

# ***List of Transparencies***

## **Chapter 1      Primitive Java    1**

- A simple first program    2
- The eight primitive types in Java    3
- Program that illustrates operators    4
- Result of logical operators    5
- Examples of conditional and looping constructs    6
- Layout of a `switch` statement    7
- Illustration of method declaration and calls    8

## **Chapter 2      References    9**

- An illustration of a reference: The `Point` object stored at memory location 1000 is referenced by both `point1` and `point3`. The `Point` object stored at memory location 1024 is referenced by `point2`. The memory locations where the variables are stored are arbitrary    10
- The result of `point3=point2`: `point3` now references the same object as `point2`    11
- Simple demonstration of arrays    12
- Array expansion: (a) starting point: `a` references 10 integers; (b) after step 1: `original` references the 10 integers; (c) after steps 2 and 3: `a` references 12 integers, the first 10 of which are copied from `original`; (d) after `original` exits scope, the original array is unreferenced and can be reclaimed    13
- Common standard run-time exceptions    14
- Common standard checked exceptions    15
- Simple program to illustrate exceptions    16
- Illustration of the `throws` clause    17
- Program that demonstrates the string tokenizer    18
- Program to list contents of a file    19

## **Chapter 3      Objects and Classes    20**

A complete declaration of an `IntCell` class 21  
IntCell members: `read` and `write` are accessible, but `storedValue` is hidden 22  
A simple test routine to show how `IntCell` objects are accessed 23  
IntCell declaration with *javadoc* comments 24  
*javadoc* output for `IntCell` 25  
A minimal `Date` class that illustrates constructors and the `equals` and `toString` methods 26  
Packages defined in this text 27  
A class `Exiting` with a single static method, which is part of the package `Supporting` 28  
Aliasing example 29  
Aliasing fixed 29  
Example of a static initializer 30

#### **Chapter 4 Inheritance 31**

Part of the `Exception` hierarchy 32  
General layout of public inheritance 33  
Constructor for new exception class `Underflow`; uses `super` 34  
Partial overriding 35  
The hierarchy of shapes used in an inheritance example 36  
Summary of `final`, `static`, `abstract`, and other methods 37  
Programmer responsibilities for derived class 38  
Basic action of insertion sort (shaded part is sorted) 39  
Closer look at action of insertion sort (dark shading indicates sorted area; light shading is where new element was placed) 40  
Basics of Interfaces 41  
Generic `MemoryCell` class; implemented via inheritance 42  
Using the generic `MemoryCell` class 43

#### **Chapter 5 Algorithm Analysis 44**

Running times for small inputs 45  
Running time for moderate inputs 46  
Functions in order of increasing growth rate 47  
The subsequences used in Theorem 5.2 48  
The subsequences used in Theorem 5.3. The sequence from  $p$  to  $q$  has sum at most that of the subsequence from  $i$  to  $q$ . On the left, the sequence from  $i$  to  $q$  is itself not the maximum (by Theorem 5.2). On the right, the sequence from  $i$  to  $q$  has already been seen. 49  
Growth rates defined 50  
Meanings of the various growth functions 51  
Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms 52  
Empirical running time for  $N$  binary searches in an  $N$ -item array 53

#### **Chapter 6 Data Structures 54**

Stack model: input to a stack is by `push`, output is by `top`, deletion is by `pop` 55

Sample stack program; output is	
Contents: 4 3 2 1 0	<b>56</b>
Queue model: input is by enqueue, output is by getFront, deletion is by dequeue	<b>57</b>
Sample queue program; output is	
Contents: 0 1 2 3 4	<b>58</b>
Link list model: inputs are arbitrary and ordered, any item may be output, and iteration is supported, but this data structure is not time-efficient	<b>59</b>
Sample list program; output is	
Contents: 4 3 2 1 0 end	<b>60</b>
A simple linked list	<b>61</b>
A tree	<b>62</b>
Expression tree for $(a+b) * (c-d)$	<b>63</b>
Binary search tree model; the binary search is extended to allow insertions and deletions	<b>64</b>
Sample search tree program;	
output is Found Becky; Mark not found;	<b>65</b>
The hash table model: any named item can be accessed or deleted in essentially constant time	<b>66</b>
Sample hash table program;	
output is Found Becky;	<b>67</b>
Priority queue model: only the minimum element is accessible	<b>68</b>
Sample program for priority queues;	
output is Contents: 0 1 2 3 4	<b>69</b>
Summary of some data structures	<b>70</b>

