🏢 Data Structures

Written by Krishna Parashar
Published by Atrus

Table of Contents

Data Structures

Sorting Algorithms

Analyses

Data Structures

Linked Lists

SLists

<table>
<tr><td>Invariants</td><td colspan="2">1. An SList's size variable is always correct.
2. There is always a tail node whose next reference is null.</td></tr>
<tr><td>Code</td><td>

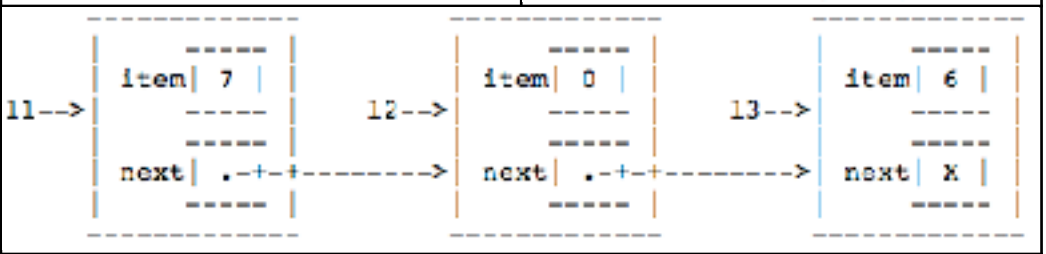
```
public class ListNode {
  public int item;
  public ListNode next;
}
public ListNode(int i, ListNode n) {
  item = i;
  next = n;
}
public ListNode(int i) {
  this(i, null);
}
```

</td><td>

```
public class SList {
  private SListNode head;
  private int size;

  public SList() {
    head = null;
    size = 0;
  }

  public void insertFront(Object item) {
    head = new SListNode(item, head);
    size++;
  }
}
```

</td></tr>
<tr><td>Example</td><td colspan="2"> 
</td></tr>
<tr><td>Run Time</td><td colspan="2">Insertion or Deletion: O(1)
Indexing or Searching: O(n)</td></tr>
</table>

DLists

<table>
<tr><td>Invariants</td><td colspan="2">1. An DList's size variable is always correct.
2. There is always a tail node whose next reference is null.
3. item.next and item.previous either points to an item or null (if tail or head).</td></tr>
<tr><td>Code</td><td>

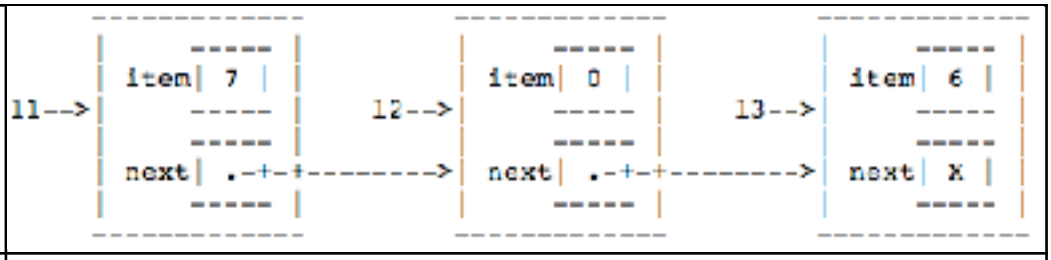
```
public class ListNode {
  public int item;
  public ListNode next;
}
public ListNode(int i, ListNode n) {
  item = i;
  next = n;
}
public ListNode(int i) {
  this(i, null);
}
```

</td><td>

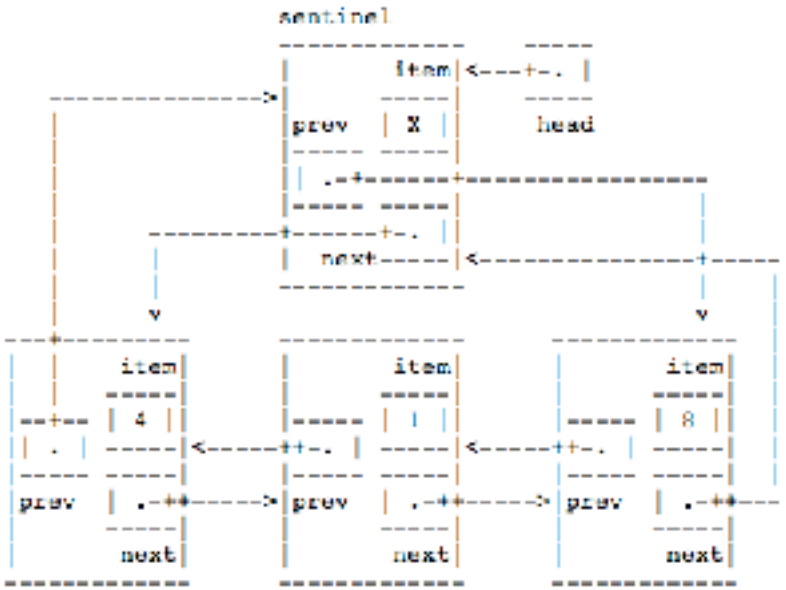
```
class DList {
    private DListNode head;
    private DListNode tail;
}

public DList() {
    head = null;
    tail = null;
    size = 0;
}
```

</td></tr>
</table>

Example	
Run Time	**Insertion or Deletion:** O(1) **Indexing or Searching:** O(n)

DLists with Sentinel

Invariants	(1) For any DList d, d.head != null. (There's always a sentinel.) (2) For any DListNode x, x.next != null. (3) For any DListNode x, x.prev != null. (4) For any DListNode x, if x.next == y, then y.prev == x. (5) For any DListNode x, if x.prev == y, then y.next == x. (6) A DList's "size" variable is the number of DListNodes, NOT COUNTING the sentinel (denoted by "head"), that can be accessed from the sentinel by a sequence of "next" references. (7) An empty DList is represented by having the sentinel's prev and next fieldspoint to itself.
Example	
Run Time	**Insertion or Deletion:** O(1) **Indexing or Searching:** O(n)

Hash Tables

<table>
<tr><td>Invariants</td><td colspan="3">• n is the number of keys (words).
• N buckets exists in a table.
• n ≤ N << possible keys.
• Load Factor is n/N < 1 (Around 0.75 is ideal).</td></tr>
<tr><td>Example Code (Strings)</td><td colspan="3"><pre>
private static int hashCode(String key) {
  int hashVal = 0;
  for (int i = 0; i < key.length(); i++) {
    hashVal = (127 * hashVal + key.charAt(i)) % 16908799;
  }
  return hashVal;
}
</pre></td></tr>
<tr><td>Algorithm</td><td colspan="3">Compression Function:
• h(hashCode) = hashCode mod N.
• Better: h(hashCode) = ((a * (hashCode) + b) mod p) mod N
• p is prime >> N, a & b are arbitrary positive ints.
• Really Large Prime: 16908799
• If hashCode % N is negative, add N.</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td>Methods</td><td>Entry insert(key, value)</td><td>Entry find(key)</td><td>Entry remove(key)</td></tr>
<tr><td>Algorithm</td><td>1. Compute the key's hash code and compress it to determine the entry's bucket.
2. Insert the entry (key and value together) into that bucket's list.</td><td>1. Hash the key to determine its bucket.
2. Search the list for an entry with the given key.
3. If found, return the entry; else, return null.</td><td>1. Hash the key to determine its bucket.
2. Search the list for an entry with the given key.
3. Remove it from the list if found.
4. Return the entry or null.</td></tr>
<tr><td>Run Time</td><td colspan="3">Best: O(1)
Worst: O(n)
Resizing: Average O(1) (Amortized)</td></tr>
</table>

Stacks

Invariants	• Crippled List: Can only operate on item at the top of stack. • Pop removes item on stack. • Push adds item to stack.

Representation	`public interface Stack {`
	`  public int size();`
	`  public boolean isEmpty();`
	`  public void push(Object item);`
	`  public Object pop();`
	`  public Object top();`
	`}`
	
	`//Use SList with insert front and remove front`

Run Time	All methods: O(1)

Queues

Invariants	• Can only read or remove item at the front of queue. • Can only add item to the back of queue.

Representation	`public interface Queue {`
	`  public int size();`
	`  public boolean isEmpty();`
	`  public void enqueue(Object item);`
	`  public Object dequeue();`
	`  public Object front();`
	`}`
	
	`//Use SList with tail pointer.`

Run Time	All methods: O(1)
Dequeues	• You can insert and remove items at both ends. • Use DList with removeFront() and removeBack() & no direct access to list nodes.

