**Tree Data Model – The Basic Parts**

**Graph Concepts**

**Node (vertex, point)**

**denoted by a point**

**Labeled Node**

**a value (or name) for each node**

**Edge (branch)**

**denoted by a line between 2 nodes**

**Directed Edge (arc)**

**an oriented line (from one node to another)**

**orientation can be explicit or implicit**

**In and Out Degrees**

**Path; Path Length; Cycle (cyclic, acyclic)**

**Connected (path from any node to any other)**

**Tree Definitions**

**Tree Definition # 1 (unoriented)**

**Root (a designated node) with no parent**

**Every node, except the root, has an edge to its unique parent**

**Connected in that all nodes have a unique path through parents to the root**

**Tree Definition # 2 (oriented from root to leaves)**

**Root – only node with in-degree of 0**

**Others have in-degree of 1 from their parents**

**Connected in that all have path from root**

**Tree Definition # 3**

**Any single node r denotes a tree with r as its root**

**If r is a new node and T1, …,Tn are trees with roots r1, …,rn, then T is a tree where T has root r and r has an edge to each of r1, …,rn. Assumes that r and all nodes of T1,…,Tn are distinct.**

**Tree Notation**

**Root**

**Parent**

**Child**

**Ancestor (proper)**

**Descendant (proper)**

**Sibling**

**Leaf (no children, out-degree of 0)**

**Frontier (catenation of labels of leaves from left to right)**

**Interior (at least one child, out-degree > 0)**

**Subtree (a node, its proper descendants with arcs)**

**Path – Unique heritage between node and root**

**Depth or level – Path length between node and root**

**Height – longest path length between node and a leaf**

**Height of tree is height of root**

**Some trees are Oriented**

**(parent-child or child-parent)**

**Some trees are Ordered**

**If Ordered then have 1st, 2nd, etc. Child**

**lower numbered siblings are to left of higher**

**If Binary then Left and Right Child**

# Also left and right subtrees (which are themselves binary)

**On ordered, if a and y are siblings and x is to left of y, then all of x’s descendants are to left of all of y’s descendants**

**Binary Tree Recursive Definition**

**A binary tree is**

**either empty // Note this is an extension to tree notion**

**or consists of a node (root) with**

**data (of some appropriate type)**

**a left child (which is also a binary tree)**

**a right child (which is also a binary tree)**

**Above lends itself to recursive divide and conquer algorithms and inductive proofs of correctness and complexity.**

## Algorithm form

**if (isEmpty()) { easy case }**

**else { process node, left child and right child in appropriate order }**

**Proof form (S(j) is property for binary trees with height j)**

**Basis: Show S(0). This is the statement of property for tree with just the root node (some proofs start with empty tree).**

**Inductive hypothesis: Assume S(k) true, 0 ≤ k < N**

**Inductive step: Prove S(N). Induction depends on fact that children of root are binary trees with height less than N.**

**Data Structure for Trees – the Obvious**

**class Tree {**

**private Object data;**

**private Tree[] children;**

**final static private int MaxChildren = 10; // or whatever**

### public Tree (Object data) {

**this.data = data;**

**this.children = new Tree[MaxChildren];**

**}**

**}**

**Given a node, we can access the i-th child in O(1) time.**

**Space utilization is quite poor as lots of children are possible, but rare.**

**Can use Vector with improve this.**

**This is usually a good structure for binary trees (ones for which every node has at most 2 children. In this case, we use fields left and right, rather than an array.**

**Expression trees are commonly represented this way since unary operators are rare and binary are common.**

**CHALLENGE: For an expression with n binary operators and no unary operators, how many child pointers are NULL; how many are non-NULL?**

**Data Structure for Trees – Less Obvious**

**An alternative data structure is**

**Leftmost-child – Right Sibling Pointers**

**class Tree {**

**private Object data;**

**private Tree leftChild;**

**private Tree rightSibling;**

**}**

**This is just a reorientation of the tree.**

**Data Structure for Trees – Unobvious**

**Another alternative data structure is**

**Parent Pointers**

**class Tree {**

**private Object data;**

**private Tree parent;**

**}**

**This changes the direction of reference so that a child points to its parent, but parents have no handle on their children. This is very space efficient since the only NULL pointer is in the root, and there is only one pointer per node. This data structure does not support many of the useful ways we traverse trees, but it is superb for certain applications, e.g., trees representing equivalence classes.**

**A fourth data structure is parent, firstChild and nextSibling parts.**

**This hedges nearly all bets.**

#### Binary Tree

**class BinaryTree {**

**Object data;**

**BinaryTree left = null;**

**BinaryTree right = null;**

**BinaryTree parent = null; // might not have this**

### BinaryTree (Object data, BinaryTree) {

**this.data = data;**

**this.parent = parent;**

**}**

**BinaryTree (Object data) {**

**this.data = data;**

**this.parent = null;**

**}**

**}**

**Common Tree Traversal Template**

**public void traverse ( ) {**

**action0;**

**Tree child = leftChild;**

**if (child == null) return;**

**child.traverse();**

**action1;**

**child = child.rightSibling;**

**if (child == null) return;**

**child.traverse();**

**action2;**

**child = child.rightSibling;**

**…**

**if (child == null) return;**

**child.traverse();**

**actionK;**

**}**

**} // traverse**

**Binary Expression Trees**

**Labels are operators or atomic operands**

### All operators are interior nodes. Common labels are +, –, \*, / and unary –

**All atomic operands are leaves**

**Inductive Definition**

**A single atomic operand is represented by a single node binary tree**

**If E and F are expressions represented by binary trees S and T, respectively, and op is a binary operator, then (E op F) is represented by the binary tree U consisting a new root labeled op with left child S and right child T.**

**If E is an expression represented by tree S and op is a prefix unary operator, then (op E) is represented by the binary tree T consisting of a new root labeled op with empty left child and right child S.**

**Example Uses**

**Interpreters and expression trees**

**Compilers and expression trees**

**Optimizing compilers and expression trees**

**Preorder Tree Traversal (Print)**

**public void preorder ( ) {**

**System.out.println(data + " ");**

**Tree child = leftChild;**

**while (child!=null) {**

**child.preorder();**

**child = child.rightSibling;**

**}**

**}**

**Assume that we use this on an expression tree with the labels being operators or simple one-letter variable names. The expression (~A - B) \* ( C / (D + E) ) is represented as**



**The preorder would print \* – ~ A B / C + D E**

**Preorder BinaryTree Traversal (Print)**

**public void preorder ( ) {**

**System.out.println(data + " ");**

**if (left!=null) left.preorder();**

**if (right!=null) right.preorder();**

**}**

**Again, assume that we use this on an expression tree with the labels being operators or simple one-letter variable names. The expression (~A - B) \* ( C / (D + E) ) is represented as**



**The preorder would print \* – ~ A B / C + D E**

**Postorder BinaryTree Traversal (Print)**

**public void postorder ( ) {**

**if (left!=null) left.postorder();**

**if (right!=null) right.postorder();**

**System.out.println(data + " ");**

**}**

**The expression (~A - B) \* ( C / (D + E) )**



**has postorder A ~ B – C D E + / \***

**Each of preorder and postorder provides an unambiguous way to denote expressions without parentheses. This is more compact than infix notation and can lead to very efficient expression evaluations.**

**Postorder IntExpressionTree Traversal (Evaluation)**

**class IntExpressionTree {**

**int data;**

**char op; // op or blank if a value**

**BinaryTree left = null;**

**BinaryTree right = null;**

**…**

**public int eval ( ) { // preorder peek helps; operation is performed postorder**

**if (op.equals(' ')) return data;**

**else {**

**switch(op) {**

**case '~' : return –right.eval(); break;**

**case '+' : return left.eval() + right.eval(); break;**

**case '-' : return left.eval() - right.eval(); break;**

**case '\*' : return left.eval() \* right.eval(); break;**

**case '/' : return left.eval() / right.eval(); break;**

**}**

**}**

**}**

**}**



**Evaluates as –28.**

**Inductive Proof of Evaluation**

**“Structural induction” is process of proving property of trees S(T) by:**

**BASIS: Show that property S(T) holds for any tree T with just one node (its root). Such a tree has height 0. Sometimes we do for null tree which has height -1.**

**INDUCTION: Assume T is a tree with a root r and children c1,…, ck, k≥1.   
Let T1, …,Tk be the subtrees rooted at c1, …,ck, respectively. The inductive step is to assume S(T1), …,S(Tk) are all true and prove S(T).**

**This really can be viewed as complete induction, where we are assuming the property for all tree with height n n or less, and proving it for a tree with height n+1.**

**Computing the Height of a Tree**

**class Tree {**

**private Object data;**

**private Tree leftChild;**

**private Tree rightSibling;**

**public int computeHT( ) {**

**int height = 0;**

**Tree child = leftChild;**

**while (child != null) {**

**height = Math.max(height, child.computeHT( )+1);**

**child = child.rightSibling;**

**}**

**return height;**

**} // computeHT**

**Inductive Proof of Correctness:**

**S(t): when the computeHT message is sent to a non-null tree, t, it returns the height of the tree.**

**BASIS: If t has just one node then t has no children. The while test in computeHT fails on the first try and hence the correct value of 0 is returned.**

**INDUCTIVE HYPOTHESIS: Assume for arbitrary non-null tree t with children t1, … tk, that S(ti) for all 1≤i≤k.**

**More on the Height of a Tree**

**INDUCTIVE STEP: Suppose t is a tree with children t1, … tk, k≥1. Then t has at least 1 child and, by definition, the height of t is one more than the maximum of the heights of its children. By the inductive hypothesis, computeHT will correctly compute the heights of all of t's children. To see that the correct value is returned for t we need to show that the while loop sets height to one more than the maximum of the heights of t’s children. This requires an embedded inductive proof.**

**S'(i): After the while loop has completed i iterations, the value of height is 1 more than the largest of the heights of the first i children of t.**

**BASIS: S'(1). Height is 0 prior to the first execution of the loop. Thus the new value of height is max(0,child.computeHT( )+1), where child is the first child. Clearly this is one more than the height of the first child.**

**INDUCTIVE HYPOTHESIS: Assume S'(n), n≥1.**

**INDUCTIVE STEP: Show S’(n+1). If the height of the n+1 -st child is less than or equal to any of the previous, then its height+1 will be less or equal to the value of height and the iteration will make no changes, as required. If, on the other hand, the n+1 -st child has a height greater than any of the preceding children than its height+1 will exceed the current value of height and the max statement will set height to this new value, as required.**

**RETURNING TO THE STRUCTURAL INDUCTION: Since the while condition and the statement that assigns each successive rightSibling to child ensures that all subtrees are inspected, we can call upon the inner induction to assure that the value of height will be correct for t when we exit the while. The remaining statement just returns the value of height as computed.**

**Inorder and Binary Expression Trees**

**public void inorder ( ) {**

**System.out.println("(");**

**if (left!=null) left.preorder();**

**System.out.println(data);**

**if (right!=null) right.preorder();**

**System.out.println(")");**

**}**

**The expression (~A - B) \* ( C / (D + E) )**



**is printed as**

**( ( ( ~ ( A ) ) – ( B ) ) \* ( ( ( C ) / ( ( D ) + ( E ) ) ) ) )**

**The correct fully parenthesized version.**

**Code Generation from Expression Tree (binary ops)**

**class ExpressionTree {**

**private String label; // no label=null tree (leaf has 2 null children)**

**int height;**

**private ExpressionTree left;**

**private ExpressionTree right;**

**public int computeHT( ) {**

**if (label == null) height = -1;**

**else height = 1 + Math.max(left.computeHT( ),   
 right.computeHT( )));**

**return height;**

**} // computeHT**

**private void genCode(register : integer) {**

**if (codeGenerator.isOperator(label) {**

**left.genCode(register);**

**right.genCode(rightChild.height);**

**codeGenerator.genOp(label, rightChild.height, register);**

**} else codeGenerator.genLoad(label, register);**

**} // genCode**

**public void generateCode() {**

**computeHT(); // set heights**

**genCode(height); // start with register height**

**}**

**Code Generator Support Routines**

**public boolean isOperator(String s){**

**return (s.equal("+")) || (s.equal("-")) || (s.equal("\*")) || (s.equal("/"));**

**}**

**public void genOp(String s, int r1, int r2) {**

**// generate assembly language opcodes**

**switch ( s.charAt(0) ) {**

**case '+' : System.out.print("ADD ");**

**case '-' : System.out.print("SUB ");**

**case '\*' : System.out.print("MUL ");**

**case '/' : System.out.print("DIV ");**

**}**

**// follow opcode by operands**

**System.out.println ("R" + r1 + ", R" + r2);**

**} // genOp**

**public void genLoad(String s, int r) {**

**System.out.println ("LOAD " + s + " R" + r);**

**} // genLoad**

**An Example of Code Generator**



**1) LOAD #0, R3**

**2) LOAD A, R0**

**3) SUB R0, R3**

**4) LOAD B, R0**

**5) SUB R0, R3**

**6) LOAD C, R2**

**7) LOAD D, R1**

**8) LOAD #7, R0**

**9) ADD R0, R1**

**10) DIV R1, R2**

**11) MULT R2, R3**

**Register Usage Optimization**

**The method computeHT is very closely related to an algorithm used to make efficient use of registers when generating code.**

**Consider if we have a commutative operator like +. We can compute**

**exp1 + exp2**

**as originally specified, or as**

**exp2 + exp1**

**The choice can affect the registers needed to complete the computation. If we exceed the number of actual registers, then we have register spill. This is a situation in which we must copy register contents to memory, and then restore these values later.**

**CHALLENGE: When is to your benefit to take advantage of commutativity? In other words what criteria should be used to reduce register consumption?**

**Dictionary ADT**

**Dictionary:** An ADT that maintains a collection of comparable (totally ordered) elements and provides three services:

**void insert(Element x) –** inserts element x into Dictionary.

**void delete(Element x) –** deletes element x from Dictionary.

**boolean lookup(Element x) –** returns true or false depending on whether or not element x is in the Dictionary.

There are several abstract implementations that are useful for trees. Two of these are the **Binary Search Tree** (**BST**) and **Trie**, both based on the tree data model.