## **Chapter 7      Recursion    71**

Stack of activation records	<b>72</b>
Ruler	<b>73</b>
Fractal star outline	<b>74</b>
Trace of the recursive calculation of the Fibonacci numbers	<b>75</b>
Divide-and-conquer algorithms	<b>76</b>
Dividing the maximum contiguous subsequence problem into halves	<b>77</b>
Trace of recursive calls for recursive maximum contiguous subsequence sum algorithm	<b>78</b>
Basic divide-and-conquer running time theorem	<b>79</b>
General divide-and-conquer running time theorem	<b>80</b>
Some of the subproblems that are solved recursively in Figure 7.15	<b>81</b>
Alternative recursive algorithm for coin-changing problem	<b>82</b>

## **Chapter 8      Sorting Algorithms    83**

Examples of sorting	<b>84</b>
Shellsort after each pass, if increment sequence is {1, 3, 5}	<b>85</b>
Running time (milliseconds) of the insertion sort and Shellsort with various increment sequences	<b>86</b>
Linear-time merging of sorted arrays (first four steps)	<b>87</b>
Linear-time merging of sorted arrays (last four steps)	<b>88</b>
Basic quicksort algorithm	<b>89</b>

The steps of quicksort **90**  
Correctness of quicksort **91**  
Partitioning algorithm: pivot element 6 is placed at the end **92**  
Partitioning algorithm:  $i$  stops at large element 8;  $j$  stops at small element 2 **92**  
Partitioning algorithm: out-of-order elements 8 and 2 are swapped **92**  
Partitioning algorithm:  $i$  stops at large element 9;  $j$  stops at small element 5 **92**  
Partitioning algorithm: out-of-order elements 9 and 5 are swapped **92**  
Partitioning algorithm:  $i$  stops at large element 9;  $j$  stops at small element 3 **92**  
Partitioning algorithm: swap pivot and element in position  $i$  **92**  
Original array **93**  
Result of sorting three elements (first, middle, and last) **93**  
Result of swapping the pivot with next-to-last element **93**  
Median-of-three partitioning optimizations **94**  
Quickselect algorithm **95**

## **Chapter 9      Randomization   96**

Distribution of lottery winners if expected number of winners is 2 **97**  
Poisson distribution **98**

## **Chapter 10     Fun and Games   99**

Sample word search grid **100**  
Brute-force algorithm for word search puzzle **101**  
Alternate algorithm for word search puzzle **102**  
Improved algorithm for word search puzzle; incorporates a prefix test **103**  
Basic minimax algorithm **104**  
Alpha-beta pruning: After  $H_{2A}$  is evaluated,  $C_2$ , which is the minimum of the  $H_2$ 's, is at best a draw. Consequently, it cannot be an improvement over  $C_1$ . We therefore do not need to evaluate  $H_{2B}$ ,  $H_{2C}$ , and  $H_{2D}$ , and can proceed directly to  $C_3$  **105**  
Two searches that arrive at identical positions **106**

## **Chapter 11     Stacks and Compilers   107**

Stack operations in balanced symbol algorithm **108**  
Steps in evaluation of a postfix expression **109**  
Associativity rules **110**  
Various cases in operator precedence parsing **111**  
Infix to postfix conversion **112**  
Expression tree for  $(a+b) * (c-d)$  **113**

## **Chapter 12     Utilities   114**

A standard coding scheme **115**  
Representation of the original code by a tree **116**  
A slightly better tree **117**

Optimal prefix code tree **118**  
Optimal prefix code **119**  
Huffman's algorithm after each of first three merges **120**  
Huffman's algorithm after each of last three merges **121**  
Encoding table (numbers on left are array indices) **122**

