Lecture 11: Hashing
(CLRS 11.1-11.3)

June 3rd, 2002

1 Maintaining ordered set

- Last time we started discussing the problem of maintaining an ordered set \( S \) under operations
  - \text{Search}
  - \text{Insert}
  - \text{Delete}
  - \text{Successor}
  - \text{Predecessor}

- We discussed several implementations
  - Array
  - Linked list
  - Skip lists

- We saw that in skip list all operations have expected running time \( O(\log n) \)
  - Next time we will discuss a data structure (red-black tree) with worst-case \( O(\log n) \) running time.

- We can argue that \( \Theta(\log n) \) time is optimal for searching in the decision tree model

Recall decision tree model:

- Binary tree where each node is labeled \( a_i \leq a_j \)
- Execution corresponds to root-leaf path
- Leaf contains result of computation

- Decision trees correspond to algorithms where we are only allowed to use comparison to gain knowledge about input.
- Decision tree for \text{Search} must have \( n \) leaves (one for each element)
  - Tree must have height \( \Omega(\log n) \)

- In the case of sorting, we saw that we could beat the \( \Omega(n \log n) \) decision tree lower bound using \textit{Indirect Addressing} (Radix sort)
  - we can also use indirect addressing idea on ordered set problem.
2 Direct Addressing

- Store element $e$ in cell $e$ of array (we assume elements are integers)

| 0 | 1 | 2 | 3 | ... | $|U|−1$ |
|---|---|---|---|---|------|
| | | | | | $e$ |

- **INSERT/DELETE/SEARCH** in $O(1)$ time
- **PREDECESSOR/SUCCESSOR** in $O(|U|)$ time ($|U|$ is the size of ”universe” $U$)

- Note: We could make PREDECESSOR/SUCCESSOR efficient by linking neighbor elements, but then Insert/Delete becomes $O(|U|)$

- Problem is that $|U|$ can be huge and often $|U| >> n$
  - 32 bit integers $\Rightarrow |U| = 2^{32}$
- We can reduce space use using ”hashing”

3 Hashing

- To introduce hashing, we look at direct addressing in a slightly different way:

```
U
```

$n$ elements in set $S$

The main idea is to fix the table size to $m = O(n)$

- now element $e$ cannot be stored in cell $e$

We introduce hash function $h(e) : U \rightarrow \{0, 1, ..., m−1\}$

```
U
```

We call the array the hash table
• Problem is of course that several elements can be stored in same cell \((m < |U|)\)
  
  – We call such an event a collision

• We solve this problem using chaining
  
  – Elements mapping to same cell are stored in linked list

  – INSERT/DELETE/SEARCH in \(O(\text{max chain length})\)
  
  – PREDECESSOR/SUCCESSOR in \(O(m + n)\) since we have to look in all cells and chains

(Note : We assume we can compute \(h(e)\) in \(O(1)\) time)

• Note: PREDECESSOR/SUCCESSOR bounds are very bad (we will not discuss them further in the following)
  
  – We call a data structure only supporting INSERT/DELETE/SEARCH a Dictionary
  
  – In a dictionary, order does not really matter
  
  – Lots of applications of dictionaries, e.g.
    * Symbol table in compilers
    * IP addresses to machine-name table

• Performance of hashing depends on how well \(h(e)\) spreads the elements in the hash table
  
  – Lets make the simple uniform hashing assumption

    | Any given element is equally likely to hash into any of the \(m\) cells |

  ↓

  – On average \(\frac{n}{m}\) elements in each chain
  ↓

  – If we choose \(m = O(n)\) we get \(O(1)\) bounds (and \(O(n)\) space)

• How do we choose a good hashing function?
  
  – Often \(h(e) = e \mod m\) is used \((e \mod m\) is remainder of \(e\) divided by \(m\))
    Example : \(m = 12, e = 100 \Rightarrow h(e) = 4\) since \(100 = 8 \cdot 12 + 4\)
  
  – \(m\) is often chosen to be a prime number far away from a power of 2

If \(m = 2^p\) then \(h(e) = \text{lowest} \ p \ \text{bits in} \ e\) which means that the hashing value only depends on some of the bits in \(e\). If data is not random—not all \(p\)-bit patterns equally likely—then this might be a very bad choice, we would rather have \(h(e)\) depend on all the bits
4 Universal Hashing

- Given hash function $h$, we can always find sets of elements that make hashing perform badly ($n$ elements that map to same location)

- Like in Quick-sort and skip lists we can make sure our data structure does not perform badly on a particular input (set of inputs) using randomization
  - We choose a hash function randomly (independent of elements) from a carefully defined set of functions
  
  - no worst case inputs
  - good average case behavior

- We want the set of hash functions to be universal

> Let $H$ be a finite collection of functions $U \rightarrow 0, 1, \ldots, m - 1$. 

$H$ is called universal if and only if for each $x, y \in U$ the number of functions $h \in H$ for which $h(x) = h(y)$ is precisely $|H|/m$.

- If we choose $h$ randomly from $H$ then the probability of collision between $x$ and $y$ is $\frac{|H|/m}{|H|} = \frac{1}{m}$
  
  - If $m > n$, then then expected number of collisions involving element $e$ is $< 1$

  $\Downarrow$

  INSERT/DELETE/SEARCH in $O(1)$ expected

- Note: The book proves the above more formally and talks about how to find universal class of hash functions (not hard but requires some number theory, so we skip it)

5 Dynamic perfect hashing

- It turns out that one can even do searches in $O(1)$ worst-case time
  - Out of scope of this class

- Idea:
  - If set of $n$ keys is static, we could potentially find a perfect hash function $h$

  - We need to be able to store description of $h$ compactly and compute $h$ fast.
– Lots of research has been done on finding perfect hash functions for a given set of elements, resulting in $O(1)$ worst-case SEARCH

– The perfect hashing idea can even be made dynamic such that one also gets $O(1)$ INSERT/DELETE expected running time.

– Lots of recent results even improve on this.