably, a reverse postorder in a DAG provides a topological order.



A digraph has a topological order if and only if it is a DAG.

Reverse postorder in a DAG is a topological sort.

With dfs, we can topologically sort a DAG in time proportional to V+E.

Strong connectivity. Strong connectivity is an equivalence relation on the set of vertices:

- Reflexive. Every vertex is strongly connected to itself.
- Symmetric. If v is strongly connected to w, then w is strongly connected to v.
- Transitive. If v is strongly connected to w and w is strongly connected to x, then v is also strongly connected to x.

Strong connectivity partitions the vertices into equivalence classes, which we refer to as strong components for short.

KosarajuSharir implements as follows:

- Given a digraph G, use dfs to compute the reverse postorder of its reverse, G^R.
- Run standard dfs on G, but consider the unmarked vertices in the order just computed instead of the standard numerical order.
- All vertices reached on a call to the recursive dfs( ) from the constructor are in a strong component, so identify them as in CC.

The Kosaraju-Sharir algorithm uses preprocessing time and space proportional to V+E to support constant-time strong connectivity queries in a digraph.

The transitive closure of a digraph G is another digraph with the same set of vertices, but with an edge from v to w if and only if w is reachable form v in G.

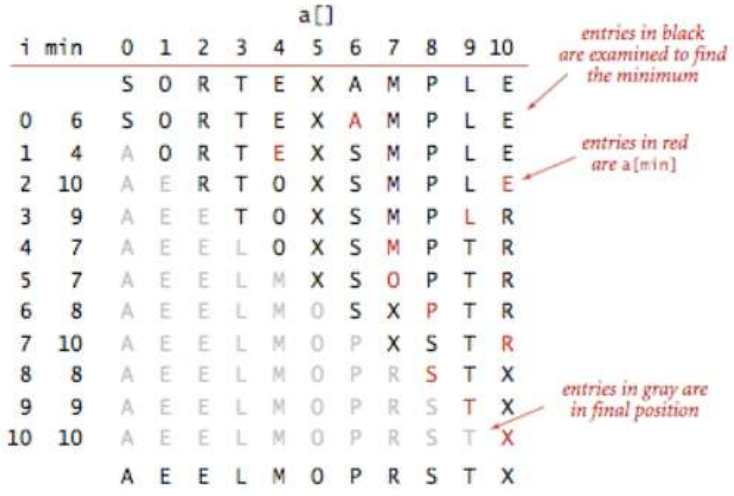
Elementary Sorts.

Our primary concern is algorithms for rearranging arrays of items where each item contains a key. The objective is to rearrange the items such that their keys are in ascending order. With but a few exceptions, our sort code refers to the data only through two operations: the method less( ) that compares objects and the method exch( ) that exchanges them.

- Sorting cost model. When studying sorting algorithms,we count **compares** and **exchanges**. For algorithms that do not use exchanges, we count array accesses.
- Extra memory. The sorting algorithms we consider divide into two basic types: those that sort **in** place (no extra memory except perhaps for a small function-call stack or a constant number of instance variables), and those that need enough **extra** memory to hold another copy of the array to be sorted.

Selection sort. One of the simplest sorting algorithms works as follows: First, find the smallest item and exchange it with the second entry. Continue in this way until the entire array is sorted. This method is called selection sort because it works by repeatedly selecting the smallest remaining item.

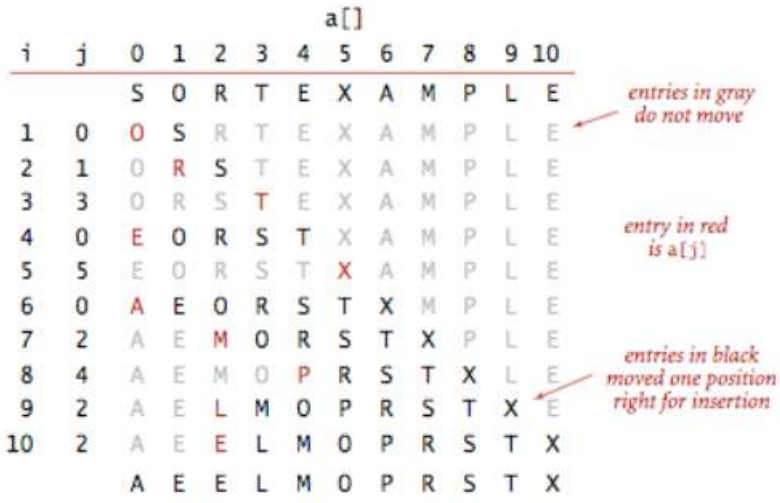
Selection sort uses $n^2/2$ compares and n exchanges to sort an array of length n.



Trace of selection sort (array contents just after each exchange)

Space complexity – O(1).

Insertion sort. Consider the cards one at a time, inserting each into its proper place among those already considered. We need to make space for the current item by moving larger items one position to the right, before inserting the current item into the vacated position.



