```
public class BST<Key extends Comparable<Key>, Value> {

   private Node root;

   private class Node
   {  /* see previous slide */  }

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Value get(Key key)
   {  /* see next slides */  }

   public void delete(Key key)
   {  /* see next slides */  }

   public Iterable<Key> iterator()
   {  /* see next slides */  }
}
```

```
private class Node {

   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int count;
}
```

```
public Value get (Key key) {

   Node x = root;

   while (x != null) {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

```
private Node put (Node x, Key key, Value val) {   initialize subtree count to 1

   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo( x.key);
   if      (cmp  < 0) x.left  = put( x.left,  key, val);
   else if (cmp  > 0) x.right = put( x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.count = 1 + size( x.left) + size( x.right);
   return x;
}
```

```
public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin (Node x) {
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.count = 1 + size(x.left) + size(x.right);
   return x;
}
```

Proposition. If N distinct keys are inserted into a BST in random order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

```
public int rank (Key key)
{  return rank(key, root);  }

private int rank (Key key, Node x) {
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return rank(key, x.left);
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);
   else if (cmp == 0) return size(x.left);
}
```

```
public Iterable<Key> keys() {

    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q) {
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```

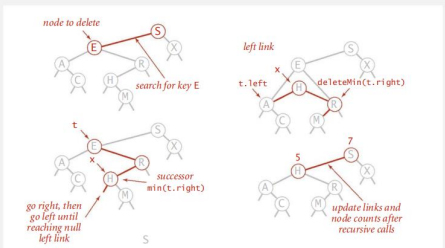
```
public Key floor (Key key) {
   Node x = floor( root, key);
   if (x == null) return null;
   return x.key;
}

private Node floor (Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp < 0)  return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```

Case 0. [0 children] Delete t by setting parent link to null.

Case 1. [1 child] Delete t by replacing parent link.

Case 2. [2 children]

- Find successor x of t. ← x has no left child
- Delete the minimum in t's right subtree. ← but don't garbage collect x
- Put x in t's spot. ← still a BST




```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete (Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

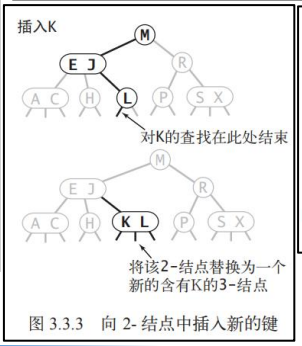


图 3.3.3 向 2- 结点中插入新的键

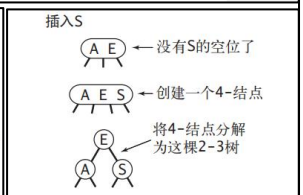


图 3.3.4 向一棵只含有一个 3- 结点的树中插入新键

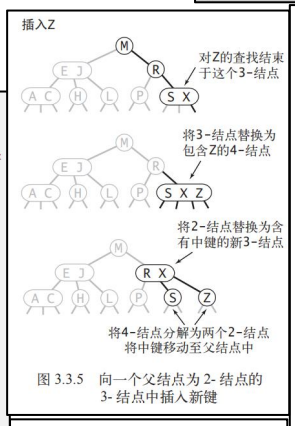


图 3.3.5 向一个父结点为 2- 结点的 3- 结点中插入新键

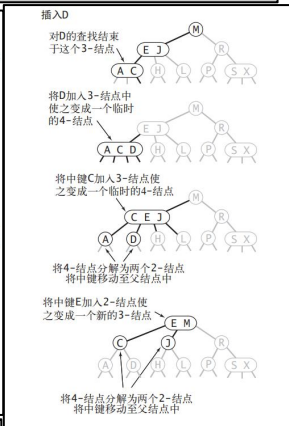


图 3.3.6 向一个父结点为 3- 结点的 3- 结点中插入新键

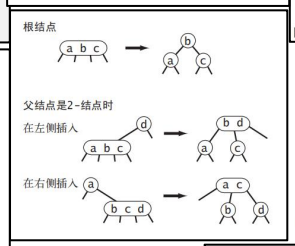
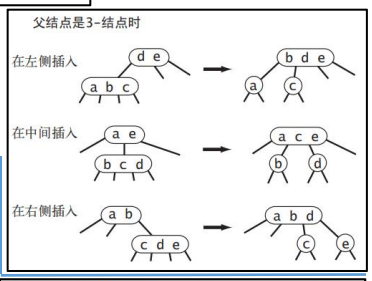


