CMSC 420: Data Structures

Spring 2001

`http://www.cs.umd.edu/~mount/420/`

Instructor: Dave Mount. Office: AVW 3209. Email: `mount@cs.umd.edu`. Office phone: (301) 405–2704. Office hours: Mon, Wed 3:00–4:00 I am also available immediately after class for questions. If the question is short (a minute or so) drop by any time, or send email. If you have trouble making my office hours contact me about scheduling another time.

Teaching Assistant: Betul Atalay. Office: AVW 1151. Email: `betul@cs.umd.edu`. Office Hours: Mon 4:00–5:00, Tue 3:30–4:30, or send her email to arrange a time.

Class Time: Tue, Thur 2:00-3:15 in CLB 0102.

Course Objectives: The goal of the course is to familiarize students with data structures, with algorithms for manipulating and retrieving information from these data structures, and for the mathematical techniques needed for analyzing their efficiency. Applications of these data structures in areas such as data processing, information retrieval, symbol manipulation, and operating systems will be discussed.

Required Texts:

- M. A. Weiss, *Data Structures and Algorithms in Java*, Addison-Wesley, 1999. I would prefer that you attempt to do the projects in Java. However, if you insist on using C++, you may purchase the C++ version (*Data Structures and Algorithms in C++*, 2nd Edition) instead.

  I will put both copies on reserve in the small library in the A.V. Williams building (AVW 3164). I am not sure just how compatible the two versions are, so beware.
- H. Samet, *Notes on Data Structures*, (Lecture notes sold at the University Book Center in the student union.)
- H. Samet, *Design and Analysis of Spatial Data Structures*, (Lecture notes sold at the University Book Center in the student union.)

Prerequisites: Grade of C or better in CMSC 330. Students are required to have a experience in programming some procedural language supporting data structures (either Java or C++). Students will be expected to design and debug fairly large programming projects. Knowledge of basic data structures (arrays, stacks, queues, and linked lists) and basic discrete mathematics (sets, simple proofs by induction, equivalence relations, permutations and combinations, manipulation of summations, graphs and trees, asymptotic notation of running times) is assumed.

Course Work: Course work will consist of a combination of written homework assignments and programming projects. Homeworks are due at the start of class. Late homeworks are not allowed (so just turn in whatever you have done by the due date). I would prefer that programming assignments be implemented in Java, but C++ is acceptable. The submission deadline will typically be due at midnight of the due date. They are subject

to the following late penalties: up to six hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for every additional day late.

There will be two exams, a midterm and a comprehensive final. Tentative weights: Homeworks and projects 35%, midterm 25%, final exam 40%. The final exam will be Friday, May 18, 10:30-12:30.

Homework assignments are to be written up neatly and clearly, and programming assignments must be clear and well-documented. A fraction of the grade for homeworks will be based on clarity, and for programming assignments will be based on the clarity of code and good documentation. Although you may develop your program on whatever system you like, for final grading your program must execute either on a Sun workstation in the WAM lab (for undergraduates) or the CSD junkfood lab (for grad students) or on the Unix cluster. If you develop your program on some other platform, it is your responsibility to see that it can be compiled and executed on one of these machines by the due date. If not, you will be asked to make whatever changes are needed and assessed an appropriate late penalty.

Some homeworks and projects will have a special challenge problem. Points from the challenge problems are *extra credit*. This means that I do not consider these points until *after* the final course cutoffs have been set. Each semester extra credit points usually account for at least few students getting one higher letter grade.

Academic Dishonesty: All class work is to be done independently. You may *discuss* class material, homework problems, and general solution strategies with classmates, but when it comes to formulating/writing/programming solutions you must work alone. Instances of academic dishonesty will be dealt with harshly, and usually result in a hearing in front of a student honor council, and a grade of XF.

Topics: The topics and order listed below are tentative and subject to change.

Introduction and Review: Basic data structures, algorithms, analysis. (1 week)

Basic Data Structures: Lists, arrays, trees, graphs and their representations. (2 weeks)

Ordered Dictionaries and Search Trees: Binary search trees, AVL trees, splay trees, skip lists, treaps, B-trees and variants (2–3, 2–4 trees, and red-black trees), optimal binary search trees, digital search trees and Patricia trees, amortized and randomized analyses. (4 weeks)

Unordered Dictionaries: Hashing functions, chaining, open addressing, linear hashing. (1 week)

Priority Queues: Binary and leftist heaps.(1 week)

Disjoint sets and Partitions: Union-Find, amortized analysis. (1 week)

Storage allocation: Dynamic storage allocation, buddy system, and garbage collection. (1 week)

Geometric Data Structures: Geometric preliminaries, grid files, and kd-trees (nearest neighbor and range queries), PM-quadtrees, segment and interval trees, range trees, R-trees. (3 weeks)

Homework 1: Basics

Handed out Tuesday, Feb 6. Due at the start of class Thursday, Feb 15. Late homeworks will not be accepted (in other words, turn in whatever you have).

General note regarding coding in homeworks: When asked to present an algorithm or data structure, do NOT give complete C++ or Java code. Instead give a short, clean pseudocode description containing only the functionally important elements, along with an English description and a short example.

Problem 1. Describe an enhanced stack data structure (storing integers say) that supports the operations `push`, `pop` and a third operation `getMin`, which returns the smallest element currently in the stack. The constructor is given the maximum stack size. All operations should run in $O(1)$ worst case time. Explain in English how your data structure works and provide pseudocode. For simplicity, you may assume that all operations are legal (i.e., no stack underflow or overflow).

Problem 2. Many numerical applications use square matrices that are *lower triangular*, that is, all entries lying strictly above the main diagonal are zero. An *upper triangular matrix* is defined similarly. Suppose that we want to implement a data structure to store such a matrix using the minimum space. To do this we store the entries in a one-dimensional array, and use an indexing function to map a row-column pair $[i, j]$ to an integer offset in this array.

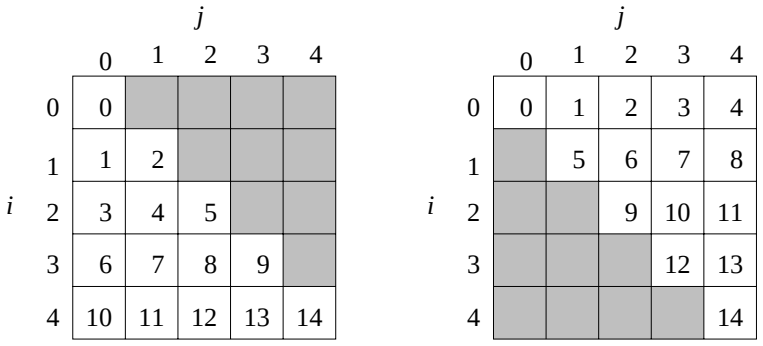


Figure 1: Lower- and upper-triangular array offsets for $n = 5$.

(a) As a function of n, derive the number of nonzero entries that an $n \times n$ lower (or upper) triangular matrix might have.

(b) Given an $n \times n$ lower triangular matrix, give the function that maps an index pair $[i, j]$ to an offset into a 1-dimensional array, as shown in the figure above left. Your function should run in $O(1)$ time.

(c) Repeat part (b) for an upper triangular matrix, using the offset method shown in the figure above right.

Problem 3. A *Fibonacci binary* tree is defined recursively as follows. A Fibonacci tree of height zero is just a single node. A Fibonacci tree of height one consists of a root node and a single right child node (the left child is null). In general, for $h \geq 2$, a Fibonacci tree of height h, denoted $F(h)$, consists of a root node, a left subtree consisting of a Fibonacci tree of height $h-2$, and a right subtree consisting of a Fibonacci tree of height $h-1$.

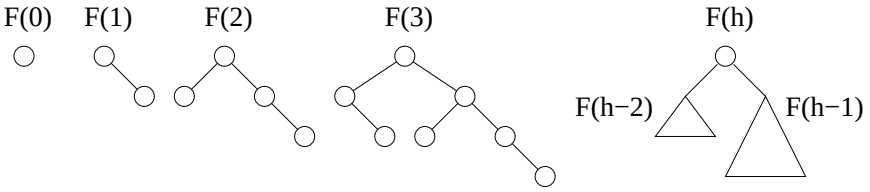


Figure 2: Fibonacci trees.