Trace of insertion sort (array contents just after each insertion)

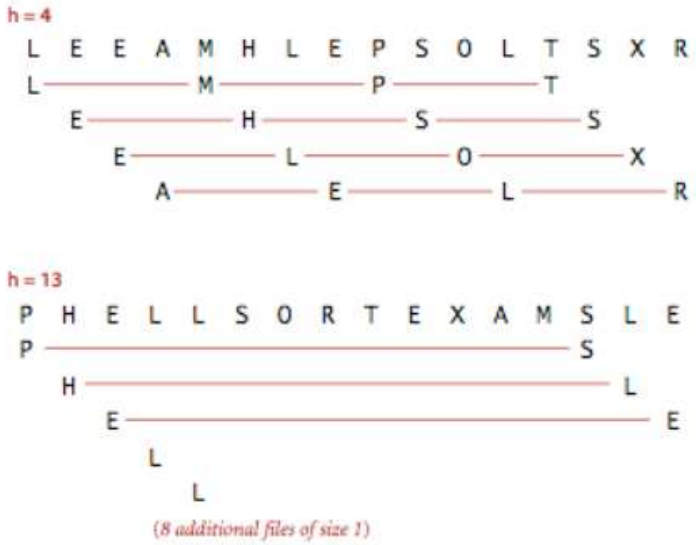
The worst case of insertion sort is $N^2/2$ compares and $N^2/2$ exchanges and the best case is N-1 compares and 0 exchanges.

Insertion sort works well for certain types of nonrandom arrays that often arise in practice, even if they are huge. If the number of inversions in an array is less than a constant multiple of the array size, we say that the array is partially sorted.

For randomly ordered arrays of distinct values, the running times of insertion sort and selection sort are quadratic and within a small constant factor of one another.

Space complexity – O(1).

Shellsort. A simple extension of insertion sort that gains speed by allowing exchanges of entries that are far apart, to produce partially sorted arrays that can be efficiently sorted, eventually by insertion sort. The idea is to rearrange the array to give it the property that taking the hth entry yields a sorted sequence. Such an array is said to be h-sorted.



An h-sorted file is h interleaved sorted files

By h-sorting for some large values of h, we can move entries in the array long distances and thus make it easier to h-sort for smaller values of h. Using such a procedure for any increment sequence of values of h that ends in 1 will produce a sorted array: that is shellsort.

The number of compares used by shellsort is bounded by a small multiple of N times the number of increments used.

Time complexity: Varet nga menyra se si perzgjidhet largesia/hapi. Rasti me i mires: NlogN, rasti me i keq $N^{3/2}$

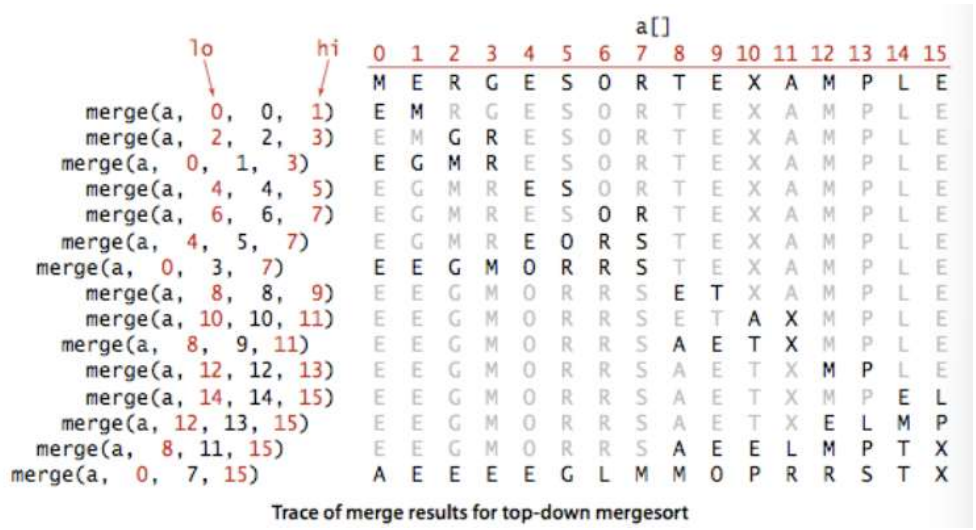
Space complexity: O(1)

Mergesort.

Merging – combining two ordered arrays to make one larger ordered array. This operation immediately lends itself to a simple recursive sort method known as mergesort: to sort an array, divide it into two halves, sort the two halves (recursively), and then merge the results.

Mergesort guarantees to sort an array of N items in time proportional to NlogN, no matter what the input. Its prime disadvantage is that it uses extra space proportional to N.

Top-down mergesort.

