

List of Transparencies

Chapter 1 Pointers, Arrays, and Structures 1

Pointer illustration 2

Result of `*Ptr=10` 3

Uninitialized pointer 4

(a) Initial state; (b) `Ptr1=Ptr2` starting from initial state; (c) `*Ptr1=*Ptr2` starting from initial state 5

Memory model for arrays (assumes 4 byte int); declaration is `int A[3]; int i;` 6

Some of the string routines in `<string.h>` 7

Two ways to allocate arrays; one leaks memory 8

Memory reclamation 9

Array expansion: (a) starting point: A2 points at 10 integers; (b) after step 1: Original points at the 10 integers; (c) after steps 2 and 3: A2 points at 12 integers, the first 10 of which are copied from Original; (d) after step 4: the 10 integers are freed 10

Pointer arithmetic: `X=&A[3]; Y=X+4` 11

First eight lines from `prof` for program 12

First eight lines from `prof` with highest optimization 12

Student structure 13

Illustration of a shallow copy in which only pointers are copied 14

Illustration of a simple linked list 15

Chapter 2 Objects and Classes 16

A complete declaration of a `MemoryCell` class 17

`MemoryCell` members: `Read` and `Write` are accessible, but `StoredValue` is hidden 18

A simple test routine to show how `MemoryCell` objects are accessed 19

A more typical `MemoryCell` declaration in which interface and implementation are separated 20

Interface for `BitArray` class 21

BitArray members 22
Construction examples 23

Chapter 3 Templates 24

Basic action of insertion sort (shaded part is sorted) 25
Closer look at action of insertion sort (dark shading indicates sorted area; light shading is where new element was placed) 26
Typical layout for template interface and member functions 27

Chapter 4 Inheritance 28

General layout of public inheritance 29
Access rules that depend on what M 's visibility is in the base class 30
Friendship is not inherited 31
Vector and BoundedVector classes with calls to `operator[]` that are done automatically and correctly 32
Vector and BoundedVector classes 33
The hierarchy of shapes used in an inheritance example 34
Summary of nonvirtual, virtual, and pure virtual functions 35
Programmer responsibilities for derived class 36

Chapter 5 Algorithm Analysis 37

Running times for small inputs 38
Running time for moderate inputs 39
Functions in order of increasing growth rate 40
The subsequences used in Theorem 5.2 41
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. 42
Meanings of the various growth functions 44
Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms 45
Empirical running time for N binary searches in an N -item array 46

Chapter 6 Data Structures 47

Sample stack program; output is
 Contents: 4 3 2 1 0 48
Stack model: input to a stack is by Push, output is by Top, deletion is by Pop 49
Sample queue program; output is
 Contents: 0 1 2 3 4 50
Queue model: input is by Enqueue, output is by Front, deletion is by Dequeue 51
Sample list program; output is Contents: 4 3 2 1 0 end 52
Link list model: inputs are arbitrary and ordered, any item may be output, and iteration is support-

ed, but this data structure is not time-efficient	53
A simple linked list	54
A tree	55
Expression tree for $(a+b) * (c-d)$	56
Sample search tree program;	
output is Found Becky; Mark not found;	57
Binary search tree model; the binary search is extended to allow insertions and deletions	58
Sample hash table program;	
output is Found Becky; Mark not found;	59
The hash table model: any named item can be accessed or deleted in essentially constant time	60
Sample program for priority queues;	
output is Contents: 0 1 2 3 4	61
Priority queue model: only the minimum element is accessible	62
Summary of some data structures	63

Chapter 7 Recursion 64

Stack of activation records	65
Trace of the recursive calculation of the Fibonacci numbers	66
Divide-and-conquer algorithms	67
Dividing the maximum contiguous subsequence problem into halves	68
Trace of recursive calls for recursive maximum contiguous subsequence sum algorithm	69
Basic divide-and-conquer running time theorem	70
General divide-and-conquer running time theorem	71
Some of the subproblems that are solved recursively in Figure 7.15	72
Alternative recursive algorithm for coin-changing problem	73

