time

cost model = array access

logN	binary search
N	find maximum
NlogN	merge sort
N^2	check pairs
N^3	check triples
2^N	check subsets
Θ(N2)	big theta 渐近
O(N2)	big oh 上限
Ω(N2)	big omega 下限
~ 10N2	tidle 主项

Selection Sort

N^2/2 compare N exchange

```
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);
```

Insertion Sort

1/4N^2 compare 1/4N^2 exchange

binary insertion sort nlgn compare

```
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;
```

Shell Sort

```
public static void sort (Comparable[] a)
int N = a.length; int h = 1;
while (h < N/3) h = 3*h + 1;
while (h >= 1)
  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;
```

Quick Sort

```
private static int partition(Comparable[] a, int lo, int hi) {
int i = lo, j = hi+1;
while (true)
  while (less(a[++i], a[lo]))   if (i == hi) break;
  while (less(a[lo], a[--j]))   if (j == lo) break;
  if (i >= j) break;   exch( a, i, j);
exch( a, lo, j);
return j;
```

space

boolean 1   byte 1 char 2 int 4 float 4

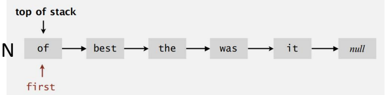
long8 double 8 array +24

object +16 padding --8 reference 8

stack

linked list 40 bytes per stack node

resizing-array between ~8N and ~32N

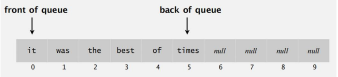


```
public class LinkedStackOfStrings
private Node first = null;
private class Node { String item; Node next;}
public boolean isEmpty() { return first == null; }
public void push(String item) {Node oldfirst = first;
first = new Node();first.item = item;first.next = oldfirst;}
public String pop() {String item = first.item;
first = first.next;return item;}
```

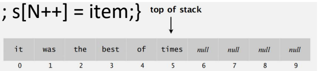
Reflexive: p is connected to p.

Symmetric: if p is connected to q, then q is connected to p.

Transitive: if p is connected to q and q is connected to r, then p is connected to r



```
public class ResizingArrayStackOfStrings
private String[] s;private int N = 0;
public ResizingArrayStackOfStrings() { s = new String[1]; }
public void push (String item)
{ if (N == s.length) resize(2 * s.length); s[N++] = item;}
private void resize (int capacity) {
String[] copy = new String[capacity];
for (int i = 0; i < N; i++)copy[i] = s[i];s = copy;}
public String pop() {String item = s[--N];s[N] = null;
if (N > 0 && N == s.length/4) resize(s.length/2);return item;}
```



Merge Sort 空间换时间 top-down扩展 bottom-up并行

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

Comparable 接口

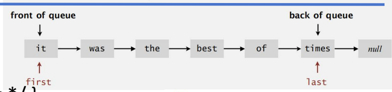
```
public interface Comparable<Item> {
   public int compareTo (Item that);
}
```

```
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 (a[j].compareTo(a[j-1]) < 0)
            exch( a, j, j-1);
         else break;
}
```

3-way quick sort

```
private static void sort (Comparable[] a, int lo, int hi) {
if (hi <= lo) return;
int lt = lo, gt = hi; Comparable v = a[lo]; int i = lo;
while (i <= gt) {int cmp = a[i].compareTo(v);
if (cmp < 0) exch( a, lt++, i++); else if (cmp > 0) exch( a, i, gt--); else i++;}
sort( a, lo, lt - 1); sort( a, gt + 1, hi);}
```

	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



```
public class LinkedQueueOfStrings
private Node first, last;
private class Node { /* same as Stack */ }
public boolean isEmpty() { return first == null; }
public void enqueue (String item) {Node oldlast = last;
last = new Node(); last.item = item; last.next = null;
if (isEmpty()) first = last;    else oldlast.next = last;}
public String dequeue() {String item = first.item;
first = first.next; if (isEmpty()) last = null;    return item;}
```

Iterator 迭代器

```
public interface Iterable<Item> {
   Iterator<Item> iterator();
}
```

```
public interface Iterator<Item> {
   boolean hasNext();
   Item next();
   void remove();   ←── optional; use
                        at your own risk
}
```

```
Iterator<String> i = stack.iterator();
while (i.hasNext()) {
   String s = i.next();
   StdOut.println(s);
}
```

Binary Heap

```
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; }
private void swim(int k)
while(k>1&& less( k/2,k)) {exch(k,k/2);   k=k/2;}
public void insert(Key X) pq[++n]= X;   swim(n);
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;}
public Key delMax() Key max = pq[1];   exch(1,n--);
sink(1);    pq[n+1]= null;    return max;
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; }
```

Heap Sort

```
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);;}
```

建立bottom-up二叉堆；取出最大项

Heap construction uses ≤ 2 n compares and ≤ n exchanges. Heapsort uses ≤ 2 n lg n compares and exchanges.

```
public class QuickFindUF union too expensive
private int[] id;
public QuickFindUF (int N)
id = new int[N]; for (int i = 0; i < N; i++) id[i] = i;
public int find(int p) { return id[p]; }
public void union (int p, int q)
{int pid = id[p]; int qid = id[q];
for (int i = 0; i < id.length; i++) if (id[i] == pid) id[i] = qid;}
public class QuickUnionUF find/connected too expensive
private int[] id;
public QuickUnionUF (int N)
{id = new int[N]; for (int i = 0; i < N; i++) id[i] = i;}
public int find (int i) {while (i != id[i]) i = id[i]; return i;}
public void union (int p, int q)
{int i = find(p); int j = find(q); id[i] = j;}
public void weightedUnion (int p, int q)
{int i = find(p); int j = find(q); if (i == j) return;
if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else { id[j] = i; sz[i] += sz[j]; }}
public int pathCompressedFind (int i)
{while (i != id[i]) {id[i] = id[id[i]];i = id[i];}return i;}
```

algorithm	worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

order of growth for M union-find operations on a set of N object

implementation	insert	del max	max
unordered array	1	n	n
ordered array	n	1	1
binary heap	$\log n$	$\log n$	1
d-ary heap	$\log_d n$	$d \log_d n$	1
Fibonacci	1	$\log n$ †	1
Brodal queue	1	$\log n$	1

Do "half-exchanges" in sink and swim.

