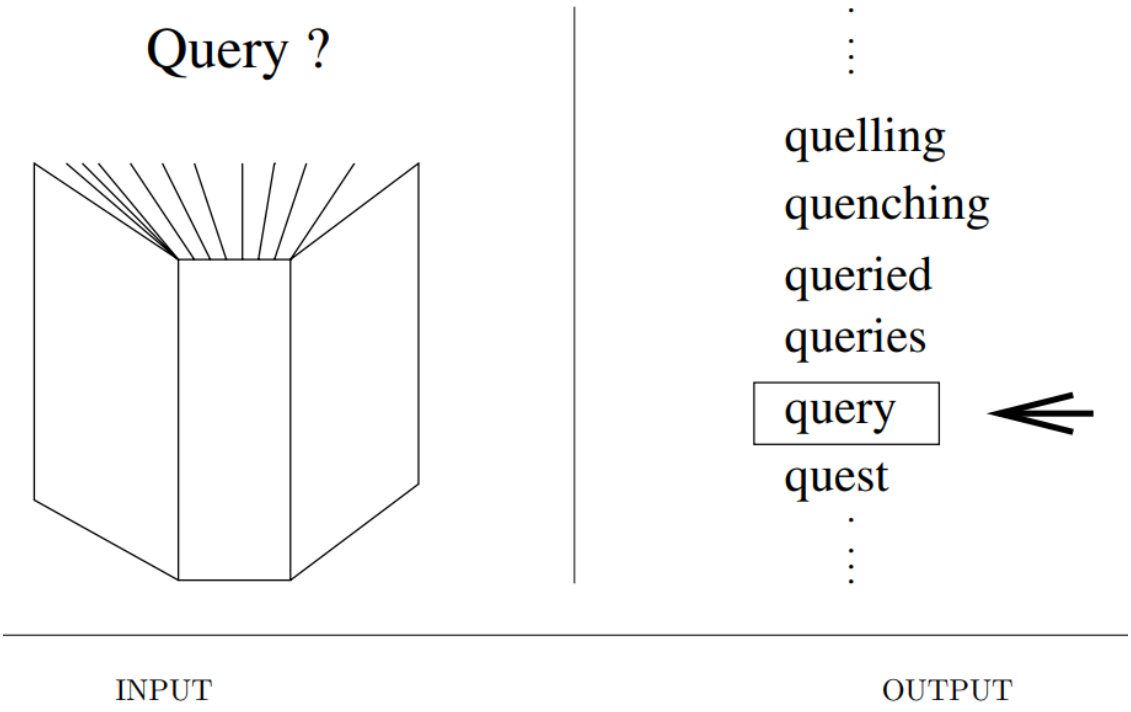
Data Structures II

June 19, 2020

1 Dictionaries



1.1 Input Description

A set of n records, each identified by one or more key fields.

1.2 Problem description

Build and maintain a data structure to efficiently locate, insert, and delete the record associated with any query key q.

1.3 Discussion

Many ways to implement dictionaries. In paratice though, it is important to avoid using bad data structures then to identify the single best option

Isolate the implementation of the dictionary data structure from its interface. Use explicit calls to

methods or subroutines that initialize search and modify the data structure, rather than embedding them within code

How many items will you have in your data structure?

1.3.1 Questions

- How many items will you have in your data structure?
- Do you know the relative frequencies of insert, delete, and search operations?
- Can we assume that the access pattern for keys will be uniform and random?
- Is it critical that individual operations be fast, or only that the total amount of work done over the entire program be minimized?

1.3.2 Implementation/Descisions

- **Unsorted linked lists or arrays**:
  - For small data sets, an unsorted array is probably the easiest data structure to maintain. Linked structures can have terrible cache performance compared with sleek, compact arrays. Once your dictionary becomes larger than 50 to 100 items, the linear search will kill you for either lists or arrays. A useful variant is called the *self-organizing list* or the LRU Cache

- **Sorted Linked Lists or Arrays:**
  - Maintaining a sorted linked list is not worth the effort unless you are trying to eliminate duplicates since we cannot perform a binary search in such data structures. Works only if there are not too many insertions or deletions