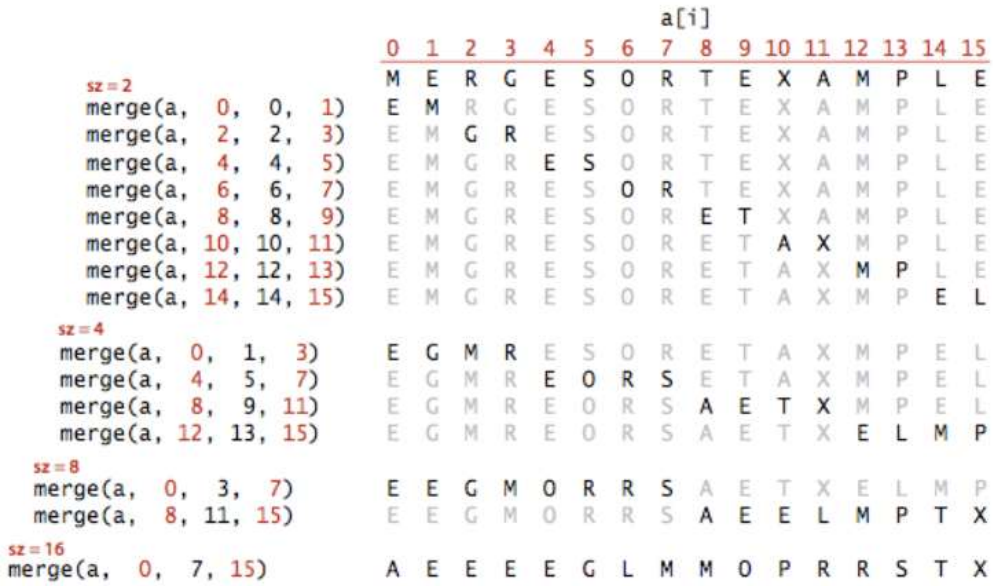


	lo	hi	a[] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	0	1	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	2	3	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 1, 3)	0	3	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	4	5	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	6	7	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	4	7	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	0	7	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	8	9	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	10	11	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 8, 9, 11)	8	11	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 12, 13)	12	13	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	14	15	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	12	15	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	8	15	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	0	15	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for top-down mergesort

Top-down mergesort uses between 1/2NlogN and NlogN compares and at most 6NlogN array accesses to sort any array of length N

Bottom-up mergesort. Another way to implement mergesort is to organize the merges so that we do all the merges of tiny arrays on one pass, then do a second pass to merge those arrays in pairs, and so forth, continuing until we do a merge that encompasses the whole array. We start by doing a pass of 1-by-1 merges (considering individual items as subarrays of size 1), then a pass of 2-by-2 merges (merge subarrays of size 2 to make subarrays of size 4), then 4-by-4 merges and so forth.



	a[i] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sz = 2																
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 4																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 8																
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 16																
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for bottom-up mergesort

Bottom – up mergesort uses between 1/2NlogN and NlogN compares and at most 6NlogN array accesses to sort any array of length N.

Quicksort.

Works well for a variety of different kinds of input data, and is substantially faster than any other sorting method. It is **in-place** (uses only a small auxiliary stack), requires time proportional to NlogN on the average to sort N items, and has an extremely short inner loop.

Quicksort is a divide-and-conquer method for sorting. It works by partitioning an array into two parts, then sorting the parts independently.

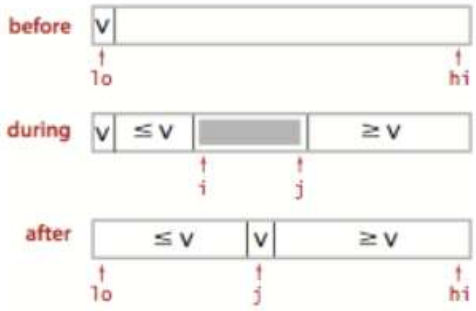
Quicksort overview

The crux of the method is the partitioning process, which rearranges the array to make the following three conditions hold:

1. The entry a[j] is in its final place in the array, for some j
2. No entry in a[lo] through a[j-1] is greater than a[j]
3. No entry in a[j+1] through a[hi] is less than a[j]

We achieve a completer sort by partitioning, then recursively applying the method to the subarrays. It is a randomized algorithm, because it randomly shuffles the array before sorting it.

Partitioning – First, we arbitrarily choose a[lo] to be the partitioning item – the one that will go into its final position. Next, we scan from the left end of the array until we find an entry that is greater than (or equal to) the partitioning item, and we scan from the right end of the array until we find an entry less than (or equal to) the partitioning item.

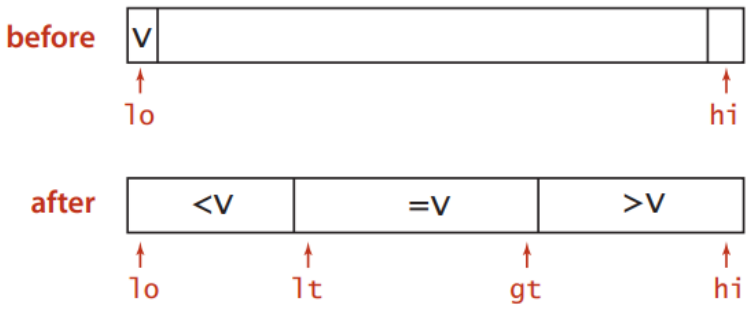


Quicksort partitioning overview

The two items that stopped the scans are out of place in the final partitioned array, so we exchange them. When the scan indices cross, all that we need to do to complete the partitioning process is to exchange the partitioning item a[lo] with the rightmost entry of the left subarray (a[j]) and return its index j.