图 3.3.8 在一棵 2-3 树中分解一个 4- 结点的情况汇总




```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;
   Value val;
   Node left, right;
   boolean color;   // color of parent link
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

null links are black

```
private Node put (Node h, Key key, Value val) {

   if (h == null) return new Node( key, val, RED);          ← insert at bottom (and color it red)
   int cmp = key.compareTo( h.key);
   if      (cmp  < 0) h.left  = put( h.left,  key, val);
   else if (cmp  > 0) h.right = put( h.right, key, val);
   else if (cmp == 0) h.val = val;

   if (isRed(h.right) && !isRed(h.left))      h = rotateLeft( h);   ← lean left
   if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight( h);   ← balance 4-node
   if (isRed(h.left)  && isRed(h.right))      flipColors( h);       ← split 4-node

   return h;
}
```

only a few extra lines of code provides near-perfect balance

```
private Node rotateLeft(Node h) {

   assert isRed( h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

右旋交换 left 和 right

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}

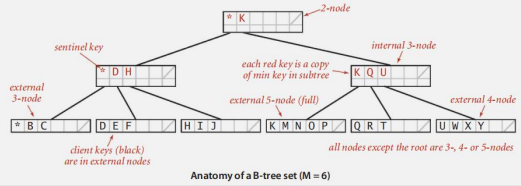
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

correct

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

choose M as large as possible so that M links fit in a page, e.g., M = 1024



Anatomy of a B-tree set (M = 6)

```
private void flipColors(Node h) {

   assert !isRed( h);
   assert isRed( h.left);
   assert isRed( h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

"Standard" recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- If field is a primitive type, use wrapper type hashCode().
- If field is null, return 0.
- If field is a reference type, use hashCode(). ← applies rule recursively
- If field is an array, apply to each entry. ← or use Arrays.deepHashCode()

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.

线性探测法

```java
public class SeparateChainingHashST<Key, Value> {

   private int M = 97;                // number of chains
   private Node[] st = new Node[M];   // array of chains

   private static class Node {

      private Object key;   ← no generic array creation
      private Object val;   ← (declare key and value of type Object)
      private Node next;
      ...
   }

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

   public Value get (Key key) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) return (Value) x.val;
      return null;
   }

}
```

```java
public class LinearProbingHashST<Key, Value> {

   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key)            {  /* as before */  }

   private void put(Key key, Value val) {  /* next slide */  }

   public Value get (Key key) {

      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
              return vals[i];
      return null;
   }

}
```

```java
public void put (Key key, Value val) {

   int i;
   for (i = hash(key); keys[i] != null; i = (i+1) % M)
      if (keys[i].equals(key))
          break;
   keys[i] = key;
   vals[i] = val;
}
```

```java
public void put (Key key, Value val) {
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next)
      if (key.equals(x.key)) { x.val = val; return; }
   st[i] = new Node(key, val, st[i]);
}
```

命题 M。在一张大小为 M 并含有 $N=\alpha M$ 个键的基于线性探测的散列表中，基于假设 J，命中和未命中的查找所需的探测次数分别为：

$$\sim \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \text{和} \sim \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

特别是当 α 约为 1/2 时，查找命中所需要的探测次数约为 3/2，未命中所需要的约为 5/2。当 α 趋近于 1 时，这些估计值的精确度会下降，但不需要担心这些情况，因为我们会保证散列表的使用率小于 1/2。

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		equals()
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	✔	compareTo()
red-black BST	2 lg N	2 lg N	2 lg N	1.0 lg N	1.0 lg N	1.0 lg N	✔	compareTo()
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		equals() hashCode()
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		equals() hashCode()