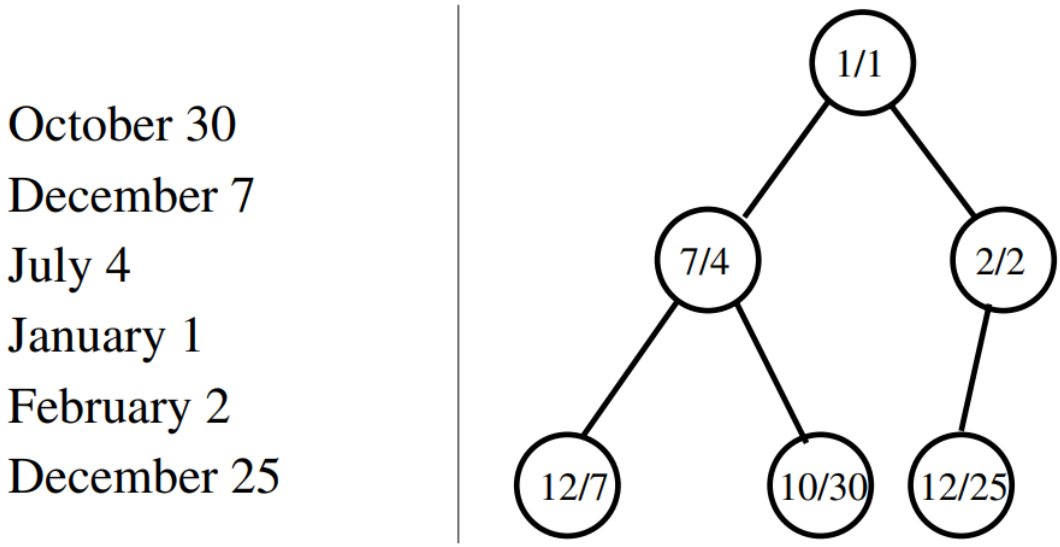
- **Hash Tables**: Anything with $ 100$ to $ 10,000,000 $ keys, a hash table is probably the right way to go. We use a function that maps keys to integers between $ 0 $ and $ m-1 $. We maintain an array of $ m $ buckets each typically implemented using an unsorted linked list. The hash function immediately identifies which bucket contains a given key. If we use a hash function that spreads the key out nicely, and a sufficently large hash table, each bucket should contain very few items, this making linear search acceptable.

  - How do i deal with collisions? Open adressing can lead to more concise tables with better cache performance than vucketing, but performance will be more brittle as the load factor (ratio of occupancy to capacity) of the hash table starts to get hight
  - How big should be the table? With bucketing $ m $ should be the same size as the max number of items you except to put in the tabl . With open adressing, make it about $ 30 $ precent larger or more. Selecting $ m $ to be a prime number minimizes the dangers of a bad hash function

- **Binary Search Trees**:
  - elegant data structures that support fast insertions, deletions and queries. In a random search tree, we simply insert a node at the leaf position where we can find it and no rebalancing takes places. Balanced search treees use local rotation operations to restructure search trees, moving more distant nodes closer to the root while maintaining the in-order search structure of the tree. Among balanced search trees, $ AVL $ and $ 2/3 $ trees are now passe and red-black trees seem more popular ._splay tree_ is an intersting self-organizing data structure which uses rotations to move any accessed key to the root. Frequently used or recently accessed nodes thus sit near the top of the tree.

- **B-Trees**:
  - For data so large that will not fit in memeory, your best bet is the **B-tree**. Once data has been stored outside of main memeory, the search time grows by several orders of magnitude. Idea behind a *B-tree* is to collapse several levels of a binary searh tree into a single large node, so that we can make the equivalent of several search steps before another disk access is needed. It is important to understand how the secondary storage device and virtual memory interact. _Cache-oblivious algorithms can migate such concerns
- **Skip Lists**
  - THese are somewhat a cult of data structures. A hierarchy of sorted linked lists is maintained, where a coin is flipped for each element to decide whether it gets copied into the next highest list. This implies roughly $ lg n $ lists, each roughly half as large as the one above it. Easier to use analyze and implement relative to balanced trees

2 priority Queues



INPUT OUTPUT

3 Input Description

A set of records with numerically or otherwise totally-ordered keys

4 Problem Description

Build and maintain a data structure for providing quck access to the *smallest* or *largest* key in the set

Called "priority" queues because they enable you to retrieve items not by insertion time (as in stack or queue), nor by key match (as in dictionary) but by which item has the highest priority of retrieval

If application performs no insertions after initial query, there is no need for an explicit priority queue. Simply sort the records by priority and proceed from top to bottom, maintaining a pointer to the last record retrieved.