Rooted Trees

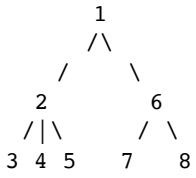
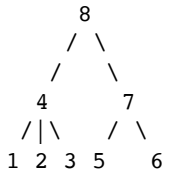
Invariants	• Exactly ONE path between any node on the tree. • Every node except root (which has ZERO) has ONE parent. • A node can have any number of children. • Depth of node is the # of nodes to traverse from root to node. • Height of node # of nodes to traverse from node to last child.

Representation	`class SibTreeNode {`
	`  Object item;`
	`  SibTreeNode parent;`
	`  SibTreeNode firstChild;`
	`  SibTreeNode nextSibling;`
	`}`
	
	`class SibTree {`
	`  SibTreeNode root;`
	`  int size;`
	`}`

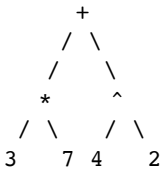
Tree Traversals

	Preorder	Postorder

Code	`public void preorder() {`
	`  this.visit();`
	`  if (firstChild != null) {`
	`  firstChild.preorder();`
	`  }`
	`  if (nextSibling != null){`
	`  nextSibling.preorder();`
	`  }`
	`}`
		`public void postorder() {`
		`  if (firstChild != null) {`
		`  firstChild.postorder();`
		`  }`
		`  this.visit();`
		`  if (nextSibling != null){`
		`  nextSibling.postorder();`
		`  }`
		`}`

Algorithm	1. Visit **current**. 2. Visit **current.left** nodes. 3. Visit **current.right** nodes.	1. Visit **current.left** nodes. 1. Visit **current.right** nodes. 2. Visit **current**.
Example		

Other Traversals

Inorder (Binary Tree Only)	
	`Prefix:  + * 3 7 ^ 4 2`
	`Infix:  3 * 7 + 4 ^ 2`
	`Postfix:  3 7 * 4 2 ^ +`
	Recursively traverse the **root's left** subtree (rooted at the left child), then the **root** itself, then the **root's right** subtree.

Level Order (Used for queues)	`+ * ^ 3 7 4 2 (From Above)` Visit the root, then all the depth-1 nodes (from left to right), then all the depth-2 nodes, etc.
Run Time	All Traversals: O(n)

Binary (Priority) Heaps

<table>
<tr><td>Invariant</td><td colspan="3">• Parent ≤ Children for ALL nodes.
• Can ONLY identify or remove the entry whose key is the lowest.
• Can insert an entry with any key at any time.</td></tr>
<tr><td>Representation</td><td colspan="3">
<pre>
Array: 0   1   2   3   4   5   6   7   8   9   10

      | | 2 | 5 | 3 | 9 | 6 | 11 | 4 | 17 | 10 | 8 |

        \--- array index 0 intentionally left empty.
 Tree:
                                        2
                                       / \
                                      /   \
                                     5   3
                                    / \   / \
                                   9   6 11 4
                                  /\   /   .
                                 17 10 8   .

public interface PriorityQueue {
  public int size();
  public boolean isEmpty();
  Entry insert(Object k, Object v);
  Entry min();
  Entry removeMin();
}
</pre>
</td></tr>
<tr><td>Indexing</td><td colspan="3">Node's index is i,
Left Child is 2i and Right Child is 2i+1,
Parent's index is floor (i/2) - Largest integer not greater than (i/2)
Smallest number is root.</td></tr>
<tr><td>Methods</td><td>removeMin()
Removes and returns the entry with the minimum key.</td><td>insert(key, value):
Adds an entry to the priority queue;</td><td>bottomUpHeap():
Turns unsorted list into binary heap.</td></tr>
<tr><td>Algorithm</td><td>1. Empty then return null.
2. Remove node, replace with last element of array.
3. Bubble root down, swapping left then right if smaller. Continue until ≤ to child or a leaf.
Note: Increase speed by bubbling up a hole, then fill in x.</td><td>1. Add as last value in array.
2. Compare with parent.
3. While new key is less, swap with parent.
4. Left to right for invariant.
5. Return Object.
Note: Increase speed by bubbling up a hole, then fill in x.</td><td>1. Make heap out of any order.
2. From heap size to one, parent checks from (right to left) & (down to up) for invariant. (Bubble down)
3. Swap if doesn't satisfy invariant. Check children for invariant conditions. (Part of swap)</td></tr>
<tr><td>Run Time</td><td>Best: Theta(1)
Worst: Theta(log n)</td><td>Best: Theta(log 1)
Worst: Theta(log n)</td><td>Best: Theta(n)</td></tr>
</table>

Binary Search Trees

<table>
<tr><td>Invariants</td><td>• Each node has only two children.
• Left Child ≤ Parent
• Right Child ≥ Parent</td></tr>
<tr><td>Representation</td><td>

```
public class BinaryTreeNode {
  Object item;
  BinaryTreeNode parent;
  BinaryTreeNode left;
  BinaryTreeNode right;

  public void inorder() {
    if (left != null) {
      left.inorder();
    }
    this.visit();
    if (right != null) {
     right.inorder();
    }
  }
}
public class BinaryTree {
  BinaryTreeNode root;
  int size;
}
```

</td></tr>
<tr><td>Binary Tree Search</td><td>

```
class BinaryTreeNode [
  Entry entry;
  BinaryTreeNode parent;
  BinaryTreeNode leftChild, rightChild;

  public BinaryTreeNode(Entry[] entries, int first, int last) {
    parent = null;   leftChild = null;   rightChild = null;
    int mid = (first + last) / 2;
    entry = entries[mid];
    if (mid > first) [
      leftChild = new BinaryTreeNode(entries, first, mid - 1);
      leftChild.parent = this;
    ]
    if (mid < last) {
      rightChild = new BinaryTreeNode(entries, mid + 1, last);
      rightChild.parent = this;
    ]
  }
}
```

</td></tr>
<tr><td>Note</td><td>Number of nodes in Balanced Trees is $2^{(height+1)} - 1$.</td></tr>
<tr><td>Algorithm</td><td>Compression Function:
• h(hashCode) = hashCode mod N.
• Better: h(hashCode) = ((a * (hashCode) + b) mod p) mod N
• p is prime >> N, a & b are arbitrary positive ints.
• Really Large Prime: 16908799
• If hashCode % N is negative, add N.</td></tr>
</table>

Methods

find(Object key)

Code	`public Entry find(Object k) {`
	`  BinaryTreeNode node = root; // Start at the root.`
	`  while (node != null) {`
	`  int comp = ((Comparable) k).compareTo(node.entry.key());`
	`  if (comp < 0) { // Repeatedly compare search`
	`  node = node.left; // key k with current node; if`
	`  } else if (comp > 0) { // k is smaller, go to the left`
	`  node = node.right; // child; if k is larger, go to`
	`  } else { /* The keys are equal */ // the right child. Stop when`
	`  return node.entry; // we find a match (success;`
	`  } // return the entry) or reach`
	`  } // a null pointer (failure;`
	`  return null; // return null).`
	`}`

Algorithm	1. Start at root. 2. Check if the node is what you want, if node is bigger go LEFT, smaller then go RIGHT. 3. Keep going until check is true, otherwise return null. **Note:** If we want to find the smallest key greater than or equal to k, or the largest key less than or equal searching downward through the tree for a key k that is not in the tree, we encounter both. Just modify the algorithm.
Run Time	Balanced: $O(\log_2 n)$ Unbalanced: Worst: $O(n)$

	Entry min() & Entry max()	**Entry insert(key, value)**
Algorithm	**min():** 1. Start at Root. 2. Empty, then return null. 3. Keep going to LEFT node until leaf. **max():** 1. Start at Root. 2. Empty, then return null. 3. Keep going to RIGHT node until leaf.	1. Start at root. 2. Check if the node is what you want, if node is bigger than insert key go RIGHT, smaller, go LEFT. 3. When you reach a null pointer, re-point to insert node (ensure invariant is satisfied). Duplicate keys are allowed. Just add to right or left branch. (See **Example** in *remove*)
Run Time	Balanced: $O(\log_2 n)$ Unbalanced: Worst: $O(n)$	

remove(Object key)

