Lecture 2: Data structures: A short background

Storing data

It turns out that depending on what we wish to do with data, the way we store it can have a significant impact on the running time. So in particular, storing data in certain "structured" way can allow for fast implementation. This is the motivation behind the field of data structures, which is an extensive area of study.

In this lecture, we will give a very short introduction, by illustrating a few common data structures, with some motivating problems.

In the last lecture, we mentioned that there are two ways of representing graphs (adjacency list and adjacency matrix), each of which has its own advantages and disadvantages. The former is compact (size is proportional to the number of edges + number of vertices), and is faster for enumerating the neighbors of a vertex (which is what one needs for procedures like Breadth-First-Search). The adjacency matrix is great if one wishes to quickly tell if two vertices i and j have an edge between them.

Let us now consider a different problem.

Example 1: Scrabble problem

Suppose we have a "dictionary" of strings $S = \{s_1, s_2, \ldots, s_N\}$ whose average length is L, and suppose we have a query string x. How can we quickly tell if $x \in S$?

Brute force. Note that the naive solution is to iterate over the strings and check if $x = s_i$ for some i. This takes time $O(NL)$.

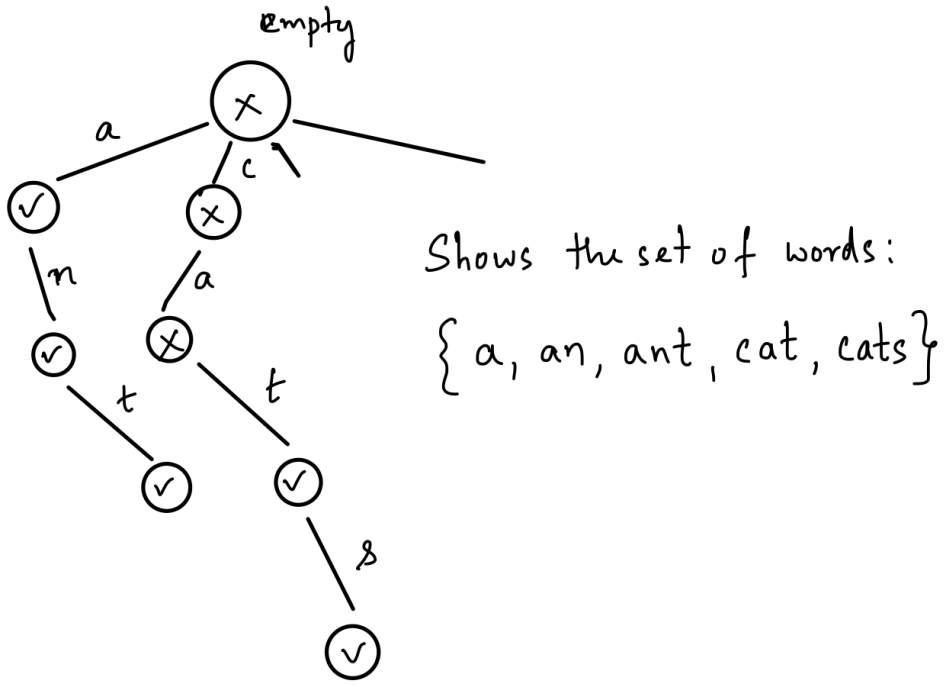
If we only care about answering the query for one x, this is not bad, as $O(NL)$ is the amount of time needed to "read the input". But of course, if we have a fixed dictionary and if we wish to check if $x \in S$ for multiple x, then this is extremely inefficient.

Second idea. Note that for integers, this "membership" problem can be solved easily by first sorting the strings and at query time, performing a binary search. Since one can do the same for strings (using the natural lexicographic ordering), we get a procedure that has pre-processing time $O(L \cdot N \log N)$, and a query time $O(L \cdot \log N)$. (The mild simplifying assumption we made here is that all the strings are of length roughly L -- i.e., lengths are not too uneven.)

The question now is: *can we get rid of the dependence on N altogether?* I.e., is there a way to store a set of strings S, so that finding out if $x \in S$ can be done in time independent of $|S|$?

We introduce a data structure known as a **prefix tree** (also called a *trie*) that achieves this. The key idea is to represent a set of strings in a tree. Let us assume that the strings all have characters {a, b, ..., z}. Now, consider a tree in which (a) the root node corresponds to the empty string "", and (b) for every node, we have 26 descendants, one for each character a-z.