4.1 Questions

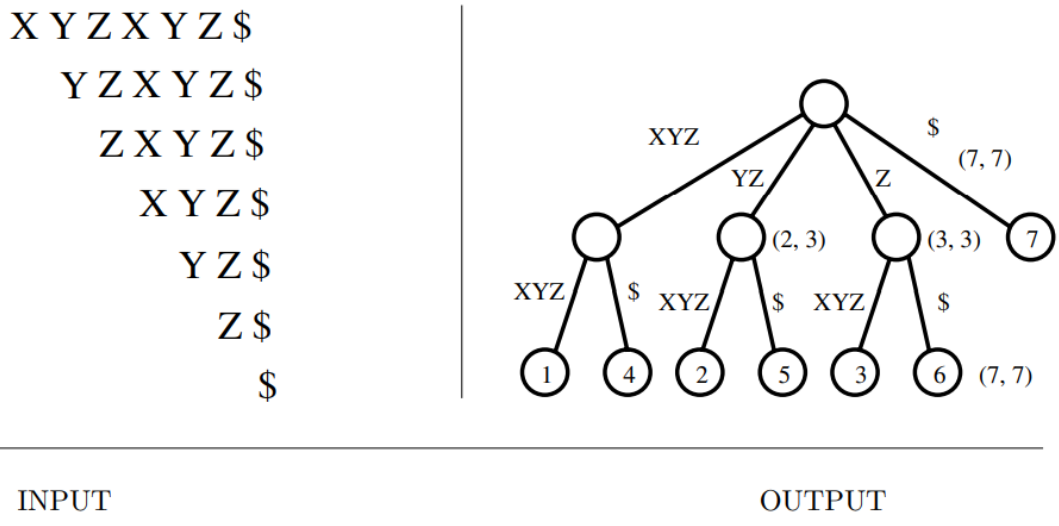
- **What other operations do you need**?
  - Will you be search for arbitrary keys, or just searching for the smallest? Will you be deleting arbitrary elements from the data, or just repeatedly deleting the top or smallest item?
- **Do you know the maximum data structure size in advance?**
  - The issue here is whether you can preallocate space for the data structure
- **Might you change thr priority of elements already in the queue?**
  - Changing the priority of elements implies that we must be able to retrieve elements from the queue based on their key, in addition to being able to retrieve the largest element

4.1.1 Implementation/Descisions

- **Sorted array or list**:
  - A sorted array is very eddicient to both identify the smallest element and delete it by decrementing the top index. Howerver, maintaining the total order makes inserting new elements slow. Sorted arrays are only suitable whne there will be few insertings into the priority queue
- **Binary heaps**
  - The simple, elegant data structure supports both insertion and extract-min in $ O (lg n) $ time each. Heaps maintain an implicit binary search tree structure in an array, such that the key at the root of the subtree is less than all of its descendents. Thus the minimum key always sits at the top of the heap. New keys can be inserted by placing them at an open leaf an percolating the element upwards untill it sits at its proper place in the partial order.
- **Bounded height priority queue**:
  - This array-based data structure permits constant-time insertion and find-min operations whenever the range of possible key value is limited. Suppose we know that all key values will be integers between $ 1 $ and $ n $. We can set up an array of $ n $ linked lists, such that the $ ith $ list serves as a bucket containing all items with key $ i $. We will maintain a top pointer to the smallest nonempty list. To insert an item with key $ k $ into the priority queue, add it to the $ kth $ bucket and set $ top= min(top, k)$. To extract the minimum, report the first tiem from bucket *top*, delete it and move *top* down if the bucket has become empty. Useful in maintaining the vertices of a graph sorted by degree, which is a fundamental operation in graph algortihms. Still they are not as widely know as they should be. They are the right priority queue for any small, discrete range of keys

- **Binary Search Trees**:
  - Binary search trees make effective priority queues since the smallest element is always the leftmost leaf and the largest is always the rightmost leaf. The min/max is found by simply tracing down left/right pointer untill the next pointer is Nil. Binary tree heaps prove most appropriate when you need other dictionary operations, or if you have an unbounded key range and do not know the maximum priority queue size in advance.

5 Suffix Trees and Arrays



5.1 Input Description

A refrence string $ S $

5.2 Problem Description

Build a data structure to quickly find all places where an arbitrary query string $ q $ occur in $ S $

5.3 Discussion

Suffix trees and arrays are phenomenally useful data strucutres for solving string problems elegantly and efficently. Proper use of a suffix tree often speeds up string processing algorithms from $ O (n^2) $ to linear time

a suffix tree is simply a *trie* of the $ n $ suffixes of an $ n$-character string $ S $.