**BST –** A binary tree whose nodes contain words, such that

The word at each node of the tree is lexically greater than the words at all nodes in its left subtree and lexically less than the words at all nodes in its right subtree.

**Trie –** A tree representing a set of words, such that

The root of the tree has a null label.

Other nodes are labeled with a letter and a Boolean flag.

The set of words represented are those formed by concatenating the labels of all nodes in some path starting at a child of the root and continuing from child to child, stopping at some node whose flag is “true”.

**Dictionaries as BSTs**

Consider how we would represent the collection of words in the sentence

"this is the time for tiny things"

**BST** (here are 2)



**Dictionaries as Tries**

Consider again how we would represent the collection of words in the sentence

"this is the time for tiny things"

Trie (example is a bit of a cheat – in general we need end of word markers.)



**Balancing BSTs**

**AVL Trees**

**For each node t, |height(left(t)) – height(right(t))|<2**

**Maintaining Balance**

**Single rotation (left, left)**

****

**Single rotation (right, right)**

****

**AVL Example**

**Example of single rotation   
(left insert created heavier left side)**

****

**Nasty Cases**

**Single rotation on left, right moves imbalance to right, left**

****

**Double rotation comes to the rescue**

****

**Mirror image for right, left**

****

**AVL Insert**

/\*\*

\* Internal method to insert into a subtree.

\* **@param** x the item to insert.

\* **@param** t the node that roots the subtree.

\* **@return** the new root of the subtree.

\*/

**private** AvlNode<AnyType> insert(AnyType x,AvlNode<AnyType> t)

{

**if**( t == **null** )

**return** **new** AvlNode<AnyType>( x, **null**, **null** );

**int** compareResult = compare( x, t.element );

**if**( compareResult < 0 )

{

t.left = insert( x, t.left );

**if**( height( t.left ) - height( t.right ) == 2 )

**if**( compare( x, t.left.element ) < 0 )

t = rotateWithLeftChild( t );

**else**

t = doubleWithLeftChild( t );

}

**else** **if**( compareResult > 0 )

{

t.right = insert( x, t.right );

**if**( height( t.right ) - height( t.left ) == 2 )

**if**( compare( x, t.right.element ) > 0 )

t = rotateWithRightChild( t );

**else**

t = doubleWithRightChild( t );

}

**else** ; // Duplicate; do nothing

t.height = Math.max(height(t.left),height(t.right)) + 1;

**return** t;

}

**AVL Insert**

**private** AvlNode<AnyType> rotateWithLeftChild( AvlNode<AnyType> k2 ){

AvlNode<AnyType> k1 = k2.left;

k2.left = k1.right;

k1.right = k2;

k2.height = Math.max(height( k2.left ), height( k2.right ) ) + 1;

k1.height = Math.max(height( k1.left ), k2.height ) + 1;

**return** k1;

}



**private** AvlNode<AnyType> doubleWithLeftChild( AvlNode<AnyType> k3 ) {

k3.left = rotateWithRightChild( k3.left );

**return** rotateWithLeftChild( k3 );

}



**B+ Trees**

**5 – 5 Tree** 

**Example Starting State**



**Insert 57 (easy case as room in leaf bin)**



**Insert 55 (split but not propagation up tree)**



**Insert 40 (split propagates up tree)**

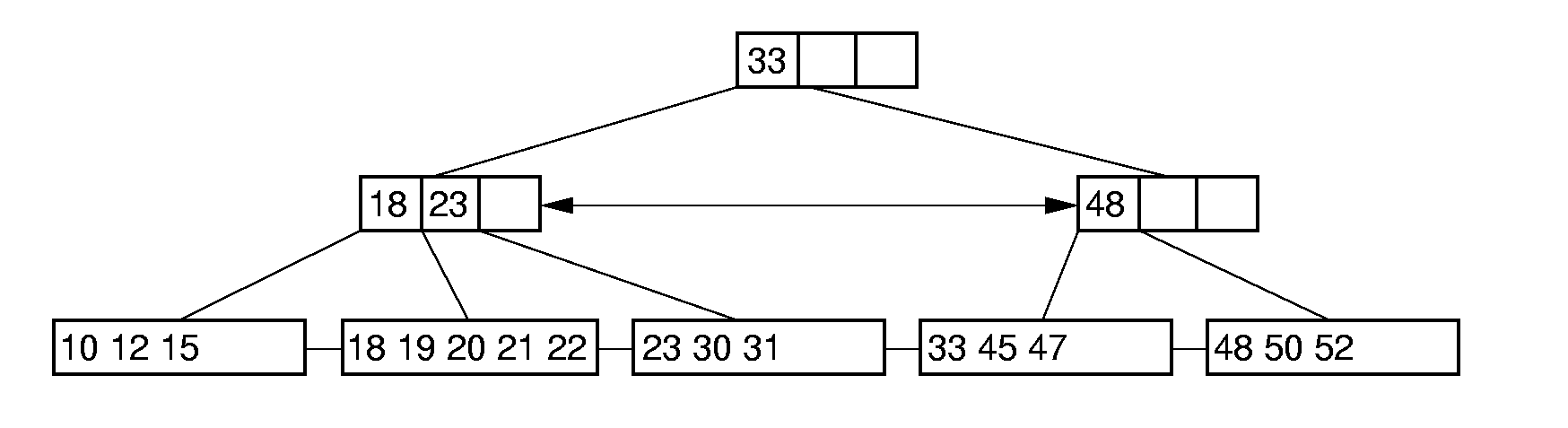


**Delete 99 (leads to merges)**



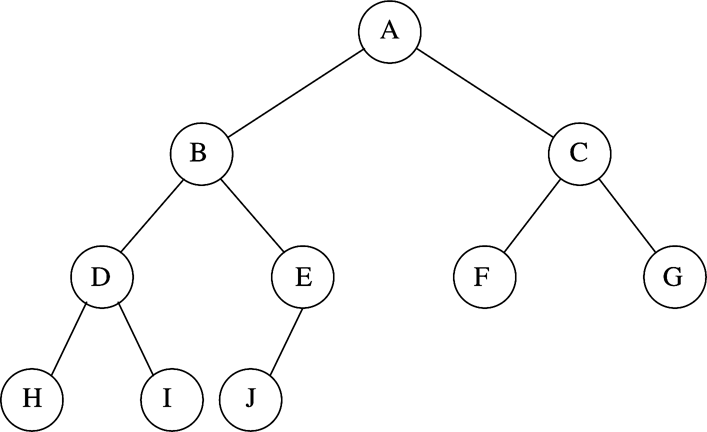
B+ uses doubly link list of nodes at each level

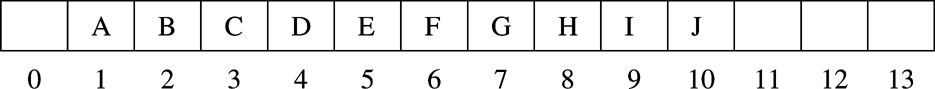
Example of 4-5



**Priority Queues**







**public** **class** BinaryHeap<AnyType **extends** Comparable<? **super** AnyType>>

{

**public** BinaryHeap( )

{ /\* See online code \*/ }

**public** BinaryHeap( **int** capacity )

{ /\* See online code \*/ }

**public** BinaryHeap( AnyType [ ] items )

{ /\* Figure 6.14 \*/ }

**public** **void** insert( AnyType x )

{ /\* Figure 6.8 \*/ }

**public** AnyType findMin( )

{ /\* See online code \*/ }

**public** AnyType deleteMin( )

{ /\* Figure 6.12 \*/ }

**public** **boolean** isEmpty( )

{ /\* See online code \*/ }

**public** **void** makeEmpty( )

{ /\* See online code \*/ }

**private** **static** **final** **int** DEFAULT\_CAPACITY = 10;

**private** **int** currentSize; // Number of elements in heap

**private** AnyType [ ] array; // The heap array

**private** **void** percolateDown( **int** hole )

{ /\* Figure 6.12 \*/ }

**private** **void** buildHeap( )

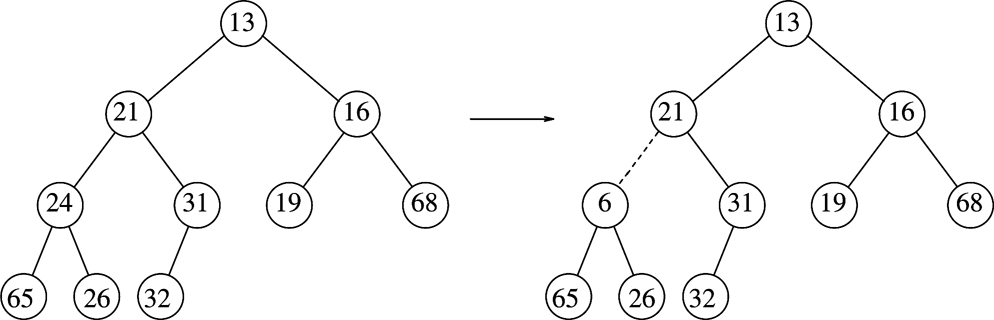
{ /\* Figure 6.14 \*/ }

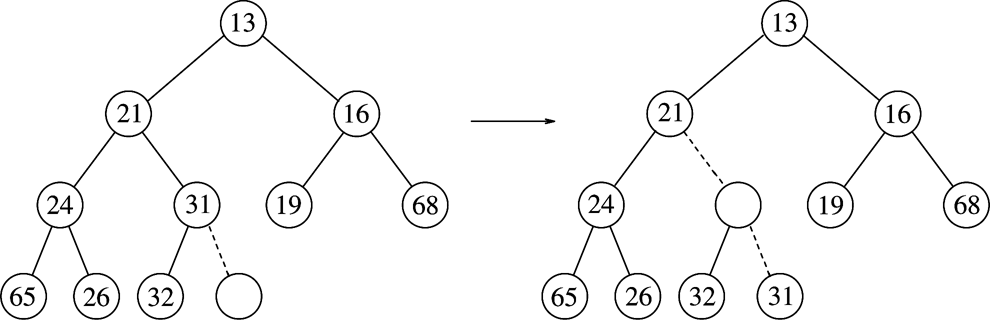
**private** **void** enlargeArray( **int** newSize )

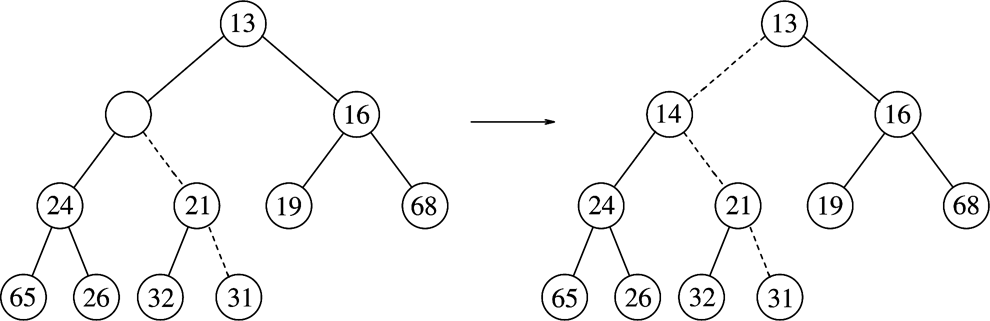
{ /\* See online code \*/ }

}

Insert value 14







**public** **void** insert( AnyType x )

{

**if**( currentSize == array.length - 1 )

enlargeArray( array.length \* 2 + 1 );

// Percolate up

**int** hole = ++currentSize;

**for**( ; hole > 1 && x.compareTo( array[ hole / 2 ] ) < 0;

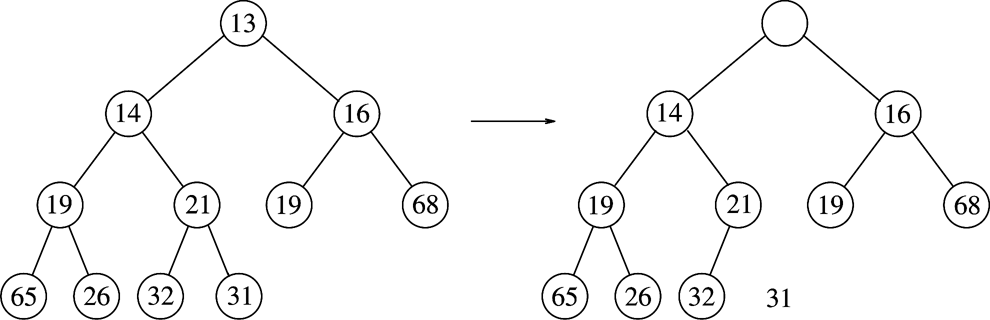
hole /= 2 )