(a) Let $N(h)$ denote the number of nodes in a Fibonacci tree of height h. Express $N(h)$ as a recurrence. (Hint: There are two basis cases, for $h = 0$ and $h = 1$.)

(b) Using the recurrence from part (a), give a proof (by induction) that for all $h \geq 4$, $N(h) \leq (1.9)^h$. Be careful: For your basis case you will probably have to establish both $N(4)$ and $N(5)$. Why is this? (Hint: It may be helpful to use the fact that $1/(1.9)^k \leq 0.077$ for all $k \geq 4$.)

Challenge Problem: (Challenge problems are graded for extra credit. These points are not considered in the determination of final grade cutoffs, but can be used to improve your final grade if you are near a cutoff.)

You are working in an application in which space is cheap but time is precious. Design an `Array` data structure (e.g. a C++ or Java class) that supports the following operations, where each operation takes $O(1)$ time.

`Array(int n, int v)`: This is the constructor, which is given a positive integer n and an integer v, this allocates an array of size n. The indices are assumed to be in the range 0 to $n-1$. The quantity v indicates the default value each array entry is to have.

`void store(int i, int x)`: Stores the value x in the ith element of the array. You may assume that $0 \leq i \leq n-1$.

`int read(int i)`: Returns the value of the ith element of the array. If a value has been stored in this entry, then this value is to be returned. If no value has been stored here, then the default value v, given in the constructor, is to be returned.

You may assume the existence of a memory allocation procedure `new int[n]`, which allocates and returns an array of n integers in $O(1)$ time. Note that this routine does *not* initialize the elements of the array, and you may *not* assume anything about the initial array contents. Furthermore, because the constructor must run in $O(1)$ time you do not have the time to initialize all the entries to v.

Present a pseudocode description of your algorithm and explain in English how it works. Give a short example to illustrate how your method works. (Hint: You will need to allocate more than one array.)

Homework 2: Trees

Handed out Thursday, March 1. Due Tuesday, March 13 at the start of class. Late homeworks will not be accepted (in other words, turn in whatever you have).

Problem 1. Suppose that you have an (unbalanced) binary search tree with inorder threads. Assume that in addition to the usual `element`, `left` and `right` fields, there are additional boolean fields `leftIsThread` (`rightIsThread`), which are true if the left (right) child is a thread. Derive pseudocode for each of the following operations.

(a) Insert a new element x into the tree.

(b) Delete a element x from the tree.

(c) Given a nonnull pointer to any node p in the tree, return a pointer to the preorder (note: *not* inorder) successor of p. (Return null if there is no preorder successor.)

Problem 2. Show that if all nodes in a splay tree are accessed (splayed) in sequential order, the resulting tree consists of a linear chain of left children.

Problem 3. Suppose that you are given a binary three which consists of a linear chain of left children. Suppose further that the tree has exactly $n = 2^k - 1$ nodes, for some integer $k \geq 1$. Derive an algorithm which using only AVL single left- and right-rotations, maps this tree into a perfectly balanced complete tree. Explain how your algorithm works and give an example. (Hint: Start by getting correct node into the root position.)

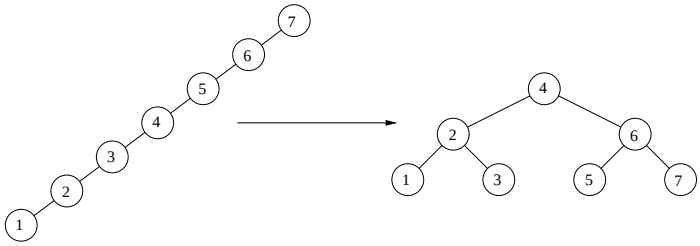


Figure 1: Problem 3

Challenge Problem: (Challenge problems are graded for extra credit.)

Suppose that you are given a linked list of records, where each record has a `key` field and a `next` field. All of the key fields are distinct, but there is no limit as to how large a key can be. You also have no idea as to how many elements are in this list. One of two cases exists, either this linked list ends at a record that contains a null next field, or else the linked list cycles back by having a next pointer that points to another record in the list. Describe an algorithm to determine which is the case. Here is the catch: You may use only $O(1)$ additional storage, and you may not alter the contents of the linked list in any way (even temporarily). The running time of your algorithm should be $O(n)$, where n is the number of elements in the list (which, by the way, you don't know).

Homework 3: Heaps and Union-Find

Handed out Tuesday, April 10. Due Thursday, April 19 at the start of class. Late homeworks will not be accepted (in other words, turn in whatever you have).

General note regarding coding in homeworks: When asked to present an algorithm or data structure, do NOT just give complete C++ or Java code. Instead give a short, clean pseudocode description containing only the functionally important elements, along with an English description, and a short example.

Problem 1. Consider the two heaps shown in the figure below.

(a) Show the result of merging these two heaps using the merge algorithm for *leftist heaps*. Show the npl value for each node in your final result.

(b) Show the result of merging these two heaps using the merge algorithm for *skew heaps*.

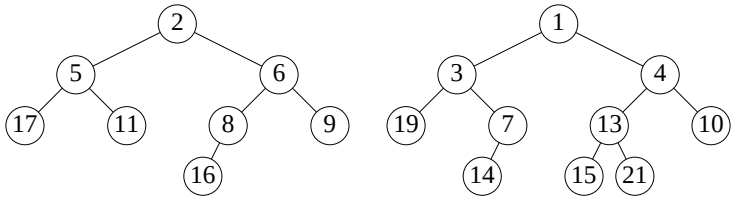


Figure 1: Problem 1.

Problem 2. Do problem 6.18 on page 220 of Weiss (on min-max heaps), only parts (a) and (b). For part (b), to illustrate your algorithm, show the result of inserting key 2 into the heap in Fig. 6.57 on page 220. Also show the result of inserting key 100 into the same heap.

Problem 3. Suppose that we are given a set of n objects (initially each item in its own set) and we perform a sequence of m unions and finds (using height balanced union and path compression). Further suppose that all the unions occur before any of the finds. Prove that after initialization, the resulting sequence will take $O(m)$ time (rather than the $O(m\alpha(m, n)$ time).

Hint: Classify the finds as being of two types: *simple*, those that traverse 0 or 1 links, and *complex*. Show that the total number of links traversed for each of these two types of finds is $O(m)$.

Challenge Problem: (Challenge problems are graded for extra credit.)

Show that if you insert the keys 1 through $2^k - 1$ into an initially empty leftist heap, then the resulting tree is a perfectly balanced tree of height k. (Hint: Use induction on k. Assume inductively that the tree containing the first $2^{k-1} - 1$ keys is perfectly balanced. Given that that you have just inserted m additional keys, for $1 \leq m \leq 2^{k-1}$, describe the structure of the resulting tree. It may help to consider the binary representation of m in doing this.)

Homework 4: Geometry and Memory Management

Handed out Tuesday, May 1. Due Tuesday, May 15 at the start of class. Late homeworks will not be accepted (in other words, turn in whatever you have).

Problem 1. In proving that range queries can be answered in $O(\sqrt{n})$ time on a kd-tree, we gave an argument that any horizontal or vertical line in the plane intersects $O(\sqrt{n})$ leaf cells of any kd-tree. This assumed that the tree was balanced, and in particular that the number of points in the left subtree is equal to the number of points in the right subtree for each node.

Generalize this argument to show that in dimension 3, an orthogonal plane (one that is orthogonal either to the x-, y-, or z-axis) intersects $O(n^{2/3})$ leaf cells of any balanced kd-tree having n total points.

Problem 2. Suppose that you are given a collection S of n intervals $[x_{\text{lo}}, x_{\text{hi}}]$ stored in an interval tree. Derive an efficient algorithm to answer the following query using the interval tree. Given a query interval $Q = [q_{\text{lo}}, q_{\text{hi}}]$ report all the intervals of S that have a nonempty intersection with Q. Your query algorithm should run in $O(k + \log n)$ time, where k is the number of intervals reported. Explain how your algorithm works, and derive its running time.