Code

```
public Entry find(Object k) {
  BinaryTreeNode node = root; // Start at the root.
  while (node != null) {
    int comp = ((Comparable) k).compareTo(node.entry.key());
    if (comp < 0) {           // Repeatedly compare search
      node = node.left;       // key k with current node; if
    } else if (comp > 0) {    // k is smaller, go to the left
      node = node.right;      // child; if k is larger, go to
    } else {    /* The keys are equal */ // the right child.  Stop
when
      return node.entry;      // we find a match (success;
    }                         // return the entry) or reach
  }                           // a null pointer (failure;
  return null;                // return null).
}
```

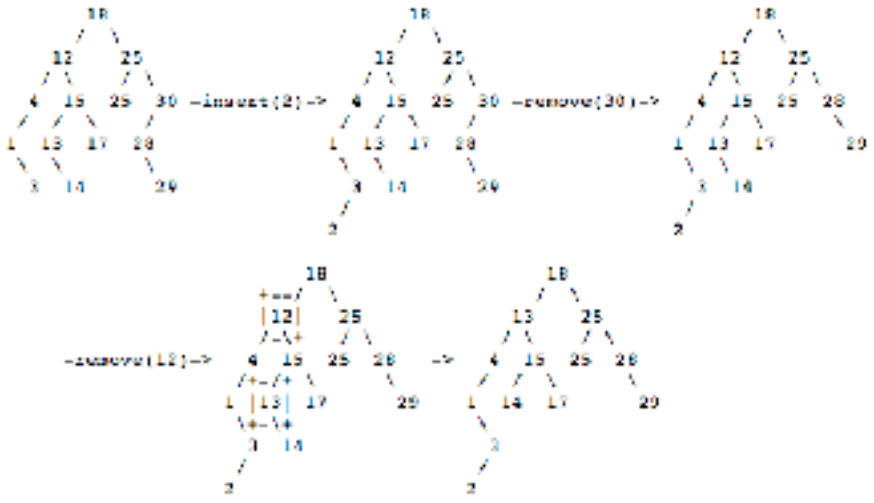
Algorithm

* Find node (call it n) using find algorithm, return null otherwise.
If n is a leaf, just get rid of it.

If n has ONE child:
1. *Replace n's place with its child's item.*
   (If coding then do the following):
2. Replace the child's place with n's item (the item you want to remove).
3. Remove item in child's place (the item you want to remove).

If n has TWO children:
1. Find the min of the right tree (call it x).
   (RIGHT then LEFTMOST)
2. Remove x.
3. Replace n's spot with with x's key.
4. (Optional) Check if invariant still hold by bubbling down.

Example



Run Time

Balanced: $O(\log_2 n)$
Unbalanced: Worst: O(n)

2-3-4 Trees

Invariants	• Perfectly Balanced. • Each node has 2, 3, or 4 children (except leaves at bottom). • Each node (no leaves) has 1 - 3 entries and # of entries + 1 children. • Leftmost child should be ≤ than 1st entry, 2nd ≤ 2nd entry, etc. • Rightmost child should be ≥ than last entry, etc.
Representation	
	Number of leaves is between 2^h or 4^h. If n is the total number of entries (including entries in internal nodes), then n >= $2^{(h+1)}$ - 1. By taking the log of both sides, h is in O(log n). Can combine all duplicate keys into one node to speed up methods.
Note	Number of nodes in Balanced Trees is $2^{(height+1)}$ - 1.

Methods

find(Object key)

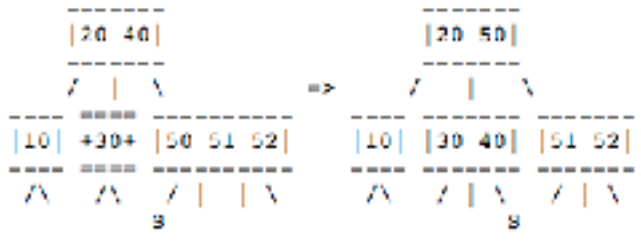
Code	`public boolean find(int key) {`
	`  Tree234Node node = root;`
	`  while (node != null) {`
	`  if (key < node.key1) { node = node.child1;}`
	`  else if (key == node.key1) { return true; }`
	`  else if ((node.keys == 1) || (key < node.key2)) {`
	`  node = node.child2; }`
	`  else if (key == node.key2) { return true; }`
	`  else if ((node.keys == 2) || (key < node.key3)) {`
	`  node = node.child3; }`
	`  else if (key == node.key3) { return true; }`
	`  else { node = node.child4; }`
	`  }`
	`  return false;`
	`}`

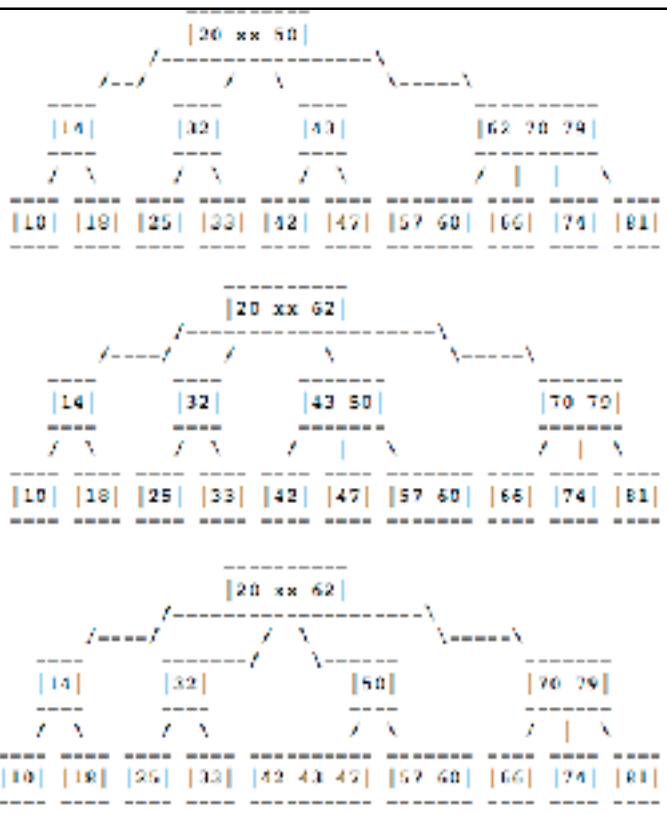
Algorithm	1. Start at root. 2. Check if the node is what you want, otherwise move left or right by comparing k with keys. 3. Return true if found or false if reach leaf and not it.
Run Time	O(log n)

insert(Object key, Object entry)

Algorithm	1. Start at root. 2. Use find algorithm to find entry with key “key”, if it finds it proceeds to entries left child and continues. 3. If you encounter a *three-key node*, eject middle key and move up with parent in appropriate spot. If root, then middle is new root. Update size! 4. Then if two nodes, squirt one of them out and attach the “key” in correct spot. **Note:** It is okay to make a new three key node along the way, although “key” should not be in a three key node.
Run Time	O(log n)

remove(Object key)

<table>
<tr><td rowspan="4">Algorithm</td><td>Find node (call it n) using find algorithm, return null otherwise</td></tr>
<tr><td>If n is a leaf:
• If it is in an internal node, replace it by its predecessor (or successor), which must be in a leaf.
• After the removal of the key value, if a leaf becomes empty, there are two possibilities:
  • If it has a 3-node sibling, a key is moved from the sibling to the parent, and the key in the parent is moved into the empty leaf.
  • If it has no 3-node sibling, it is merged with a sibling with key from the parent. This process may repeat at the parent. If the root become empty, it is removed.</td></tr>
<tr><td>If n has ONE child:
1. Tries to steal a key from an adjacent sibling,
2. If parent has 2+ keys and sibling has 2+ keys:
Move the key from adjacent sibling to the parent, and move a parent's key down to node n. We also move the left subtree of the sibling and make it the left sibling of the node n. (Known as rotation) (Example 1)
3. If parent has 2+ keys and sibling has only 1 key:
Steal a key from the parent (parent was treated same way and has two for sure, unless root) and absorb sibling to make a 3 key node. (Known as fusion) (Example 2)
4. If parent has 1 key & is root and sibling has only 1 key:
The current node, parent, and sibling are fused into one three key node and size--. (Example 3)
5. Soon we reach a leaf and we do the stuff above to make sure we are left with a two or three key node, then we delete the key and there is still one left in the leaf.</td></tr>
<tr><td>If n has TWO children:
1. Find the min of the right tree (call it x). (RIGHT then LEFTMOST)
2. Remove x.
3. Replace n's spot with with x's key.
4. (Optional) Check invariant still hold, by bubbling down.</td></tr>
<tr><td>Example 1</td><td>  </td></tr>
</table>