As described, this is an infinite tree. Now corresponding to every node in the tree, we can associate a string, which consists of the letters used in the path from the root to that node. Likewise, given any string "$x_1 x_2 \ldots x_L$", we can find a node that corresponds to it by traversing down the tree, starting with the root, going down to the descendant corresponding to x_1, then down to the one for x_2, and so on.



Now, how do we keep track of a *set* of strings using this tree? We can imagine that at every node, we have a boolean variable that indicates if the string corresponding to the node is present in the set S. For instance, in the figure above, the nodes with the check-mark correspond to the set S indicated.

Avoiding infinities. As defined, the prefix tree is an infinite sized object. Of course, to store it in practice, we need to only create descendants "if necessary". For instance, if the dictionary contains the word "cat" but no words starting with "catb", then the node corresponding to "cat" does not have a descendant corresponding to the character 'b'.

Query time. Once we have built the prefix tree, checking if a string $q = q_1 q_2 \ldots q_L$ is present in the tree or not is fairly simple: we simply start at the root, go down q_1, then down q_2, and so on. If we encounter a descendant that is not present, we return NO. Otherwise, we go all the way to the node corresponding to q, and return the boolean value stored at the node (which would have been YES if q was in the dictionary and NO otherwise). The running time is $O(L)$.

Adding/deleting from dictionary. The procedure for adding a word $x = x_1 x_2 \dots x_L$ is also similar. We go down the tree, adding descendants if necessary. Finally, once we reach the node corresponding to x, we set the boolean value at the node to be YES (or 'true'). This takes $O(L)$ time. Deletion is similar (we go down the tree and set the boolean to NO/'false'). This does not take into account "garbage collection", i.e., removing branches in which all the words have been deleted, but it turns out that this can be implemented relatively easily.

Building the tree, i.e., pre-processing time. For most data structures, we need to first "create" the data structure from the "base" representation of the data. In this case, we need to go from the collection of strings we have to the prefix tree. This time is usually known as the pre-processing time for creating the data structure. In many contexts, one creates the data structure before-hand, and only performs "queries" or "tweaks" (a few add/delete operations) as time proceeds.

In the example of prefix trees, building the tree using the add operations takes time $= O(NL)$ (using our analysis of the add operation).

Summary: "API" for prefix trees. Note that a prefix tree, as we defined, is a data structure that stores *a set of strings*. It uses space $O(NL)$ (can be smaller if the strings share prefixes). We also showed that the operations to add, delete and query a string x can all be performed in time $O(\text{length}(x))$.

Abstracting data structures

The final summary in our discussion of prefix trees is something that we should keep in mind whenever we study any data structure. I.e.,

1. what is being stored? (in the prefix tree e.g., this is a set of strings)
2. what operations does the data structure allow? (for prefix trees, add, delete, query)
3. what is the time complexity of each operation?
4. what is the "pre-processing" time, i.e., the time needed to build the data structure?

Given the answers to these questions, we can make use of the data structure without going into the details of how the operations are implemented.

Let us now give a second example. The goal is not really to introduce a novel data structure, but to give another example of how one stores the input affects the time for computation.

Exampe 2: Web search

Suppose we have a corpus of N documents, each of which has roughly m words. Given a set of query words $q_1, q_2, \dots$, find all the documents that contain all the query words.

Naive solution. The trivial solution is one that goes through the documents keeping track of whether the query words appear in the current document. This procedure clearly takes $O(Nm)$ time (there's also a factor depending on the number of query words).

Once again, if we have the same set of documents and one makes multiple queries (typical in web search applications), this can be very inefficient. So the question is: *given that most words don't appear in most documents, can we avoid this running time?*

Turns out a nice data structure for solving this problem is what is known as the "inverted index". It is similar to the "Index" that appears at the end of most textbooks. For each word, suppose we store the list of documents containing that word. Now, given queries $q_1, q_2, \ldots$, we can look up the lists for each word q_i (call it $InvIdx(q_i)$), and output $InvIdx(q_1) \cap InvIdx(q_2) \cap \ldots$.

This results in a query time that is proportional to the sum of the sizes of the inverted index (assuming that we compute the intersection in the naive way; we will see a slightly better way in the HW).

Pre-processing. Note that creating the inverted index takes time roughly $O(Nm)$ (we simply have to make a pass over all the documents, adding the document-ID to the list corresponding to each word). In practice, this is not too much, given that "collecting" the data takes roughly the same amount of time.