Problem 3. Give pseudocode for a procedure `dealloc(p)` that deallocates a block of memory p that was allocated using the dynamic storage allocation method described in Lecture 22. You may assume that simple utilities have been provided for manipulating the doubly-linked available space list. Remember that after deallocation, there should be no two consecutive available blocks. You may assume that the block being deallocated is not the first or last block in the heap. Your procedure should run in $O(1)$ time.

As always, do not just give the code. Explain how your code works in English, and if helpful give an illustration of the various cases that arise.

Challenge Problem: (Challenge problems are graded for extra credit.)

Some query problems on complex geometric objects can be reduced to query problems on points. For example, consider a vertical line segment whose upper endpoint is (x, y_{hi}) and whose lower endpoint is (x, y_{lo}). We can represent such a segment as a point in a 3-dimensional space with coordinates $(x, y_{\text{lo}}, y_{\text{hi}})$. Suppose we are given a collection of n vertical line segments S, which we have mapped to points in 3-space using this representation. Build a 3-dimensional range tree for this set of n points.

Explain how to use this range tree to answer 2-dimensional window queries for the line segment. In particular, given a rectangular query window Q in the plane given by its two corner points, Q_{lo} and Q_{hi}, describe how to use the range tree to report all the segments of S that intersect this rectangle in $O(k + \log^3 n)$ time, where k is the number of segments that have been reported.

Practice Problems for the Midterm

The midterm will be on Thursday, March 15. The exam will be closed-books, closed-notes, but you will be allowed one cheat-sheet, front and back to use for the exam. These problems do not necessarily reflect the actual difficulty of problems on the exam or the total length of the exam.

Problem 1. Short answer questions.

(a) Consider the function $f(n)$ defined: $f(n) = \sum_{i=1}^{n} i$. Express f asymptotically using big-Oh notation.

(b) In both skip lists and B-trees, the size of a node (i.e. the number of pointers that this node contains) is variable, ranging from some minimum to some maximum value. In skip lists we allocated nodes of variable size, and in B-trees we allocate nodes of the maximum size. For each of these data structures, explain why we did this.

(c) Suppose you know that a very small fraction of the keys in a data structure are to be accessed most of the time, but you do not know which these keys are. Among the data structures we have seen this semester, which would be best for this situation? Why?

(d) Consider the following dictionary structures: Unbalanced binary search trees, AVL trees, splay trees, B-trees, skip-lists. Suppose we insert one key into such a data structure that contains n nodes. For which of these data structures is it true that the number of structural changes is $O(\log n)$ in the worst case? (A *change* is any local alteration of the structure: modification of node contents, rotation, node split, etc.)

Problem 2. Suppose you are given a binary search tree in which each node contains an additional field, `size`, which contains the number of keys in the subtree rooted at this node. (Thus the `size` field of a leaf is 1, and the `size` field of the root is the total number of nodes in the tree, n. Using this size field, give pseudocode for a procedure `findKth(t, k)`, which returns the k-th smallest element in the search tree t. (You may assume that $1 \leq k \leq n$.)

Problem 3. Consider an *external binary search tree* in which data (key and associated data) is only stored at the leaves (external nodes) and the interior nodes only hold key values used to locate the leaf node. Each nonleaf (internal) node has exactly two children. An example is shown below, with internal nodes shown as circles and leaves as squares.

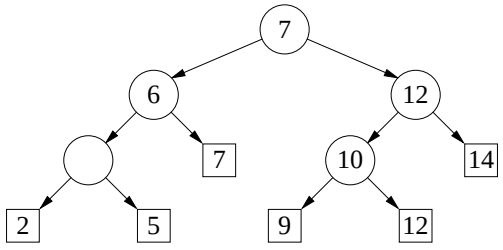


Figure 1: Leaf level data storage.

To implement these trees, we may assume each node has an additional field `nodeType` which is either `INTERNAL` or `LEAF`. Modify the tree insertion procedure for unbalanced binary trees to insert a key x into this type of search tree. (Make sure to modify the `element` fields of any ancestor internal nodes if needed.)

Problem 4.

(a) Show that the number of *node splits* performed when inserting a single node into a 2-3 tree can be as large as $O(\log n)$, where n is the number of keys in the tree. (E.g. give a general, worst-case example.)

(b) Show that the number of *node merges* performed when deleting a node from a 2-3 tree can be as large as $O(\log n)$, where n is the number of keys in the tree.

(c) Prove that if you start with an empty 2-3 tree, and perform n insertions (no deletions), the total number of *node splits* and *key rotations* is $O(n)$.

Problem 5. Consider the following recursive procedure, which, given a binary tree t, recursively visits both children at all the even levels of the tree, but visits only the left child at the odd levels of the tree. The initial call is `visit(root, 0)`.

```
visit(t, depth) {
    if (t != null) {
        visit(t.left, depth+1);
        if (depth is even) visit(t.right, depth+1);
    }
}
```

Suppose that t is a complete binary tree of height h such that every level of the tree has the maximum possible number of nodes. Recall that the root is at level 0.

(a) If i is even, how many nodes at level i are visited by this procedure? (Give your answer as an exact function of i.)

(b) If i is odd, how many nodes at level i are visited by this procedure? (Give your answer as an exact function of i.)

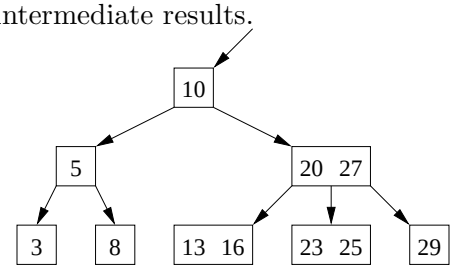
(c) Letting n denote the total number of leaves in this tree (all of which reside at level h), how many nodes are visited at the leaf level by this algorithm? You may express your answer using "big-O" notation, as a function of n. Explain.

CMSC 420:Spring 2001 Dave Mount

Midterm Exam

This exam is closed-book and closed-notes. You may use 1 sheet of notes (front and back). Write answers in the exam booklet. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (10 points) Consider the 2-3 tree shown below. Show the result of inserting key 14 into this tree. Show the intermediate results.



Problem 2. (40 points, 8 points each) Short answer questions.

(a) Given an undirected graph with V vertices and E edges, what is the average degree of a vertex in this graph?

(b) A binary tree is *full* if every node either has 0 or 2 children. Given a full binary tree with n total nodes, what is the maximum number of leaf nodes? What is the minimum number? Give your answer as a function of n (no explanation needed).

(c) Suppose you insert a new node into an AVL tree containing n keys. What is the maximum number of rotations that might be needed as part of this operation?

(d) Suppose that you delete an edge from a graph containing V vertices and E edges. What is the maximum number of edge changes (insertions and deletions) that might result in the MST? 1, 2, 3, all of them? Explain.

(e) Consider a splay tree containing n nodes, and let $x < y < z$ be three consecutive keys in the tree (that is, y is the only key in the tree whose value is between x and z). Suppose you do `splay(x)` followed immediately by `splay(z)`. What is the maximum depth of node y in the resulting tree? (Recall that the *depth* of the node is the number of edges between the root and this node.)

Problem 3. (15 points) Recall that a multiway tree T can be represented as a binary tree T' by using the firstChild and nextSibling pointers. If we think of the firstChild link as being the left link and the nextSibling link as being the right link, then we can traverse this binary tree T' as we would any binary tree. Answer each of the following and give a brief explanation or an example.

(a) A *preorder* traversal of T' is equivalent to which of the following traversals of T? Preorder, postorder, neither.

(b) An *inorder* traversal of T' is equivalent to which of the following traversals of T? Preorder, postorder, neither.

(c) A *postorder* traversal of T' is equivalent to which of the following traversals of T? Preorder, postorder, neither.

Problem 4. (20 points) Suppose we have a skip list with n nodes in which, rather than promoting each node to the next higher level with probability $1/2$, we promote each node with probability p, for $0 < p < 1$.