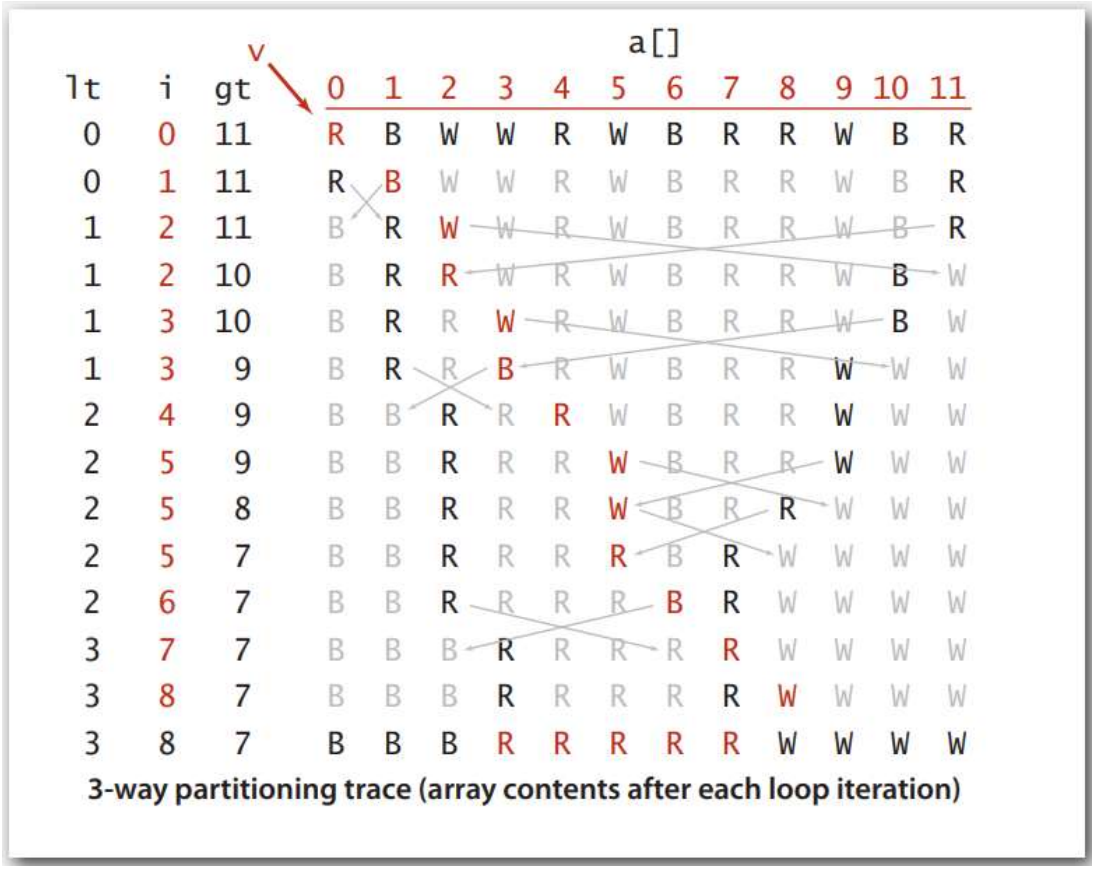
3-way partitioning: Qellimi eshte te ndahet matrica ne 3 pjese te tilla qe:

- Elementet midis lt dhe gt jane te njejta me pivot
- Nuk ka elemente me te medhenj se pivot ne te majte te lt
- Nuk ka elemente me te vegjel se pivot ne te djathte te gr



Menyra e implementimit:

- Shenojme me v: pivot (elementi a[lo])
- Skanojme i nga e majta ne te djathte
  1. (a[i] <v): Shkembe a[lt] me a[i], inkremento lt edhe i
  2. (a[i] > v): shkembe me a[i], dekremento gt
  3. (a[i] == v): inkremento i



3-way partitioning trace (array contents after each loop iteration)

Priority Queues.

Priority Queues are characterized by the remove the maximum and insert operations. By convention, we will compare keys only with a less( ) method, as we have been doing for sorting. Thus, if records can have duplicate keys, maximum means any record with the largest key value.

public class MaxPQ<Key extends Comparable<Key>>	
MaxPQ()	*create a priority queue*
MaxPQ(int max)	*create a priority queue of initial capacity* max
MaxPQ(Key[] a)	*create a priority queue from the keys in* a[]
void insert(Key v)	*insert a key into the priority queue*
Key max()	*return the largest key*
Key delMax()	*return and remove the largest key*
boolean isEmpty()	*is the priority queue empty?*
int size()	*number of keys in the priority queue*

The binary heap is a data structure that can efficiently support the basic priority-queue operations. In a binary heap, the items are stored in an array such that each key is guaranteed to be larger than (or equal to) the keys at two other specific positions. In turn, each of those keys must be larger than two more keys, and so forth.

A binary tree is heap-ordered if the key in each node is larger than the keys in that nodes two children.

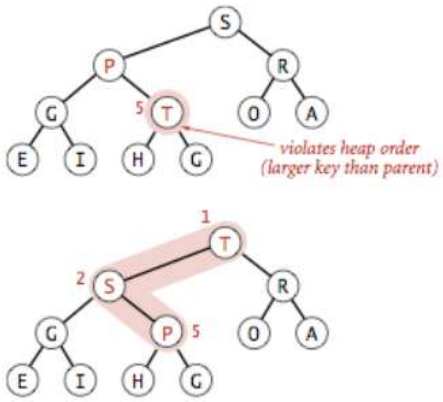
The largest key in a heap-ordered binary tree is found at the root.

We represent complete binary trees sequentially within an array by putting the nodes with level order, with the root at position 1, its childrent at positions 2 and 3, their children in positions 4, 5, 6 and 7, and so on.