Example 3: Dynamic arrays

The final example is that of dynamic arrays (i.e. arrays whose size changes as the algorithm proceeds). The goal is to illustrate an interesting aspect of many data structures, concerning point (3) above: *what is the time complexity of each operation?* It turns out that in some data structures, operations can be very expensive "in the worst case", but one can formally prove that they are cheap "on average".

The goal in dynamic arrays (DA) is to store a sequence of elements, and to support the following operations: (a) add one element to the end of the DA (b) remove an element from the end of the DA (c) return the ith element

We wish to have $O(1)$ time for each of the operations. Furthermore, we wish to have (at any point of time), the total memory footprint of the data structure to be $O(n)$, where n is the number of elements currently in the DA. (E.g., we do not want a huge array.)

An example of a dynamic array implementation is the vector<> class in C++.

How are dynamic arrays implemented? The idea is to store the DA as a pair (n, A), where n is the number of elements in the DA, and A is a "regular" array (essentially a known block of memory; size(A) must be $\geq n$). This way, returning the ith element can be done by simply returning $A[i]$. So we only need to discuss how to add and remove elements.

At initialization, the DA (assuming it is empty) will be set to $(0, A)$, where A is an array of size n_0, for some fixed constant (for concreteness, assume that $n_0 = 2$).

Now imagine that elements are added one by one to the end of the DA (n, A). The **add procedure** works as follows: as long as size$(A) > n$, one can simply add the element being added as $A[n]$, and increment n. Now, if we have size$(A) = n$, this can no longer be done. In this case, we create a new array A' of size $2n$, and copy over all the current elements of A to A', along with the new element being added, and set the DA to $(n+1, A')$.

Running time of the add procedure. Note that when size$(A) = n$, adding *one element* takes time $O(n)$, which is basically the current size of the array. But on the other hand, as we add more and more elements, this "extreme case" occurs less and less frequently. Let us compute the total time complexity when we add $n = 2^r$ elements to the DA one after another.

Adding elements 1 and 2 takes only 1 step (as we assumed $n_0 = 2$).

Adding element 3 takes 4 steps (need to allocate a new array of size 4 (1 step -- assuming "malloc" is constant time), need to copy 2 elements (2 step), need to add final element (1 step)).

Adding element 4 takes only one step.

Adding element 5 takes $1 + 4 + 1$ steps (as before).

Adding elements 6, 7, 8 takes 1 step each.

...

In general, adding elements $2^i + 2, \ldots, 2^{i+1}$ takes 1 step each. Adding the $2^i + 1$st element takes $1 + 2^i + 1$ steps. Thus, the time taken to add elements $2^i + 1, 2^i + 2, \ldots, 2^{i+1}$, for any $i \geq 1$, is $1 + 2^{i+1}$.

Thus the total time, summing this over $i = 1, 2, \ldots, (r-1)$ is $2^r + 2^{r-1} + \ldots 2 = 2^{r+1}$. In short, adding n elements to the DA takes a total time of $2n$ (ignoring lower order additive terms).

Remark. This is a very simple example of the *"doubling trick"*, a very useful tool in algorithm design.

Running time of the remove procedure. Removing an element of the DA (n, A) can be as simple as decrementing n . However, this results in a wastage of space when a lot of elements are added and then removed. For instance, suppose we started with an empty array, added a million elements and then deleted them all. Then A will still be of size $\geq$ one million, even though the DA is empty. One idea is to use a similar approach as in the add case -- if half the elements of A are empty, i.e., if size$(A) \geq 2n$, then we create a new array of half the size and copy the elements across.

This works, modulo a slight technical problem: suppose that right after the size of A was halved, we perform an add operation. Then we would end up creating a new array and copying elements across. Thus, alternating delete/add operations like this would result in a situation where each operation takes $O(n)$ time. A simple trick to avoid this is to move to an array of half the size only if 3/4'th of the elements of A are empty.

Amortized analysis

The example of a dynamic array showed that sometimes, operations can be expensive ($O(n)$ in that case), but one can prove (formally) that any sequence of n operations takes only $O(n)$ time in total, for all n. In this case, we say that the operations have an "*amortized* running time of O(1) per operation".

Amortized analysis is an important paradigm in studying data structures. In many applications, one only cares about the total running time of an algorithm. Individual steps taking longer does not matter.