(a) Given a skip list with n nodes, show that the expected number of list entries of height $\log_{1/p} n$ is $O(1)$. Briefly explain.

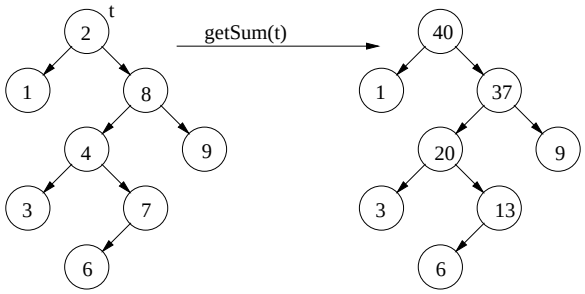
(b) Show that (excluding the header and sentinel nodes) the total number of links in such a skip list (that is, the total size of all the skip list nodes) is expected to be at most $n/(1-p)$.

Problem 5. (15 points) Consider a binary search tree, in which in addition to the standard fields (data, left and right) each node has an integer field called *size*, which stores the number of elements in the subtree rooted at this node. In a *range query* we are given two key values $x_1 \leq x_2$ and wish to return a count the number of elements in the tree whose key value x satisfies $x_1 < x \leq x_2$. Give pseudocode for this operation, and briefly explain how your algorithm works. Your algorithm should run in $O(h)$ time, where h is the height of the tree (but I'll give partial credit for any correct algorithm).

Practice Problems for the Final Exam

The final will be on Friday, May 18, 10:30–12:30. The exam will be closed-books, closed-notes, but you will be allowed two cheat-sheets, front and back to use for the exam. These problems do not necessarily reflect the actual difficulty, length, or coverage of problems on the exam or the total length of the exam.

Problem 1. Let t be a standard binary tree with the usual fields `data`, `left` and `right`. Write a procedure `makeSum(t)`, which replaces the `data` field of each node in the tree, with the sum of its data fields of it and all of its descendents in the original tree. An example is shown below.



Problem 2. Suppose you want to store a large set of integer (x, y)-coordinates of points in 2-dimensional space. Each point is associated with a string identifier. Which of the following data structures would you choose to answer each of the following queries. You want to optimize the speed of the query in the expected case: Skip-lists, B-trees, splay trees, Union-Find trees, leftist heaps, skew-heaps, kd-trees, interval trees, patricia tries, hashing? Explain.

(a) `nearest(x,y)` returns the nearest point in the set to (x, y).

(b) `find(x,y)` returns true if the point with coordinates (x, y) is in the set and false otherwise. (Note that the coordinates must match exactly.)

(c) `locate(id)` returns the coordinates of the point whose string identifier matches the string id. An error is produced if no such point exists.

(d) `prefix(id)` is given a string id. If there is a unique string having id as its prefix, the coordinates of the associated point are returned. If there are zero or two or more such points, then an error is produced.

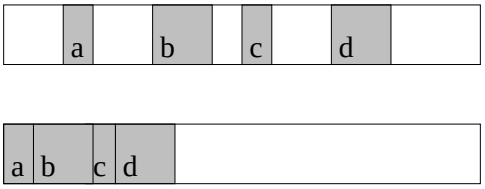
Problem 3. Suppose that rather than using heights (rank) to determine how to merge trees in Union/Find, we instead use *size*, i.e. number of nodes. In particular, in the root of each subtree we store the the number of nodes in this subtree. When we Union two trees, we link the smaller tree as a child of the root of the larger tree.

Prove the following (by induction on the number of Union's performed). Let T be a Union/Find tree that was built using Union's *by size*. Let n be the number of nodes in this tree, and let h be its height. Then $h \leq \lg n$. (Recall that $\lg n = \log_2 n$.)

Problem 4. We know that if we insert n points randomly into a binary tree, then the tree will be balanced, in the expected case, meaning that it will have height $O(\log n)$. If we insert them in sorted order the tree will have height $O(n)$, however.

Suppose that we have n uniformly distributed points in the plane. These points might be generated by calling a random number generator to create each x and y coordinate. We know that if we insert these points randomly into a point k-d tree, the expected height of the k-d tree will be $O(\log n)$. Suppose that we sort the points according to their x-coordinates (assume there are no duplicate x-coordinates) and we insert them into a k-d in sorted order. What can we say about the expected height of the resulting tree? Explain.

Problem 5. Professor Flintstone proposed the following improvement to the stop-and-copy garbage collection method. Rather than maintaining two banks of memory, we maintain only one bank. When compressing memory we simply move each block up to the next available location in the bank. What problems (if any) will be experienced in implementing Professor Flintstone's approach?



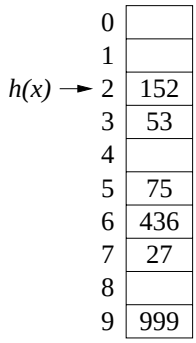
CMSC 420:Spring 2001 Dave Mount

Final Exam

This exam is closed-book and closed-notes. You may use 2 sheets of notes (front and back). Write answers in the exam booklet. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (15 points)

Consider an insertion of they key $x = 222$ into the hash table shown in the figure. For each of the following probing methods, indicate the sequence of table entries that would be probed, and the final location of insertion for the key. Assume that $h(x) = 2$. (In each case start with the same initial table.)



(a) Linear probing.

(b) Quadratic probing.

(c) Double hashing, where $g(x) = 7$.

Problem 2. (35 points, 5 points each.) Short answer questions.

(a) Of the following dictionary structures, which could be used to answer the following query efficiently. Given a key x, find the three smallest items in the dictionary whose key value is greater than or equal to x. Pick any/all that apply: Sorted array, AVL tree, skip list, hashing. (Explain briefly.)

(b) Consider a connected, undirected graph G with positive edge weights, where no two edges have the same weight. True or false: The edge of minimum weight must be in G's MST. True or false: The edge of maximum weight must *not* be in G's MST.

(c) What is the definition of *npl* (null path length) used in leftist heaps? Draw a leftist heap with at least 7 nodes and indicate the npl values for each node.

(d) Suppose that you modify the union/find data structure so that it performs balanced unions (using tree height), but it does not perform path compression. Starting with an initial collection of n isolated sets (each containing a single element) what is the worst case total running time for a sequence of m unions and finds? (You may assume $m \geq n$.)

(e) What is the difference between internal and external fragmentation?

(f) Consider a binary heap with n nodes. How long does it take to find the third smallest key in the heap? (You do not have to delete this key, just report its value.) Explain briefly.

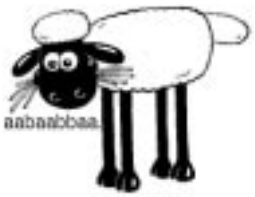
(g) Consider a buddy system allocation. What is the size and starting address of the buddy of a block of size 2 located at starting address 12?

Problem 3. (10 points)

Consider a multiway tree implemented using the firstChild-nextSibling representation. You may assume that each node t has a parent link, `t.parent`. Given a node t in such a tree, present pseudocode for the operation `evert(t)`, which makes t the root of its tree. (Same as in the project). If it helps, you may assume there exist operations `cut(t)`, which unlinks a node t from the tree (by deleting the edge to its parent) and `link(s, t)`, which links a tree root s as a child of t.

Problem 4. (15 points)

Consider a string $s =$ "aabaabbaa.". Assume that the alphabet consists of the three symbols, {'a','b','.'}.



(a) What are the 10 *substring identifiers* for s?

(b) Draw a suffix tree for s. You may use a standard trie. (If you do not know how to answer part (a), show a trie on any 10 different substrings of s.)

Problem 5. (10 points) In class we showed that given a balanced kd-tree with n points, an *orthogonal line* intersects at most $O(\sqrt{n})$ cells of the tree.

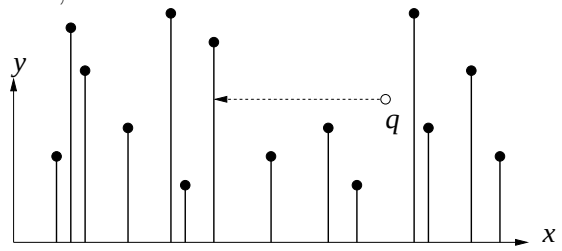
(a) Show that for every n, it is possible to construct a balanced kd-tree with n points such that there is a line (nonorthogonal) such that this line intersects all (or at least half) of the cells of the kd-tree.