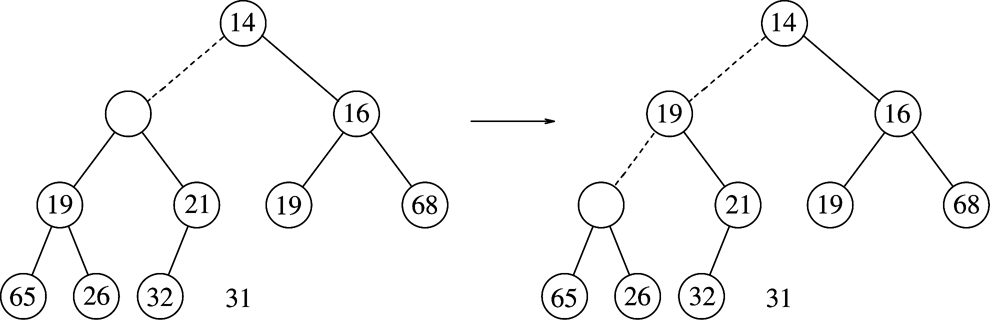
array[ hole ] = array[ hole / 2 ];

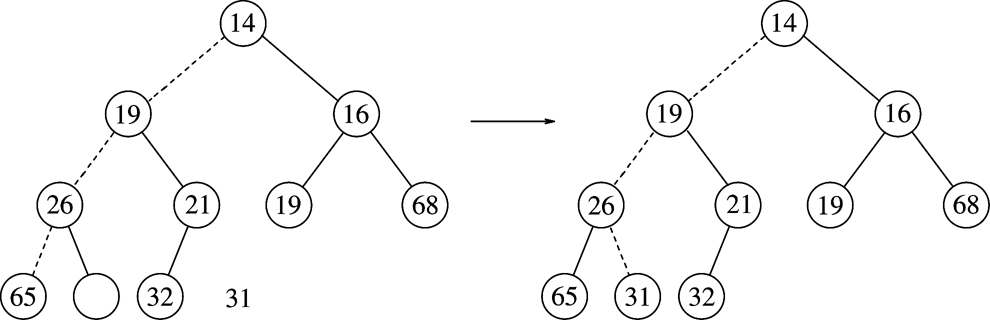
array[ hole ] = x;

}

deleteMin







**public** AnyType deleteMin( )

{

**if**( isEmpty( ) )

**throw** **new** UnderflowException( );

AnyType minItem = findMin( );

array[ 1 ] = array[ currentSize-- ];

percolateDown( 1 );

**return** minItem;

}

**private** **void** percolateDown( **int** hole )

{

**int** child;

AnyType tmp = array[ hole ];

**for**( ; hole \* 2 <= currentSize; hole = child )

{

child = hole \* 2;

**if**( child != currentSize &&

array[ child + 1 ].compareTo( array[ child ] ) < 0 )

child++;

**if**( array[ child ].compareTo( tmp ) < 0 )

array[ hole ] = array[ child ];

**else**

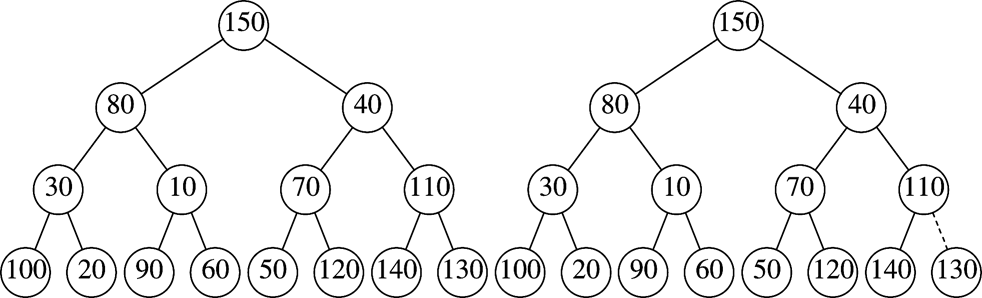
**break**;

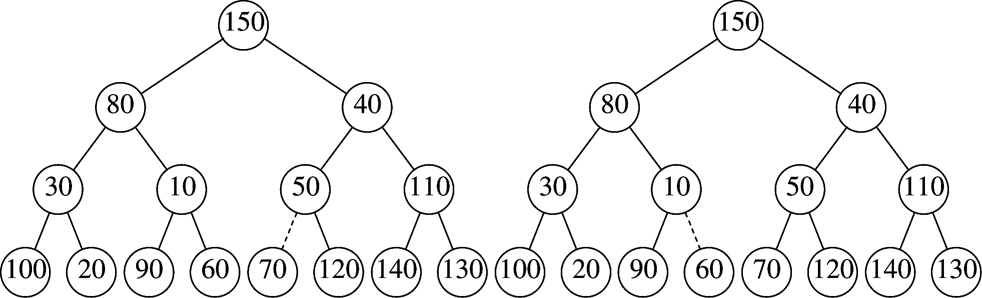
}

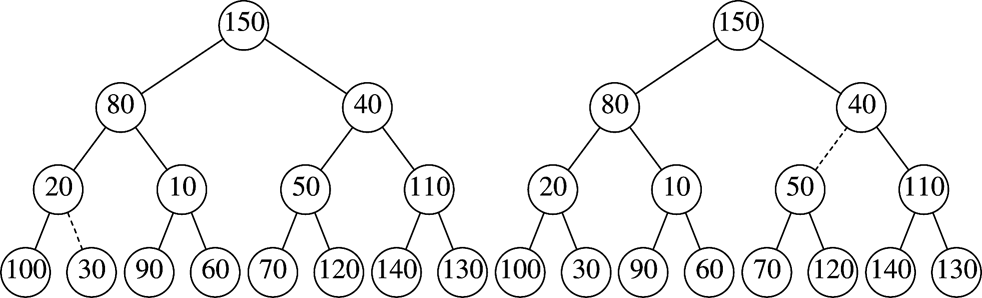
array[ hole ] = tmp;

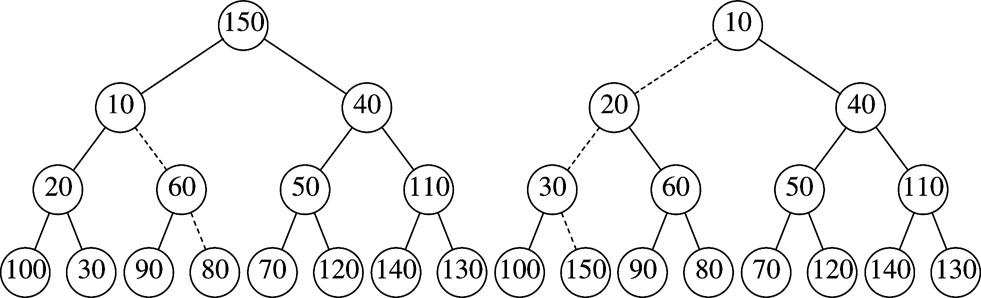
}

**heapify (buildHeap)**









**heapify (buildHeap) Implementation**

/\*\*

\* Construct the binary heap given an array of items.

\*/

**public** BinaryHeap( AnyType [ ] items )

{

currentSize = items.length;

array = (AnyType[]) **new** Comparable[( currentSize + 2 ) \* 11 / 10];

**int** i = 1;

**for**( AnyType item : items )

array[ i++ ] = item;

buildHeap( );

}

/\*\*

\* Establish heap order property from an arbitrary

\* arrangement of items. Runs in linear time.

\*/

**private** **void** buildHeap( )

{

**for**( **int** i = currentSize / 2; i > 0; i-- )

percolateDown( i );

}

**heapify (buildHeap) Correctness**

A balanced priority-ordered tree (BPOT) is a complete binary tree that satisfies the priority-ordered property (the root of each subtree is the highest priority item in that tree).

Prove the tree produced from buildHeap is a BPOT. Proof by induction on the subtree roots (j) skipped or visited directly (not recursively or iteratively) by the call to percolateDown in buildHeap. Here, we will count from j=1 to currentSize, starting our count from the last node in the list (currentsize is the root index).

**heapify (buildHeap) Correctness**

Basis (j=1 to ⎡currentSize / 2⎤): The leaves of the tree are not visited and each is a BPOT by definition of a single node tree.

IH(j=k, k≥⎡currentSize / 2⎤): Assume that all subtrees rooted at nodes through the k-th are BPOTs.

IH(j=k+1): When the (k+1)-st node is passed to percolateDown, its tree is balanced since we have made no structural changes and its subtrees are BPOTs by the IH. percolateDown, when called on the root of a tree whose subtrees are BPOTs produces a BPOT that includes all elements in the tree. Therefore IH(k+1) and the hypothesis is proven.**percolateUp and percolateDown Correctness**

percolateUp can be shown correct by a simple deductive argument. Essentially, if we consider its branch a list, percolateUp just inserts the new value placed at the end of this list into its proper position, moving all other (higher valued) items down to make room. When an item is moved down it maintains BPOT property because it does not change the structure and it alters each changed subtree to have a smaller (higher priority) root.

percolateDown can be proven correct by an inductive argument on the height of the tree with the new value at its root. Our hypothesis is then that percolateDown, given a tree whose left and right subtrees of the root are BPOTs, will swap items so the new element (initially at the root) is incorporated correctly into the BPOT.

**percolateUp and percolateDown Correctness**

Base (h=0): A tree with just one element is a BPOT.

IH(h=k, k≥0): Assume percolateDown works correctly for all trees of height less than or equal to k.

IS(h=k+1): The first iteration of percolateDown guarantees that the smallest item is at the root and, at worst, one of the subtrees of height at most k (either k or k-1) has a root value that is larger than one or more of its children nodes, each of which roots a BPOT. By the IH, subsequent iterations of percolateDown will restructure that subtree to be a BPOT that properly incorporates the new root and the entire tree will then be a BPOT the incorporates the new value.

**Heap Sort Analysis**

Naive:

Each percolateDown (bubbleDown) could require log2N moves, so   
buildHeap (heapify) takes N log2N

Intelligent:

The lower 1/2 of all nodes are never bubbled down.

The second 1/4 of all nodes can only bubble down 1 place.

The second 1/8 can only go down 2 places.

Only one element can bubble down log2N - 1 places.

**Heap Sort Analysis**

Analysis:

Can show time is

N (1/4 + 2/8 + 3/16 + 4/32 + 5/64 + … + k/2k+1 + …)

= N/2 (1/2 + 2/4 + 3/8 + 4/16 + 5/32 + … + k/2k + …)

One way to find closed form is

1/2 + 2/4 + 3/8 + 4/16 + 5/32 + … + k/2k + …

= 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + … + 1/2k + …

+ 1/4 + 1/8 + 1/16 + 1/32 + … + 1/2k + …

+ + 1/8 + 1/16 + 1/32 + … + 1/2k + …

+ …

= 1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + … + 1/2k + … = 2

so N/2 (1/2 + 2/4 + 3/8 + 4/16 + 5/32 + … + k/2k + …) = N/2 (2)= N

Another way is

S = 1/2 + 2/4 + 3/8 + 4/16 + 5/32 + … + k/2k + …

S/2 = 1/4 + 2/8 + 3/16 + 4/32 + 5/64 + … + k/2k+1 + …

S - S/2 = 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + … + 1/2k + … = 1

So S/2 = 1, S = 2, and the sum is once again N/2 (2) = N

Selection – find k-th smallest (largest)

heapify first – O(N)

deleteMin k times – O(k \* lg N)

Thus, using a heap we can find k-th priority item in O(N + k lg N).

This is better than O(N lg N) so long as k is not O(N). In fact, if k = O(N/lg N) then running time is O(N).

Of course if we use this for median, k is ⎡N/2⎤. and so the algorithm is Θ(N lg N) and we don’t do any better than a sort.

Selection – find k-th smallest (largest)

heapify first k elements – O(k)

for all remaining N-k elements

Compare next element, e, to last, hk, in heap.   
If e < hk, make e the k-th and percolateUp.

When done, the k-th smallest item is in the heap. In fact, it will be the largest in there and so will be one of the leaves.

Time = O(k + (N-k)lg k).

This algorithm is Θ(N lg N) if k is median.

**Disjoint Set Problem (Intentionally Lame Text Book Approach)**

/\*\*

\* Construct the disjoint sets object.

\* **@param** numElements the initial number of disjoint sets.

\*/

**public** DisjSets( **int** numElements )

{

s = **new** **int** [ numElements ];

**for**( **int** i = 0; i < s.length; i++ )

s[ i ] = -1;

}

/\*\*

\* Union two disjoint sets.

\* For simplicity, we assume root1 and root2 are distinct

\* and represent set names.

\* **@param** root1 the root of set 1.

\* **@param** root2 the root of set 2.