```java
public class Graph  {

   private final int V;
   private Bag<Integer>[] adj;

   public Graph (int V) {

      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge (int v, int w) {
      adj[v].add( w);
      adj[w].add( v);
   }

   public Iterable<Integer> adj (int v)
   {  return adj[v];  }
}
```

```java
public class DepthFirstPaths
{
   private boolean[] marked;
   private int[] edgeTo;
   private int s;

   public DepthFirstPaths(Graph G, int s)
   {
      ...
      dfs(G, s);
   }

   private void dfs(Graph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w])
         {
            dfs(G, w);
            edgeTo[w] = v;
         }
   }
}
```

```java
public boolean hasPathTo (int v) {
   return marked[v];
}

public Iterable<Integer> pathTo (int v) {
   if (!hasPathTo(v)) return null;
   Stack<Integer> path = new Stack<Integer>();
   for (int x = v; x != s; x = edgeTo[x])
      path.push( x);
   path.push( s);
   return path;
}
```

```java
public class BreadthFirstPaths {

   private boolean[] marked;
   private int[] edgeTo;
   private int[] distTo;

   …

   private void bfs(Graph G, int s) {
      Queue<Integer> q = new Queue<Integer>();
      q.enqueue( s);
      marked[s] = true;
      distTo[s] = 0;
      while (!q.isEmpty()) {
         int v = q.dequeue();
         for (int w : G.adj(v)) {
            if (!marked[w]) {
               q.enqueue( w);
               marked[w] = true;
               edgeTo[w] = v;
               distTo[w] = distTo[v] + 1;
            }
         }
      }
   }
```

```java
public class CC {

   private boolean[] marked;
   private int[] id;
   private int count;
```

```java
public int count()
{  return count;  }

public int id(int v)
{  return id[v];  }

public boolean connected (int v, int w)
{ return id[v] == id[w];  }
```

```java
private void dfs (Graph G, int v) {
   marked[v] = true;
   id[v] = count;
   for (int w : G.adj(v))
      if (!marked[w])
         dfs( G, w);
}
```

```java
public CC (Graph G) {
   marked = new boolean[G.V()];
   id = new int[G.V()];

   for (int v = 0; v < G.V(); v++) {
      if (!marked[v]) {
         dfs(G, v);
         count++;
      }
   }
```

problem	BFS	DFS	time
path between s and t	✔	✔	$E+V$
shortest path between s and t	✔		$E+V$
connected components	✔	✔	$E+V$
biconnected components		✔	$E+V$
cycle	✔	✔	$E+V$
Euler cycle		✔	$E+V$
Hamilton cycle			$2^{1.657\,V}$
bipartiteness	✔	✔	$E+V$
planarity		✔	$E+V$
graph isomorphism			$2^{c\sqrt{V}\log V}$

Algorithm

- A problem-solving method suitable for implementation as a computer program

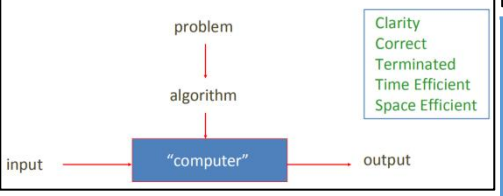
Data structures

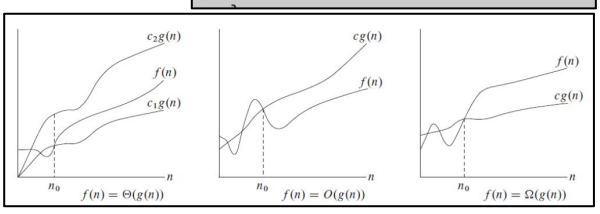
- Objects created to organize data used in computation
- A way to store and organize data in order to facilitate access and modifications

Data structure exist as the by-product or end product of algorithms

- Understanding data structure is essential to understanding algorithms and hence to problem-solving
- Simple algorithms can give rise to complicated data-structures
- Complicated algorithms can use simple data structures

An *algorithm* is a sequence of unambiguous instructions for solving a computation problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.





DFS，BFS 把 Graph 改成 Digraph

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

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