(b) Using your answer to (a), give an example of a rotated (nonorthogonal) range query, in which none of the points are in the range, but every cell of the kd-tree would have to be visited. Explain.

Problem 6. (15 points)

Design a data structure and a search algorithm to solve the following problem. It should be able to answer the query in $O(\log^2 n)$ time. A high-level description of your data structure, the search algorithm, and its running time are sufficient.

You are to store a set of n vertical segments, where each segment extends from a point $p_i = (x_i, y_i)$ down to the x-axis. You may assume that all coordinates are positive. The query consists of a single point $q = (q_x, q_y)$. Imagine that you shoot a bullet horizontally to the left from q. Return the *first* vertical line segment in the data structure that is hit by this bullet path. If no segment is hit, then return `null`.



Programming Assignment 1: Dynamic Minimum Spanning Trees

Handed out Thur, Feb 22. The program must be submitted to the grader by the Tuesday Apr 3 (any time up to midnight). Information on submissions will be forthcoming. See the syllabus for the late policy.

Overview:

One of the challenges of real data structure design is that data structures do not operate in isolation, but rather many data structures interact with each other to produce an combined objective. This project will explore this by having you to implement a number of different data structures (graphs, multiway trees, and heaps) to achieve the objective of maintaining a minimum spanning tree (MST).

Minimum spanning trees are important in networking applications, since they provide a way of connecting all the nodes in a graph together in a minimal structure. In modern wireless and mobile ad-hoc networking (MANET) systems, devices are constantly being added and removed from the network, and transmission delays along links are variable. Thus it is important to be able to maintain an MST as the underlying network characteristics change. Due to transmission failures, the graph may not be connected, and hence it is necessary to maintain a collection (or forest) of MST's.

The goal of this project is to input an undirected graph with weighted edges, compute an initial MST for the graph. Then as changes occur in the graph structure, the MST is to be updated. These changes include the insertion and deletion of vertices, insertion and deletion of edges, and changing the weights of edges.

Graph Input:

All input will occur from the standard input and output to the standard output. The input begins with a description of the graph. This description starts with the number of vertices, followed by a list of information about the vertices, one per line. Each vertex has an associated string *identifier* and a location given as its (x, y) coordinates (as floats). You may assume that each string identifier consists of at most 20 characters. The location information is not used in this project, but will be used in the subsequent project. For example, here is the input for three vertices.

```
3                                       (number of vertices)
V128.8.10.14      123.4    276.52       (vertex identifier and x,y coords)
V128.8.128.423    1.203   -2.773
V128.8.10.268     324.2   -934.72
```

To specify the edges, the number of edges is given followed by a list of edges, each consisting of the identifiers of the endpoints and the edge weight (float). You may assume that all weights are positive.

```
2                                       (number of edges)
V128.8.10.14   V128.8.10.268   43.56    (endpoints and edge weight)
V128.8.10.268  V128.8.128.423  170.36
```

You may assume that the first part of the input follows these specification (that is, you do not have to check for input format errors), but you should verify that the vertex identifiers are unique (a duplicate vertex should simply be ignored), and that the edges are distinct (no multiple edges) and the endpoints are valid vertices.

MST by Prim's Algorithm:

You may assume that this initial graph is connected. After inputting the graph, compute its MST using Prim's algorithm. For consistency, you should use the first vertex input (V128.8.10.14 in this case) as the starting vertex for the algorithm. Output the sequence of edges that have been added to the tree and their associated weights.

Update Directives:

The rest of the input consists of a sequence of directives, each indicating an operation to be performed on the graph. For those directives that alter the graph, you should incrementally update your MST after this operation. Note that in the course of performing these operations, the graph may become disconnected. In this case you will need to maintain a separate MST for each connected portion of the graph.

Print the MST: Print the contents of the current MST(s), given a particular vertex v as the root. Details on the output format will be provided later, but basically involve printing the MST according to a preorder traversal of the tree.

Path: Given two vertex identifiers, compute and print the path between them in the MST. If they are not in the same tree, then print a message indicating this.

Insert vertex: Insert a vertex (with no edges) in the graph given its identifier and x and y coordinates.

Insert edge: Insert an edge between two existing vertices into the graph.

Decrease weight: Decrease the weight on an existing edge in the graph by the given amount.

Delete vertex: Delete the given vertex from the graph and all its incident edges.

Delete edge: Delete the given edge from the graph.

Increase weight: Increase the weight on an existing edge by the given amount.

Here is a summary of the update directives.

```
print-mst       u       Print the MST using u as the root
path            u v     Print the path from u to v in the MST
insert-vertex   u x y   Insert vertex u in the graph
```

```
insert-edge     u v w   Insert edge (u,v) with weight w in the graph
decrease-weight u v w   Decrease the weight of edge (u,v) by w units
delete-vertex   u       Delete vertex u from the graph
delete-edge     u v     Delete edge (u,v) from the graph
increase-weight u v w   Increase the weight of edge (u,v) by w units
quit                    This is the last directive.
```

You may assume that the input will adhere to these conventions. However if any operation is illegal (attempting to insert a vertex that already exists or deleting an edge that does not exist), the operation should be ignored and an error message should be printed. After each update operation, print a message indicating what operation was performed, and which edges were added to or deleted from the MST.

Implementation Requirements:

You are required to implement at least three separate data structures for the assignment: (1) an undirected (weighted) graph, (2) a multiway tree, and (3) a binary heap. The undirected graph must be implemented using an adjacency list, and edges must have cross links between (as discussed at the end of Lecture 4 on Feb 8). The MST is to be stored in the multiway tree, using the firstChild and nextSibling representation discussed in class. It will also be necessary to have a parent pointer for each node for path finding operations. The binary heap is used in Prim's algorithm. (See Section 6.3 in Weiss for an example, but you may implement the heap however you like.) The update operations must be performed incrementally, by making the fewest possible changes to the MST. In particular, it is illegal to completely rebuild the MST from scratch after each operation using Prim's algorithm.

I would suggest that you begin by implementing just Prim's algorithm and the procedure for printing the MST. Then go about adding the other operations. This way you can get partial credit if the other operations are not working. Implement the update operations one by one, so that if you run out of time, at least you will have a partial subset that is supported. We will provide test data later.

You are not required to provide an efficient method for mapping a vertex identifier to a vertex object. (This will be addressed in the next project.) I used the built-in Hashtable class in java.util, but you can just do a linear search if you like.

You are allowed to use simple, linear data structures from the Java or C++ libraries (e.g. strings, linked-lists, vectors, stacks, queues). You are not allowed to use anything more sophisticated though (no trees, dictionaries, priority queues, etc.). The only exception is you may use any data structure you like for mapping vertex identifiers to actual vertex objects, as mentioned in the previous paragraph.

Implementation Suggestions:

This project involves the integration of a number of different data structures. The easy but ugly way to do this would be to make one huge class in which you store all the information for each vertex (adjacency list, multiway tree, information for Prim's algorithm, etc). However, this is not at all modular. In C++ this could be handled cleanly by multiple inheritance, but Java does not support multiple inheritance. For full credit, you should modularize the major project components.

The binary heap must support an operation `decreaseKey()`, which decreases the key value of an entry in the heap, and rearranges the heap contents accordingly. Note that the hard part of this operation is locating where the item is stored in the heap. There is no fast method for searching a heap for a given key other than checking every value in the heap (think about this). You should provide a cross-reference mechanism by which each vertex can find its corresponding node in the heap in $O(1)$ time.

The graph can be represented by a straightforward implementation of the adjacency list described in class along with cross links between the edges. Note that you are not told how many vertices there are. Your data structure should be capable of handling any number of vertices in an efficient manner.