A **binary heap** is a set of nodes with keys arranged in a complete heap-ordered binary tree, represented in level order in an array (not using the first entry). In a heap, the parent of the node in position k is in position k/2; and conversely, the two children of the node in position k are in positions 2k and 2k+1. We can travel up and down by doing simple arithmetic on array indices: to move up the tree from a[k] we set k to k/2; to move down the tree we set k to 2k or 2k+1.

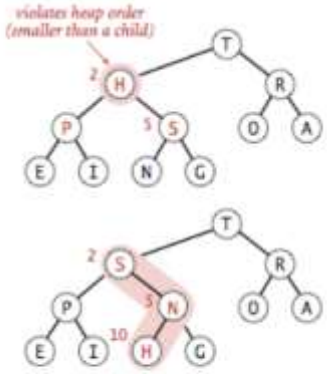
We represent a heap of size n in private array pq[] of length n+1, with pq[0] unused and the heap in pq[1] through pq[n]. We access keys only through private helper functions less( ) and exch( ). The heap operations that we consider work by first making a simple modification that could violate the heap condition, then traveling through the heap, modifying the heap as required to ensure that the heap condition is satisfied everywhere. We refer to this process as reheapifying, or restoring heap order.

- Swim: If the heap order is violated because a node's key becomes larger than that node's parents key, then we can make progress toward fixing the violation by exchanging the node with its parent. After the exchange, the node is larger than both its children but the node may still be larger than its parent. We can fix that violation in the same way, and so forth, moving up the heap until we reach a node with a larger key, or the root.




```
private void swim(int k) {
   while (k > 1 && less(k/2, k)) {
      exch(k, k/2);
      k = k/2;
   }
}
```