\*/

**Disjoint Set Problem (Intentionally Lame Text Book Approach)**

**public** **void** union( **int** root1, **int** root2 )

{

s[ root2 ] = root1;

}

**public** **int** find( **int** x )

{

**if**( s[ x ] < 0 )

**return** x;

**else**

**return** find( s[ x ] );

}

**Much Better Solution (Smart Compression Text Book Approach)**

**public** **void** union( **int** root1, **int** root2 )

{

**if**( s[ root2 ] < s[ root1 ] ) // root2 is deeper

s[ root1 ] = root2; // Make root2 new root

**else**

{

**if**( s[ root1 ] == s[ root2 ] )

s[ root1 ]--; // Update height if same

s[ root2 ] = root1; // Make root1 new root

}

}

**public** **int** find( **int** x )

{

**if**( s[ x ] < 0 )

**return** x;

**else**

**return** s[ x ] = find( s[ x ] );

}

**AbstractPartition (Disjoint Sets Contract)**

**package** COP3503H;

**public** **interface** AbstractPartition {

/\*

\* Unions the partition containing element1 with that containing element2.

\* Reduces number of partitions by one, provided the elements are not

\* already in same partition.

\*/

**void** union(**int** element1, **int** element2);

// Returns “representative element” in partition containing element.

**int** find(**int** element);

}

**ParentTree Class**

**package** COP3503H;

**class** ParentTree {

ParentTree parent = **null**;

**int** element = 0;

**public** ParentTree(**int** element) {

**this**.element = element;

}

}

**Partition#1 (Naïve) – Disjoint Set**

**package** COP3503H;

// Actual DumbPartition,

// But acts as SuperClass for SmartPartition which is superclass of CompressPartition

**public** **class** Partition **implements** AbstractPartition {

**static** **int** *inspected* = 0, *finds* = 0;

ParentTree[] partitions = **null**;

**public** Partition(**int** size) {

partitions = **new** ParentTree[size];

**for** (**int** i=0; i<size; i++) partitions[i] = **new** ParentTree(i);

}

**public** **void** resetStats() {

*inspected* = 0; *finds* = 0;

}

**Partition#2 (Naïve) – Disjoint Set**

**private** **int** doFind(**int** element) {

*inspected*++;

**if** (partitions[element].parent == **null**) **return** element;

**else** **return** doFind(partitions[element].parent.element);

}

**public** **int** find(**int** element) {

*finds*++;

**return** doFind(element);

}

**public** **void** union(**int** el1, **int** el2) {

**int** root1 = find(el1); **int** root2 = find(el2);

**if** (root1 != root2) partitions[root2].parent = partitions[root1];

}

**public** String statString() {

String stat = "Finds = " + *finds* + "; Calls = " + *inspected*;

**int** averagePathLength = 0;

// rint rounds the result, choosing the closest integer

// simple recasting truncates rather than rounds

**if** (*finds*>0)

averagePathLength = (**int**)Math.*rint*((**double**)*inspected*/*finds*);

stat += "; Average path length per Find = " + averagePathLength;

**return** stat;

}

**Partition#3 (Naïve)**

**public** String toString() {

**boolean**[] processed = **new** **boolean**[partitions.length+1];

**for** (**int** i=0; i<=partitions.length; i++) processed[i] = **false**;

**int** start = 0;

String trace = "";

**while** (start < partitions.length) {

**while** (processed[start]) start++;

**if** (start < partitions.length) {

trace += start;

**int** current = start;

**boolean** done = **false**;

**while** (!done) {

processed[current] = **true**;

ParentTree parent = partitions[current].parent;

trace += ((parent == **null**) ?

"::" :

("->" +(**new** Integer(parent.element).toString())));

**if** (parent != **null**) current = parent.element;

done = (parent == **null**) || processed[current];

}

trace += " ";

}

}

**return** trace;

}

}

**Partition Test Program**

**package** COP3503H;

**import** java.io.\*;

**import** java.util.\*;

**class** PartitionTest {

**public** **static** **void** main(String[] args) **throws** IOException {

**for** (**int** SZ=16; SZ<=1024; SZ <<= 2) {

Random r = **new** Random(168886542);

Partition p = **new** Partition(SZ);

System.*out*.println(p.toString());

**for** (**int** i = 0; i < SZ-1; i++) {

**int** rep1, rep2;

**do** {

**int** el1 = Math.*abs*(r.nextInt())%SZ;

**int** el2 = Math.*abs*(r.nextInt())%SZ;

rep1 = p.find(el1); rep2 = p.find(el2);

} **while** (rep1 == rep2);

p.union(rep1,rep2);

}

System.*out*.println(p.toString());

p.resetStats();

**for** (**int** i = 0; i < 10000; i++) p.find(Math.*abs*(r.nextInt())%SZ);

System.*out*.println(p.statString()+"\n");

}

System.*out*.println("\*\*\*\* Press Enter \*\*\*\*"); System.*in*.read();

}

}

**Quicksort main and helper**

/\*\*

\* Quicksort algorithm.

\* **@param** a an array of Comparable items.

\*/

**public** **static** <AnyType **extends** Comparable<? **super** AnyType>>

**void** quicksort( AnyType [ ] a )

{

quicksort( a, 0, a.length - 1 );

}

/\*\*

\* Return median of left, center, and right.

\* Order these and hide the pivot.

\*/

**private** **static** <AnyType **extends** Comparable<? **super** AnyType>>

AnyType median3( AnyType [ ] a, **int** left, **int** right )

{

**int** center = ( left + right ) / 2;

**if**( a[ center ].compareTo( a[ left ] ) < 0 )

swapReferences( a, left, center );

**if**( a[ right ].compareTo( a[ left ] ) < 0 )

swapReferences( a, left, right );

**if**( a[ right ].compareTo( a[ center ] ) < 0 )

swapReferences( a, center, right );

// Place pivot at position right - 1

swapReferences( a, center, right - 1 );

**return** a[ right - 1 ];

}

**Quicksort algorithm**

**private** **static** <AnyType **extends** Comparable<? **super** AnyType>>

**void** quicksort( AnyType [ ] a, **int** left, **int** right )

{

**if**( left + CUTOFF <= right ) // 5 ≤ CUTOFF ≤ 20

{

AnyType pivot = median3( a, left, right );

// Begin partitioning

**int** i = left, j = right - 1;

**for**( ; ; )

{

**while**( a[ ++i ].compareTo( pivot ) < 0 ) { }

**while**( a[ --j ].compareTo( pivot ) > 0 ) { }

**if**( i < j )

swapReferences( a, i, j );

**else**

**break**;

}

swapReferences( a, i, right - 1 ); // Restore pivot

quicksort( a, left, i - 1 ); // Sort small elements

quicksort( a, i + 1, right ); // Sort large elements

}

**else** // Do an insertion sort on the subarray

insertionSort( a, left, right );

}

**Quicksort fails to do well on small N, so use insertionSort**

/\*\*

\* Simple insertion sort.

\* **@param** a an array of Comparable items.

\*/

**public** **static** <AnyType **extends** Comparable<? **super** AnyType>>

**void** insertionSort( AnyType [ ] a )

{

**int** j;

**for**( **int** p = 1; p < a.length; p++ )

{

AnyType tmp = a[ p ];

**for**( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )

a[ j ] = a[ j - 1 ];

a[ j ] = tmp;

}

}

**Quickselect algorithm**

**private** **static** <AnyType **extends** Comparable<? **super** AnyType>>

**void** quickSelect( AnyType [ ] a, **int** left, **int** right, **int** k )

{

**if**( left + CUTOFF <= right )

{

AnyType pivot = median3( a, left, right );

// Begin partitioning

**int** i = left, j = right - 1;

**for**( ; ; )

{

**while**( a[ ++i ].compareTo( pivot ) < 0 ) { }

**while**( a[ --j ].compareTo( pivot ) > 0 ) { }

**if**( i < j )

swapReferences( a, i, j );

**else**

**break**;

}

swapReferences( a, i, right - 1 ); // Restore pivot

**if**( k <= i )

quickSelect( a, left, i - 1, k );

**else** **if**( k > i + 1 )

quickSelect( a, i + 1, right, k );

}

**else** // Do an insertion sort on the subarray

insertionSort( a, left, right );

}

**Properties of Odd-Even Transposition**

**In Some Things Luck Shines Our Way**

• Algorithm Uses Compare / Exchange Operations

• The Algorithm is Oblivious

• Communication Independent of Prior Results

• All Oblivious Comparison-Exchange (OCE) Algorithms are Easy to Analyze

**How Can Algorithm Fail?**

**Properties of a Faulty Permutation Sort**

• Assume X1, X2, … , Xn is to be Sorted

• Assume X(1), … , X(N) is a Correct Sort

• Assume X(1), … , X(N) is an Incorrect Permutation Produced by Some Faulty "Sort"

• Let k be Smallest Index Where X(k) > X(k)

• Then, the Permutation is Correct up to k-1

X(1) = X(1), X(2) = X(2), …, X(k-1) = X(k-1)

**The 0-1 Sorting Lemma**

**A Faulty OCE Sort also Fails on 0, 1 Data**

• Define Yi = 0 if Xi ≤ X(k) ,

Yi = 1 if Xi > X(k)

• Xi ≤ Xj if Yi ≤ Yj, Since Oblivious

• Thus, Output on 0,1 Data is

• 0, 0, 0, …, 0, 1, …, 1, 0, …,

• There is a 0 after the 1 in the k-th cell

•  Any Faulty Sort Fails on Some 0, 1 Data

**Correctness of Odd-Even Sort**

**Proof Based on 0-1 Sorting Lemma**

• Consider Rightmost Cell Pk Containing a 1

• If k is even then it won't move at step 1

• But it will shuttle right at all subsequent steps until it reaches N-th cell

• If k is odd, it starts moving at step 1

• And it will shuttle right at all subsequent steps until it reaches N-th cell

• Consider the i-th Rightmost 1

• By step i+1, no 1's block right shuttle

• So i-th 1 starts moving by step i+1

• i-th rightmost 1 is home (cell N-i+1) in at most N-i moves

• i-th rightmost is home no later than by i+(N-i) = N-th step

• This Shows Correctness and Timing

**An Improved Parallel Sort**

**The Shearsort**

• Assume N is a Perfect Square

• Organize into a N N Array of Cells

• Alternately Sort Rows and Columns  
(In the Manner of Shearing Sheep)

• Sort Odd Numbered Rows Left to Right  
Sort Even Numbered Rows Right to Left

• Conceptually Algorithm Uses Two Clocks

• Standard clock tells everyone to participate in one more step of a simple row or column sort

• Added clock tells cells to alternate between rows and columns

• IDs N(i-1)+1 to iN Cooperate on Row i;  
i, i+N, .., i+N–N Cooperate on Column i

**Shearsort Algorithm**

**Managing with One Clock**

At Each Clock Tick and For Each Pi do

Step := Step+1;

MajorStep := (Step-1) div N + 1;

if odd(MajorStep) then

if odd((i-1) div N + 1) then

SortLeftToRight // Bubble N

else SortRightToLeft // Bubble N

else

SortTopToBottom // Bubble N

**An Example Shearsort**



• N = 16, N = 4: Use 4  4 Matrix

**Shearsort Pass#1**

**Shearsort Pass#2**



**Shearsort Pass#3**



**Shearsort Pass#4**



**Shearsort Pass#5**



**Efficiency of Shearsort**

**How’d We Do? Not Bad!**

• T1(N) = N log N Optimal Sequential

• TN(N) = N  logN Parallel Shearsort

• SN(N) = N Speedup

• CN(N) = W N(N) = N  N  logN Cost

• EN(N) = 1 / N Efficiency

**Correctness of Shearsort**

**Proof Based on 0-1 Sorting Lemma**

• Each Pair of Passes Sorts at Least Half of the Unsorted Rows

• To See This, Consider Three Categories

• All 0 rows

• All 1 rows

• Dirty rows - some 0's, some 1's

• Can Divide Rows into Categories

• Upper all 0-rows

• Lower all 1-rows

• Dirty rows in middle

**Halving in Shearsort**

**Each Pair of Passes Cuts Dirty Rows in Half**

• A Row Sorting Pass Will Leave Dirty Pairs  
0…0…01…1 0…01 …… 1 0 … 01 … 1  
1…10 …… 0 1…1…10…0 1 … 10 … 0  
(more 0's) (more 1's) (equal 0's 1's)

• Dirty Pairs After Column Sorting Pass  
0……………0 0…01…10…0 0 … …0  
1…10…01…1 1……………1 1 … …1  
(more 0's) (more 1's) (equals)

**Convergence of Shearsort**

**How Much Work Before It’s Sorted?**

• Number of Halvings is Bounded by log N

• But We Do Two Passes per Halving

• Number of Passes is 2  log N + 1 =  
log N2 + 1 = log N + 1  
The + 1 is for One Dirty Row Left

• Each Pass Requires a Sort of N Cells  
We Can Parallel Bubble Sort in N Steps

• Total is N  (log N + 1) = O(N  log N)

**Shearsort Proof – Prelims.**

The crux of this correctness and analysis proof for ShearSort is to show that each row/column pair of sorts reduces the number of dirty rows by ½. Moreover, after each row/column sort, all the clean 0 rows are at the top (lowered numbered rows), and all the clean 1 rows are at the bottom (higher numbered rows.)

Notation: Assume R rows and C columns.

Lemma 1. After any row sort, each even/odd pair of rows is sorted low to high, high to low, respectively.