The multiway tree will need to provide a number of operations to support dynamic maintenance of the MST. For example, you will need to be able to merge trees (when edges are added), split trees (when edges are deleted), reorganize the tree to make an arbitrary node the new root (which is required for the print-MST operation), determine whether two vertices are in the same tree, compute the path between two vertices in the tree, determine whether a given graph edge is in the tree, etc. You will also need to be cross referenced with the graph, so that given an edge of the graph the corresponding edge in the tree can be found (if it is in the tree) and given an edge in the tree the corresponding edge in the graph can be found.

MST Updates:

The incremental MST update operations should be performed in a reasonably efficient manner. The insert edge operation may generally result in the addition of a new edge into the MST. To determine this, let u and v be the endpoints of this edge. Find the path from u to v in the current MST. If they are in different components, then just add the edge to the MST (thus merging the two trees). Otherwise, find the maximum weight edge (x, y) on the path between them. If $w(u, v) < w(x, y)$, then replace edge (x, y) with edge (u, v) in the MST. Otherwise there is no change. Decreasing the weight of a edge operates in much the same way, since the edge may now be added to the tree.

Edge deletion is more complex because the loss of an edge (u, v) may result in splitting the MST into two separate trees. You need to find the lowest weight edge (if any) connecting these two subgraphs. To do this you can label (or color) all the vertices as being in one subgraph or the other by traversing the tree. Then enumerate all the edges of the graph, searching for the lowest weight edge with one endpoint in one subgraph and one endpoint in the other. This edge is then added to the MST. Increasing the weight of an edge is similar because this edge may no longer be part of the MST. Deleting a vertex from the graph is quite messy, because it may result in many complex changes to the tree. One simple implementation involves deleting the edges incident to this vertex, one by one, and then removing the vertex itself.

Programming Assignment 1: Further Information

Major Data Structures:

The main data structures consist of the graph, which is represented using an adjacency list representation and a multiway tree. The graph data structure will likely consist of at least two types of nodes: *vertices* (shaded in the figure below) and *edges*. Each vertex node contains information such as the vertex identifier, coordinates and a pointer to the adjacency list. Each edge node contains information about each edge, including the neighboring vertex, the edge weight, and the cross link to the twin edge (not shown in the figure). In the figure the edge neighbor is given as an identifier, but this would actually be a pointer (or index) to the vertex node. In addition there will be a link for accessing the next element in the adjacency list.

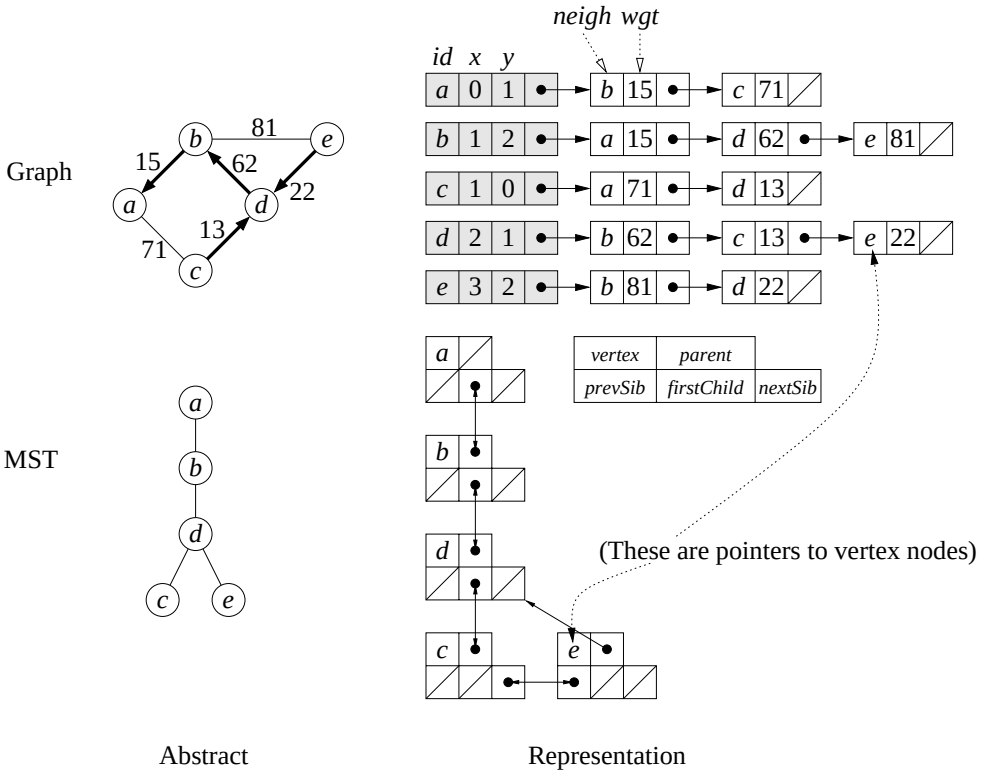


Figure 1: Overview of major data structures.

When Prim's algorithm runs, the link from each vertex u to its predecessor corresponds to a link in the MST. Each MST is a multiway tree. Each node of this data structure contains a reference to the associated vertex in the graph. (In the figure this is given as the vertex identifier, but actually this would be a pointer to the vertex node.) In addition the node might contain pointers to the first child, next sibling, previous sibling, and parent in the multiway tree. Note that there may be

many MST's, one for each connected component, and the associated data structure should allow for this possibility.

Evert:

The evert operation is not a requirement of the project, but I found it to be a very useful operation, and it simplified many aspects of the project. Given an MST (represented as multiway rooted tree) and given any node u in this tree, the operation `evert(u)` produces a modified tree which has the same edge structure but in which u is made the root of the tree. The figure below shows an example of the evert operation, on the abstract multiway tree (left) and on the firstChild-nextSibling representation (right). We have shown parent links as dotted lines.

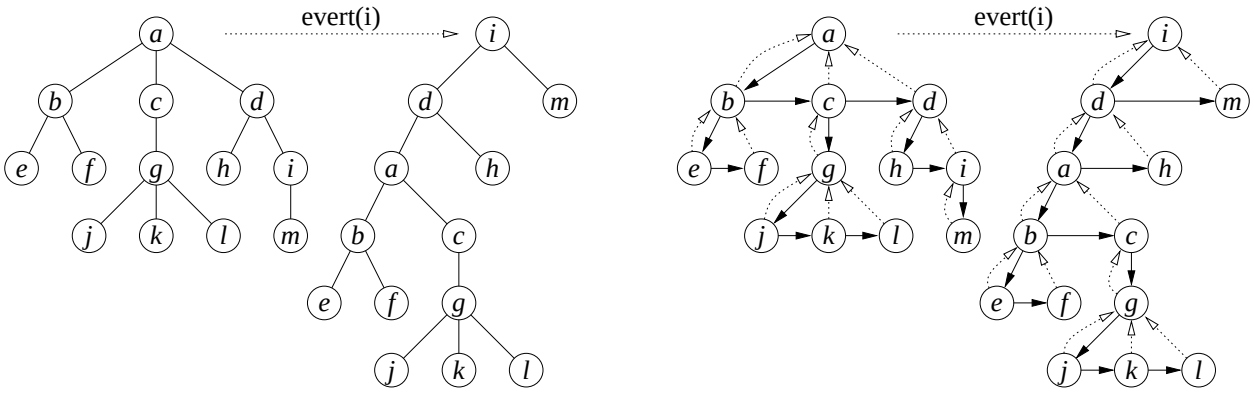


Figure 2: Evert operation on a multiway tree.

As a hint to implementing evert, you may want begin by implementing two utility operations, `cut(u)`, which cuts a node u and its associated subtree out of a tree by deleting the link to its parent in the multiway tree, and `link(u,v)`, which assumes that u is the root of a tree which has been cut, and links this subtree rooted at u as a new child of node v. The figure below show the result of applying `cut(c)` and `link(c,d)`. Evert can be implemented as a sequence of cuts and links.