A trie is a tree structure, where each edge represents one character and the root represents the null string

Thus each path from the root represents a string, described by the characters labeling the edges traversed. Any finite set of words defines a trie, and two words with common prefixes branch off from each other at the first distinguishing character

Tries are useful for testing whether a given query string $ q $ is in the set. We traverse the trie from the root along branches defined by successive characters of $ q $. If a branch does not exist in the trie, then $ q $ cannot be in the set of strings. Otherwise we find $ q $ in $ | q | $ character comparisions regardles of how many strings are in the trie

Tries are very simple to build (insering new strings) and very fast to search, although they can be expensive in terms of memeory

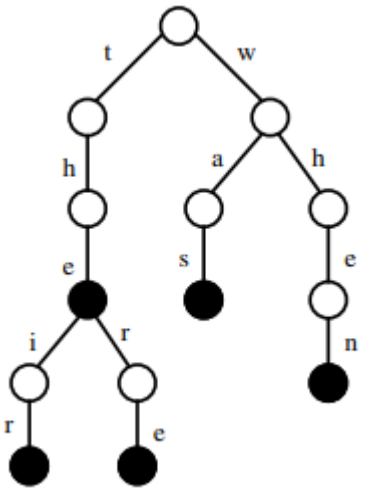


Figure 12: A trie on strings *the* , *their* , *there* , *was* and *when*

A suffix tree is simply a trie of all the proper suffixes of $ S $. The suffix tree enables you to test whether $ q $ is a substring of $ S $, because any substring of $ S $ is the prefix of some suffix

The search time is again linear in the lenght of $ q $

The catch is that constructing a full suffix tree in this manner can require $ O(n^2)$ time and even worse $ O(n^2) $ space, since the average lenght of the $ n $ suffixes is $ n / 2 $.

Even better, there exist O(n) algorithms to construct this collapsed tree, by making clever use of pointers to minimize construction time. These additional pointers can also be used to speed up many applications of suffix trees.

5.3.1 Applications of Tries

- **Find all occurrences of q as a substring of $ S $**
  - Jut as with a trie, we can walk from the root to the node $ n_q $. In collapsed suffix trees, it takes $ O(|q| + k) $ time to find the $ k $ occurences of $ q $ in $ S $
- **Longest substring common to a set of strings**:
  - Build a single collapsed suffix tree containing all suffixes of all strings, with each leaf labeled with its orginal string. In the course of doing a depth-first search on this tree, we can label each node with both the lenght of its common prefix and the number of distinct strings that are children of it.
- **Find the longest palindrome in $ S $**:
  - A *palindrome* is a string that reads the same if the order of characters is reversed, such as *madam*. To find the longest palinfrome in a string $ S $, build a suffix tree containing

all suffixes of $ S $ and the reversal of $ S $, with each leaf identified by its starting position. A palindrom is defined by any node in this tree that has forward and reversed children from same position

5.3.2 Suffix Arrays

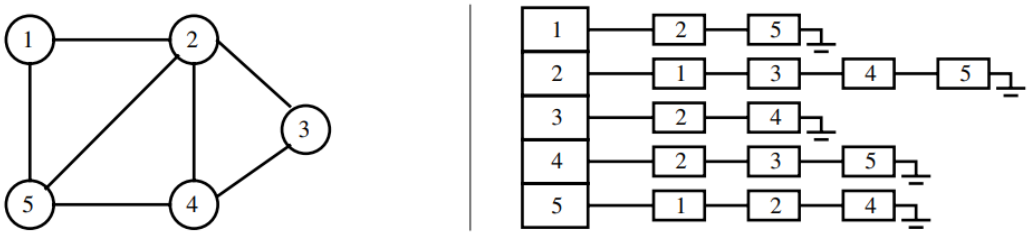
Suffix arrays do most of what suffix trees do, while using roughly four times less memory. They are also easier to implement. A suffix array is in principle just an array that contains all the $ n $ suffixes of $ S $ in sorted order.

Thus a binary search of this array for string $ q $ suffices to locate the prefix of a sufix that matches $ q $, permitting an efficient substring search in $ O(lg n) $ string comparisions.

With the addition of an index specifiying the common prefix lenght of all bounding suffixes, only $ lg n + |q| $ character comparisions needs be performed on any query, since we can identify the next character that must be tested in the binary search

Constructing suffix arrays are difficult due to the $ O (n^2) $ characters in the strings beign sorted. One solution is to first build a suffix tree, then perform an in-order traversal of it to read the strings off in an sorted order!