Example 2	
Example 3	 The last step is to remove 42 from the leaf and replace "xx" with 42.
Run Time	O(log n)

Directed Graphs

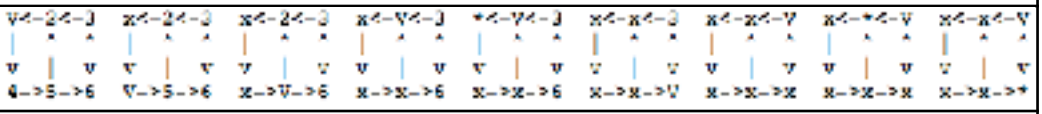
Invariants	• *Directed/Diagraph:*
	  • Every edge e is directed from some vertex v to some vertex w.
	  • The vertex v is called the origin of e, w is the destination of e.
	• *Undirected:* e = (v, w) = (w, v). No direction is favored.
	• **No multiple copies of an edge**; Directed graph may contain both (v, w) and (w, v).
	• *Strongly Connected* - each path has a path from each vertex to every other vertex.
	• Degree - number of edges coming out of the vertex.
	• For directed graphs there is *indegree* (incoming edges) & *out-degree* (outgoing edges).

Representation (Adjacency Matrix)	A complete graph is a graph having every possible edge: for every vertex u and every vertex v, (u, v) is an edge of the graph. (Adjacency Matrix: For Complete Graphs) 1 2 3 4 5 6 Alb Ken Eme Ber Oak Pie 1 - - - T - - Albany - T - T - - 2 T - - - - - Kensington T - - T - - 3 - T - - - T Emeryville - - - T T - 4 - - - - T - Berkeley T T T - T - 5 - T - - - T Oakland - - T T - T 6 - - T - - - Piedmont - - - - T -
Representation (Adjacency List)	(Adjacency List : Memory Efficient & Good for Everything else.) 
Note	Total memory used the lists is Theta(IVI + IEI)

Depth & Breadth First Search

Depth First Search (DFS)

Code	`public class Graph {`
	`  // Before calling dfs(),`
	`  // set every "visited" flag`
	`  // to false; O(|V|) time`
	`  public void dfs(Vertex u) {`
	`  u.visit();`
	`  // Do some thing to u`
	`  u.visited = true;`
	`  // Mark the vertex u visited`
	`  // for (each vertex v such that`
	`  // (u, v) is an edge in E) {`
	`  if (!v.visited) {`
	`  dfs(v);`
	`  }`
	`  }`
	`  }`
	`}`

Algorithm	No visiting twice. Starts at an arbitrary vertex and searches a graph as "deeply" as possible as early as possible. Visits a vertex u. Checks every vertex v that can be reached by some edge (u, v). If v has not yet been visited, DFS visits it recursively.
Example	
Run Time	Adjacency List: O(IVI + IEI) Adjacency Matrix: O(IVI^2)

Breadth First Search (BFS)

<table>
<tr><td>Code</td><td>

Breadth-first search (BFS) is a little more complicated than depth-first search, because it's not naturally recursive. We use a queue so that vertices are visited in order according to their distance from the starting vertex.

```
public void bfs(Vertex u) {
  for (each vertex v in V) { v.visited = false; }            // O(|V|) time
  u.visit(null);                                  // Do some unspecified thing to u
  u.visited = true;                                    // Mark the vertex u visited
  q = new Queue();                                                  // New queue...
  q.enqueue(u);                                       // ...initially containing u
  while (q is not empty) {
    v = q.dequeue();
    for (each vertex w such that (v, w) is an edge in E) {
      if (!w.visited) {
        w.visit(v);                               // Do some unspecified thing to w
        w.visited = true;                              // Mark the vertex w visited
        q.enqueue(w);
      }
    }
  }
}
```

Notice that when we visit a vertex, we pass the edge's origin vertex as a parameter. This allows us to do a computation such as finding the distance of the vertex from the starting vertex, or finding the shortest path between them. The visit() method at right accomplishes both these tasks.

```
public class Vertex {
  protected Vertex parent;
  protected int depth;
  protected boolean visited;

  public void visit(Vertex origin) {
    this.parent = origin;
    if (origin == null) {
      this.depth = 0;
    } else {
      this.depth = origin.depth + 1;
    }
  }
}
```

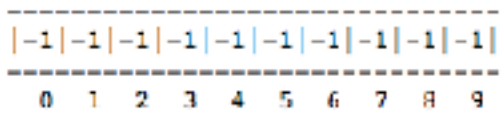
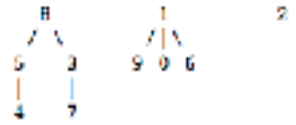
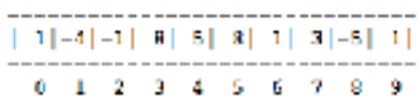
</td></tr>
<tr><td>Algorithm</td><td>The starting vertex is enqueued first, then all the vertices at a distance of 1 from the start, then all the vertices at a distance of 2, and so on. Dequeue and process all depth 1, then depth 2, etc.</td></tr>
<tr><td>Example</td><td>  </td></tr>
<tr><td>Run Time</td><td>Adjacency List: O(|V| + |E|)
Adjacency Matrix: O(|V|^2)</td></tr>
</table>

Kruskal's Algorithm

Algorithm	[1] Make a new graph T with the same vertices as G, but no edges (yet). [2] Make a list of all the edges in G. [3] Sort the edges by weight, from lowest to highest. [4] Iterate through the edges in sorted order. For each edge (u, w): [4a] If u and w are not connected by a path in T, add (u, w) to T. (**Using DFS**) **IGNORE Self Edges** Because this algorithm never adds (u, w) if some path already connects u and w, T is guaranteed to be a tree (if G is connected) or a forest (if G is not).
Run Time	O(|V| + |E| log |V|)

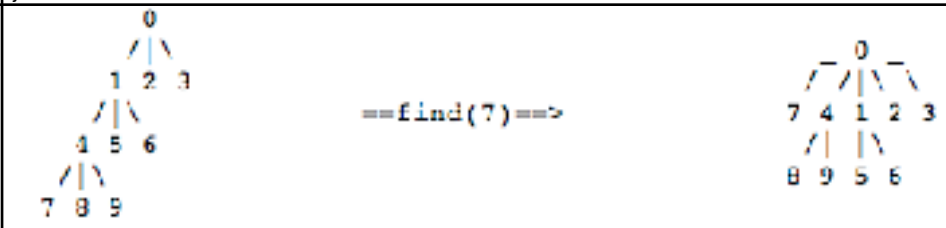
Disjoint Sets

<table>
<tr><td>Definitions</td><td>• Partition is a collection of disjoint sets because item s are partitioned among the sets.
• Universe is made up of all of the items that can be a member of a set</td></tr>
<tr><td>Invariants</td><td>• No item is found in more than one set. (Disjoint Property) Thus every item is in only one set.
• Each set references a list of the items in that set.
• Each item references the set that contains it.

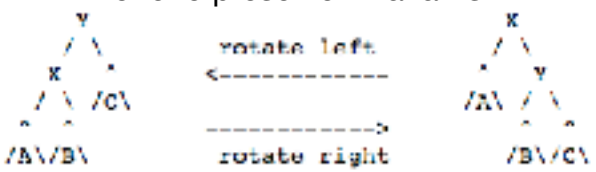
• An item references its parent as it array element.
• If an item has no parent, we'll record the size of its tree. We record the size s as the negative number -s (negative “size”).
• Initially every item is a root of itself so its array element is -1.</td></tr>
<tr><td>Example</td><td>Initially:
 
Tree:
 
Array:
  </td></tr>
<tr><td colspan="2">Using Quick Union Algorithm</td></tr>
<tr><td></td><td>union(int root1, int root2)</td></tr>
<tr><td>Code</td><td><pre>
public void union(int root1, int root2) {
  if (array[root2] < array[root1]) { // root2 has larger tree
    array[root2] += array[root1]; // update # of items in root2's tree
    array[root1] = root2;   // make root2 parent of root1
  }
  else {   // root1 has equal or larger tree
    array[root1] += array[root2]; // update # of items in root1's tree
    array[root2] = root1;   // make root1 parent of root2
  }
}
</pre></td></tr>
<tr><td>Run Time</td><td>Theta(1)</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td></td><td>find(int number)</td></tr>
</table>

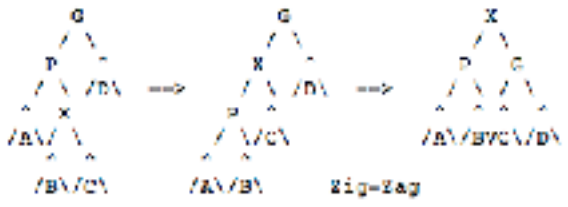
Note	Updates trees every time you call number and puts it near top.

