UBC CS500 lecture notes: X-tries

9th October lecture, scribed by Dmitry Borzov

October 16, 2013

1 Overview

In the previous lecture we started our discussion of the dictionary problem. We considered a special case of the dictionary when its keys make up a subset S of some known range of m integers of some universe $U, |U| = m$, that is $S \in U$.

In this lecture we discuss algorithms for such a dictionary that allow for performing of the following operations efficiently:

1. **Member search**. Looking up if the key $i \in U$ is in S.
2. **Closest element search**. For a given $i \in U$ finding the closest/predecessor/successor element that is in S.
3. **Insert/delete**. Updating the dictionary: inserting or deleting new keys to S.
4. **Max/min** Finding the maximum/minimum key values.

We start with the recap of the design of the Direct Access Table. We see that the Direct Access Table enables an effective member search, but also has several drawbacks.

One of them is considerable initialization time with $O(m)$ upper bound. We discuss a modified design where the Direct Access Table stores pointers to another array of records for the S dictionary keys. By storing the reverse pointers for each of these keys we can introduce a pointer checking procedure that would make the 'proper initialization' of each of the m elements in the Direct Access Table redundant. We show that this makes the necessary initialization time upper bound to be of $O(|S| \log m)$, which is quite an improvement.

Then we proceed to algorithms for tackling the predecessor search problem. We discuss a design where we simply store the record for the predecessor/successor element for each element in U. While it indeed makes the predecessor search very effective, the tradeoffs are slow insert/delete time bounds ($O(m)$) and worrying memory space requirements ($O(m \log m)$).

We then consider using an auxiliary data structure called x-fast trie on top of the Direct Access Table to keep track of the closest element information efficiently. We show how this makes it possible to both search for the closest element and update the dictionary not slower than $\log m$ in execution time upper bound asymptotics.

Discussion of these topics in more detail is presented below.

2 Dictionary problem

A dictionary problem involves storing pairs of keys and values. Each of the keys must be unique.

In our discussion of the lower bounds for various algorithmic tasks on the previous lecture we saw that there is a lower bound of $\Omega(\log S)$ for member search using the comparison-based model. Are there ways to bypass this limitation?

One way is to exploit the structure of the specific set of keys we have for the problem at hand. Let us consider the case where the keys are a set of integers $S = \{s_i | i = 0...n\}$ within some given range/universe U (or could be unabvivalently mapped to such a representation).

One way is to define the Direct Access Table for this set. We have an array of m elements and for each key in the dictionary we mark the element with the value of the key itself:

$$A[0:m-1], \forall i \in U : A[i] = \begin{cases} 1, & \text{if } i \in S \\ 0, & \text{if } i \notin S \end{cases}$$

Now in order to find out if the key is in the set we simply look up the value at the table with the corresponding key value position.

Let us look at the trade offs the Direct Access Table brings.

The advantages are:

1. Looking up if the element is in set is of $O(1)$.
2. Inserting/deleting the element is straightforward: we only mark one element in the specific position: $O(1)$.

Here are the drawbacks:

1. Preprocessing time. Initialization of the Direct Access Table includes going through each element of the table and assigning the appropriate value, with the time proportional to the size of U, $O(m)$.
2. Closest element (predecessor/ successor) search is not very effective: we have to move along the table until we stumble upon the 'marked' element. Worst case bound: $O(m)$
3. Space constraints, memory needed is proportional to the size of universe U, $O(m)$

Can this drawbacks be addressed? We will now look at various design improvements that tackle the first and the second of the ones listed here.

3 Initializing in $O(|S| \log m)$ time

Is there a way to modify the Direct Access Table design in such a way so that the initialization procedure would involve only going through each of the values in S, not U? Here is an approach to do that. Let us set up another auxiliary array with $|S|$ elements that contains records for each of these keys. We now go through each of the elements in $|S|$ and for the corresponding position in the Direct Access Table we record not simply the occupied/unoccupied marker as before, but a pointer to the corresponding element in this new auxiliary array of ours. Now for a specific query key we can read the pointer to the corresponding element in the auxiliary array and read its value.