Proof: This is a direct consequence of the proof that the Even-Odd Transposition algorithm works.

Lemma 2. After the first column comparison exchange (there are number of rows of these CE’s for a complete column sort), the number of dirty rows is reduced to no more than half of what there were after the preceding row sort. Moreover, each clean 0 row will be in the lower numbered row of such a pair, and each clean 1 row will be in the higher numbered row.

Proof: This is done by showing the interaction of all possible combinations of dirty pairs (more 0’s than 1’s, more 1’s than 0’s, equal number of 0’s and 1’s.) There is an overhead that does this.

Lemma 3: Each column comparison-exchange (there are number of rows of them for a complete column sort) in which there is pair of clean rows leaves each clean 0 row in the lower numbered row of such a pair (on the top), and each clean 1 row will be in the higher numbered row (on the bottom).

Proof: This case is no different than the other case above in which we had an equal number of 0’s and 1’s in dirty rows.

**Shearsort Proof**

Theorem 1: Each row/column pair of sorts reduces the number of dirty rows by at least one half. Moreover, after each row/column, all the clean 0 rows are at the top (lowered numbered rows), and all the clean 1 rows are at the bottom (higher numbered rows) of the mesh.

Proof: By Lemma 1, we know that each column sort starts with even-odd pairs of rows sorted low to high and high to low, respectively. By Lemmas 2 and 3, the first comparison-exchange operation of the column sort will result in all row pairs being of one of the forms clean 0/dirty, dirty/clean 1 or clean 0/clean 1. The remaining R-1 comparison-exchange operations of the column sort will, in effect, move each clean 0 row up, so they are all together at the top (low numbered) rows of the mesh. Similarly, the clean 1’s will move down to be together at the bottom of the mesh. The reason that this is doable in the R-1 remaining passes is that the clean 0’s are already on the top of the pairs from which they formed and the clean 1’s are on the bottom of these pairs, leaving at most R-1 moves to find their respective destinations.

Theorem 2: The parallel ShearSort algorithm correctly sorts an R×C mesh in O(R+C)lg R steps.

Proof: By Theorem 1, the parallel ShearSort sorts all but, perhaps, one row of an R×C mesh containing 0-1 data in lg R row/column parallel even-odd transposition sorts. By previous analysis, we know that the row sorts take R steps and the column sorts take C steps. Thus, each row/column pair of sorts takes (R+C) steps. The final cleanup of the last unordered row takes R steps, so the total number of steps is (R+C)lg R + R. By the OCE (oblivious comparison exchange) lemma, an OCE sort that works on 0-1 data works on arbitrary data. Thus, the parallel ShearSort is correct and runs in O(R+C)lg R. When the mesh is square, this is 2√N lg √N = √N lg N.

**Revsort: An Improvement of Shearsort**

**The Revsort**

• Revsort is a Column / Row Alternating Sort

• For Convenience We Number Cells from 0

• Define rev(i) = Bit Reversal of i

• Revsort Sorts the Columns Downwards

• It Then Sorts Rows to the Right,   
Viewing Row i as Cyclically Starting at Column rev(i)

• Clearly the Wraparound Property That We Didn't Need in Shearsort is Critical Here

**Boundary Conditions and Complexity**

• Revsort does Not Actually Complete a Sort

• But It Leaves at Most 8 Dirty Rows

• These Rows Can be Handled by Shearsort

• Let d be the Number of Dirty Rows

• On Each Column / Row Pass With d > 8

Reduces Dirty Rows by O(√d)

• Running Time is 2 × √N × (lg lg √N + 2)

**Revsort on an Old Example**



• We Underscored the Starting Column of Each Row.

**Revsort Pass#1**

**Revsort Pass#2**



**Revsort Pass#3**



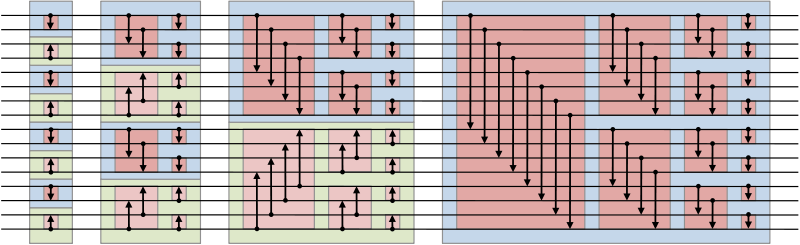
**Revsort Pass#4**



**Sort – Making a List Bitonic**

A sorted sequence is a monotonically non-decreasing (or non-increasing) sequence.

A *bitonic* sequence is a sequence with x_0 \leqq \ldots \leqq x_k \geqq \ldots \geqq x_{n-1} for somek, 0 \leqq k < n, or a circular shift of such a sequence.



O(*n* log2(*n*)) comparators and have a time delay of O(log2(*n*)), where *n* is the number of items to be sorted

**Bitonic Sort – Making a List Bitonic**



**Bitonic Merge – Finishing the Sort**



**Bitonic Sort**



Time =  = (lg n + 1) lg n / 2 = O(lg2n).

**Bitonic Sort on Hypercube**

• The Mapping is Natural – Use 3-Cube for 8 Values





**Use Hypercube to Make List Bitonic**

Phase 1 Phase 2, Steps 1 & 2 – Bitonic Now

**Use Hypercube to Sort Bitonic List**

Phase 3, Steps 1, 2 & 3 – Sorting the Bitonic List

**Classification of Algorithms**

Classification of Algorithms

Divide and Conquer – Searching, Sorting, Multiplication, Parsing

Greedy

Scheduling, cost optimizing while spanning a circuit, bin packing

Dynamic Programming

Divide and conquer to its extreme

**Divide and Conquer**

The problem is either small and easy to solve, or it can be naturally broken up into several sub-problems, each of which can be solved by the exact same approach as was applied to the original problem.

When the sub-problems have been solved, it is possible to merge the solutions into one that is correct for the larger original problem.

Thus, we look for three characteristics:

(a) Easy to split into sub-problems;

(b) Sub-problems are simpler instances of original;

1. Sub-problem results can be easily combined to solve the original problem.

D&C -- Algorithmic Form

algorithm p (s);

if (small (s)) then return easy\_attack (s);

else {

[ s1, s2 , ..., sk ] = divide (s);

return (combine ( p(s1), p(s2), …, p(sk) ) )

}

In some cases, we can divide and immediately reject one sub-problem, pursuing only the other. BST search is such an example.

**Greedy**

Want to Max or Min some objective function.

Solution must satisfy some feasibility constraint.

Any solution satisfying constraints is feasible.

A feasible solution that maxes or mins the objective is optimal.

Greedy solutions are often sub-optimal, but are always feasible solutions.

For example, First Fit BinPacking never overfills a trunk, so it always returns a feasible solution.

Its solutions are, however, not guaranteed to be optimal.

Greedy -- Algorithmic Form

solution = { };

for ( int i = 1 ; i <= numberOfChoices; i++) {

x = select (a); // where select is simple

if (feasible (solution ∪ x))

solution = solution ∪ x;

}

return solution;

**Dynamic Programming**

Based on "Principal of Optimality"

Technique requires that a problem have multiple stages at which decisions are made. We talk about stage i, and indicate that the system is in state i.

The Principal of Optimality says that the choice that is made at this stage is dependent only on state i and not on the policy that was used to make decisions at stages 1 to i-1.

Stated differently – the solution from stage i to n (the last stage) must be optimal in order for 1 to n to be optimal.

**Greedy – Basics**

**Want to Max or Min some objective function. Solution must satisfy some feasibility constraint.**

**Any solution satisfying constraints is feasible.**

**A feasible solution that maxes or mins the objective is optimal.**

**Greedy solutions are often suboptimal, but are always feasible solutions.**

**For example, a First Fit never overfills a trunk, so it always return a feasible solution. Its solutions are, however, not guaranteed to be optimal.**

**General Form of Greedy Algorithm:**

**solution := {};**

**FOR i:=1 to NumberOfChoices DO**

**X := Select (A); (\* where Select is simple \*)**

**IF Feasible (Solution  X) THEN**

**Solution := Solution  X**

**RETURN Solution**

**Spanning Trees**

**Assume that G = (V, E), where G is an undirected graph, V is the set of vertices (nodes), and E is the set of edges.**

**A spanning tree of G is a subgraph which is a tree that encompasses all nodes in the original graph. Such a tree will commonly include just a subset of the original edges. Here, by tree, we mean a graph with no simple cycles. We ignore the designation of a root and we do not order nodes.**

**If G is a single connected component, then there is always a spanning tree.**

**Adding weights to edges gives us the minimum spanning tree problem, where we wish to span with edges whose sum is minimum among all spanning trees.**

**Weights could be distances, costs, signal degradation, …**

**Spanning Trees**

**Consider four nodes, fully connected as below,**



**The spanning trees are:**



**Min Spanning Tree (MST)–Kruskal's Algorithm**

Greedy – We grab the lightest weight edge that does not create a cycle.

Feasible – There are no simple cycles at every stage.

Minimal as any other choices will have the same or higher weight edges to bring in new nodes.

There are lots of ways to implement Kruskal’s algorithm.

We will study a number of these, each using an adjacency (edge) list.

#### Kruskal’s Algorithm in Action (Greedy across all)



**Edge Cost Graph**

(1,2) 10 

(3,6) 15 

(4,6) 20 

(2,6) 25 

(1,4) 30, Reject

(3,5) 35 

(2,5), (1,5), (2,3) 40, 45, 50, Reject

**Last 3 rejects are skipped if we count edges added.**

**MST–Kruskal's Algorithm**

Feasible – There are no simple cycles (only repeats are start/finish vertices) at every stage

**import java.util.\*;**

**……**

**List kruskalMinSpan** **(int** n**, List<Edge>** edges**) {**

**CompressedPartition** p = new **CompressedPartition**( n );

**List<Edge>** spanningEdges = new **ArrayList<Edge>**();

**Collections**.sort(edges); // sorted low to high by cost

**for** (**Edge** edge : edges **{**

int p1 = p.find(edge.node1); int p2 = p.find(edge.node2);

if (p1 != p2) {

p.union(edge.node1, edge.node2);

spanningEdges.add(edge);

}

}

**return** spanningEdges;

**}**

#### This implementation uses an adjacency (edge) list and starts with a sort of that list based on edge weights. This is |E| lg |E| = |E| lg |V| since |E| is bounded by |V|2. It then costs barely over |E| for the for-each loop over edges since union/find is essentially constant. We could skip the sort and search for the least weight edge when needed but that is |E| each time through the loop for |E|2. A better option is to heapify the edges by weight in |E| time. This will require a lg |E| deleteMin operation in the for-each loop, so cost is |E| lg |E| = |E| lg |V|, as before, but it is actually unlikely that you will need to iterate over all edges as an early break can be made when the spanning tree has added |V|-1 edges. In the worst case, we will still need to go over all edges, but this is rare. Kruskal’s Algorithm from Text

**void kruskal( ) {**

**int edgesAccepted = 0;**

**DisjSet ds = new DisjSets( NUM\_VERTICES );**

**PriorityQueue<Edge> pq = new PriorityQueue<Edge>( getEdges( ) );**

**Edge e;**

**Vertex u, v;**

**while( edgesAccepted < NUM\_VERTICES - 1 ) {**

**e = pq.deleteMin( ); // Edge e = (u. v)**

**SetType uset = ds.find( u );**

**SetType vset = ds.find( v );**

**if( uset != vset ) {**

**// Accept the edge**

**edgesAccepted++;**

**ds.union( uset, vset );**

**}**

**}**

**}**

**Min Spanning Tree–Prim's Algorithm**

Greedy – We grab the closest node to one of the ones that has already been included. Can start with any node unlike Kruskal’s, which requires we start with lightest weight edge. Always is a tree, unlike Kruskal’s, which can have a forest at intermediate stages.

Feasible – There are no simple cycles at every stage.

Minimal as any other choices will have the same or higher weight edges to bring in new nodes.

There are lots of ways to implement Prim’s algorithm.

We will study an O(|V|2) way. This is optimal for dense graphs. In fact the algorithm uses an adjacency matrix which is itself |V|2 in size even for a sparse graph.

Other implementations are O(M lg |V|), where M = max (|E|, |V|).

**MST–Prim’s Algorithm (Greedy from some start point)**



Optimization– At each point choose edge (u,v) so (u,v) is minimum weight edge allowing A  (u,v) to be a tree

# Edge Cost Tree

(1,2) 10 

(2,6) 25 

(3,6) 15 

(6,4) 20 

(1,4) Reject

(3,5) 35 

**Min Spanning Tree–Prim's Algorithm**

void PrimMinSpan (int N; AdjacencyMatrix Adjacency) {/\* pseudo code \*/

int j,k;

/\* Assume N nodes, labeled 1 to N \*/

V = new Set of 1..N;

Dist = new Array [1..N];

Source = new Array [1..N];

Dist = Adjacency[1]; // a list of |V| distances

V = [2..N];

Source[1] = 0; // Root has no source

f**or** j **in** V **do**

Source[j] = 1; // Distances are from root

**while** V <> [ ] **do {**

k = index in V with smallest value in Dist;

V = V – [k];

**for** j **in** V **{**

**if** Dist[j] > Adjacency[k,j] **then** {

Dist[j] = Adjacency[k,j]; Source[j] = k; }

**}**

**}**

Outside main loop we take |V| time to initialize. Main loop runs |V| times. We must find minimum weight from nodes already added. That could be |V| if we keep unordered or lg |V| if we use a min heap. No matter what, we will pay |V| to update the weights. If we keep a heap, we could pay |V| lg |V| to update the least weight from nodes in tree; that may not be worth it. Of course this is all dependent on our use of an adjacency array, not a list.