Chapter 8 Sorting Algorithms 74

Examples of sorting	75
Deriving the relational and equality operators from <code>operator<</code>	76
Shellsort after each pass, if increment sequence is {1, 3, 5}	77
Running time (milliseconds) of the insertion sort and Shellsort with various increment sequences	78
Linear-time merging of sorted arrays (first four steps)	79
Linear-time merging of sorted arrays (last four steps)	80
Basic quicksort algorithm	81
The steps of quicksort	82
Correctness of quicksort	83
Partitioning algorithm: pivot element 6 is placed at the end	84
Partitioning algorithm: <i>i</i> stops at large element 8; <i>j</i> stops at small element 2	84
Partitioning algorithm: out-of-order elements 8 and 2 are swapped	84
Partitioning algorithm: <i>i</i> stops at large element 9; <i>j</i> stops at small element 5	84
Partitioning algorithm: out-of-order elements 9 and 5 are swapped	84
Partitioning algorithm: <i>i</i> stops at large element 9; <i>j</i> stops at small element 3	84
Partitioning algorithm: swap pivot and element in position <i>i</i>	84

Original array 85
Result of sorting three elements (first, middle, and last) 85
Result of swapping the pivot with next to last element 85
Median-of-three partitioning optimizations 86
Quickselect algorithm 87
Using an array of pointers to sort 88
Data structure used for in-place rearrangement 89

Chapter 9 Randomization 90

Distribution of lottery winners if expected number of winners is 2 91
Poisson distribution 92

Chapter 10 Fun and Games 93

Sample word search grid 94
Brute-force algorithm for word search puzzle 95
Alternate algorithm for word search puzzle 96
Improved algorithm for word search puzzle; incorporates a prefix test 97
Basic minimax algorithm 98
Alpha-beta pruning: After H2A is evaluated, C2, which is the minimum of the H2's, is at best a draw. Consequently, it cannot be an improvement over C1. We therefore do not need to evaluate H2B, H2C, and H2D, and can proceed directly to C3 99
Two searches that arrive at identical positions 100

Chapter 11 Stacks and Compilers 101

Stack operations in balanced symbol algorithm 102
Steps in evaluation of a postfix expression 103
Associativity rules 104
Various cases in operator precedence parsing 105
Infix to postfix conversion 106
Expression tree for $(a+b) * (c-d)$ 107

Chapter 12 Utilities 108

A standard coding scheme 109
Representation of the original code by a tree 110
A slightly better tree 111
Optimal prefix code tree 112
Optimal prefix code 113
Huffman's algorithm after each of first three merges 114
Huffman's algorithm after each of last three merges 115
Encoding table (numbers on left are array indices) 116
IdNode data members: Word is a String; Lines is a pointer to a Queue 117
The object in the tree is a copy of the temporary; after the insertion is complete, the destructor is

called for the temporary **118**

Chapter 13 Simulation 119

The Josephus problem **120**

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 **121**

Steps in the simulation **122**

Priority queue for modem bank after each step **123**

Chapter 14 Graphs and Paths 124

A directed graph **125**

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 **126**

Information maintained by the Graph table **127**

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) **128**

Graph after marking the start node as reachable in zero edges **129**

Graph after finding all vertices whose path length from the start is 1 **130**

Graph after finding all vertices whose shortest path from the start is 2 **131**

Final shortest paths **132**

How the graph is searched in unweighted shortest path computation **133**

Eyeball is at v ; w is adjacent; D_w should be lowered to 6 **134**

If D_v is minimal among all unseen vertices and all edge costs are nonnegative, then it represents the shortest path **135**

Stages of Dijkstra's algorithm **136**

Graph with negative cost cycle **137**

Topological sort **138**

Stages of acyclic graph algorithm **139**

Activity-node graph **140**

Top: Event node graph; Bottom: Earliest completion time, latest completion time, and slack (additional edge item) **141**

Chapter 15 Stacks and Queues 142

How the stack routines work: empty stack, $\text{Push}(A)$, $\text{Push}(B)$, Pop **143**