There is one issue however. The 'empty' elements of the Direct Access Table that weren't assigned any value after the initial memory allocation may potentially have an arbitrary value. They can be wrongly read as pointers to some element in the auxiliary array. How can we make sure it won't happen?

The way to do this is to store a reverse pointer to the Direct Access Table address in each of the elements of the auxiliary array. This allows us to check if the pointer from the Direct Access Table is authentic and not just an error.

So here is the summary of how we define and use such a design:

1. We define an auxiliary array with records for each of the S elements in any order. There we store the key value/pointer to the corresponding Direct Access Table element.
2. We allocate memory for the Direct Access Table and for all the elements corresponding to the keys in the dictionary we record the pointers to the corresponding auxiliary array element.

To look up if the key k is in the dictionary we

1. Read the pointer in the kth position of the Direct Access Table.
2. Read the value for that pointer lead us to in the auxiliary array which should contain the pointer back to kth element of the Direct Access Table.
3. If this pointer indeed points back and everything checks out, that means this key is indeed in the dictionary. Otherwise, it is not.

One can see that this improvement allows for the initialization time upper bound to be of $O(|S| \log m)$. Indeed, at the initialization we go through each of the elements in $|S|$ and recording the pointer to an element in the Direct Access Table goes as $\log m$.

4 Predecessor/Successor problem

Let us look at how we can design an algorithm for an efficient predecessor element search.

4.1 Naive search (linear search)

We may start with analysing how fast is the naive way of looking for the predecessor element. This 'naive' or linear search would involve simply going 'up' along the Direct Access Table elements starting from the query one until we stumble upon the element in S or just reach the end of the table.

Worst case for such an approach would be going through the whole table, $O(m)$.

How hard is it to add/delete keys to S? We only need to mark the corresponding value at the Direct Access Table, $O(1)$.

4.2 An array of pointers

How can we improve this linear upper bound time we had for the linear search?

One of the design improvements is to just store pointers to the closest elements for each of the elements in the U universe.

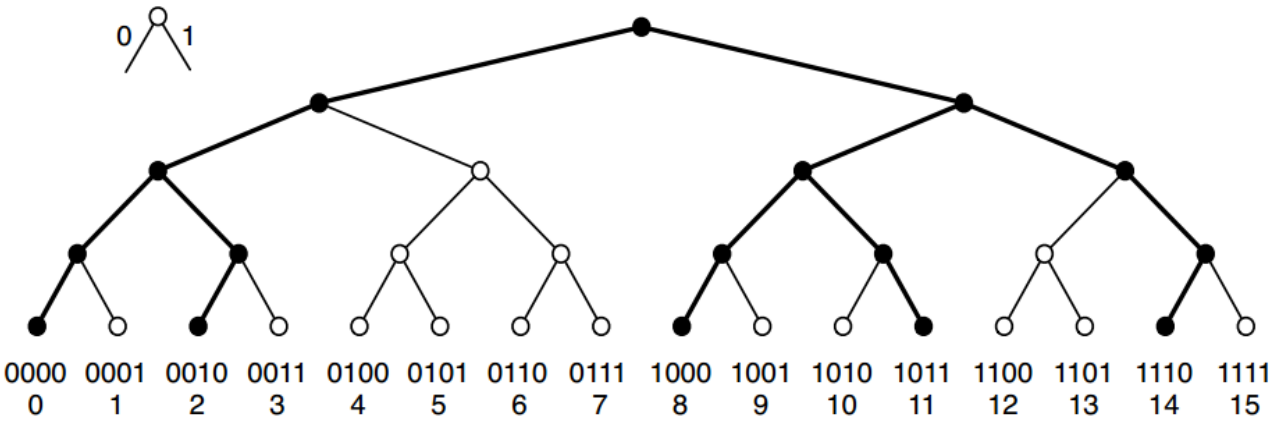
This makes looking up the successor elements very efficient. We just read the corresponding value in the table, $O(1)$. However, it also means that we have to use much more memory space, specifically $m \log m$, as we need to store the pointers of size $\log m$ for each of the elements in U.