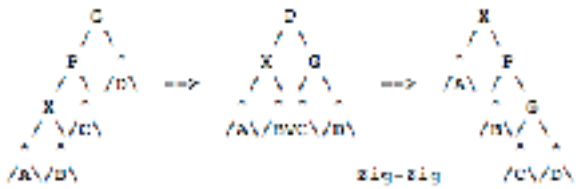
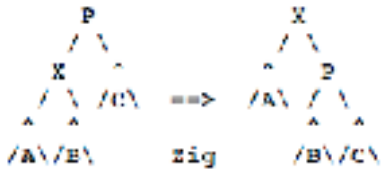
Code	`public int find(int x) {`
	`  if (array[x] < 0) { return x; } // x is the root of the tree; return it`
	`  else {`
	`  // Find out who the root is; compress path by making the root x's parent.`
	`  array[x] = find(array[x]);`
	`  return array[x];  // Return the root`
	`  }`
	`}`

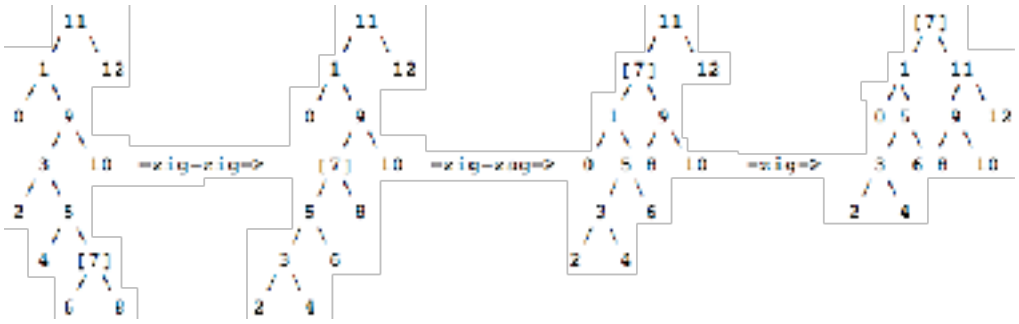
Example	
Run Time	Average (aka Amortized): **Theta(1)** **Technically:** Theta(u + f *alpha*(f + u, u)) where *alpha* is Inverse Ackermann Function which is always ≤ 4

Splay Trees

Invariants	• Each node has only two children. • Left Child ≤ Parent • Right Child ≥ Parent
Note	A splay tree is a type of balanced binary search tree.
Rotations	Done to preserve Invariants: 
Runtime	Average: **O(log n)** where *n* is number of entries. (Amortized) Any sequence of k ops. with tree initially empty & never > n items, takes O(k log n) worst case time.

Splaying	
X is node we are splaying to the root. **P be the parent of X.** **G be the grandparent of X.**	
Repeatedly **apply zig-zag and zig-zig rotations** to X (each rotation move X two levels up) until **X reaches the root** (and we're done) or **X becomes the child of the root** (in which case we zig).	
Zig-Zag	**X is the RIGHT child of a LEFT child :** 1. Rotate X and P left. 2. Rotate X and G right. (**Shown below**)
	X is the LEFT child of a RIGHT child : 1. Rotate X and P right. 2. Rotate X and G left.
	

Zig-Zig	**X is the LEFT child of a LEFT child :** 1. Rotate G and P right. 2. Rotate P and X right.
	X is the RIGHT child of a RIGHT child 1. Rotate G and P left. 2. Rotate P and X left.
	
Zig	X's parent P is the root: we rotate X and P so that X becomes the root.
	

find(Object key)	
Algorithm	1. Start at root. 2. Check if the node is what you want, if node is bigger go LEFT, smaller then go RIGHT. 3. **Splay** node to root. Return it. 4. If **not found** splay node where search ended.
Run Time	Average: **O(log n)** where *n* is number of entries.
Example	

remove(Object key)	
Algorithm	* Find node (call it n) using find algorithm, return null otherwise. **If n is a leaf, just get rid of it.**
	If n has ONE child: 1. *Replace n's place with its child's item.* (If coding then do the following): 2. Replace the child's place with n's item (the item you want to remove). 3. Remove item in child's place (the item you want to remove).
	If n has TWO children: 1. Find the min of the right tree (call it x). (RIGHT then LEFTMOST) 2. Remove x. 3. Replace n's spot with with x's key. 4. (Optional) Check if invariant still hold by bubbling down.
	<u>THEN:</u> After the node is removed, its parent splays to the root. If the key k **is not in the tree**, splay the node where the search ended to the root, just like in a find() operation.
	

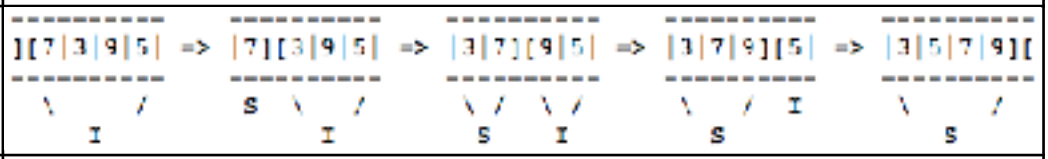
	Entry min() & Entry max()	Entry insert(key, value)
Algorithm	**min():** 1. Start at Root. 2. Empty, then return null. 3. Keep going to LEFT node until leaf. 4. **Splay** node to the rode. Return it. **max():** 1. Start at Root. 2. Empty, then return null. 3. Keep going to RIGHT node until leaf. 4. **Splay** node to the rode. Return it.	1. Start at root. 2. Check if the node is what you want, if node is bigger than insert key go RIGHT, smaller, go LEFT. 3. When you reach a null pointer, repoint to insert node (ensure invariant is satisfied). 4. **Splay** node to the rode (root actually). Return it. Duplicate keys are allowed. Just add to right or left branch.
Run Time	Average: **O(log n)** where *n* is number of entries.	

Sorting Algorithms

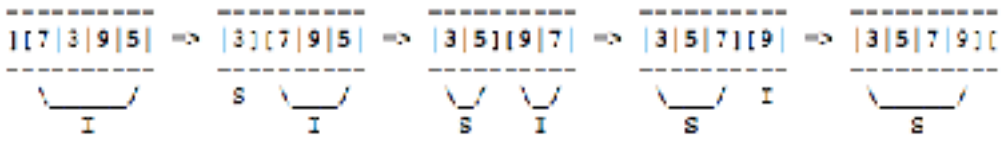
Insertion Sort + Selection Sort

Insertion Sort

In-Place Sort is a sorting algorithm that keeps the sorted items in the same array that initially held the input items.

Invariant	List/Linked List/Array *S* is Sorted. **Stable**
Algorithm	Start with an empty list S and the unsorted list I of n input items. for (each item x in I) : insert x into the list S, positioned so that S remains in sorted order.
Example	
Run Time	**List:** O(n^2) **Linked List:** Theta(n) (Worst Case) **Array:** O(log n) (Binary Search) **Worst Case Array**: Theta(n^2) **Almost Sorted:** Theta(n) **Balanced Tree:** O(n log n)

Selection Sort

Invariant	List/Linked List/Array *S* is Sorted. **Stable**
Algorithm	Start with an empty list S and the unsorted list I of n input items. for (i = 0; i < n; i++) : Let x be the item in I having smallest key. Remove x from I. Append x to the end of S.
Example	
Run Time	**Always**: Theta(n^2)

Heapsort

Heapsort

Invariant	Selection Sort on Heap I, where Parent ≤ Children for ALL nodes.

Algorithm	Start with an empty list S and an unsorted list I of n input items.
	Toss all the items in I onto a heap h (ignoring the heap-order property).
	h.bottomUpHeap(); // Enforces the heap-order property
	for (i = 0; i < n; i++) :
	x = h.removeMin();
	Append x to the end of S.