### **Chapter 13      Simulation    123**

The Josephus problem **124**  
Sample output for the modem bank simulation: 3 modems; a dial-in is attempted every minute; average connect time is 5 minutes; simulation is run for 19 minutes **125**  
Steps in the simulation **126**  
Priority queue for modem bank after each step **127**

### **Chapter 14      Graphs and Paths   128**

A directed graph **129**  
Adjacency list representation of graph in Figure 14.1; nodes in list  $i$  represent vertices adjacent to  $i$  and the cost of the connecting edge **130**  
Information maintained by the Graph table **131**  
Data structures used in a shortest path calculation, with input graph taken from a file: shortest weighted path from A to C is: A to B to E to D to C (cost 76) **132**  
If  $w$  is adjacent to  $v$  and there is a path to  $v$ , then there is a path to  $w$  **133**  
Graph after marking the start node as reachable in zero edges **134**  
Graph after finding all vertices whose path length from the start is 1 **135**  
Graph after finding all vertices whose shortest path from the start is 2 **136**  
Final shortest paths **137**  
How the graph is searched in unweighted shortest path computation **138**  
Eyeball is at  $v$ ;  $w$  is adjacent;  $D_w$  should be lowered to 6 **139**  
If  $D_v$  is minimal among all unseen vertices and all edge costs are nonnegative, then it represents the shortest path **140**  
Stages of Dijkstra's algorithm **141**  
Graph with negative cost cycle **142**  
Topological sort **143**  
Stages of acyclic graph algorithm **144**  
Activity-node graph **145**  
Top: Event node graph; Bottom: Earliest completion time, latest completion time, and slack (additional edge item) **146**

### **Chapter 15      Stacks and Queues   147**

How the stack routines work: empty stack, push(A), push(B), pop **148**  
Basic array implementation of the queue **149**  
Array implementation of the queue with wraparound **150**  
Linked list implementation of the stack **151**  
Linked list implementation of the queue **152**

enqueue operation for linked-list-based implementation **153**

## **Chapter 16      Linked Lists    154**

Basic linked list **155**

Insertion into a linked list: create new node (*tmp*), copy in *x*, set *tmp*'s next reference, set current's next reference **156**

Deletion from a linked list **157**

Using a header node for the linked list **158**

Empty list when header node is used **159**

Doubly linked list **160**

Empty doubly linked list **161**

Insertion into a doubly linked list by getting new node and then changing references in order indicated **162**

Circular doubly linked list **163**

## **Chapter 17      Trees        164**

A tree **165**

Tree viewed recursively **166**

First child/next sibling representation of tree in Figure 17.1 **167**

Unix directory **168**

The directory listing for tree in Figure 17.4 **169**

Unix directory with file sizes **170**

Trace of the `size` method **171**

Uses of binary trees: left is an expression tree and right is a Huffman coding tree **172**

Result of a naive merge operation **173**

Aliasing problems in the merge operation; *T1* is also the current object **174**

Recursive view used to calculate the size of a tree:  $S_T = S_L + S_R + 1$  **175**

Recursive view of node height calculation:  $H_T = \max(H_L + 1, H_R + 1)$  **176**

Preorder, postorder, and inorder visitation routes **177**

Stack states during postorder traversal **178**

## **Chapter 18      Binary Search Trees    179**

Two binary trees (only the left tree is a search tree) **180**

Binary search trees before and after inserting 6 **181**

Deletion of node 5 with one child, before and after **182**

Deletion of node 2 with two children, before and after **183**

Using the `size` data field to implement `findKth` **184**

Balanced tree on the left has a depth of  $\log N$ ; unbalanced tree on the right has a depth of  $N-1$  **185**

Binary search trees that can result from inserting a permutation 1, 2, and 3; the balanced tree in the middle is twice as likely as any other **186**

Two binary search trees: the left tree is an AVL tree, but the right tree is not (unbalanced nodes are darkened) **187**