- Top-down heapify (sink): If the heap order is violated because a node's key becomes smaller than one or both of that node's children's keys, then we can make progress toward fixing the violation by exchanging the node with the larger of its two children. This switch may cause a violation at the child; we fix that violation in the same way, and so forth, moving down the heap until we reach a node with both children smaller, or the bottom.




```
private void sink(int k) {
   while (2*k <= N) {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

Heap-based priority queue.

- Insert. We add the new item at the end of the array, increment the size of the heap, and then swim up through the heap with that item to restore the heap condition.
- Remove the maximum. We take the largest item off the top, put the item from the end of the heap at the top, decrement the size of the hep, and then sink down through the heap with that item to restore the heap condition.

In an n-item priority queue, the heap algorithms require no more than 1 + logn compares for insert and no more than 2lgn compares for remove maximum.

Heapsort.

We can use any priority queue to develop a sorting method. We insert all the keys to be sorted into a minimum-oriented priority queue, then repeatedly use remove the minimum to remove them all in order. When using a heap for the priority queue, we obtain heapsort.

We use swim( ) and sink( ) directly. Doing so allows us to sort an array without needing any extra space, by maintaining the heap within the array to be sorted.

Heapsort breaks into two phases: heap construction, where we reorganize the original array into a hep, and the sortdown, where we pull the items out of the heap in decreasing order to build the sorted result.

- Heap construction: We can accomplish this task in time proportional to nlgn, by proceeding from left to right through the array, using swim( ) to ensure that the entries to the left of the scanning pointer make up a heap-ordered complete tree. A clever method that is much more efficient is to proceed from right to left, using sink( ) to make subheaps as we go. Every position in the array is the root of a small subheap; sink( ) works or such subheaps, as well. If the two children of a node are heaps, then callin sink( ) on that node makes the subtree rooted there a heap.
- Sortdown. Most of the work during heapsort is done during the second phase, where we remove the largest remaining items form the heap and put it into the array position vacated as the heap shrinks.

Sink-based heap construction is linear time.

Heapsort uses fewer than 2nlogn compare and exchanges to sort n items.

Hash Tables

If keys are small integers, we can use an array to implement a symbol table, by interpreting the key as an array index so that we can store the value associated with key i in array position i. With hashing, we reference key-value pairs using arrays by doing arithmetic operations to transform keys into array indices.

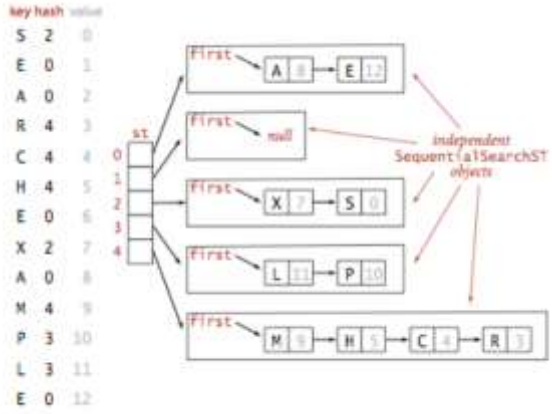
Search algorithms that use hashing consist of two separate parts. The first step is to compute a hash function that transforms the search key into an array index. Ideally, different keys would map to different indices. This ideal is generally beyond our reach, so we have to face the possibility that two or more different keys may hash to the same array index. Thus, the second part of a hashing search is a collision-resolution process that deals with this situation.

Hash functions: If we have an array that can hold M key-value pairs, then we need a function that can transform any given key into an index into that array: an integer in the range [0, M-1]. We seek a hash function that is both easy to compute and uniformly distributes the keys.

We have three primary requirements in implementing a good hash function for a given data type:

- It should be deterministic – equal keys must produce the same hash value
- It should be efficient to compute
- It should uniformly distribute the keys