It also makes updating the set of keys harder. Now we have to update records for all the adjacent elements, for which the new key was the predecessor or successor. Worst case time for that would be going through the whole table, $O(m \log m)$.

4.3 X-fast trie

X-fast tries is a data structure that can be built on top of the Direct Access Table to store the 'neighbouring elements' information. It enables the predecessor search that does not quite have the drawbacks of the algorithms discussed before.

Here is how it can be constructed. We build a binary tree with the lowest child nodes corresponding to the direct access table records and each parent node connecting the adjacent child notes. Here is a figure depicting an example built on top of the 16-element U universe (reproduced from [1]):



$$S = \{0, 2, 8, 11, 14\} = \{0000_2, 0010_2, 1000_2, 1011_2, 1110_2\}$$

One can see that for $m = 2^k$ we have $k = \log m$ layers of node parents. Now for each node in the Direct Access Table with the corresponding key in the dictionary S we mark with node value 1 each of the parent nodes. Every other node is marked with 0 record (illustrated with black and white nodes on the figure above).

Here is how we can search for the closest element using this x-fast trie graph:

1. We search among all the ancestor nodes of the query element for the deepest one that has 'occupied'/1 value (lowest 'blackened' node on the figure above).

2. If there is one we found, we now know that this node has an element with the key in S as a child node. We go down the tree selecting the marked child nodes to this specific element.

What is the worst case bound for such an algorithm? We need to search among the $\log m$ parent elements for the lowest marked node, so it is $O(\log m)$.

How much time is needed to update the x-fast trie? In order to insert an element we go up along the tree from that new element and if they are not 'marked', update the values. So it would be only $\log m$ operations for the worst case scenario, $O(\log m)$.

Similarly, in order to delete an element we go through all the parents, with the only exception that for each parent node we also look up the value of the other child and update the value accordingly. We have the same bound $O(\log m)$.

Let us count how many nodes we have in this graph. We have one 'common ancestor' node and the number of elements doubling with each layer as we go down along the tree:

$$1 + 2 + 2^2 .. + 2^{\log_2 m} = \frac{1 - 2^{\log_2 m + 1}}{1 - 2} = 2 \cdot 2^{\log_2 m} - 1 = 2m - 1$$

We see that the space needed to store the x-fast trie is only two times as big as the plain Direct Access Table record.

4.4 Implementing an x-fast trie

How can we implement that x-fast trie structure in practice effectively?

Let us define an array with the number of elements in the graph: $2m - 1$. We now can assign the first element to the common ancestor node value, the second two to the two consequetive nodes in the second layer, and so on. Then the last m elements would correspond to the deepest, or the Direct Access Table one, layer.

Such an arrangement allows for a straightforward way to find parent/children nodes for a fixed element i(assuming they are within a range to exist):

$$\begin{aligned} \text{parent of}(i) &= i/2 \\ r\text{th ancestor of}(i) &= i/2^r \\ \text{left child of}(i) &= 2 * i \\ \text{right child of}(i) &= 2 * i + 1 \end{aligned}$$

My example implementation of the x-trie in Python is available online at the Github website [2].

4.5 Summary

Let us summarize the properties of the three discussed algorithms for predecessor element search in a table:

Algorithm	Search time	Inserting/deleting an element
Naive (linear search)	$O(m)$	$O(1)$
Store pointers	$O(1)$	$O(m)$
X-fast trie	$O(\log m)$	$O(\log m)$

Table 1: Comparison of the bounds on the asymptotic execution time for the algorithms for search for the predecessor/successor elements.

References

[1] Talk slides: "Predecessor Data Structures", Philip Bille, 2011, Some slides I found that discuss various data structures for effective predecessor element search and also have some helpful illustrations.

[2] Github: dborzov/XLtrie