Floyd's "bounce" heuristic.先沉到最底

Multiway heaps.多个分支

Binary Search

```
public Value get (Key key) {
if (isEmpty()) return null;
int i = rank(key);
if (i < N && keys[i].compareTo(key) == 0)
return vals[i]; else return null;}
private int rank (Key key) {
int lo = 0, hi = N-1; while (lo <= hi) {
int mid = lo + (hi - lo) / 2;
int cmp = key.compareTo( keys[mid]);
if (cmp < 0) hi = mid - 1;
else if (cmp > 0) lo = mid + 1;
else if (cmp == 0) return mid;}
return lo;}
```

判断相等前

```
if (y == this) return true;
if (y == null) return false;
if (y.getClass() != this.getClass()) return false;
```

For any x y z

Reflexive: x.equals(x) is true.

Symmetric: x.equals(y) iff y.equals(x).

Transitive: if x.equals(y) and y.equals(z), then x.equals(z).

Non-null: x.equals(null) is false.

Immutable. String, Integer, Double, Color, Vector, Transaction, Point2D.

Mutable. StringBuilder, Stack, Counter, Java array.

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

binary heap 根节点最大
binary search tree 左节点<父节点<右节点
2-3 tree 两个节点 三个孩子
red-black tree 2-3数放平

Binary Search Tree

```
public class BST<Key extends
Comparable<Key>, Value> {
private Node root;
private class Node {
private Key key; private Value val;
private Node left, right; private int count;
public Node(Key key,Value val) {
this.key = key;this.val = val;}}
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;}
public void put(Key key, Value val)
{ root = put(root, key, val); }
private Node put (Node x, Key key, Value val) {
if (x == null) return new Node(key, val);
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 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 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;}
```

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing from v?
list of edges	E	1	E	E
adjacency matrix	V 2	1†	1	V
adjacency lists	E + V	1	outdegree(v)	outdegree(v)

红黑树

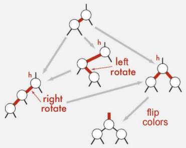
```
private Node put(Node h, Key key, Value val)
{if (h == null) return new Node(key, val, RED);
int cmp = key.compareTo(h.key);
if (cmp < 0) h.left = put(h.left, key, val);
else if (cmp > 0) h.right = put(h.right, key, val);
else if (cmp == 0) h.val = val;
if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
if (isRed(h.left) && isRed(h.right)) flipColors(h);
return h;}
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;}
private Node rotateRight(Node h){
assert isRed( h.left);    Node x= h.left;    h.left = x.right;    x.right = h;
x.color = h.color;    h.coor = RED;    return x;}
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;    return x;}
```

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	√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()

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			$2^{c\sqrt{V \log V}}$



Hash Table - Seperate Chaining

```
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;}
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]);}
```

Hash Table - Linear Probing

```
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;}
public void put (Key key, Value val) {
for (int i = hash(key); keys[i] != null; i = (i+1) % M)
if (keys[i].equals(key)) break;
keys[i] = key;   vals[i] = val; }
```

spanning tree: Connected,相连
Acyclic, 非循环结构
Includes all of the vertices包含全部点

Greedy Algorithm:
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.

无向图

```
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);}有向图不需要
public Iterable<Integer> adj (int v)
{ return adj[v]; }}
private void dfs (Graph G, int v) {marked[v] = true;
preorder.enqueue(v);
for (int w : G.adj(v))  if (!marked[w]) dfs(G, w);
postorder.enqueue(v);
reversePostorder.push(v);}
```

Key-indexed Counting

```
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++) countlr+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];
a[i]---LSD:a[i].charAt(d); MSD:charAt(a[i],d)+1;
```

Edge-weighted Graph

```
public class EdgeWeightedGraph {
private final int V;    private final Bag<Edge>[] adj;
public EdgeWeightedGraph (int V) {
constructor 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]; }}
```

operation	frequency	time per op
build pq	1	E
delete-min	E	log E
union	V	log* V †
connected	E	log* V †

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()

3-way string quicksort: 分割string,对后面进行排序

```
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;
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;}
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;
public class CC
private boolean[] marked;   private int[] id;  private int count;
public CC(Graph G)
marked = new boolean[G.V()]; id = new int[G.V()];
DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
for (int v : dfs.reversePostorder())有向图使用
for (int v = 0; v < G.V(); v++)    if (!marked[v])    dfs(G, v); count++;
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]; }
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);
```

Kruskal 选两个集合间的最小边，不构成环

```
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; }}
```

Prim每次添加一个点，eager使用优先级队列管理

```
public class LazyPrimMST {
private boolean[] marked; private Queue<Edge> mst;
private MinPQ<Edge> pq;
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; }
```

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