Hashing with separate chaining. The second component of a hashing algorithm is collision resolution: a strategy for handling the case when two or more keys to be inserted hash to the same index. A straightforward approach to collision resolution is to build, for each of the M array indices, a linked list of the key-value pairs whose keys hash to that index. The basic idea is to choose M to be sufficiently large that the lists are sufficiently short to enable efficient search through a two-step process: hash to find the list that could contain the key, then sequentially search through that list for the key.



Hashing with separate chaining for standard indexing client

In a separate-chaining hash table with M lists and N keys, the probability that the number of keys in a list is within a small constant factor of N/M is extremely close to 1.

In a separate-chaining hash table with M lists and N keys, the number of compares for search and insert is proportional to N/M

Hashing with linear probing. Another approach to implementing hashing is to store N key-vale pairs in a hash table of size M>N, relying on empty entries in the table to help with collision resolution. Such methods are called open-addressing hashing methods. The simplest open-addressing method is called linear probing: when there is a collision (when we has to a table index that is already occupied with a key different from the search key), then we just check the next entry in the table (by incrementing the index). There are three possible outcomes:

- Key equal to search key: search hit
- Empty position (null key at indexed position): search miss
- Key not equal to search key: try next entry

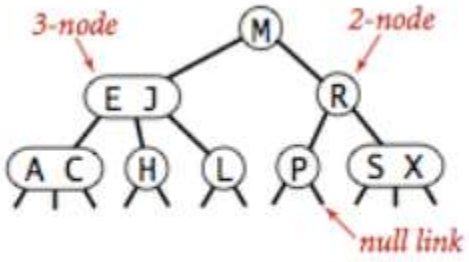
As with separate chaining, the performance of open-addressing methods is dependent on the ratio N/M, but we interpret it differently. For separate chaining the raport is the average number of items per list and is generally larger than 1. For open addressing, is it the percentage of table positions that are occupied; it must be less than 1. We refer to N/M as the load factor of the hash table.

Balanced Search Trees

2-3 search trees: The primary step to get the flexibility that we need to guarantee balance in search trees is to allow the nodes in our trees to hold more than one key.

A 2-3 search tree is a tree that either is empty or:

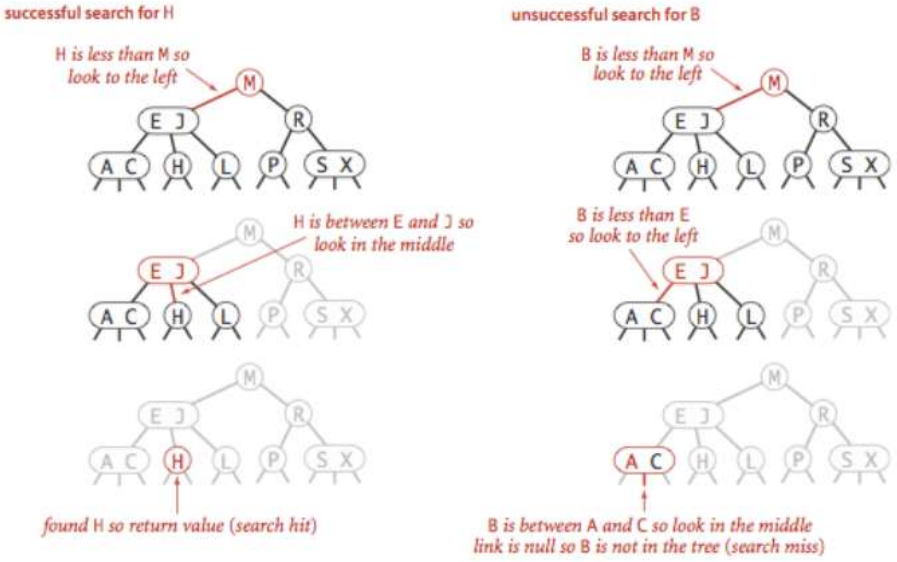
- A 2-node, with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A 3-node, with two keys (and associated values) and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys and a right link to a 2-3 search tree with larger keys



Anatomy of a 2-3 search tree

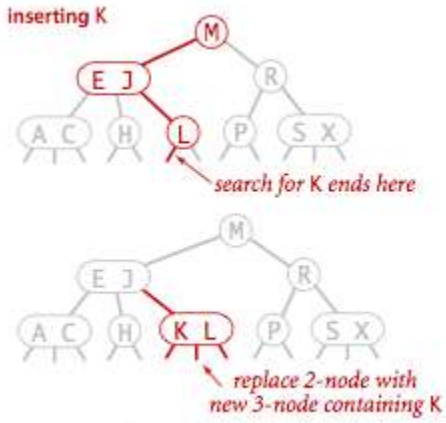
A perfectly balanced 2-3 search tree is one whose null links are all the same distance from the root.

- Search. To determine whether a key is in a 2-3 tree, we compare it against the keys at the root: If it is equal to any of them, we have a search hit; otherwise, we follow the link from the root to the subtree corresponding to the interval of key values that could contain the search key, and then recursively search in that subtree.



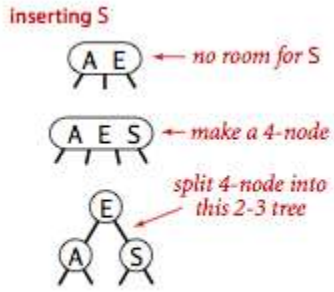
Search hit (left) and search miss (right) in a 2-3 tree

- Insert into a 2-node. To insert a new node in a 2-3 tree, we might do an unsuccessful search and then hook on the node at the bottom, as we did with BSTs, but the new tree would not remain perfectly balanced. It is easy to maintain perfect balance if the node at which the search terminates is a 2-node: We just replace the node with a 3-node containing its key and the new key to be inserted.



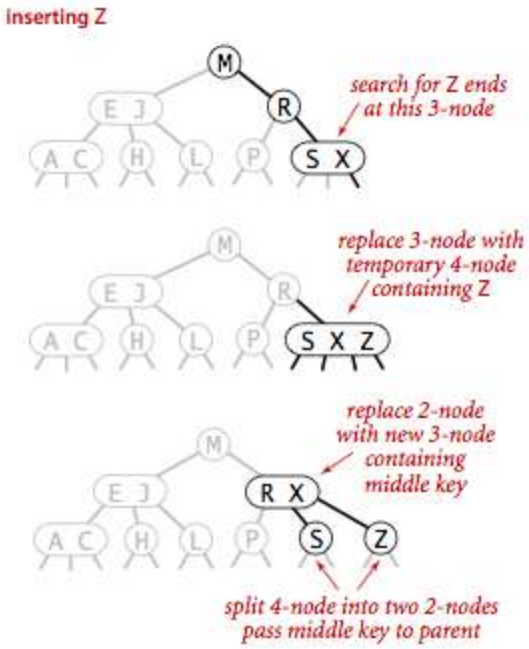
Insert into a 2-node

- Insert into a tree consisting of a single 3-node. Suppose that we want to insert into a tiny 2-3 tree consisting of just a single 3-node. Such a tree has two keys, but no room for a new key in its one node. To be able to perform the insertion, we temporarily put the new key into a 4-node, a natural extension of our node type that has three keys and four links. Creating the 4-node is convenient because it is easy to convert it into a 2-3 tree made up of three 2-nodes, one with the middle key (at the root), one with the smallest of the three keys (pointed to by the left link of the root), and one with the largest of the three keys (pointed to by the right link of the root).



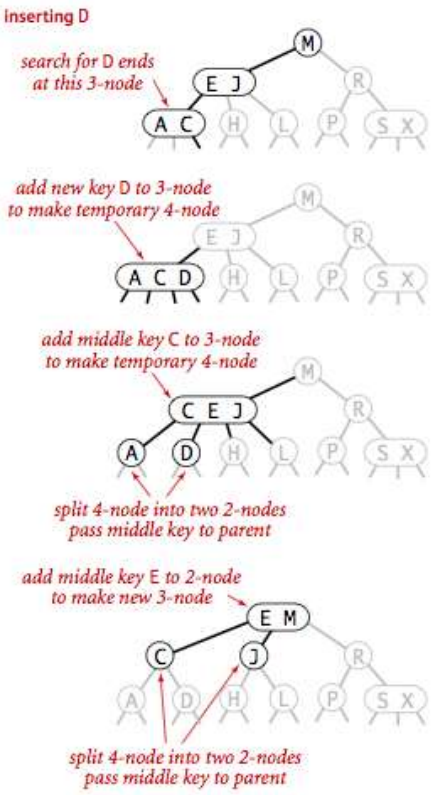
Insert into a single 3-node