How would you recast with an adjacency list?

**Applying Prim’s Algorithm**



**Node Dist/Source Cost Tree**

1 [**0**/0,10/1,∞/1,30/1,45/1,∞/1]

2 [**0**/0,**10**/1,50/2,30/1,40/2,25/2] 10 

6 [**0**/0,**10**/1,15/6,20/6,40/2,**25**/2] 25 

3 [**0**/0,**10**/1,**15**/6,20/6,35/3,**25**/2] 15 

4 [**0**/0,**10**/1,**15**/6,**20**/6,35/3,**25**/2] 20 

5 [**0**/0,**10**/1,**15**/6,**20**/6,**35**/3,**25**/2] 35 

**Reflexive Transitive Closure**

**The Problem:**

**Given a graph, G, determine for which pairs of nodes, (A,B), there is a path between A and B.**



**Array representation – 1 is True; 0 is False**

# A B C D E F G

**A 1 0 1 1 0 0 1**

**B 0 1 0 0 0 0 0**

**C 0 0 1 0 1 1 0**

**D 0 0 0 1 1 0 1**

**E 0 1 0 0 1 0 0**

**F 0 1 0 0 0 1 0**

**G 0 1 0 0 1 0 1**

**Warshall’s**

**public void warshallsAlgorithm() {**

**//for each pivot try all pairs of nodes**

**for (int pivot = 0; pivot < N; pivot++)**

**for (int v = 0; v < N; v++)**

**for (int w = 0; w < N; w++)**

**if (v != w)**

**connectedMatrix[v][w] = connectedMatrix[v][w] ||**

**(connectedMatrix[v][pivot] && connectedMatrix[pivot][w]);**

**}**

**Analysis easily shows that this is O(N3).**

**In comparison N DFS’s take O(NM), where M is max of N and number of arcs.**

**In the very worst case, DFS is O(N3).**

**Proof of correctness:**

**If there is a path between i and j involving intermediate nodes with indices less than or equal to k, then such a path will be discovered, and only paths of this sort will be discovered.**

**Basis case: If i and j are connected directly (no intermediate nodes) then we discover this as part of initialization.**

**IH(k-1, k>=0): Assume if i and j are connected with intermediate indices that never exceed k-1, then we will discover this in the k-th iteration.**

**IS(k): If the path from i to j involves the k-th node as an intermediary, and we exclude cycles, then i is connected to k and k to j by paths involving no intermediaries with indices greater than k-1. By IH, we will have discovered the fact that i is connected to k and k to j. The k+1-st iteration will use pivot = k to discover that i is connected to j. Q.E.D.Weary Traveler – Shortest Path**

**The Problem:**

**Given a graph (a dag), G, with weighted arcs, and two nodes, A and B, determine the minimum weight path from A to B.**

**Greedy fails here: Get 3 + 6 + 6 = 15; but can get 5 + 3 + 5 = 13**



Greedy fails here: Get 3 + 6 + 6 = 15; but can get 5 + 3 + 5 = 13

**Array representation**

### A B C D E F G

**A 0 ∞ 5 3 ∞ ∞ 14**

**B ∞ 0 ∞ ∞ ∞ ∞ ∞**

**C ∞ ∞ 0 ∞ 3 7 ∞**

**D ∞ ∞ 11 0 7 ∞ 6**

**E ∞ 5 ∞ ∞ 0 ∞ ∞**

**F ∞ 7 ∞ ∞ ∞ 0 ∞**

**G ∞ 6 ∞ ∞ 7 ∞ 0**

**Lousy Weary Traveler Solution**

**const INFINITY = 9999;**

**FirstCity = 'A';**

**LastCity = 'G';**

**type City = FirstCity .. LastCity;**

**var Dist : array[City] of array[City] of Word;**

**procedure Weary ( Source, Sink : City) : Word;**

**var cost, c : Word;**

**intermediary : City;**

**begin**

**if Source = Sink then cost := 0**

**else begin**

**cost := INFINITY;**

**for intermediary := FirstCity to LastCity do**

**if (Dist[Source, intermediary] < INFINITY) and**

**(Source <> intermediary) then**

**cost := min(cost, Dist[Source,intermediary]+Weary(intermediary,Sink));**

**end;**

**Weary := cost**

**end; (\*Weary\*)**

**OR**

**begin**

**if Source = Sink then cost := 0**

**else begin**

**cost := INFINITY;**

**for intermediary := FirstCity to LastCity do**

**if (Dist[intermediary, Sink] < INFINITY) and**

**(Sink <> intermediary) then**

**cost := min(cost,**

**Dist[intermediary,Sink]+Weary(Source,intermediary));**

**end;**

**Weary := cost**

**end; (\*Weary\*)**

**Analysis of Lousy Weary Algorithm**

**We would like to determine the number of times we execute the loop body in the procedure Weary. That is the dominant factor.**

**Consider the first call. The number of cities = N. Thus, we will execute the loop body exactly N times, plus the times associated with each recursive call. At worst, the source is connected to all other nodes. So there can be up to N-1 calls to Weary. Each one of them will do N iterations, plus of course the work done in their recursive calls. But now, the maximum number of node connections is only N-2, since there cannot be cycles, and therefore the source is unreachable.**

**Looking at a timing function, where K is the number of nodes that can be directly connected to the current source.**

**T(K) = N + (K) \* T( K - 1)**

**T(0) = N**

**We would start at T(N-1), since the first source can be connected to at most N-1 other nodes. Clearly, if we ignore the N+ part, we have K!, or (N-1)! for the problem at hand. In fact a careful analysis shows this is even worse.**

**DFS-Based Unweighted Shortest Path Algorithm**

**void unweighted( Vertex s ) {**

**Queue<Vertex> q = new Queue<Vertex>( );**

**for each Vertex v**

**v.dist = INFINITY;**

**s.dist = 0;**

**q.enqueue( s );**

**while( !q.isEmpty( ) ) {**

**Vertex v = q.dequeue( );**

**for each Vertex w adjacent to v**

**if( w.dist == INFINITY ) {// unvisited**

**w.dist = v.dist + 1;**

**w.path = v;**

**q.enqueue( w );**

**}**

**}**

**}Floyd’s All Shortest Paths Algorithm**

**final int INFINITY = Integer.MAX\_VALUE; // choose value not used in weights**

**private boolean connected(int v, int w) {**

**return adjacencyMatrix[v][w] != INFINITY)**

**}**

**public void floydsAlgorithm() {**

**for (int pivot = 0; pivot < N; pivot++)**

**for (int v = 0; v < N; v++)**

**for (int w = 0; w < N; w++) {**

**if (connected(v,pivot) && connected (pivot,w)) {**

**int tempDistance = adjacencyMatrix[v][pivot] +**

**adjacencyMatrix[pivot][w];**

**if (tempDistance < adjacencyMatrix[v][w])**

**adjacencyMatrix[v][w] = tempDistance;**

**}**

**}**

**Analysis again shows that this is O(N3).**

**Is there a way to get close to the potential gain we saw in DFS versus Warshall’s.**

**Adjacency Lists for Shortest Path**



**Graph Representation Using Adjacency Lists**

**G (Graph)**

**city (index) dist (from A) toPOT adjacency (name, label)**

**A 0 1 ((C,5),(D,3),(G,14))**

**B ∞ 2 ()**

**C ∞ 3 ((E,3),(F,7))**

**D ∞ 4 (C,11),(E,7),(G,6))**

**E ∞ 5 ((B,5))**

**F ∞ 6 ((B,7))**

**G ∞ 7 ((B,6),(E,7))**

**Note: dist is an unsigned integer as we don’t allow negative weights**

**Heap for Shortest Path**

**POTCities (POT) – note that heap is organized by G[POTCities[index]].dist**

**index city = POTCities[index]**

**1 A**

**2 B**

**3 C**

**4 D**

**5 E**

**6 F**

**7 G  last**

#### Dijkstra’s Shortest Paths Algorithm

**const FirstCity = 'A'; LastCity = 'G'; // or whatever**

**type City = FirstCity .. LastCity;**

**POTCity = 1..(ord(LastCity)-ord(FirstCity))+1;**

**Graph = array[City] of (\* see above pic of abstract data structure \*)**

**POT = array[POTCity] of City; (\* heap values are in G[city].dist \*)**

**var G : Graph; POTCities: POT; last: POTCity = ord(LastCity);**

**procedure swap(a,b : POTCity);**

**var temp : City;**

**begin**

**temp := POTCities[b];**

**POTCities[b] := POTCities[a]; POTCities[a] := temp;**

**G[POTCities[a]].toPOT := a; G[POTCities[b]].toPOT := b**

**end;**

**function compare(a,b : POTCity) : Integer;**

**begin**

**compare := G[POTCities[a]].dist – G[POTCities[b]].dist;**

**end;**

#### Dijkstra’s Shortest Paths Algorithm

**procedure Dijkstra;**

**var u, v : City; p : List;**

**begin**

**Initialize; (\* initialize all data structures and variable**

**while last>1 do begin**

**v := deleteMin(POTCities);**

**(\***

**v := POTCities[1];**

**swap(1, last); last := last-1; bubbleDown(1);**

**\*)**

**p := G[v].adjacency;**

**while p<>NIL do begin**

**u := p^.name;**

**if (G[u].dist > G[v].dist+p^.label) then begin**

**G[u].dist := G[v].dist+p^.label;**

**bubbleUp(G[u].toPOT);**

**end;**

**p := p^.next**

**end**

**end**

**end; (\* Dijkstra \*)**

#### Analysis of Dijkstra’s Algorithm

**This is a Greedy algorithm in that it always does the best it can at each step.**

**To run it for one node takes O(M  log2 N) where M is max of N = |V| and number of arcs = |E|. This is just a log factor away from the time it takes to look at the graph. To get “All Paths”, we need to run this N times, getting O(N  M  log2 N). If M is close to N2, then this runs worse than Floyd’s algorithm. If M is small then this is better. There is an N2 implementation of Dijkstra’s algorithm which can be used to create a competitive N3 all path algorithm, but it’s more complicated than Floyd’s.**

**The N2 algorithm is on the next page.**

**Greedy N2 Shortest Paths Algorithm**

**const FirstCity = 'A'; LastCity = 'G';**

**type City = FirstCity .. LastCity;**

**var Dist: array[City] of array[City] of Word; (\* unsigned integer \*)**

**procedure Greedy;**

**var u, v : City; shortest : Word;**

**short: array[City] of Word; (\* from source \*) settled, unsettled : Set of City;**

**begin**

**(\* initialize the shortest paths from FirstCity \*)**

**for u:=FirstCity to LastCity do**

**short[u] := Dist[FirstCity,u];**

**short[FirstCity] := 0;**

**(\* initially only FirstCity is a settled node \*)**

**settled := [FirstCity];**

**unsettled := [succ(FirstCity) .. LastCity];**

**(\* iterate until all nodes are settled \*)**

**while unsettled <> [ ] do begin**

**(\* greedily pick next one to settle \*)**

**shortest := INFINITY; (\* a global const \*)**

**for v in unsettled do**

**if short[v] <= shortest then begin u := v; shortest := short[v] end;**

**settled := settled + [u]; unsettled := unsettled - [u];**

**(\* fix the current shortest paths from FirstCity \*)**

**for v in unsettled do**

**short[v] := min(short[v],short[u]+Dist[u,v])**

**end**

**end; (\* Greedy \*)**

**This is N2. Why use <= in greedy part? What would happen to complexity if short were kept as a heap?**

**Flow Graphs**

**A flow graph G = (N, E, s) refers to a directed graph (N, E) and an initial node s in N, where there is a path from s to every node of G. Nodes can be statements or basic blocks (single entry, single exit). Commonly, they are the latter.**

**Program SquareRoot;**

**var L, N, K, M : integer; C : boolean;**

**begin**

**read(L); (\* start of block B1 \*)**

**N := 0; K := 0; M := 1; (\* end of block B1 \*)**

**loop**

**K := K + M; (\* start of block B2 \*)**

**C := K > L;**

**if C then break; (\* end of block B2 \*)**

**N := N + 1; (\* start of block B3 \*)**

**M := M + 2 (\* end of block B3 \*)**

**end loop;**

**write(N) (\* all of block B4 \*)**

**end. (\* SquareRoot \*)**

****

**A More Complex Flow Graph**

****

**Partitions and Connected Components**

**import java.util.\*;**

**……**

**Partition connectedComponents(int** n**, List<Edge>** edges**) {**

p = new **CompressedPartition**( n );

**for** (**Edge** edge : edges)

p.union(edge.node1, edge.node2);

**return p;**

**}**

Assume m is max of n = |V|, the number of nodes, and e = |E|, number of edges, then this takes O(m\*f), where f is cost of a Find operation (the basis for Union). Fast union/find shows this can be done in almost O(m), so that’s a clue that there may be a direct O(m) algorithm – there is one!