Basic array implementation of the queue **144**

Array implementation of the queue with wraparound **145**

Linked list implementation of the stack **146**

Linked list implementation of the queue **147**

Enqueue operation for linked-list-based implementation **148**

Chapter 16 Linked Lists 149

Basic linked list **150**

Insertion into a linked list: create new node (Tmp), copy in X, set Tmp's next pointer, set Current's next pointer **151**
 Deletion from a linked list **152**
 Using a header node for the linked list **153**
 Empty list when header node is used **154**
 Doubly linked list **155**
 Empty doubly linked list **156**
 Insertion into a doubly linked list by getting new node and then changing pointers in order indicated **157**
 Circular doubly linked list **158**

Chapter 17 Trees 159

A tree **160**
 Tree viewed recursively **161**
 First child/next sibling representation of tree in Figure 17.1 **162**
 UNIX directory **163**
 The directory listing for tree in Figure 17.4 **164**
 UNIX directory with file sizes **165**
 Trace of the Size function **166**
 Uses of binary trees: left is an expression tree and right is a Huffman coding tree **167**
 Result of a naive Merge operation **168**
 Aliasing problems in the Merge operation; T1 is also the current object **169**
 Recursive view used to calculate the size of a tree: $ST = SL + SR + 1$ **170**
 Recursive view of node height calculation: $HT = \text{Max}(HL+1, HR+1)$ **171**
 Preorder, postorder, and inorder visitation routes **172**
 Stack states during postorder traversal **173**

Chapter 18 Binary Search Trees 174

Two binary trees (only the left tree is a search tree) **175**
 Binary search trees before and after inserting 6 **176**
 Deletion of node 5 with one child, before and after **177**
 Deletion of node 2 with two children, before and after **178**
 Using the Size data member to implement FindKth **179**
 Balanced tree on the left has a depth of $\log N$; unbalanced tree on the right has a depth of $N-1$ **180**
 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 **181**
 Two binary search trees: the left tree is an AVL tree, but the right tree is not (unbalanced nodes are darkened) **182**
 Minimum tree of height H **183**
 Single rotation to fix case 1 **184**
 Single rotation fixes AVL tree after insertion of 1 **185**
 Symmetric single rotation to fix case 4 **186**
 Single rotation does not fix case 2 **187**
 Left-right double rotation to fix case 2 **188**

- Double rotation fixes AVL tree after insertion of 5 **189**
- Left-right double rotation to fix case 3 **190**
- Red black tree properties **191**
- Example of a red black tree; insertion sequence is 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55) **192**
- 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 **193**
- 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 **194**
- If *S* is red, then a single rotation between the parent and grandparent, with appropriate color changes, restores property 3 between *X* and *P* **195**
- Color flip; only if *X*'s parent is red do we continue with a rotation **196**
- Color flip at 50 induces a violation; because it is outside, a single rotation fixes it **197**
- Result of single rotation that fixes violation at node 50 **198**
- Insertion of 45 as a red node **199**
- Deletion: *X* has two black children, and both of its sibling's children are black; do a color flip **200**
- Deletion: *X* has two black children, and the outer child of its sibling is red; do a single rotation **201**
- Deletion: *X* has two black children, and the inner child of its sibling is red; do a double rotation **202**
- 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 **203**
- AA-tree properties **204**
- AA-tree resulting from insertion of 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55, 35 **205**
- Skew is a simple rotation between *X* and *P* **206**
- Split is a simple rotation between *X* and *R*; note that *R*'s level increases **207**
- After inserting 45 into sample tree; consecutive horizontal links are introduced starting at 35 **208**
- After Split at 35; introduces a left horizontal link at 50 **208**
- After Skew at 50; introduces consecutive horizontal nodes starting at 40 **208**
- After Split at 40; 50 is now on the same level as 70, thus inducing an illegal left horizontal link **209**
- After Skew at 70; this introduces consecutive horizontal links at 30 **209**
- After Split at 30; insertion is complete **209**
- When 1 is deleted, all nodes become level 1, introducing horizontal left links **210**
- Five-ary tree of 31 nodes has only three levels **211**
- B-tree of order 5 **212**
- B-tree properties **213**
- B-tree after insertion of 57 into tree in Figure 18.70 **214**
- Insertion of 55 in B-tree in Figure 18.71 causes a split into two leaves **215**
- Insertion of 40 in B-tree in Figure 18.72 causes a split into two leaves and then a split of the parent node **216**
- B-tree after deletion of 99 from Figure 18.73 **217**