Example	
Run Time	O(n log n)
Notes	Good for **Arrays**, bad for linked lists. **In-place** if done as above. **NEVER Stable**

Merge Sort

Invariant	Selection Sort in which I is a Heap

Code	`Let Q1 and Q2 be two sorted queues.  Let Q be an empty queue.`
	`while (neither Q1 nor Q2 is empty):`
	`  item1 = Q1.front();`
	`  item2 = Q2.front();`
	`  move the smaller of item1 and item2`
	`  from its present queue to end of Q.`
	`concatenate the remaining non-empty queue (Q1 or Q2) to the end of Q.`

Algorithm	1. Start with the unsorted list I of n input items. 2. Break I into two halves I1 and I2, having ceiling(n/2) and floor(n/2) items. 3. Sort I1 recursively, yielding the sorted list S1. 4. Sort I2 recursively, yielding the sorted list S2. 5. Merge S1 and S2 into a sorted list S.
Example	
Run Time	Each level of the tree involves O(n) operations O(log n) levels. Runs in **O(n log n)**
Note	Stable, NOT In-place

Quicksort + Quickselect

Quicksort

Code

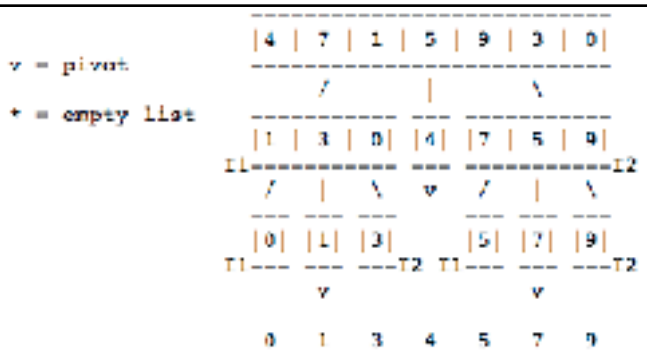
```
public static void quicksort(Comparable[] a, int low, int high) {
  // If there's fewer than two items, do nothing.
  if (low < high) {
    int pivotIndex = random number from low to high;
    Comparable pivot = a[pivotIndex];
    a[pivotIndex] = a[high];              // Swap pivot with last item
    a[high] = pivot;
    int i = low - 1;
    int j = high;
    do {
      do { i++; } while (a[i].compareTo(pivot) < 0);
      do { j--; } while ((a[j].compareTo(pivot) > 0) && (j > low));
      if (i < j) { swap a[i] and a[j]; }
    }
    while (i < j) {
      a[high] = a[i];
      a[i] = pivot;       // Put pivot in the middle where it belongs
      quicksort(a, low, i - 1);       // Recursively sort left list
      quicksort(a, i + 1, high);      // Recursively sort right list
  }
} // IN PLACE SORT & NOT STABLE // THIS IS FOR ARRAYS // LINKED LIST is Stable
```

Algorithm

1. Start with the unsorted list I of n input items.
2. Choose a pivot item v from I. Generally is done randomly.
3. Partition I into two unsorted lists I1 and I2.
   - I1 contains all items whose keys are smaller than v's key.
   - I2 contains all items whose keys are larger than v's.
   - Items with the same key as v can go into either list.
   - The pivot v, however, does not go into either list.
4. Sort I1 recursively, yielding the sorted list S1.
5. Sort I2 recursively, yielding the sorted list S2.
6. Concatenate S1, v, and S2 together, yielding a sorted list S.

Note: If I1 or I2 is equal to pivot, add it to S1 or S2

Example



Run Time

If Properly Designed: **O(n log n)** Always
Worst: $O(n^2)$

Quickselect

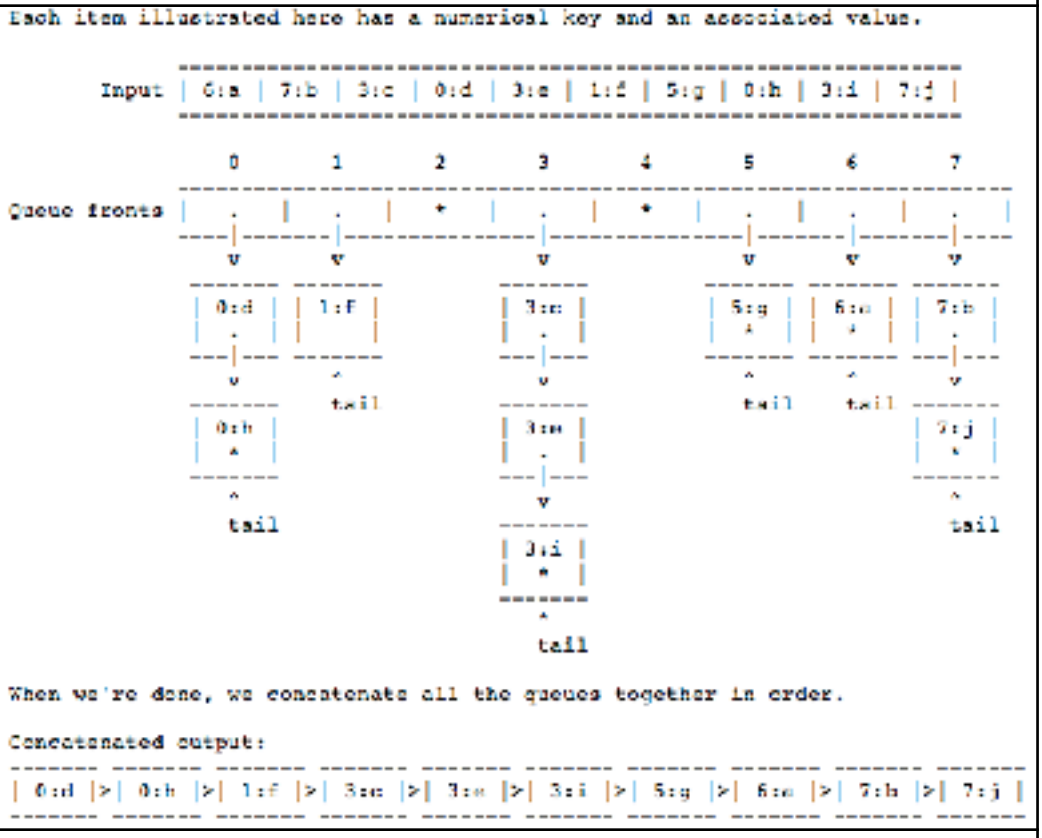
Goal	Find the kth smallest key in a list Use quicksort to find index j = floor(n / 2) where j = (k-1)

Algorithm	Start with an unsorted list I of n input items.
	Choose a pivot item v from I.
	Partition I into three unsorted lists I1, Iv, and I2.
	- I1 contains all items whose keys are smaller than v's key.
	- I2 contains all items whose keys are larger than v's.
	- Iv contains the pivot v.
	- Items with the same key as v can go into any of the three lists.
	(In list-based quickselect, they go into Iv;
	In array-based quickselect, they go into I1 and I2)
	if (j < |I1|):
	Recursively find the item with index j in I1; return it.
	else if (j < |I1| + |Iv|):
	Return the pivot v.
	else: // j >= |I1| + |Iv|.
	Recursively find the item with index j - |I1| - |Iv| in I2;
	return it.

Run Time	Theta(n)

EVERY comparison-based sorting algorithm takes Omega(n log n) worst-case time.

Bucket Sort

Note	**Good for Linked Lists** For array of *q* queues/buckets, good for code in a small range [0, *q* - 1], with number of items *n* > *q*. This is a ***stable*** sort if items with equal keys come out in the same order they went in.
Example	
Run Time	Theta(q + n) = Theta(num. of queues/buckets/counts + num. of items) **If q is in O(n) =** num of possible keys isn't much larger than the num of items. **Theta(q)** time to initialize the buckets + **Theta(n)** time to put all the items in their buckets.