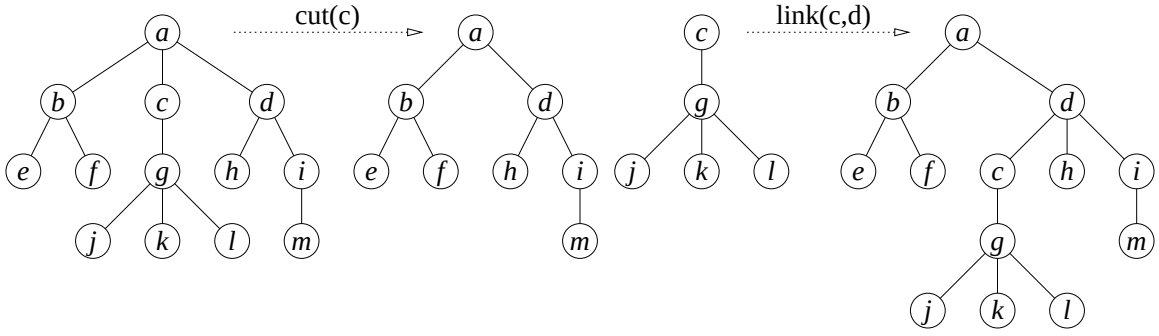


Figure 3: Application of `cut(c)` and `link(c,d)`.

Output Format: Your output be should similar to the output in our test*.out files (not the test*.debug file) ignoring case of letters and blanks. See below for how to get a copy of the files. For each directive output the directive, for the path and print-mst directives print the associated output, and print any error message, if needed.

Each MST should be printed in preorder. Start each line with a string ". . . " to indicate depth. For consistency of output, the children of each node should be listed in increasing order by the string identifier. One easy way to implement this ordering is that whenever a node u is added as a child to a another node v, insert u in sorted order among v's children.
For example, for the tree shown in Figure 2, the operation "`print-mst i`" would produce the output shown on the right. This can be done by invoking `evert(i)`, and then applying a preorder traversal of the tree.

```
Directive----------------> print-mst i
  i
  . d
  . . a
  . . . b
  . . . . e
  . . . . f
  . . . c
  . . . . g
  . . . . . j
  . . . . . k
  . . . . . l
  . . h
  . m
```

Test Data: Test data can be found either as a zip file in either of the following locations, one is a zip file with all the data and the other directs you to an ftp site from which the data can be downloaded.

```
ftp://ftp.cs.umd.edu/pub/faculty/mount/420/420Stuff.zip    (zip file)
ftp://ftp.cs.umd.edu/pub/faculty/mount/420/420Stuff        (directory)
```

Submission Details: Your submission will consist of an encapsulation of files which you will email directly to the grader (betul@cs.umd.edu). Please read the following instructions carefully, since significant deviations can result in the loss of points. Your program will be graded based on how it runs on the AITS cluster. (It doesn't matter how it runs on your PC.) The encapsulation must include the following items:

README: There must be a file called `README`, which contains any helpful information to the grader on how to compile and run your program. This includes:

Your name: (Very important.)

How to compile: You should provide a `Makefile` and the grader should just have to enter "make" to compile your program.

How to run it: Explain how to execute your program on a given input file.

Special Features: List special features or extensions, which you would like the grader to consider for extra credit.

Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications. (As a service to the grader, please be complete here. If she finds bugs that were not listed here, she reserves the right to double the normal deduction.)

File directory: If you have multiple source or data files, (other than those created by the compiler), please explain the purpose of each.

Source Files: All the source files and header files. (Do not include systems files that are available on your platform).

Input Files (if applicable): If you would like us to run your your program on specific test data (e.g. either because of special features you have or limitations that prohibit us from using our test data) please include that.

DO NOT INCLUDE: Because the grader's disk quota is limited, please delete all executable and object (.o or .class) files prior to submission.

To submit, store everything (README, Makefile, source) in a directory whose name is your login name (or something easily identifiable) e.g. `smith`. In particular, DO NOT send directories with names like `project1`, since it will make it harder for the grader to keep track of submissions. Be sure to delete any unnecessary files (executable or ".o" files).

Then email everything to the TA. There are a few ways to do this. I prefer either the first or second method below. If you are short on disk space, then try the third option.

Use the submit420 script (preferred): Use the following script, which is accessible from my WAM, CSD junkfood or cluster accounts:

```
% ~mount/submit420 smith          (from WAM or CSD)
% ~dm42001/submit420 smith        (from the AITS cluster)
```

where "smith" is the name of the directory with your files. Watch for any error messages as the script runs. It is not very robust.

Encapsulated attachment: This is just a manual version of the previous.

(1) Encapsulate everything into one file, either by tar'ing your directory (`tar -cvf smith.tar smith`), and then compress the tar file using gzip (or compress) (`gzip smith.tar`).

(2) Send the resulting bundle as an email attachment to the grader (`betul@cs.umd.edu`). If your mail system does not support attachments, you can use mpack: for example you could use `mpack smith.tar.gz betul@cs.umd.edu`.

If you discover an error in an earlier submission, you may repeat your submission. But in consideration to the grader's time and disk quota, please keep the number of submissions to a minimum. She will grade the last submission she receives. Server errors are rare but can occur. Be sure to save your final submission somewhere safe (very important).

Please check your email the next day after submitting (very important). If the grader has problems unpacking your files, she will ask you to resubmit. The grader reserves the right to deduct points for people that deviate significantly from these instructions.

Programming Assignment 2: Euclidean MST's and kd-trees

Handed out Tue, April 10. The program must be submitted to the grader by the Thu, May 3 (any time up to midnight). Use the same submission method as in Project 1.

Overview:

In Part 1 of the programming assignment you implemented a program that inputs a graph, computes it minimum spanning tree (MST), and supports various sorts of updates. Recall that the application was for dynamic support of mobile wireless networks. In real mobile networks, network nodes reside somewhere in space, which we will model by assigning an (x, y) coordinate to each vertex. Node communication is most reliable when nodes are close to one another, and hence it makes sense to connect the nodes into an MST, where the weight of an edge is the Euclidean distance between two points. Recall that the Euclidean distance between points p and q is

$$dist(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Your program will input a set of n points (but no edges). As in Project 1, each point will consist of an string identifier and (x, y) coordinates. Implicitly this defines a graph, called the *complete Euclidean graph*, whose vertices are these points and whose edges are all pairs of vertices, and whose the weight of an edge is the distance between these points. However, the degree of each vertex in such a graph would be $n - 1$, which is unacceptably large for large point sets. Instead, you will compute a sparse graph, called the *8-sector graph* in which the average degree of each vertex is at most 16 (and typically much smaller). This graph will have the nice property that the MST of this graph will be the same as the MST of the complete Euclidean graph.

The 8-sector graph:

The 8-sector graph is defined as follows for a set of points S (in the plane). For each point p, we partition the space about p into eight 45 degree angular sectors, as shown in the figure below. For each of these sectors we find the closest point of S to p lying within the sector. (Note that some sectors contain no point.) We then create an undirected edge from p to each such point. By repeating this for each point in S, the resulting graph is the 8-sector graph. If a point lies on the boundary between two sectors, you may assign it to either one. If two or more points are equidistant to p within a given sector, you may connect p to either of them.

Note that a vertex may have more than eight neighbors. This is because another point (such as q in the figure) may add an edge going to p, whereas p does not create an edge going to this point. Note that each point of S can cause at most eight new edges to come into existence, implying that the total number of edges is $E \leq 8n$. Since the average vertex degree in a graph is $2E/n$ it follows that the average degree in the 8-sector graph is at most 16. The final sector graph and resulting MST are shown in the figure on the right. (Note: I did this by hand, so there may be some errors in the figure. If anything looks fishy, check with me.) The 8-sector graph is always connected.