5.4 Graph Data Structures



INPUT OUTPUT

5.5 Input Description

A graph $ G $

5.6 Problem Description

Represent the graph $ G $ using a felxible, efficent data structure

5.7 Discussion

The two basic data structures for represeting graphs are *adjacency matrices* and *adjacency lists*.

In general, for most things, adjacency lists are the way to go

5.8 Implementations/Descisions

- **How big will your graph be?**:
  - How many vertices will it have, both typically and in the worst case? Ditto for the number of edges? Graphs with $ 1000 $ vertices imply adjacency matrices with $ 1,000,000 $ entries. This seems too be the boundary of reality. Adjaceny matrices make sense only for small or very dense graphs
- **How dense will your graph be?**:
  - If your graph is very dense, meaning that a large fraction of the vertex pairs define edges, there is a probably no compelling reason to use adjacency lists. You will be doomed to using $ Θ(n^2) $ space anyways. Indeed, for complete graphs, matrices will be more concise due to the elimination of pointers
- **Which algortihms will you be implementing?**:
  - Certain algorithms are more natural on adjacency matrices (such as all-pairs shortest path) and other favor adjacency lists (such as most **DFS-based algortihms**. Adjacency matrices win for algorithms that repeatedly ask "Is $(i, j) $ in $ G $?" Howerver, most graph algorithms can be designed to eliminate such queries
- **Will you be modifiying the graph over the course of your application?**
  - Efficient static graph implementations can be used when no edges insertion/deleting operations will done following initial construction. Ineed more common than modifying the topology of the graph is modifying the attributes of a vertex or edge of the graph, such as size, wieght, label or color. Attributes are best handled as extra fields in the vertex or edge records of adjacency lists

5.9 Planer Graphs

Planer graphs are those that can be drawn in the plane so no two edges cross. Graphs arising in many applications are planer by definition, such as maps of countries. Others are planar by happenstance, like trees. Planar graphs are always sparse, since any $ n-$ vertex planer graph can have at most $ 3n - 6 $ edges.

Thus they should be represented using adjacency lists. If the planar drawing (or embedding) of the graph is fundamental to what is being computed, planar graphs are the best represented geometrically

5.10 HyperGraphs

Are generalized graphs where each edge may link subsets of more than two vertices. Suppose we want to represent who is on which congressional committee. The vertices of our hypergraph would be individual congressmen, while each hyperedge would represent one committee. Such arbitrary collections of subsets of a set are naturally though of as hypergraphs

There are two basic data structures for hypergraphs

5.11 Incidence matrices

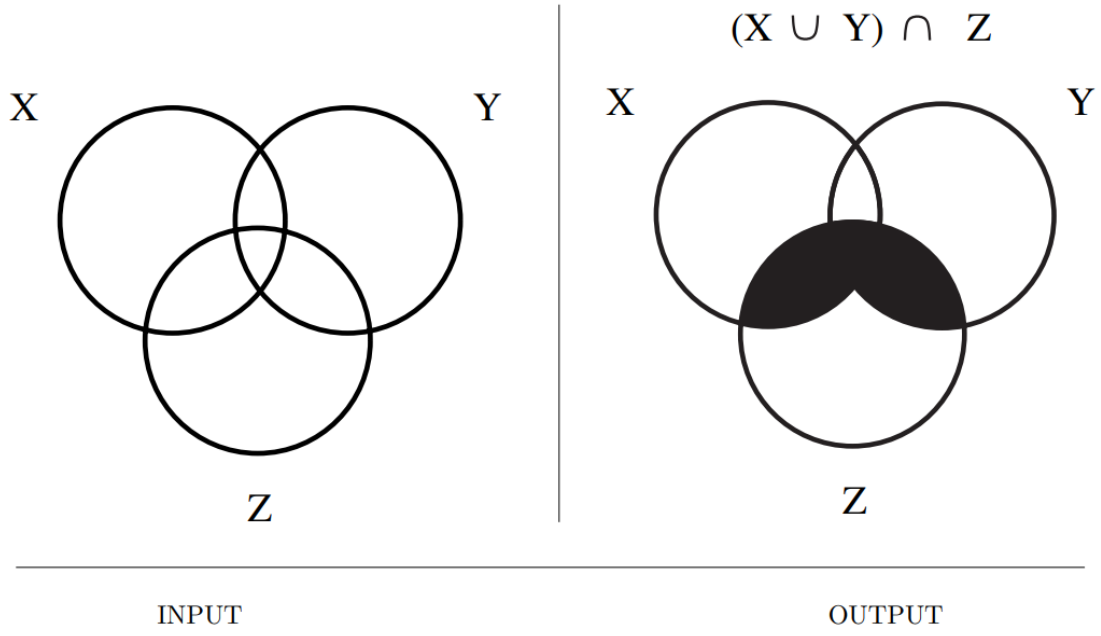
Which are analogous to adjacency matrices. They require $ n * m $ space, where $ m $ is the number of hyperedges. Each row corresponds to a vertex, and each column to an edge, with a nonzero entry in $ M[i, j]$ iff vertex $ i $ is incident to edge $ j $. On standard graphs there