Processes Scheduling Problem

A Process Scheduling Problem can be described by

m processors P1, P2, …, Pm,

processor timing functions S1, S2, …, Sm, each describing how the corresponding processor responds to an execution profile,

additional resources R1, R2, …, Rk, e.g., memory and other serially reusable items,

a transmission cost matrix Cij (1 ≤ i , j ≤ m), based on processor data sharing,

tasks to be executed T1, T2, …, Tn,

task execution profiles A1, A2, …, An,

a partial order defined on the tasks  
such that Ti < Tj means that Ti must complete before Tj can start execution,

a communication matrix Dij (1 ≤ i , j ≤ n) where Dij can be non-zero only if Ti < Tj,

weights W1, W2, …, Wn interpreted as the cost of deferring execution of a task.

**Scheduling of Processes and NP-Completeness**

The intent of a scheduling algorithm is to minimize the sum of the weighted completion times of all tasks, while obeying the constraints of the task system. Weights can be made unusually large to impose actual deadlines.

The general scheduling problem is quite complex, but even simpler instances, where the processors are uniform, there are no additional resources, there is no data transmission, the execution profile is just processor time and the weights are uniform, are very hard.

In fact, if we just specify the time to complete each task and we have no partial ordering, then finding an optimal schedule on two processors is an NP-complete problem. (The notion of NP Complete is on other overheads.)

**2-Processor Scheduling**

The problem of optimally scheduling n tasks T1, T2, …, Tn onto 2 processors with an empty partial order < is the same as that of dividing a set of positive whole numbers into two subsets, such that the numbers are as close to evenly divided. So, for example, given the numbers

3, 2, 4, 1

we could try a “greedy” approach as follows:

put 3 in set 1

put 2 in set 2

put 4 in set 2 (total is now 6)

put 1 in set 1 (total is now 4)

This is not the best solution. A better option is to put 3 and 2 in one set and 4 and 1 in the other. Such a solution would have been attained if we did a greedy solution on a sorted version of the original numbers. In general, however, sorting doesn’t always work.

**2-Processor Scheduling**

Try the unsorted list

7, 7, 6, 6, 5, 4, 4, 5, 4

Greedy (Always in one that is least used)

7, 6, 5, 5 = 23

7, 6, 4, 4, 4 = 25

Optimal

7, 6, 6, 5 = 24

7, 4, 4, 4, 5 = 24

Sort it

7, 7, 6, 6, 5, 5, 4, 4, 4

7, 6, 5, 4, 4 = 26

7, 6, 5, 4 = 22

Even worse than greedy unsorted

**2-Processor with Partial Ordering**



**Anomalies Everywhere**



**More Anomalies**



**Critical Path or Level Strategy – UET**

A UET is a Unit Execution Tree. Our Tree is funny. It has a single leaf by standard graph definitions.

1. Assign L(T) = 1, for the leaf task T

2. Let labels 1, …, k-1 be assigned. If T is a task with lowest numbered immediate successor then define L(T) = k (non-deterministic)

This is an order n labeling algorithm that can easily be implemented using a breadth first search.

Note: This can be used for a forest as well as a tree. Just add a new leaf. Connect all the old leafs to be immediate successors of the new one. Use the above to get priorities, starting at 0, rather than 1. Then delete the new node completely.

Note: This whole thing can also be used for anti-trees. Make a schedule, then read it backwards. You cannot just reverse priorities.

**Applying Level Strategy to UET**



**Theorem:** Level Strategy is optimal for unit execution, m arbitrary, forest precedence

**Level Strategy – DAG with Unit Time**

1. Assign L(T) = 1, for an arb. leaf task T

2. Let labels 1, …, k-1 be assigned. For each task T such that  
  
{L(T’) is defined for all T’ in Successor(T)}  
  
Let N(T) be decreasing sequence of set members in  
  
{S(T’) | T’ is in S(T)}  
  
Choose T\* with least N(T\*).  
Define L(T\*) = K.

This is an order n2 labeling algorithm. Scheduling with it involves n union / find style operations. Such operations have been shown to be implementable in nearly constant time using an “amortization” algorithm.

**Theorem**: Level Strategy is optimal for unit execution, m=2, dag precedence.

**Examples of Relational Operators**

# EMPLOYEES

|  |  |
| --- | --- |
| **NAME** | **ID** |
| **Smith, Mary** | **027** |
| **Arco, Max** | **145** |
| **Simmons, Richard** | **037** |
| **Gonzalez, Rafael** | **111** |
| **Jones, Atticus** | **621** |
| **Torey, Phyllis** | **006** |
| **Casper, Leona** | **427** |

**EMPLOYEES union SHAREHOLDERS**

|  |  |
| --- | --- |
| **NAME** | **ID** |
| **Arco, Max** | **145** |
| **Blackman, Tonya** | **088** |
| **Casper, Leona** | **427** |
| **Gonzalez, Rafael** | **111** |
| **Jones, Atticus** | **621** |
| **Kolomotov, Karyl** | **077** |
| **Pham, Carole** | **777** |
| **Simmons, Richard** | **037** |
| **Smith, Mary** | **027** |
| **Ting, Xin** | **099** |
| **Torey, Phyllis** | **006** |
| **Torres, Alejandro** | **174** |

**SHAREHOLDERS**

|  |  |
| --- | --- |
| **NAME** | **ID** |
| **Simmons, Richard** | **037** |
| **Pham, Carole** | **777** |
| **Torres, Alejandro** | **174** |
| **Blackman, Tonya** | **088** |
| **Ting, Xin** | **099** |
| **Gonzalez, Rafael** | **111** |
| **Smith, Mary** | **027** |
| **Kolomotov, Karyl** | **077** |

**EMPLOYEES intersect SHAREHOLDERS**

|  |  |
| --- | --- |
| **NAME** | **ID** |
| **Gonzalez, Rafael** | **111** |
| **Simmons, Richard** | **037** |
| **Smith, Mary** | **027** |

**EMPLOYEES minus SHAREHOLDERS**

|  |  |
| --- | --- |
| **NAME** | **ID** |
| **Arco, Max** | **145** |
| **Blackman, Tonya** | **088** |
| **Casper, Leona** | **427** |
| **Jones, Atticus** | **621** |
| **Torey, Phyllis** | **006** |

**Examples of Relational Operators**

**EMPLOYED\_BY**

|  |  |
| --- | --- |
| **NAME** | **FIRM** |
| **Smith, M.** | **RCA** |
| **Arco, M.** | **Mitre** |
| **Sim, R.** | **Apple** |
| **Garcia, R.** | **Mitre** |
| **Jones, A.** | **TCBY** |
| **Torey, P.** | **RCA** |
| **Carr, L.** | **RCA** |

**EMPLOYED\_BY join HEALTH\_PLANS**

|  |  |  |
| --- | --- | --- |
| **NAME** | **FIRM** | **HMO** |
| **Smith, M.** | **RCA** | **PPC** |
| **Smith, M.** | **RCA** | **AVMED** |
| **Smith, M.** | **RCA** | **Humana** |
| **Arco, M.** | **Mitre** | **Humana** |
| **Sims, R.** | **Apple** | **Kaiser** |
| **Garcia,R.** | **Mitre** | **Humana** |
| **Jones, A.** | **TCBY** | **PPC** |
| **Torey, P.** | **RCA** | **PPC** |
| **Torey, P.** | **RCA** | **AVMED** |
| **Torey, P.** | **RCA** | **Humana** |
| **Carr, L.** | **RCA** | **PPC** |
| **Carr, L.** | **RCA** | **AVMED** |
| **Carr, L.** | **RCA** | **Humana** |

**HEALTH\_PLANS**

|  |  |
| --- | --- |
| **FIRM** | **HMO** |
| **RCA** | **PPC** |
| **RCA** | **AVMED** |
| **RCA** | **Humana** |
| **Mitre** | **Humana** |
| **TCBY** | **PPC** |
| **Apple** | **Kaiser** |

**FIRM=Mitre (EMPLOYED\_BY)**

|  |  |
| --- | --- |
| **NAME** | **FIRM** |
| **Arco, M.** | **Mitre** |
| **Garcia, R.** | **Mitre** |

**π HMO** **(HEALTH\_PLANS)**

|  |
| --- |
| **HMO** |
| **PPC** |
| **AVMED** |
| **Humana** |
| **Kaiser** |

**Complexity of Relational Operators**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **MaxSz** | **MinSz** | **Naive** | **Pre-Sort;**  **Post-Sort** | **Indexed** |
| **R  S** | **t = n+m** | **max(n,m)** | **nm** | **n log n +  m log m;**  **t log t** | **t=n+m** |
| **R  S** | **min(n,m)** | **0** | **nm** | **n log n +  m log m;**  **t log t** | **t=n+m** |
| **R – S** | **n** | **0** | **nm** | **n log n +  m log m;**  **t log t †** | **t=n+m** |
| **C (R)** | **n** | **0** | **n** | **no gain;**  **no gain** | **k *Lucky!*** |
| **π– (R)** | **n** | **n, usual**  **1, rare** | **n^2** | **nlog n;**  **nlog n** | **n** |
| **R ∞ S** | **nm** | **0** | **nm** | **no gain;**  **k+t log t ††**  ***sort-join*** | **k+n or k+m**  ***index-join*** |

**Assumes |R| = n, |S| = m, t = n+m, and |Result| = k**

**† An extra field is initially added to each result tuple to identify the relation from which this tuple came.  
†† (a,k) in R becomes (k,a,R) and (k,b) in S is (k,b,S).**

**When doing a sequence of operations, it is critical to keep the size of intermediary results as small as possible. Reordering and deferring operations can be very helpful. In arithmetic A\*B+A\*C = A\*(B+C), but second expression is usually faster than first.**

**Algebraic Laws for Relational Operators**

**Laws for Join**

**Limited Associativity**

**((R ∞ A=B S) ∞ C=D T)  (R ∞ A=B (S ∞ C=D T))**

**provided A is an attribute of R, B and C are different attributes of S, and D is an attribute of T.**

**Laws for Selection**

**Selection Pushing below Joins**

**(C (R ∞ S))  (C (R) ∞ S)**

**provided all attributes of C are in R**

**(C (R ∞ S))  (R ∞ C (S))**

**provided all attributes of C are in S**

**Selection Splitting**

**(C and D(R))  (C (D (R))**

**Selection Commutivity**

**(C (D (R))  (D (C (R))**

**Algebraic Laws for Relational Operators -- Continued**

###### Laws for Projection

**Projection Pushing below Unions**

**(L (R  S))  (L (R)  L (S))**

**Limited Projection Pushing below Joins**

**(L (R ∞ A=B S))  (L (M (R) ∞ A=B N (S)))**

**where**

**1) M is attributes of L from R followed by A, if not in L,**

**2) N is attributes of L from S followed by B, if not in L**

**Projection Identity**

**L (R)  R, when L is all attributes of R**

**Query on a Relational Database**

**RELATIONS**

**1. CSG (Course-StudentId-Grade)**

**2. SNAP (StudentId-Name-Address-Phone)**

**3. CDH (Course-Day-Hour)**

**4. CR (Course-Room)**

**QUERY**

**“Where is C. Brown 9AM on Mondays?”**

**An Approach**

**( ( ( ( SG ∞ SNAP ) ∞ CDH ) ∞ CR )**

**Gets Tuples**

**(c, s, g, n, a, p, d, h, r)**

**Can Select**

**Name = “C. Brown”**

**and (Day=“M”) and (Hour=“9AM”)**

**and Project Room**

**Query Tree**

**Leaf nodes in the tree are relations.**

**Interior nodes are relational operators.**

**Query can be interpreted from leaves towards root, with intermediate results generated at each node.**



**The final Join is enormous, even though we require just one item in the result.**

**Optimization by Pushing Selection**

**Push Selection below Join**

**Push in both directions, but remove slection on right branch, since CR has no Name, Day or Hour field.**



**This makes final Join trivial, since there will now be only one tuple coming up the left branch.**

**Optimization by Splitting Selection**

**Split Selection to prepare for sending only relevant parts of the Selection down each branch on the next Join.**



**Could Split again, but that won’t help**

**Push Selections on Separate Paths**

**Push Selections down only those paths that involve selected attributes**



**Day / Hour apply to only CDH.**

**Name applies to only SNAP, so keep on Pushing**

**Push Name Selections to SNAP**

**Push Name Selection down toward SNAP since CSG does not involve Selected attributes**



**Can’t Push Selections any farther down.**

**Push Projection Down**

**Must Push Projection on attributes of Join as well as that in the original Projection.**



**Join attribute is Course, so we project to Course alone on left, since Room is not an attribute on left. Pushing a Projection of Room and Course to right is useless.**

**Push Projection Down Farther**

**Push Projection down middle Join.**



**Course Projection can restrict size on both sides.**

**Push Projection Down to Bottom**

**Push Projection down as far as possible.**



**Just need the attributes that play a role in Join and are kept after Projection.**

**Relax on Some Projections**

**There’s little to be gained by Projecting out attributes that disappear in next step. We relax by not removing the Grade attribute since it disappears at next Join.**