```java
public class Digraph {

   private final int V;
   private final Bag<Integer>[] adj;

   public Digraph (int V) {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge (int v, int w) {
      adj[v].add(w);
   }

   public Iterable<Integer> adj (int v)
   {  return adj[v];  }
}
```

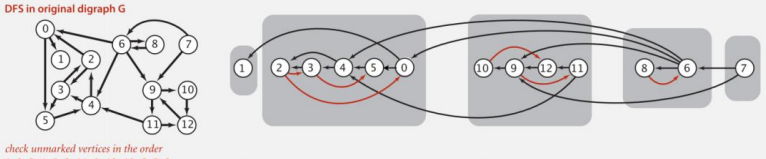
```java
public class DepthFirstOrder {

   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   public DepthFirstOrder (Digraph G) {
      reversePostorder = new Stack<Integer>();
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) dfs(G, v);
   }

   private void dfs (Digraph G, int v) {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
      reversePostorder.push(v);
   }

   public Iterable<Integer> reversePostorder()
   {  return reversePostorder;  }
}
```

```
dfs(0)
  dfs(1)
    dfs(4)
    4 done
  1 done
  dfs(2)
  2 done
  dfs(5)
    check 2
  5 done
0 done
check 1
check 2
dfs(3)
  check 2
  check 4
  check 5
  dfs(6)
    check 0
    check 4
  6 done
3 done
check 4
check 5
check 6
done
```

Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G, considering vertices in order given by first DFS.

DFS in original digraph G



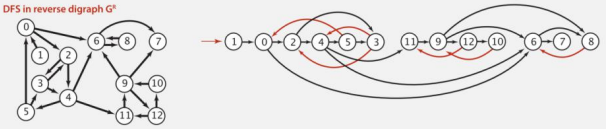
check unmarked vertices in the order
1 0 2 4 5 3 11 9 12 10 6 7 8

```
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
  dfs(8)
    check 6
  8 done
  check 0
6 done
dfs(7)
  check 6
  check 9
7 done
check 8
```

Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G, considering vertices in order given by first DFS.

DFS in reverse digraph G^R



check unmarked vertices in the order
0 1 2 3 4 5 6 7 8 9 10 11 12

reverse postorder for use in second dfs()
1 0 2 4 5 3 11 9 12 10 6 7 8

```
dfs(0)
  dfs(6)
    dfs(8)
      check 6
    8 done
    dfs(7)
    7 done
  6 done
  dfs(2)
    dfs(4)
      dfs(11)
        dfs(9)
          dfs(12)
            check 11
            dfs(10)
              check 9
            10 done
          12 done
          check 7
          check 6
  . . .
```

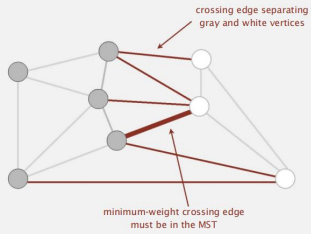
```
public class KosarajuSharirSCC {

   private boolean marked[];
   private int[] id;
   private int count;

   public KosarajuSharirSCC (Digraph G) {

      marked = new boolean[G.V()];
      id = new int[G.V()];
      DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
      for (int v : dfs.reversePostorder()) {

         if (!marked[v]) {
            dfs(G, v);
            count++;
         }
      }
   }

   private void dfs (Digraph G, int v) {
      marked[v] = true;
      id[v] = count;
      for (int w : G.adj(v))
         if (!marked[w])
            dfs(G, w);
   }

   public boolean stronglyConnected (int v, int w)
   {  return id[v] == id[w];  }
}
```

Def. A spanning tree of G is a subgraph T that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.
Def. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.