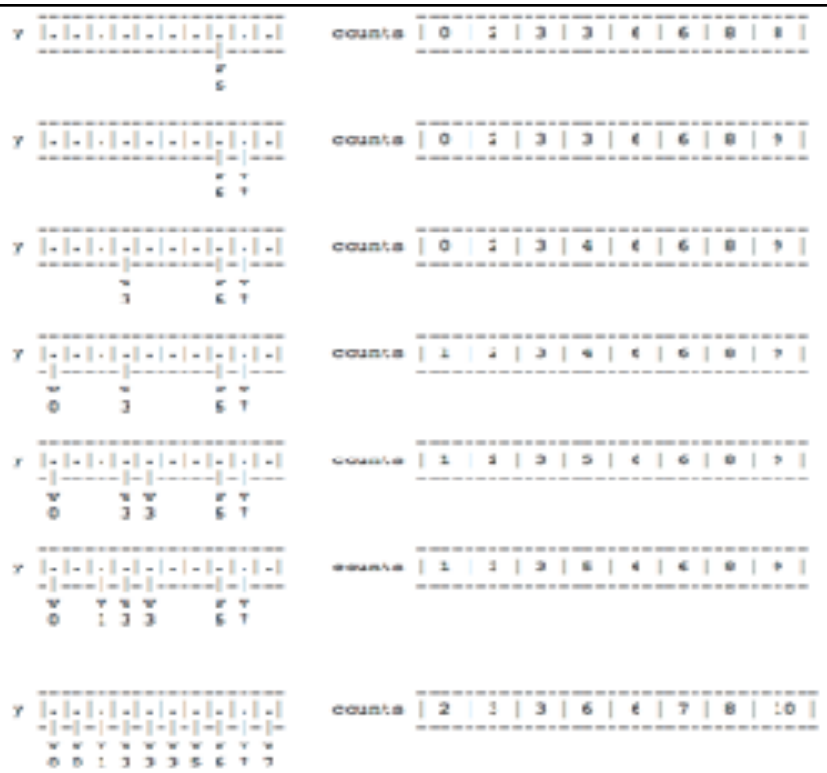
Counting Sort + Radix Sort

Counting Sort

<table>
<tr>
<td>Code</td>
<td>Good for Arrays
<pre>
(1) for (i = 0; i < x.length; i++) :
      counts[x[i].key]++;
(2) total = 0;
    for (j = 0; j < counts.length; j++) :
      c = counts[j];
      counts[j] = total;
      total = total + c;
(3) for (i = 0; i < x.length; i++) :
      y[counts[x[i].key]] = x[i];
      counts[x[i].key]++;
OTHER:
public static int[] countingSort(int[] keys, int whichDigit) :
    int[] count = new int[(keys.length)];
    int divisor = (int) Math.pow(RADIX, whichDigit);
    for (int i = 0; i != keys.length;) :
      for (int j = 0; j < RADIX; j++) :
        for (int k = 0; k < keys.length; k++) :
          if ((keys[k]/divisor & (Length of key - 1)) == j) :
            count[i] = keys[k];
            i++;
    return count;
</pre></td>
</tr>
<tr>
<td>Algorithm</td>
<td>1. Begin by counting the keys in x.
2. Do a scan of the counts array so that counts[i] contains the number of keys less than i.
3. Walk through the array x and copy each item to its final position in output array y. When you copy an item with key k, you must increment counts[k] to make sure that the next item with key k goes into the next slot.</td>
</tr>
</table>

<table>
<tr><td>Example
With Only Keys</td><td>If items are alone keys without values, we can use counting sort. Point is to keep count of the number of copies of a key:
Input: 6 7 3 0 3 1 5 0 3 7

Representation:
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>counts</td><td>2</td><td>1</td><td>0</td><td>3</td><td>0</td><td>1</td><td>1</td><td>2</td></tr>
</table>

Output: 0 0 1 3 3 3 5 6 7 7.</td></tr>
<tr><td>Example
With Keys & Values</td><td>  </td></tr>
<tr><td>Run Time</td><td>O(q + n) = O(num. of queues/buckets/counts + num. of items)
If q is in O(n) = num of possible keys isn't much larger than the num of items.</td></tr>
</table>

Radix Sort

Note	Good for when number of keys is >> number of items.
Algorithm	1. Start with the last place digit (usually ones place) from smallest to largest 2. Sort by each digit going right (tens, hundreds, thousands, etc) the same way. This works because Radix Sort is **Stable.** Sort 2 digits, q = Radix 10^2 buckets; 3 digits, q = radix 10^3 buckets.
Example	Sort on 1s: 771 721 822 955 405 5 925 825 777 28 829 Sort on 10s: 405 5 721 822 925 825 28 829 955 771 777 Sort on 100s: 5 28 405 721 771 777 822 825 829 925 955
Run Time	O(n + (b / log n) n) **Number of Passes:** O((n + q) ceiling(b / $\log_2$q)).

Analyses

Asymptotic Analysis

Big-Oh (O)

- Big-Oh notation compares how quickly two functions grow as n -> ∞.
- n be the size of input; T(n) is a function. f(n) be another function (a basic function from the table) [T(n) is the algorithm's running time in milliseconds]
- Let O(f(n)) is the SET of ALL functions T(n) that satisfy:

  There exist positive constants c and N such that, for all n >= N, T(n) <= c f(n)

- Gives **Upper Bound** or **Worst Case. (Your algorithm is at least this good!)**

Note: N and c are rather arbitrary, you must find them to ensure the inequality is satisfied.

Note: Confused about logs? $\log_a b = x \Longrightarrow a^x = b$

(In CS we assume a = 2, known as base 2)

n -> ∞	Function	English Name
	O(1)	constant
is a subset of	O(log n)	logarithmic
is a subset of	$O(\log^2 n)$ aka $O(\log n)^2$	log squared
is a subset of	O(sqrt(n)) aka O(root(n))	root n
is a subset of	O(n)	linear
is a subset of	O(n log n)	n times log n
is a subset of	$O(n^2)$	quadratic
is a subset of	$O(n^3)$	cubic
is a subset of	$O(n^4)$	quartic
is a subset of	$O(2^n)$	exponential
is a subset of	$O(e^n)$	bigger exponential
is a subset of	O(n!)	factorial

Remember: Constants can be important!

Omega (Ω)

- Reverse of Big-Oh
- Omega(f(n)) is the set of all functions T(n) that satisfy:
  There exist positive constants d and N such that, for all n >= N, T(n) >= d f(n)
- Gives **Lower Bound** or **Best Case. (Your algorithm is at least this bad.)**
- **Examples: 2n is in Ω(n) BECAUSE n is in O(2n).**
  n^2 is in Ω(n) BECAUSE n is in O(n^2).

Theta (Θ)

- Theta(f(n)) is the set of all functions that are in both of O(f(n)) and Ω(f(n)).
- Theta is symmetric:
  - If f(n) is in Θ(g(n)), then g(n) is in Θ(f(n)).
  - If n^3 is in Θ($3n^3$ - n^2), and 3 n^3 - n^2 is in Θ(n^3).
  - n^3 is not in Θ(n), and n is not in Θ(n^3)
- Theta is NOT easy to calculate, it can be complicated.

letter	bound	growth
(Theta) Θ	upper and lower, tight[1]	equal[2]
(big-oh) O	upper, tightness unknown	less than or equal[3]
(small-oh) o	upper, not tight	less than
(big omega) Ω	lower, tightness unknown	greater than or equal
(small omega) ω	lower, not tight	greater than

algorithm	performance
o(n)	< n
O(n)	≤ n
Θ(n)	= n
Ω(n)	≥ n
ω(n)	> n

Remember: "O" is NOT a synonym for "worst-case running time," and "Ω" is not a synonym for "best-case running time." The function has to be specified. Constants are Important.

Common Complexities

Searching

Heaps	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(log(n))	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Binomial Heap	-	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Data Structures

Algorithm	Data Structure	Time Complexity		Space Complexity
		Average	Worst	Worst
Depth First Search (DFS)	Graph of |V| vertices and |E| edges	-	O(|E| + |V|)	O(|V|)
Breadth First Search (BFS)	Graph of |V| vertices and |E| edges	-	O(|E| + |V|)	O(|V|)
Binary search	Sorted array of n elements	O(log(n))	O(log(n))	O(1)
Linear (Brute Force)	Array	O(n)	O(n)	O(1)
Shortest path by Dijkstra, using a Min-heap as priority queue	Graph with |V| vertices and |E| edges	O((|V| + |E|) log |V|)	O((|V| + |E|) log |V|)	O(|V|)
Shortest path by Dijkstra, using an unsorted array as priority queue	Graph with |V| vertices and |E| edges	O(|V|^2)	O(|V|^2)	O(|V|)
Shortest path by Bellman-Ford	Graph with |V| vertices and |E| edges	O(|V||E|)	O(|V||E|)	O(|V|)

Graphs

Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

\|V\| denotes the number of vertices; \|E\| denotes the number of edges.

