CS 008
Lecture notes
4/25/24

1 Direct Access Arrays and Hash Tables

1.1 Outline

- Pre-Lecture questions
- The WORD RAM model (the main computational model for CS008)
- Direct access arrays
- Hash tables

2 Pre-lecture questions

Important question to keep in mind for the lecture:

Big lecture question: Is it possible to execute the function $find(k)$ any more quickly than $O(log(n))$?

3 Word RAM model

Any region of memory can be accessed in $O(1)$ complexity time. In reality it is not but for the most part this statement is true. In the word RAM model, anything that is within 64-bits is within $O(1)$. When we work with a data structure with n elements, $n < 2^w$. For most computers, $w = 64$ and we call this a **word**. Memory is divided into w-bit chunks and each chunk can be read and written in $O(1)$.

4 Comparison model of computation

The comparison model of computation is more restrictive than the Word RAM Model. Like what the name implies, the comparison model can only perform comparisons (==, !=, , , $\leq$, $\geq$)

5 Direct access arrays

Going back to our lecture question:

> Is it possible to execute a $find(k)$ function any more quickly than $O(log(n))$?

Using direct access arrays, you can execute a find function in $O(1)$.
A direct access array is an array where every index in the array is associated with a specific element. Each element has it's own place in the array. Direct access arrays are very fast but have a major flaw where it takes up too much space.

Operations:

As previously stated, direct access arrays are fast however they are very spacially inefficient.

- $build() = O(n)$
- $find(k) = O(1)$
- $add(k) = O(1)$
- $delete(k) = O(1)$

These are fast! However very inefficient space wise.

In summary, direct access arrays are great for speed however for larger ranges of values, the size of the array would be too big. With smaller ranges of numbers, it is OK. Before making a direct access array, you need to know the largest potential value you can have. It is dependent on the largest *potential* element that can be in the array and contains an index for every single element up until that point.

6 Hashmaps

As said before, the biggest flaw to using direct access arrays is space. When working with direct access arrays, you have an index for every possible element.

However, we can reduce the hashmap to a smaller direct access array that leads to lists. We call this a **hash function** but the basic gist of it is that it reduces the large direct access array to a smaller array where the index represents a certain pattern / group of elements.

Consider:
Direct access array: $[a_0, a_1, a_2, \ldots, a_{n-1}]$ where a_i represents a potential value.

Now, consider a hash function using modulus (%). We can define some hash function k where $h(k) = k\%n$. Each group would be called a **key**. So, for example, we could say any value of the direct access array where $h(k) = k\%n \equiv 0$ has a key of 0.

The main advantage of a hash table is how efficient it is for space and it is also a lot less intense to build compared to a direct access array where the array needs an index for every *potential* element. In summary, hash tables are used as a more space efficient way to store values than direct access array while still being fast.