```
public class KruskalMST {

   private Queue<Edge> mst = new Queue<Edge>();

   public KruskalMST (EdgeWeightedGraph G) {

      MinPQ<Edge> pq = new MinPQ<Edge>( G.edges());

      UF uf = new UF(G.V());

      while (!pq.isEmpty() && mst.size() < G.V()-1) {
         Edge e = pq.delMin();
         int v = e.either(), w = e.other(v);
         if (!uf.connected( v, w)) {
            uf.union( v, w);
            mst.enqueue( e);
         }
      }
   }

   public Iterable<Edge> edges()
   {  return mst;  }
}
```

```
public class Edge implements Comparable<Edge> {

   private final int v, w;
   private final double weight;

   public Edge (int v, int w, double weight) {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int either()
   {  return v;  }

   public int other (int vertex) {
      if (vertex == v) return w;
      else return v;
   }

   public int compareTo (Edge that) {
      if      (this.weight < that.weight) return -1;
      else if (this.weight > that.weight) return +1;
      else                                return  0;
   }
}
```

```
public class EdgeWeightedGraph {

   private final int V;
   private final Bag<Edge>[] adj;

   public EdgeWeightedGraph (int V) {
      this.V = V;
      adj = (Bag<Edge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Edge>();
   }

   public void addEdge (Edge e) {
      int v = e.either(), w = e.other(v);
      adj[v].add(e);
      adj[w].add(e);
   }

   public Iterable<Edge> adj (int v)
   {  return adj[v];  }
}
```

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1 †	$\log V$ †	1 †	$E + V \log V$

Prim's algorithm (lazy) demo

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V - 1$ edges.

```
public class LazyPrimMST {

   private boolean[] marked;    // MST vertices
   private Queue<Edge> mst;     // MST edges
   private MinPQ<Edge> pq;      // PQ of edges

   public LazyPrimMST (WeightedGraph G) {

      pq = new MinPQ<Edge>();
      mst = new Queue<Edge>();
      marked = new boolean[G.V()];
      visit(G, 0);

      while (!pq.isEmpty() && mst.size() < G.V() - 1) {
         Edge e = pq.delMin();
         int v = e.either(), w = e.other(v);
         if (marked[v] && marked[w]) continue;
         mst.enqueue(e);
         if (!marked[v]) visit(G, v);
         if (!marked[w]) visit(G, w);
      }
   }
}
```

```
   private void visit(WeightedGraph G, int v)
   {
      marked[v] = true;
      for (Edge e : G.adj(v))
         if (!marked[e.other(v)])
            pq.insert(e);
   }

   public Iterable<Edge> mst()
   {  return mst;  }
```

```
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];
```

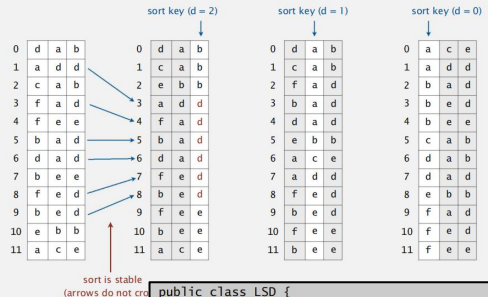
i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

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

LSD string (radix) sort.

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

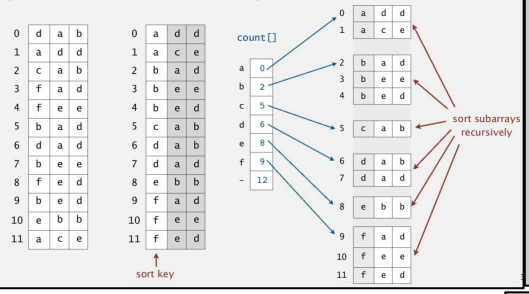


algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✔	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✔	compareTo()
quicksort	$1.39\, N \lg N$ *	$1.39\, N \lg N$	$c \lg N$		compareTo()
heapsort	$2\, N \lg N$	$2\, N \lg N$	1		compareTo()
LSD sort †	$2\, W\, (N + R)$	$2\, W\, (N + R)$	$N + R$	✔	charAt()
MSD sort ‡	$2\, W\, (N + R)$	$N \log_R N$	$N + D\, R$	✔	charAt()
3-way string quicksort	$1.39\, W\, N \lg R$ *	$1.39\, N \lg N$	$\log N + W$		charAt()