**This may or may not avoid some wasted effort.**

**A reasonably Fast, Inefficient Max**

**• Quick, but Not Blindingly Fast**

**• Use a Doubly Logarithmic-Depth Tree**

• If N=22k, then root has 22k–1 children

• At ith level, 0≤i<k, each node has 22k–i–1 children

• At level k, each node has 2 leaves as children

**Example of Doubly Log Depth Tree**

**If N = 64K = 216 = 224, then**

level 0 (root) has 256 = 28 = 223 children

level 1 nodes have 16 = 24 = 222 children

level 2 nodes have 4 = 22 = 221 children

level 3 nodes have 2 = 21 = 220 children

level 4 nodes have 2 children

Number of leaves = 2×2×4×16×256 = 21+1+2+4+8 = 216

**• Each Internal Node Gets Max of Subtree**

• Using super fast max, each level takes O(1),   
so T(N) = O(lg lg n)

• Work is O(N lg lg N), E = 1 / lg lg N – Non-Optimal

**Doubly Logarithmic Max**

Doubly Logarithmic Tree Algorithms (N = 22k):

we count levels from leaves up

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| level | #trees | #kids | time | | work/tree | | work | |
|  |  | /tree | fast | lg | fast | lg | fast | lg |
| 0 | N/2 | 2 | 1 | 1 | 1 | 1 | N/2 | N/2 |
| 1 | N/22 | 2 | 1 | 1 | 1 | 1 | N/4 | N/4 |
| 2 | N/24 | 22 | 1 | 2 | 24 | 22 | N | N/22 |
| 3 | N/28 | 24 | 1 | 4 | 28 | 24 | N | N/24 |
| 4 | N/216 | 28 | 1 | 8 | 216 | 28 | N | N/28 |
| • | • | • | • | • | • | • | • | • |
| • | • | • | • | • | • | • | • | • |
| k-1 | 22k-1 | 22k-2 | 1 | 2k-2 | 22k-1 | 22k-2 | N | N/22k-2 |
| k | 1 | 22k-1 | 1 | 2k-1 | 22k | 22k-1 | N | N/22k-1 |
| **Order of Totals** | | | lglgN | lgN |  |  | Nlg lgN | N |

Conclusions:

• Doubly Logarithmic Max with fast algorithm is fast and reasonably efficient.

* Doubly Logarithmic Max with tree algorithm is no faster than standard binary tree reduction algorithm but is still work efficient.

**Fast, Efficient CRCW Max**

**• Accelerated Cascading**

• Use Work Optimal Binary Tree Reduction Algorithm to Get Problem Size Reduced -- Don’t go too far; don’t quit too soon

• Finish with Work Suboptimal, Fast Algorithm

**• In Case of max**

• Use lg N algorithm for lg lg lg N levels

• Reduces size to N / 2 lg lg lg N = N/lg lg N elements in   
lg lg lg N steps = O(lg lg N), taking O(N) work.

• Next, use CRCW doubly log-depth super fast algorithm.

• This requires no more than O(lg lg N) steps,   
Work is O(N/lg lg N × lg lg (N / lg lg N)) = O(N)

**• The Total is O(lg lg N) Time and O(N) Work**

**Summary of Accelerated Cascading**

Accelerated Cascading:

• Use work optimal, not super fast algorithm to reduce problem size.

• Use work suboptimal, super fast algorithm on remaining subproblem.

Reduce problem using lg tree algorithm for lg lg lg N levels

Work is O(N) since work for lg N levels is O(N)

Time is number of levels = lg lg lg N = O(lg lg N)

# of nodes left is N / 2 lg lg lg N = N / lg lg N

Attack remaining problem using CRCW doubly log algorithm

Time is clearly O(lg lg N) since it is this fast on N values

Work is O(N/lg lg N × lg lg (N / lg lg N)) = O(N)

Conclusions:

• Accelerated Cascading Max is fast and optimally work efficient.

• This is a case of using two types of specialists.

One is work efficient and reasonably fast

The other is very fast, but not real work efficient

The sum total is a fast, work efficient algorithm

**Program Flow Analysis**

**Basic type is Scalar Analysis**

**Concentrates on simple variable names**

**Indexed array ref. A[I] is treated as a reference to all of object A**

**This basic coverage ignores aliasing (multiple names for same object)**

**Basic Block**

**One in, one out sequence of code**

**Local Analysis – done on single basic blocks**

**Intraprocedural Analysis – done within procedures**

**Interprocedural Analysis – done across procedures**

**Control Flow**

**intra creates flow graph with procedure entry as initial node**

**inter creates a call graph with main body as initial node**

**Data Flow**

**determines accessibility of definitions and uses to each other**

**UD chaining – given a variable use, what definitions reach this use**

**DU chaining – given a variable definition, what uses are made of it**

**Data Flow Notations**

**Program P consists of procedures, one of which is denoted p.**

**local[p] = variables locally declared in p**

**formal[p] = set of parameters declared in p’s prototype**

**global[p] = global vars and arrays accessible to p**

**visible[p] = local[p] + formal[p] + global[p]**

**var(p) = visible(p)**

**We assume one entry / one exit procedures.**

**A flowgraph G = (N, E, s) refers to a directed graph (N, E) and an initial node s in N, where there is a path from s to every node of G. Nodes can be statements or basic blocks. Commonly, they are the latter.**

**Extracting Loops**

**Let G = (N,E,s)**

**(1) a node s’ ∈ N is the entry point for a loop in G iff there is an n’ ∈ N such that   
(n’,s’) ∈ E and s’ ≤ n’. (n’ branches back)**

**(2) Let s’ be an entry point of a loop. The max loop with entry s’ is G’ = (N’,E’,s’), where  
N’ = {n” | ∃ a path from n” to s’ which contains only nodes “dominated” by s’}.  
s’ dominates n” if s’ is on every path from s (start node) to n”.   
E’=E ∩ (N’×N’)**

**To do data flow analysis we wish to obey dominances, doing loop entries before their bodies, if conditions before their choices, etc.**

**More Notation**

**S\_DEFS = { s | s is a statement that defines variables }**

**S\_USES = { s | s is a statement that uses variables }**

**DEF[s] = { v | s is a definition of variable v }**

**USE[s] = { v | s is a use of variable v }**

**DEF[n] = { v | ∃ an outward exposed defn of v in n }**

**USE[n] = { v | ∃ an outward exposed use of v in n }**

**PRE[n] = VAR – DEF[n] /\* preserved defs, where VAR is set of all visible variables \*/**

**S\_DEF[n] = { s | s is an outward exposed defn in n }**

**S\_USE[n] = { s | s is an outward exposed use in n }**

**S\_PRE[n] = { s’ | s’ ∈ S\_DEFS and, for all  
s ∈ S\_DEF[n], DEF[s'] ≠ DEF[s] }**

**Reaching Definitions**

**RD[n] = { s | s ∈ S\_DEFS and s reaches n }**

**UD[n, v] = { s’ | s’ ∈ RD[n] and v ∈ DEF[s'] }**

**DU[n’, v] = {s | s ∈ S\_USE[n] for some n ∈ N,  
v ∈ USE[s] and s’ ∈ UD[n, v] }**

**Types of Data Flow**

**Notation: For any node n, pred[n] are all immediate predecessors of n and succ[n] are all immediate successors.**

**ReachIn[n] = { s | p ∈ pred[n] and s ∈ ReachOut[p] }**

**ReachOut[n] = (ReachIn[n] ∩ S\_PRE[n]) ∪ S\_DEF[n]**

**In some papers this is (In[n] - Kill[n]) + Gen[n]**

**In any case, we have a recurrence relation and hence seek a fixed point. We want the least fixed point.**

**MAY – determine if a property may be possible. This is attacked by assuming no elements satisfy, then union in all those that might have the property. By starting with the empty set, we get the Least Upper Bound (LUB). This is conservative.**

**MUST – determine if a property must be true. This is attacked by assuming all elements satisfy, then intersecting all those that must have the property. By starting with the everything, we get Greatest Lower Bound (GLB). This is conservative.**

**FORWARD FLOW – information flows from the root towards leaves of the control flow graph.**

**BACKWARD FLOW – information goes from the leaves towards the root of the control flow graph.**

**Reaching Definitions is MAY / FORWARD FLOW**

**Dataflow also often refers to ud and du chaining, meaning use/definition and definition/use chaining.**

**Reaching Definitions Algorithm**

**For i := 1 to NBlocks do begin**

**ReachOut[i] := S\_DEF[i];**

**ReachIn[i] := { }**

**end;**

**change := true;**

**while change do begin**

**change := false;**

**for i := 1 to NBlocks do begin**

**newIn := { s | p ∈ pred[n] & s ∈ ReachOut[p] };**

**if ReachIn[i] ≠ newIn then begin**

**ReachIn[i] := newIn;**

**oldOut := ReachOut[i];**

**ReachOut[i] :=**

**(ReachIn[i] ∩ S\_PRE[i]) ∪ S\_DEF[i];**

**if oldOut ≠ ReachOut[i] then change := true**

**end**

**end**

**end**

**Scalar Data Dependence**

**S1: A := 1.0;**

**S2: B := A + 3.1415;**

**S3: A := .333 \* (C – D);**

**… …**

**S4: A := (B \* 3.8) / 2.718;**

**S2 is true dependent on S1**

**S3 is anti-dependent on S2**

**S4 is output dependent on S3**

****

**Can use scalar data flow analysis to determine these dependencies.**

**Vector Data Dependence**

**for i := 1 to 100 do begin**

**S: A[2\*i] := B[i] + 1;**

**S’: D[i] := A[2\*i + 1]**

**end**

**If treat A, B and D as scalars then S’ is true dependent on S and S is anti-dependent on S’. But it can’t be so since S references only even numbered elements of A and S’ references only off numbered elements of A. Thus, we can do the iterations independently. But how do we recognize this? The basis is Diophantine analysis – provided indices are linear in the *for* variable. In above, we can ask if there is an integral solution to**

**1 ≤ X, Y ≤ 100 such that 2X = 2Y + 1**

**The answer is no, hence the indices cannot overlap. Even if we had for i:=1 to N, we can determine this.**

**for i := 2 to 10 do begin**

**S: A[i] := B[i] + 1;**

**S’: D[i] := A[i – 1]**

**end**

**The relation is X = Y – 1, for 2 ≤ X, Y ≤ 10. Can solve for all 2 ≤ X ≤ 9, so there is true dependence.**

**Testing Data Dependence**

**There are exact and inexact (but faster) tests for the existence of solutions to linear Diophantine equations. There is no test for polynomials of degree ≥ 4, and in fact exact solutions for lower degree polynomials are very hard.**

**One simple test is the GCD (Greatest Common Divisor) test. It is easiest seen by example.**

**for i := 1 to N do**

**for j := 2 to M do begin**

**S: A[2\*i + 2\*j] := …;**

**… …**

**S’: … := A[4\*i – 6\*j + 3]**

**end**

**These are independent if there is no solution to 2 A + 2 B = 4 C – 6 D + 3**

**Can rewrite as 2 A + 2 B – 4 C + 6 D = 3**

**But evenness of left says no solution is possible. This is recognized by gcd(left) = 2, gcd(right) = 3, but 2 is not a divisor of 3.**

**The technique is conservative, especially since it ignores regions. So, it says the following are possibly dependent**

**for i := 0 to 10 do**

**for j := 0 to 10 do begin**

**S: A[2\*i + j] := …;**

**… …**

**S’: … := A[–i + 2\*j – 21]**

**end**

**which translates to 2 A + B + C – 2 D = -21. gcd(left)=1; gcd(right) = 21. But the restriction that   
0 ≤ A, B, C, D ≤ 10 can be used to deny a solution since the left side can be no smaller than -20.**

**Examples of Vectorizing**

**for i := 1 to N do**

**S: A[i + 1] := A[i] \* B[i] (\* True Dependence \*)**

**==============================================**

**for i := 1 to 100 do begin**

**S: D[i] := A[i – 1] \* D[i]; (\* S depends on S’ \*)**

**S’: A[i] := B[i] + C[i]**

**end**

**Reorder S and S’**

**for i := 1 to 100 do begin**

**S’: A[i] := B[i] + C[i]**

**S: D[i] := A[i – 1] \* D[i]; (\* S depends on S’ \*)**

**end**

**Loop Distribution**

**for i := 1 to 100 do**

**S’: A[i] := B[i] + C[i]**

**for i := 1 to 100 do**

**S: D[i] := A[i – 1] \* D[i]; (\* S depends on S’ \*)**

**Change to Vector Operations**

**S’: A[1:100] := B[1:100] + C[1:100]**

**S: D[1:100] := A[0:99] \* D[1:100];**

**==============================================**

**for i := 1 to N do**

**for j := 1 to N do**

**C[i, j] := c[i – 1, j] – D[i – 1, j + 1]**

**Dependence is on outer loop only, so vectorize as**

**for i := 1 to N do**

**C[i, 1:N] := c[i – 1, 1:N] – D[i – 1, 2:N+1]**

**Dynamic Planning (Programming)**

**Based on "Principal of Optimality"**

**Technique requires that a problem have multiple stages at which decisions are made. We talk about stage i, and indicate that the system is in state i.**

**The Principal of Optimality says that the choice that is made at this stage is dependent only on state i and not on the policy that was used to make decisions at stages 1 to i-1.**

**Stated differently – the solution from stage i to n (the last stage) must be optimal in order for 1 to n to be optimal.**