In order to build the 8-sector graph, you will need to write a procedure which given a point p, a sector i, returns the nearest point of S in the sector (or returns null if no such point exists). This

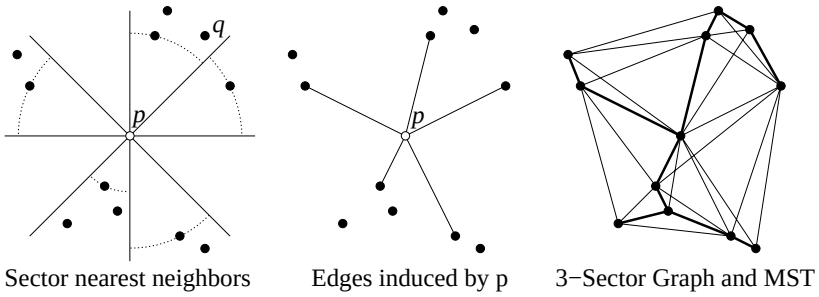


Figure 1: The 8-sector graph.

is called a *sector nearest neighbor query*. The sector i can be given as an integer from 0 to 7. You will do this by maintaining a kd-tree for the point set S. Whenever a point is added to the graph, you will perform eight sector nearest neighbor queries to your kd-tree, and insert an edge into your graph for each non-null result. Then you can insert the new point into the kd-tree. To compute the MST, invoke Prim's algorithm on the resulting undirected graph. (The fact that the MST is a subgraph of the 8-sector graph is an interesting geometric theorem, which will be discussed in class. In fact, it works even if the 60 degree angular sectors are used. The reason for using 45 degree sectors is that it is easier to process the sector nearest neighbor queries.) The same procedure is invoked whenever a new point is added.

Deleting a point p from the 8-sector graph is tricky. It is not sufficient to just delete every edge incident to p. The reason is that p may have been a sector nearest neighbor for some other point (such as q in the figure). Thus the deletion of p would require q to compute its new sector nearest neighbor for this sector. To delete a point p, we first delete p from the kd-tree (even though it is still in the graph). We enumerate all the neighbors of p in the current graph. For each such neighbor q, we determine the sector that p lies in relative to q. Then we recompute q's sector nearest neighbor using the kd-tree, and (if it is non-null) we add this edge to the graph. When this has been done for each of the neighboring points of p, we delete p from the graph and all of its incident edges. You can reuse the code for Project 1 for maintaining the MST under insertion and deletion of edges.

Input:

As with Project 1, read input from standard input and write output to standard output. The first line of the input contains four floating point numbers, $x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$. These define a rectangle which will contain all the subsequent points, and will be used in the construction of the kd-tree. The next line contains the number of points, followed by a list of information about the points, one per line. As in the first project, each point has an associated string *identifier* and a *location* given as its (x, y) coordinates (as floats). As mentioned earlier, no edge information is given. You may assume that no two point coordinates are identical.

After inputting the points compute a kd-tree for the points (see the description below) and output the contents of the kd-tree. Compute and output the 8-sector graph. Then compute the graph's MST using Prim's algorithm. Use the first vertex input as the starting vertex for Prim's algorithm. Output the sequence of edges that have been added to the MST and their associated weights.

Update Directives:

The rest of the input consists of a sequence of directives, each indicating an operation to be performed on the point set. Each operation may result in a change to the kd-tree and/or the MST. You should incrementally update each after the operation.

Print the Graph: Print the contents of the graph. Format: `print-graph`. When printing vertices, print both the identifier and coordinates.

Print the kd-tree:

Print the contents of the kd-tree according to an in-order traversal of the tree. First print the left subtree, then the root, then the right subtree, along with a prefix to indicate the depth in the tree. Also print an indicator ('`|`' or '`--`') to indicate whether the cutting dimension is x or y, respectively. Format: `print-kd`.

```
. . d:(120.0, 60.0) |
. . . e:(260.0, 100.0)--
. b:(200.0, 160.0)--
. . f:(160.0, 320.0) |
a:(360.0, 300.0) |
. c:(520.0, 220.0)--
. . g:(440.0, 340.0) |
```

Print the MST: Print the contents of the current MST, given a particular vertex u as the root, using a preorder traversal as in Project 1. Format: `print-mst u`.

Path: Given two vertex identifiers, compute and print the path from u to v in the MST. Format: `path u v`.

Insert point: Insert a point in the graph given its identifier and x and y coordinates. Add the point to the kd-tree and create edges as described above by computed the 8-sector nearest neighbors, and update the MST. Format: `insert-point u x y`.

Delete point: Delete the given point from the graph and kd-tree. Modify the 8-sector graph as described above and update the MST. Format: `delete-point u`.

Quit: Terminate the program. Format: `quit`.

You may assume that the input will adhere to these conventions. You may assume that there will be no attempt to insert a duplicate vertex or delete a nonexisting point. After each update operation, print a message indicating what operation was performed, and which edges were added to or deleted from the MST.

Implementation Requirements:

The only new data structure requirement is that you implement the kd-tree and that you implement your sector nearest neighbor queries in a reasonably efficient manner.

As before, you are allowed to use simple, linear data structures from the Java or C++ libraries (e.g. strings, linked-lists, vectors, stacks, queues). You are not allowed to use anything more sophisticated though (no trees, dictionaries, priority queues, etc.). The only exception is you may use any data structure you like for mapping vertex identifiers to actual vertex objects, as mentioned in the previous paragraph.

Implementation Suggestions:

Much of the basic kd-tree code (e.g. for insertion and deletion) will be given in class, and you may adapt this code for your purposes. The trickiest thing to implement will probably be the sector nearest neighbor queries. As with any query to a kd-tree, when you visit a node, you should pass in the bounding rectangle for the node. The key question is whether you need to continue to process this node (since it might provide a nearest neighbor) or ignore it.

In my implementation I encoded each sector as an integer from 0 to 7 (working counterclockwise based on angle). There are two utilities that you will find useful. (If you do not see why these utilities are useful, you should think more about how sector nearest neighbor queries are answered.) The first is given two points q and p and a sector index i. If p lies in sector i relative to q, then it returns the distance from q to p. Otherwise it returns ∞. The second one is given a point q, a rectangle R (a cell in the kd-tree), and a sector index i. It returns the minimum distance from q to any point of R that lies within sector i about q. If R does not intersect the sector then ∞ is returned.

Rather than writing eight different distance routines one for each sector, I would suggest that you write a function, which given a point q, sector index i, and point p, transforms p in sector i to a point p' in sector 0. (To transform a rectangle, just transform its corner points.) Then just write all your utilities assuming that the sector index is 0. For example, assuming that q is at the origin, you can map point $p = (x, y)$ in sector 2 to the point $p' = (y, -x)$ in sector 0 (see the figure). Observe that no complex trigonometric functions are required here. The sector containing a point depends only on the signs of the coordinates of the point and which coordinate is larger in absolute value. The transformation generally involves negating either or both of x and y and/or transposing the coordinates. The implementation details are left to your creativity.

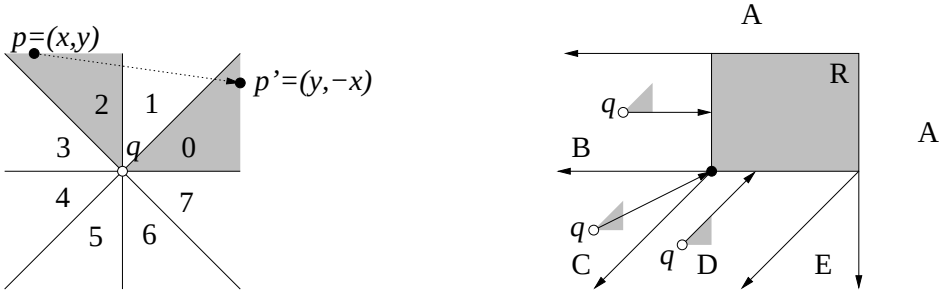


Figure 2: Transforming to sector 0, and sector-0 distance to rectangle.

The other tricky issue is how to compute the distance between a point q and a rectangle R in some sector. By the above comments, you may assume that everything has been mapped to sector 0. My suggestion for handling this would be to determine the location of q relative to R (see the figure). If q lies inside R then the distance is 0. If q lies above or to the right of R (region A) then the distance is ∞. If q lies to the left of R (region B) then the distance is the horizontal distance to the left wall of R. If it lies diagonally below and left (region C) then the distance is the distance to the lower left corner of R. If it lies diagonally below (region D), then the distance is the diagonal distance to the bottom of R (which is the same as the vertical distance multiplied by $\sqrt{2}$). Otherwise, it is in in region E and the distance is ∞.