- Insert into a 3-node whose parent is a 2-node. Suppose that the search ends at a 3-node at the bottom whose parent is a 2-node. In this case, we can still make room for the new key while maintaining perfect balance in the tree, by making a temporary 4-node as just described, then splitting the 4-node as just described, but then, instead of creating a new node to hold the middle key, moving the middle key to the nodes parent.



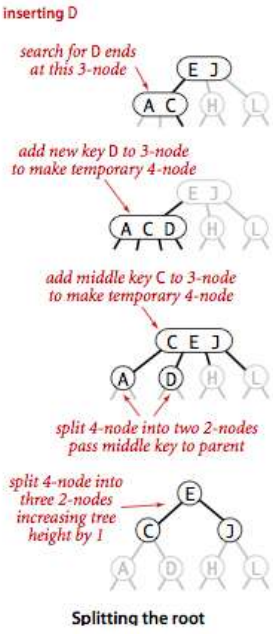
Insert into a 3-node whose parent is a 2-node

- Insert into a 3-node whose parent is a 3-node. Now suppose that the search ends at a node whose parent is a 3-node. Again, we make a temporary 4-node as just described, then split it and insert its middle key into the parent. The parent was a 3-node, so we replace it with a temporary new 4-node containing the middle key from the 4-node split. Then, we perform precisely the same transformation on that node. That is we split the new 4-node and insert its middle key into its parent. Extending to the general case is clear: we continue up the tree, splitting 4-nodes and inserting their middle keys in their parents until reaching a 2-node, which we replace with a 3-node that does not to be further split, or until reaching a 3-node at the root.



Insert into a 3-node whose parent is a 3-node

- Splitting the root. If we have 3-nodes along the whole path from the insertion point to the root, we end up with a temporary 4-node at the root. In this case we split the temporary 4-node into three 2-nodes.

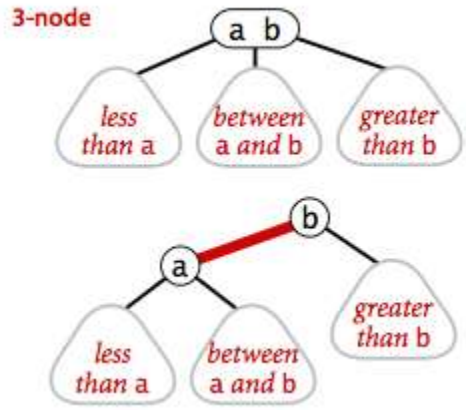


Splitting the root

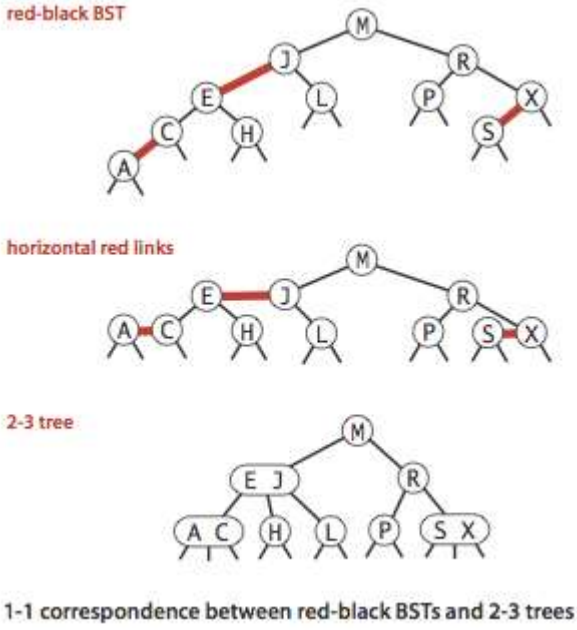
Search and insert operations in a 2-3 tree with N keys are guaranteed to visit at mos lgN nodes.

Red-Black BSTs.

- Encoding 3-nodes. The basic idea behind red-black BSTs is to encode 2-3 trees by starting with standard BSTs and adding extra information to encode 3-nodes. We think of the links as being of two different types: **red** links, which bind together two 2-nodes to represent 3 nodes, and **black** links, which bind together the 2-3 tree. Specifically, we represent 3-nodes as two 2-nodes connected by a single red link that leans left. We refer to BSTs that represent 2-3 trees in this way as red-black BSTs.



- A 1-1 correspondence. Given any 2-3 tree, we can immediately derive a corresponding red-black BST, just by converting each node as specified. Conversely, if we draw the red links horizontally in a red-black BST, all of the null links are the same distance from the root, and if we then collapse together the nodes connected by red links, the result is a 2-3 tree.

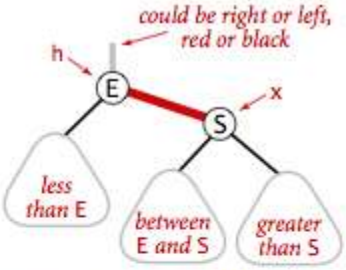


1-1 correspondence between red-black BSTs and 2-3 trees

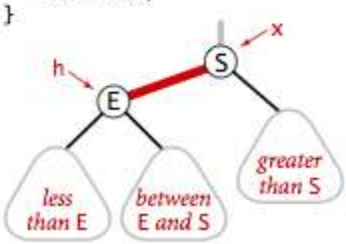
- Rotations. The implementation that we will consider might allow right-leaning red links or two red-links in a row during an operation, but it always corrects these conditions

before completion, through judicious use of an operation called rotation that switches orientation of red links. First, suppose that we have a right-leaning red link that needs to be rotated to lean to the left. This operation is called a left rotation. Implementing a right rotation that converts a left-leaning red link to a right-leaning one amounts to the same code, with left and right interchanged.

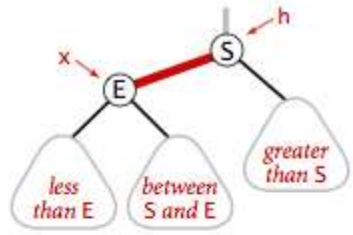
- Flipping colors. The implementation that we will consider might also allow a black parent to have two red children. The color flip operation flips the colors of the the two red children to black and the color of the black parent to red.



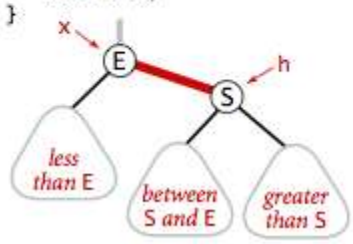

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Left rotate (right link of h)




```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Right rotate (left link of h)

- Insert into a single 2 node
- Insert into a 2 node at the bottom
- Insert into a tree with two keys (in a 3 node)
- Keeping the root black
- Insert into a 3- node at the bottom
- Passing a red link up the tree

In a red-black BST, the following operations take logarithmic time in the worst case: search, insertion, finding the minimum, finding the maximum, floor, ceiling, rank, select, delete the minimum, delete the maximum, delete and range count.

The average length of a path from the root to a node in a red-black BST with N nodes is 1.00lgN.