Chapter 19 Hash Tables 218

- Linear probing hash table after each insertion **219**
- Quadratic probing hash table after each insertion (note that the table size is poorly chosen because it is not a prime number) **220**

Chapter 20 A Priority Queue: The Binary Heap 221

A complete binary tree and its array representation 222

Heap order property 223

Two complete trees (only the left tree is a heap) 224

Attempt to insert 14, creating the hole and bubbling the hole up 225

The remaining two steps to insert 14 in previous heap 226

Creation of the hole at the root 227

Next two steps in DeleteMin 228

Last two steps in DeleteMin 229

Recursive view of the heap 230

Initial heap (left); after PercolateDown(7) (right) 231

After PercolateDown(6) (left); after PercolateDown(5) (right) 231

After PercolateDown(4) (left); after PercolateDown(3) (right) 232

After PercolateDown(2) (left); after PercolateDown(1) and FixHeap terminates (right) 232

Marking of left edges for height one nodes 233

Marking of first left and subsequent right edge for height two nodes 233

Marking of first left and subsequent two right edges for height three nodes 234

Marking of first left and subsequent right edges for height 4 node 234

(Max) Heap after FixHeap phase 235

Heapsort algorithm 236

Heap after first DeleteMax 237

Heap after second DeleteMax 237

Initial tape configuration 238

Distribution of length 3 runs onto two tapes 239

Tapes after first round of merging (run length = 6) 239

Tapes after second round of merging (run length = 12) 239

Tapes after third round of merging 239

Initial distribution of length 3 runs onto three tapes 240

After one round of three-way merging (run length = 9) 240

After two rounds of three-way merging 240

Number of runs using polyphase merge 241

Example of run construction 242

Chapter 21 Splay Trees 243

Rotate-to-root strategy applied when node 3 is accessed 244

Insertion of 4 using rotate-to-root 245

Sequential access of items takes quadratic time 246

Zig case (normal single rotation) 247

Zig-zag case (same as a double rotation); symmetric case omitted 247

Zig-zig case (this is unique to the splay tree); symmetric case omitted 247

Result of splaying at node 1 (three zig-zigs and a zig) 248

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) 249

Top-down splay rotations: zig (top), zig-zig (middle), and zig-zag (bottom) **250**

Simplified top-down zig-zag **251**

Final arrangement for top-down splaying **252**

Steps in top-down splay (accessing 19 in top tree) **253**

Chapter 22 Merging Priority Queues 254

Simplistic merging of heap-ordered trees; right paths are merged **255**

Merging of skew heap; right paths are merged, and the result is made a left path **256**

Skew heap algorithm (recursive viewpoint) **257**

Change in heavy/light status after a merge **258**

Abstract representation of sample pairing heap **259**

Actual representation of above pairing heap; dark line represents a pair of pointers that connect nodes in both directions **259**

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 **260**

`CompareAndLink` merges two trees **261**

Chapter 23 The Disjoint Set Class 262

Definition of equivalence relation **263**

A graph G (left) and its minimum spanning tree **264**

Kruskal's algorithm after each edge is considered **265**

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 **266**

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 **267**

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 **268**

Forest and its eight elements, initially in different sets **269**

Forest after `Union` of trees with roots 4 and 5 **269**

Forest after `Union` of trees with roots 6 and 7 **270**

Forest after `Union` of trees with roots 4 and 6 **270**

Forest formed by union-by-size, with size encoded as a negative number **271**

Worst-case tree for $N=16$ **272**

Forest formed by union-by-height, with height encoded as a negative number **273**

Path compression resulting from a `Find(14)` on the tree in Figure 23.12 **274**

Ackerman's function and its inverse **275**

Accounting used in union-find proof **276**

Actual partitioning of ranks into groups used in the union-find proof **277**