```
public class LSD {

   public static void sort (String[] a, int W) {

      int R = 256;
      int N = a.length;
      String[] aux = new String[N];

      for (int d = W-1; d >= 0; d--) {

         int[] count = new int[R+1];
         for (int i = 0; i < N; i++)
            count[a[i].charAt(d) + 1]++;
         for (int r = 0; r < R; r++)
            count[r+1] += count[r];
         for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]++] = a[i];
         for (int i = 0; i < N; i++)
            a[i] = aux[i];
      }
   }
}
```

MSD string (radix) sort.

- Partition array 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).

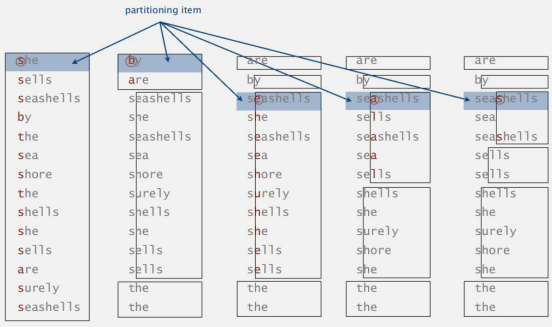



```
public static void sort(String[] a) {
   aux = new String[a.length];
   sort(a, aux, 0, a.length - 1, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d) {

   if (hi <= lo) return;
   int[] count = new int[R+2];                    key-indexed counting
   for (int i = lo; i <= hi; i++)
      count[charAt(a[i], d) + 2]++;
   for (int r = 0; r < R+1; r++)
      count[r+1] += count[r];
   for (int i = lo; i <= hi; i++)
      aux[count[charAt(a[i], d) + 1]++] = a[i];
   for (int i = lo; i <= hi; i++)
      a[i] = aux[i - lo];

   for (int r = 0; r < R; r++)                    sort R subarrays recursively
      sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

recycles aux[] array but not count[] array

```
public class KMP
{
   private String pat;
   private int[][] dfa;
   public KMP(String pat)
   {  // 由模式字符串构造DFA
      this.pat = pat;
      int M = pat.length();
      int R = 256;
      dfa = new int[R][M];
      dfa[pat.charAt(0)][0] = 1;
      for (int X = 0, j = 1; j < M; j++)
      {  // 计算dfa[][j]
         for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];              // 复制匹配失败情况下的值
         dfa[pat.charAt(j)][j] = j+1;           // 设置匹配成功情况下的值
         X = dfa[pat.charAt(j)][X];             // 更新重启状态
      }
   }
   public int search(String txt)
   {  // 在txt上模拟DFA的运行
      int i, j, N = txt.length(), M = pat.length();
      for (i = 0, j = 0; i < N && j < M; i++)
         j = dfa[txt.charAt(i)][j];
      if (j == M) return i - M;  // 找到匹配（到达模式字符串的结尾）
      else        return N;      // 未找到匹配（到达文本字符串的结尾）
   }
}
```

3-way string quicksort: trace of recursive calls




```
private static void sort (String[] a)
{  sort( a, 0, a.length - 1, 0);  }

private static void sort (String[] a, int lo, int hi, int d) {

   if (hi <= lo) return;
   int lt = lo, gt = hi;
   int v = charAt(a[lo], d);                    3-way partitioning (using dth character)
   int i = lo + 1;

   while (i <= gt) {                            to handle variable-length strings
      int t = charAt( a[i], d);
      if      (t < v) exch( a, lt++, i++);
      else if (t > v) exch( a, i, gt--);
      else            i++;
   }

   sort(a, lo, lt-1, d);
   if (v >= 0) sort( a, lt, gt, d+1);           sort 3 subarrays recursively
   sort(a, gt+1, hi, d);
}
```