are two nonzero entries in each column. The degre of each vertex governs the number of nonzero entries in each row

5.12 Bipartite Incidence Structures

Are analogous to adjacency lists, and hence suited for sparse hypergraphs. There is a vertex of the incidence structure associated with each edge and vertex of the hypergraphs, and an edge $ (i, j) $ in the incidence structure if vertex $ i $ of the hypergraph appears in edge $ j $ of the hypergraph. Adjacency lists are typically used to represent this incidence structure. Drawing the associated bipartite graph provides an natural way to visualize the hypergraph

6 Set Data Structure



6.1 Input Description

A universe of items $ U = {u_1,..., u_n} $ on which is defined a collection of subsets $ S = {S_1,..., S_m}$

6.2 Problem description

Represent each subset so as to efficently $ (1) $ test whether $ u_i $ in S_j, (2) compute the union or intersection of $ S_i $ and $ S_j $, and $ (3) $insert or delete members of $ S $

6.3 Discussion

A set is an unordered collection of objects drawn from a fixed universal set. Howerver, it is usually useful for implementation to represent each set in a single *cononical order* , typically sorted, to speed up or simplify various operations.

Sorted order truns the problem of finding the union or intersection of two subsets into a linear-time operation- just sweep from left to right and see what you are missing It makes possible element searching in sublinear time. Finally, printing the elements of a set in a cononical order paradoxically reminds us that order really doesnt matter

We distinquish sets from two other kinds of objects: dictionaries and strings. A collection of objects *not* drawn from a fixed-size universal set is best though of as a *dictionary.*

Strings are structures where order matters

Multisets permit elements to have more than one occurence. Data structures for sets can generall ybe extened to multisets by maintaining a count field or linked list equivalent entries for each element

6.4 Implementation/Descisions

- **Bit vectors**:
  - An n-bit vector or array can represent any subsets $ S $ on a universal set $ U $ containing $ n $ items. Bit $ i $ will be $ 1 $ if $ i $ inE $ S $ and $ 0 $ if not. Since only one bit is needed per element, bit vectors can be very space efficient for surprisingly large values of $ | U | $. Element insertion and deletion simply flips the appropriate bit.
  - Intersection and union are done by "and-ing" or "or-ing" the bits together. The only drawback of a bit vector is its performace on sparse subsets. For example, it takes $ O(n) $ time to explicitly identify all memebers of sparse (even empty) subset $ S $
- **Containers or dictionaries**
  - A subset can also be represented using a linked list, array or dictionary containing exactly the elements in the subset. No notion of a fixed univerasal set is needed for such a data structure. For sparse subsets, dictionaries can be more space and time efficient than bit vectors and easier to work with and program. For efficient union and intersection operations, it pays to keep the elements in each subset sorted, so linear-time traversal through both subsets identifies all duplicates
- **Boom Filters**:
  - We can emulate a bit vector in the absence of a fixed universal set by hashing each subset element to an integer from $ 0 $ to $ n $ and setting the corresponding bit. Thus, bit $ H(e) $ will be $ 1 $ if $ e E S $. Collisions leave some possibility for error under this scheme, howerver, because a different key might have hashed to the same position
  - Boom fileter use several (say k) different hash functions $ H_1, ..., H_k $, and set all $ k $ bits $ H_i(e)$ upon insertion of key $ e $. Now $ e $ is in $ S $ only if all $ k $ bits are $ 1 $. The probability of false positives xan be made arbitratily low by increasing the number of hash functions $ k $ and table size $ n $. With the proper constants, each subset element can be represented using a constant number of bits, independent of the size of the universal set.
  - This hashing based data strucutre is much more space-efficent than dictionaries for static subset applications that can tolerate a small probability of error. Many can. For instance, a spell checker that left a rare random string undected would prove no great tragedy