Minimum tree of height  $H$  **188**

- Single rotation to fix case 1 **189**
- Single rotation fixes AVL tree after insertion of 1 **190**
- Symmetric single rotation to fix case 4 **191**
- Single rotation does not fix case 2 **192**
- Left-right double rotation to fix case 2 **193**
- Double rotation fixes AVL tree after insertion of 5 **194**
- Left-right double rotation to fix case 3 **195**
- Red-black tree properties **196**
- Example of a red-black tree; insertion sequence is 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55) **197**
- If  $S$  is black, then a single rotation between the parent and grandparent, with appropriate color changes, restores property 3 if  $X$  is an outside grandchild **198**
- If  $S$  is black, then a double rotation involving  $X$ , the parent, and the grandparent, with appropriate color changes, restores property 3 if  $X$  is an inside grandchild **199**
- If  $S$  is red, then a single rotation between the parent and grandparent, with appropriate color changes, restores property 3 between  $X$  and  $P$  **200**
- Color flip; only if  $X$ 's parent is red do we continue with a rotation **201**
- Color flip at 50 induces a violation; because it is outside, a single rotation fixes it **202**
- Result of single rotation that fixes violation at node 50 **203**
- Insertion of 45 as a red node **204**
- Deletion:  $X$  has two black children, and both of its sibling's children are black; do a color flip **205**
- Deletion:  $X$  has two black children, and the outer child of its sibling is red; do a single rotation **206**
- Deletion:  $X$  has two black children, and the inner child of its sibling is red; do a double rotation **207**
- $X$  is black and at least one child is red; if we fall through to next level and land on a red child, everything is good; if not, we rotate a sibling and parent **208**
- AA-tree properties **209**
- AA-tree resulting from insertion of 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55, 35 **210**
- skew is a simple rotation between  $X$  and  $P$  **211**
- split is a simple rotation between  $X$  and  $R$ ; note that  $R$ 's level increases **212**
- After inserting 45 into sample tree; consecutive horizontal links are introduced starting at 35 **213**
- After split at 35; introduces a left horizontal link at 50 **213**
- After skew at 50; introduces consecutive horizontal nodes starting at 40 **213**
- After split at 40; 50 is now on the same level as 70, thus inducing an illegal left horizontal link **214**
- After skew at 70; this introduces consecutive horizontal links at 30 **214**
- After split at 30; insertion is complete **214**
- When 1 is deleted, all nodes become level 1, introducing horizontal left links **215**
- Five-ary tree of 31 nodes has only three levels **216**
- B-tree of order 5 **217**
- B-tree properties **218**
- B-tree after insertion of 57 into tree in Figure 18.70 **219**
- Insertion of 55 in B-tree in Figure 18.71 causes a split into two leaves **220**
- Insertion of 40 in B-tree in Figure 18.72 causes a split into two leaves and then a split of the parent node **221**
- B-tree after deletion of 99 from Figure 18.73 **222**

**Chapter 19 Hash Tables 223**

Linear probing hash table after each insertion 224

Illustration of primary clustering in linear probing (middle) versus no clustering (top) and the less significant secondary clustering in quadratic probing (bottom); long lines represent occupied cells; Load factor is 0.7 225

Quadratic probing hash table after each insertion (note that the table size is poorly chosen because it is not a prime number) 226

**Chapter 20 A Priority Queue: The Binary Heap 227**

A complete binary tree and its array representation 228

Heap-order property 229

Two complete trees (only the left tree is a heap) 230

Attempt to insert 14, creating the hole and bubbling the hole up 231

The remaining two steps to insert 14 in previous heap 232

Creation of the hole at the root 233

Next two steps in `deleteMin` 234Last two steps in `deleteMin` 235

Recursive view of the heap 236

Initial heap (left); after `percolateDown(7)` (right) 237After `percolateDown(6)` (left); after `percolateDown(5)` (right) 237After `percolateDown(4)` (left); after `percolateDown(3)` (right) 238After `percolateDown(2)` (left); after `percolateDown(1)` and `fixHeap` terminates (right) 238