type	bytes
char[]	$2N + 24$
int[]	$4N + 24$
double[]	$8N + 24$

one-dimensional arrays

```
public static int binarySearch (int[] a, int key) {
   int lo = 0, hi = a.length-1;

   while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      if      (key < a[mid]) hi = mid - 1;
      else if (key > a[mid]) lo = mid + 1;
      else return mid;
   }
   return -1;
}
```

```
public class Insertion {

   public static void sort (Comparable[] a) {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch( a, j, j-1);
            else break;
   }
```

```
public class Shell {

   public static void sort (Comparable[] a) {

      int N = a.length;

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

      while (h >= 1) {
         // h-sort the array.
         for (int i = h; i < N; i++) {
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch( a, j, j-h);
         }

         h = h/3;
      }
   }
```

```
public class Selection {

   public static void sort (Comparable[] a) {
      int N = a.length;
      for (int i = 0; i < N; i++) {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch( a, i, min);
      }
   }
```

```
public class Merge {

   private static void merge (...) {
      /* as before */
   }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort (Comparable[] a) {
      Comparable[] aux = new Comparable[a.length];
      sort( a, aux, 0, a.length - 1);
   }
}
```

```
private static int partition(Comparable[] a, int lo, int hi) {

   int i = lo, j = hi+1;

   while (true) {

      while (less(a[++i], a[lo]))                find item on left to swap
         if (i == hi) break;

      while (less(a[lo], a[--j]))                find item on right to swap
         if (j == lo) break;

      if (i >= j) break;                         check if pointers cross
      exch( a, i, j);                            swap
   }

   exch( a, lo, j);                              swap with partitioning item
   return j;                                     return index of item now known to be in place
}
```

```
private void swim( int k) {

   while (k > 1 && less( k/2, k)) {

      exch(k, k/2);

      k = k/2;
   }
}
```

parent of node at k is at k/2

```
public class MaxPQ<Key extends Comparable<Key>> {

   private Key[] pq;
   private int n;

   public MaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity+1];  }

   public boolean isEmpty()
   {  return n == 0;  }
   public void insert(Key key)  // see previous code
   public Key delMax()

   private void swim(int k)
   private void sink(int k)

   private boolean less(int i, int j)
   {  return pq[i].compareTo(pq[j]) < 0;  }
   private void exch(int i, int j)
   {  Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
}
```

```
public void insert (Key x) {
   pq[++n] = x;
   swim(n);
}
```

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {

   for (int k = lo; k <= hi; k++)               copy
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++) {
      if      (i > mid)                         a[k] = aux[j++];      merge
      else if (j > hi)                          a[k] = aux[i++];
      else if (less(aux[j], aux[i]))            a[k] = aux[j++];
      else                                      a[k] = aux[i++];
   }
}
```

```
public class Heap {

   public static void sort (Comparable[] a) {

      int n = a.length;
      for (int k = n/2; k >= 1; k--)
         sink( a, k, n);

      while (n > 1) {
         exch( a, 1, n);
         sink( a, 1, --n);
      }
   }

   but make static (and pass arguments)
   private static void sink(Comparable[] a, int k, int n)
   {  /* as before */  }

   private static boolean less(Comparable[] a, int i, int j)
   {  /* as before */  }

   private static void exch(Object[] a, int i, int j)
   {  /* as before */  }
```

but convert from 1-based indexing to 0-base indexing

```
private static void sort (Comparable[] a, int lo, int hi) {

   if (hi <= lo + CUTOFF - 1) {
      Insertion.sort( a, lo, hi);
      return;
   }
   int j = partition( a, lo, hi);
   sort( a, lo, j-1);
   sort( a, j+1, hi);
}
```

优化 1

原 sort 去掉白色部

```
public Key delMax () {
   Key max = pq[1];
   exch(1, n--);
   sink(1);
   pq[n+1] = null;
   return max;
}
```

```
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;
   }
}
```

children of node at k are 2*k and 2*k+1

a[i]

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

Heapsort trace (array contents just after each sink)

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✔		$n \log_3 n$	?	$c\, n^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✔	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✔		$n \lg n$	$2\, n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2\, n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✔		$3\, n$	$2\, n \lg n$	$2\, n \lg n$	$n \log n$ guarantee; in-place
?	✔	✔	n	$n \lg n$	$n \lg n$	holy sorting grail