6.5 Set Partition

many applications involve collections of subsets that are pairwise disjoint, meaning that each element is in exactly one subset

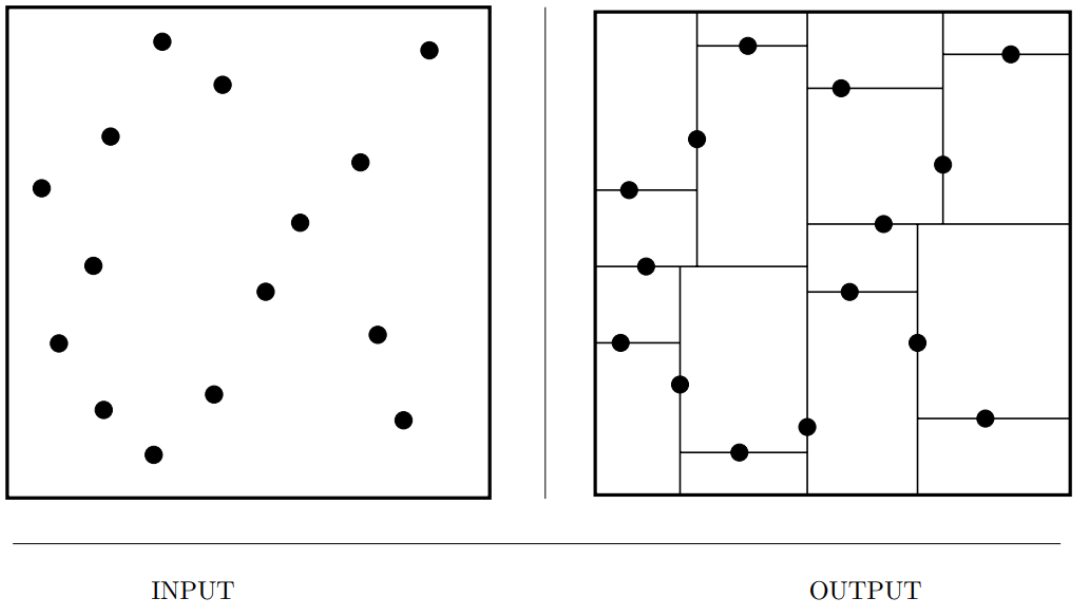
Each vertex is in exactly one component. Such a system of subsets is called a *set partition*.

Problme with set partation data structures is maintaining changes over time. Perhaps as edges are added or party memebrs defect. Typical queries include "which set is a particular item in?" and "are two items in the same set?" as we modify the set by $ (1) $ changing one item, $ (2)$ merging or unioning two sets, or $(3) $ breaking a set apart

Operations

- **Collection of containers**: representing each subset in its own container/dictionary permits fast access to all the elements in the subset, which facilitates union and intersection operations. The cost comes in membership testing, as we must search each subset data structure independently until we find our target
- **Generalized bit vector**: Let the $ ith $ element of an array denote the number/name of the subset that contains it. Set identification queries and single elements modifications can be performed in constant time. Howerver operations like performing the union of two subsets take time proportional to the size of the universe, since each element in the two subsets must be identifies and (at least one subset's worth) must have its name changed
- **Dictionaries with a subset attribute**: Similarly, each item in a binary tree can be associated a field that records the name of the subset it is in. Set identification queries and single element modifications c an be performed in the time it takes to search in the dictionary. Howerver, union/intersection operatios are again slow.
- Union-find data structure: we represent a subset using a rooted tree where each node points to its parent instead of its children. The naame of each subset will be the name of the item at the root. Finding out which subset we are in is simple, for we keep traversing up the parent pointers untill we hit the root. Unioning two subsets is also easy. Just assign the root of one of two trees to point to the other, so now all elements have the same root and hence the same subset name
- Implementation details have a big impact on asympototic performance. Always selecting the larger (or taller) tree as the root in a merger guarantees logarithmic hegiht trees. Retraversing the path traced on each find and explicitrly pointing all nodes on the path to the root (path compression) reduces the tree to almost constant height. Union find is a fast, simple data strucutre every programmer should know.

7 Kd-Trees



INPUT OUTPUT

7.1 Input description

A set $ S $ of $ n $ points or more complicated geometric objects in $ k $ dimensions

7.2 Problem Description

Construct a tree that partitions space by half-places such that each object is contaned in its own box-shaped region

7.3 Discussion

Kd-tree and related spacial data structures hierarchically decompose space into small number of cells, each containing a few representives from an input set of point .