Marking of left edges for height-one nodes 239

Marking of first left and subsequent right edge for height-two nodes 239

Marking of first left and subsequent two right edges for height-three nodes 240

Marking of first left and subsequent right edges for height-four node 240

(Max) Heap after `fixHeap` phase 241

Heapsort algorithm (in principle) 242

Heap after first `deleteMax` 243Heap after second `deleteMax` 243

Initial tape configuration 244

Distribution of length 3 runs onto two tapes 245

Tapes after first round of merging (run length = 6) 245

Tapes after second round of merging (run length = 12) 245

Tapes after third round of merging 245

Initial distribution of length 3 runs onto three tapes 246

After one round of three-way merging (run length = 9) 246

After two rounds of three-way merging 246

Number of runs using polyphase merge 247

Example of run construction 248

**Chapter 21 Splay Trees 249**

Rotate-to-root strategy applied when node 3 is accessed 250



Insertion of 4 using rotate-to-root **251**  
 Sequential access of items takes quadratic time **252**  
 Zig case (normal single rotation) **253**  
 Zig-zag case (same as a double rotation); symmetric case omitted **253**  
 Zig-zig case (this is unique to the splay tree); symmetric case omitted **253**  
 Result of splaying at node 1 (three zig-zigs and a zig) **254**  
 The `remove` operation applied to node 6: First 6 is splayed to the root, leaving two subtrees; a `findMax` on the left subtree is performed, raising 5 to the root of the left subtree; then the right subtree can be attached (not shown) **255**  
 Top-down splay rotations: zig (top), zig-zig (middle), and zig-zag (bottom) **256**  
 Simplified top-down zig-zag **257**  
 Final arrangement for top-down splaying **258**  
 Steps in top-down splay (accessing 19 in top tree) **259**

## Chapter 22 Merging Priority Queues 260

Simplistic merging of heap-ordered trees; right paths are merged **261**  
 Merging of skew heap; right paths are merged, and the result is made a left path **262**  
 Skew heap algorithm (recursive viewpoint) **263**  
 Change in heavy/light status after a merge **264**  
 Abstract representation of sample pairing heap **265**  
 Actual representation of above pairing heap; dark line represents a pair of references that connect nodes in both directions **265**  
 Recombination of siblings after a `deleteMin`; in each merge the larger root tree is made the left child of the smaller root tree: (a) the resulting trees; (b) after the first pass; (c) after the first merge of the second pass; (d) after the second merge of the second pass **266**  
`compareAndLink` merges two trees **267**

## Chapter 23 The Disjoint Set Class 268

Definition of equivalence relation **269**  
 A graph  $G$  (left) and its minimum spanning tree **270**  
 Kruskal's algorithm after each edge is considered **271**  
 The nearest common ancestor for each request in the pair sequence  $(x,y)$ ,  $(u,z)$ ,  $(w,x)$ ,  $(z,w)$ ,  $(w,y)$ , is  $A$ ,  $C$ ,  $A$ ,  $B$ , and  $y$ , respectively **272**  
 The sets immediately prior to the return from the recursive call to  $D$ ;  $D$  is marked as visited and  $NCA(D, v)$  is  $v$ 's anchor to the current path **273**  
 After the recursive call from  $D$  returns, we merge the set anchored by  $D$  into the set anchored by  $C$  and then compute all  $NCA(C, v)$  for nodes  $v$  that are marked prior to completing  $C$ 's recursive call **274**  
 Forest and its eight elements, initially in different sets **275**  
 Forest after `union` of trees with roots 4 and 5 **275**  
 Forest after `union` of trees with roots 6 and 7 **276**  
 Forest after `union` of trees with roots 4 and 6 **276**  
 Forest formed by union-by-size, with size encoded as a negative number **277**  
 Worst-case tree for  $N=16$  **278**

Forest formed by union-by-height, with height encoded as a negative number **279**  
Path compression resulting from a `find(14)` on the tree in Figure 23.12 **280**  
Ackermann's function and its inverse **281**  
Accounting used in union-find proof **282**  
Actual partitioning of ranks into groups used in the union-find proof **283**