Algorithm	Data Structure	Time Complexity			Worst Case Auxiliary Space Complexity
		Best	Average	Worst	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)	O(1)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	Array	O(n+k)	O(n+k)	O(n^2)	O(nk)
Radix Sort	Array	O(nk)	O(nk)	O(nk)	O(n+k)

Heaps

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	-	O(1)	O(n)	O(n)	-	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)

Searching

Amortised Analysis

Averaging Method (Example)

Note	**Amortized analysis is an upper bound.** **it's the average performance of each operation in the worst case.** Any operation that doesn't resize the table takes O(1) time. ≤ 1 second.
Example	Take a hash table n : number of items in hash table N: number of buckets One second to insert new item, n++, and check if n=N. Is so, double size from N to 2N, taking 2N seconds. Ensure load factor n/N < 1. Suppose new hash table has n = 0, N =1. After i inserts, n ≤ i. N must be power of 2, can't be ≤ n. Thus N is smallest power of 2 > n, which is ≤ 2n. Total seconds for all resizing operations is: 2+ 4+ 8 + … + N/4 + N/2 + N = 2N-2 Cost of i inserts is ≤ i + 2N - 2 seconds <------ N ≤ 2n ≤ 2i ≤ 5i - 2 seconds Average time of inset ≤ (5i - 2) / i = 5 - (2/i) seconds which exists in O(1). Amortized Running Time of Insert Exists O(1). Worst-Case Time of Insert Exists O(n).

Accounting Method (Example)

Invariant	The Bank Balance Must **Never Fall Below Zero**!
Note	Dollar: Unit of time to execute slowest constant time computation in our algorithm. **Each operation has:** - An Amortized Cost: # of dollars we "charge" to do operation. (fixed fiction of n) - An Actual Cost: Actual # of O(1) time computations. (varies wildly) **When amortized cost > actual cost:** Extra $$ saved in bank to be spent on later operations. **When actual > amortized:** withdraw $$ to pay.
Example (Hash Tables)	Every operations costs $1 actual time + resizing (if any). - insert doubles table size if n = N AFTER new item is inserted, n++. - taking $2N to resize to 2N buckets. - Load factor always < 1. - remove halves table if n = N/4 AFTER item removed, n--, - taking $N to resize to N/2 buckets. - Load factor always > 1/4. Either way, a hash table that has just been resized has n = N/2. **Amortized Costs:** insert: $5 remove: $5 find: $1 At anytime, we know $$ in bank from n & N. Every insert/remove costs one actual dollar, puts $4 in the bank. There must be 4|n - N/2| dollars saved. Function of the data structure; Does NOT depend on History. insert() resized table if n reaches N. THere are ≥ $2N in the bank. Resizing from N to 2N buckets cost $2N. remove() resizes table if n drops to N/4 -> THere are ≥ $N in the bank. Resizing from N to N/2 buckets costs $N. The Bank Balance Thus Never Falls Below Zero! -> amortized costs are valid, are in O(1).

Randomised Analysis

Note	Like Amortized Analysis except over infinite number of runs.
Expected Values	E[*Variable*] = The **Average Running Time** a call to the code makes.
Linearity of Expectation	**E[T]** = E[X + Y] **= E[X] + E[Y] (always true, even if Y depends on X)** The *Total Expected* time is the sum of each *Average Running Time* of the code. This holds **whether or not** X and Y **are independent**.
Example (Hash Table)	• Hash table uses chaining (no duplicate keys). • Each key is mapped to a random hash code. If key is the same as already existing, replace it anyways. • We run find(k), and the key *k* hashes to a bucket *b*. • Bucket b has only one entry with key *k*, so the cost of the search is $1, plus another $1 for **every** entry stored in bucket b whose key is not *k*. If there are n keys in the table besides k: • Let V_1, V_2, ..., Vn be random variables such that for each key k_i, the variable V_i = 1 if key k_i hashes to bucket b, and V_i is zero otherwise. **The total cost for find(k)**: T = 1 + V1 + ... + Vn **The expected cost of find(k)**: E[T] = 1 + E[V_1] + ... + E[V_n] If there are N buckets and the hash code is random: E[V_i] = 1/N E[T] = 1 + n/N So if n/N is kept below some constant c as n grows, find operations cost expected O(1) time

<table>
<tr><td>Example
(Quicksort)</td><td colspan="2">
• There are at least floor(n/4) keys.
• The greatest floor(n/4) keys are bad pivots, and the other keys are good pivots.
• Since there are at most n/2 bad pivots, the probability of choosing a bad pivot is <= 0.5.

1/4 - 3/4 split or better is a Good Pivot.
Key k will go into a subset containing at most 3/4 of the keys, sorted recursively. Bad pivots might just put everything in one subset.

• D(n) is a random variable equal to the lowest depth at which key k appears when we sort n keys.
• D(n) varies from run to run, but we can guesstimate.

E[D(n)] <= 1 + 0.5 E[D(n)] + 0.5 E[D(3n / 4)].

Multiplying by two and subtracting E[D(n)] from both sides gives:
E[D(n)] <= 2 + E[D(3n / 4)].

The base cases for this recurrence are D(0) = 0 and D(1) = 0.
E[D(n)] <= 2 $\log_{4/3} n$.

O(log n) levels of the recursion tree and O(log n) from partitioning work. Sum is still O(log n) work for each of the n keys.
Quicksort runs in expected O(n log n) time.
</td></tr>
<tr><td>Example
(Quickselect)</td><td colspan="2">
P(n) be a random variable equal to the total number of keys partitioned, summed over all the partitioning steps.
Running time: Theta(P(n)).

We choose a good pivot at least half the time, so
E[P(n)] <= n + 0.5 E[P(n)] + 0.5 E[P(3n / 4)],

Solution: E[P(n)] <= 8n.
Running time of Quickselect on n keys is in O(n).
</td></tr>
<tr><td rowspan="5">Amortized
Time
vs.
Expected
Time</td><td>Splay trees</td><td>O(log n) amortized time / operation</td></tr>
<tr><td>Disjoint sets (tree-based)</td><td>O(alpha(f + u, u)) amortized time</td></tr>
<tr><td>Quicksort</td><td>O(n log n) expected time</td></tr>
<tr><td>Quickselect</td><td>Theta(n)</td></tr>
<tr><td>Hash tables</td><td>Theta(1)</td></tr>
</table>

Garbage Collection

Prime Directive	Garbage collection's prime directive is to never sweep up an object your program might possibly use or inspect again (live).
Roots	• Are any object references your program can access **directly**, without going through another object. • Every local variable (also parameters) in every stack frame on the program stack is a root if it is a reference. • Primitive types like ints are not roots; only references are. • Every class variable (*static* field) that is a reference is a root.
Live Objects	Objects should not be garbage collected if **(recursively)**: • it is referenced by a root (obviously), or • it is referenced by a field in another live object. Run DFS on all the roots and discard node objects that remain unvisited.
Note	Java is a tricksy bastard and hides where it keeps objects. Never. Trust. It.
Mark & Sweep GC	1. The *mark* phase does a DFS from every root, and marks all the live objects. 2. The *sweep* phase does a pass over all the objects in memory. 3. Each object that was not marked as being live is garbage, and is thusly EXTERMINATED.
References	A reference isn't a pointer. A reference is a *handle* which is a pointer to a pointer. 
Copying GC	**Advantage:** Fast. **Disadvantage:** Half the amount of heap memory to your program.

Old Gen vs. New Gen (BATTLE!)	Old objects last longer, so the old generation uses a compacting **mark-and-sweep collector**, because speed is not critical, but memory efficiency might be. Tends to remain **compact**. The young generation has **Eden**, and it is the space where all objects are born, and most die. When Eden fills up (rare), it is GC and the surviving objects are copied into one of two *survivor spaces*. If **a lot of objects** survive Eden, **the survivor spaces** expand to make room for additional objects. Objects move back and forth between the two survivor spaces until they move to the old generation. **Types of GC:** **Minor Collections:** happens frequently, only affects the young generation, saves time, **Major Collections:** happens less often but cover all the objects in memory. **Problem:** Suppose a young object is live only because an old object references it. How does the minor collection find this out, if it doesn't search the old generation? **Solution:** References from old objects to young objects tend to be rare, because old objects are set in their ways and don't change much. JVM keeps a special table of them (updates whenever such a reference is created). The table of references is **added to the roots** of the young gen's copying collector.