This provides a fast way to access any object by position. We traverse down the hierachy untill we find the smallest cell containing it, and then scan through the object in the cell to identify the right one

Typical algorithms constuct **kd-Trees** by partitioning point sets. Each node in the tree is defined by a plane cutting through one of the dimensions. Ideally, this plane equally partitions the subset of points into left/right (or up/down) subsets

These children are again partitioned into equal halves, using planes through a different dimesnsion. Partitioning stops after $ lg n $ levels, with each point in its own leaf cell

The cutting planes along any path from the root to another node defines a unique box-shaped region of space. Each subsequent plane cuts this box into two boxes. Each box-shaped region is defined by $ 2k $ planes, where $ k $ is the number of dimensions

Indeed the $ kd $ in kd-tree is short for k-dimensional

We maintain the region of intrest defined by the intersection of these half-spaces as we move up the tree

7.3.1 Splitting the Plane

Flavors of $ kd $ trees differ in exactly how the splitting plane is selected. Options include

- **Clycling though the dimensions** -partionion first on $ d_1 $ then $ d_2, ..., d_k $ before cycling back to $ d_1 $
- **Cutting along the largest dimension** -select the partion dimension to make the resulting boxes as squares or cube-like as possible. Selecting a plane to partition the points in half does not mean selecting a splitter in the middle of the box-shaped regions, since all the points may lie in the left side of the box
- **Quadtrees or Octtree** - Instead of partitiong with single planes, use all axis-parallel planes that pass through a given partition point. In two dimensions, this means creating four child cells; in 3D, it means eight child cells. Quadtrees seem particulary popular on image data, where leaf cells imply that all pixels in the regions have the same color
- **BSP Trees**: - Binary space partions use general (i.e., not just axis-parallel cutting planes to carve up space into cells so that each cell ends up containing only one object (say a polygon). Such partitions are not possible using only axis-parallel cuts for certain sets of objects. The downside is that such polyhedral cell boundaries are more complicated to work with than boxes
- **R-trees** -This is another spatial data structure useful for geometic objects that cannot be parititoned into axis-oriented boxed without cutting them into pieces. At each level, the objects are partitioned into a smaller number of (possibly-overlapping) boxes to construct searchable hierarchies without partitioning objects

Ideall the partitions split both the space (ensuring fat, regular regions) and the set points (ensuring a log height tree) evenly, but doing both simultaneously can be impossible on a given input

The advantage of fat cells become clear in many applications of kd-trees

7.3.2 Advantage

- **Point location** -to identify which cell a query point $ q $ lie in, we start at the root and test which side of the parition plane contains $ q $. Bt repeating the process on the appropriate child node, we travel down the tree to find the leaf cell containing $ q $ in time proprotinal to its height
- **Nearest neighbor search** - to find the point in $ S $ to a query point $ q $, we perform point location to find the cell $ c $ contains $ q $. Since $ c $ is borderred by some point $ p $, we can compute the distance $ d(p, q) $ from $ p $ to $ q $. Point $ p $ is likely close to $ q $, but it might not be the single closest neighbor might lie just to the left of the boundary in another cell. Thus, we must travel all cells that lie within a distance of $ d(p, q) $ of cell $ c $ and verify that none of them contain closer points. In trees, with nice fat cells, very few cells should need to be tested
- **Range Search** - Which points lie within a query box or region? Starting from the root, check whether the query region intersects (or contains) the cell defining the current node. If

it does, check the children; if not, none of the leaf cells below this node can possibly be of intrest. We quickly prune away irrelevant portions of the space

- **Partial key search** - SUppose we want to find a point $ p $ in $ S $, but we do not have full information about $ p $. Say we are looking for someone of age $ 35 $ and height $ 5'8 $ but of unknown weight in a $ 3D-tree$ with dimensions of age, weight and height. Starting from the root, we can identify the correct descendant for all but the weight dimension. To be sure we find the right point, we must search both children of these nodes. The more fields we know the better, but such partial key search can be substantially faster than checking all points

7.3.3 Summary

$ kd $ trees are most useful for a small to moderate number of dimensions, say from $ 2 $ up to maybe $ 20 $ dimensions. They lose effectiveness because the ratio of the volume of a unit sphere in $ k $ dimensions shirnks exponentially compared to the unit cube