(first module after the midterm)

Datastructures 1
Hash Tables
Red Black Trees
Week 8 Objectives

- Hash Tables, Hashing functions
- Red-Black Trees
Arrays VS Hash Tables

- **typical computer storage is (key,value) pair**
- **arrays must have keys as integers**
  - keys=indices=positions
  - due to how they work in computer’s memory
  - have to be continuous

- **Hash Table also stores (key,value) pairs**
  - keys can be anything, like peoples names
  - keys cannot be used as positions/indices
Basic hashing

- Arrays are very nice, but keys have to be integers
  - Keys from 0 to N-1

- Hashes very useful when keys are not integers
  - Names, words, addresses, phone numbers etc
  - Even if key=integer (like phone #) they are not the integers we want as indices

- Text processing: natural keys are words/n-grams/phrases

- Databases: natural keys can be anything
Hashing for integer keys

- Even if the keys are integers, they might be inappropriate for storage indices.
- Typically the case of few keys in a very large range.
- Example: phone numbers.
  - Might have to use about 10,000 phone numbers as keys
  - If each is used as a index, the resulting array must allocate 9Billion locations (U.S. phone numbers have 10 digits)
Hash Tables

- key -> index -> use array[index] = value
Hash Tables – Collisions

- when several keys (words) map to the same key (index)
- have to store the actual keys in a list
  - list head stored at the index
- key -> index -> list_head -> search for that key

![Diagram of hash table with collisions]
Hash Tables – Collisions with chaining

- when several keys (words) map to the same key (index)
- have to store the actual keys in a list
  - list head stored at the index
- key -> index -> list_head -> search for that key
Hash Tables—Collisions with chaining

- \( n \) = number of keys; \( m = \text{MAXHASH} \); \( \alpha = n/m \)

- **simple uniform hashing**: any key \( k \) equally likely to be mapped on any of the indices \([0...m)\)

- If collisions are handled with chaining linked lists, assuming simple uniform hashing:
  - unsuccessful search for a key takes \( \Theta(1 + \alpha) \)
  - successful search for a key also takes \( \Theta(1 + \alpha) \)
  - proof in the book
Hash Function

• Easy for humans to use such a hash table
• but not easy for a computer
  – need integer memory locations
  – we have to map keys (names, colors etc) into integers

• hash function $h$: take input any key, returns an index $(\text{int}) \ h(\text{key})=\text{index}$

• basic operations: INSERT, DELETE, SEARCH; all use the mapped value $h(\text{key})$
Hash Function

- Usually two stages
  - convert key to a [large] integer (not necessary if keys are already large integers like phone numbers)
  - map the integer in interval \([0, \text{MAXHASH})\)
Simple hash function for words

- return a simple combination of characters, modulo MAXHASH
- int MAXHASH=100000;
- Example hashing word “Virgil” based on ASCII codes

<table>
<thead>
<tr>
<th>V</th>
<th>i</th>
<th>r</th>
<th>g</th>
<th>i</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>86 * 1^2</td>
<td>105 * 2^2</td>
<td>114 * 3^2</td>
<td>103 * 4^2</td>
<td>105 * 5^2</td>
<td>108 * 6^2</td>
</tr>
</tbody>
</table>

- int hash_function(char[]) // returns integers between 0 and MAXHASH
  - int sum=0, i=0;
  - while(char[i]>0) {sum+=char[i] * ++i*i;}
  - return sum % MAXHASH;
Hash function: two qualities

• quality ONE: one-to-one (injection). Different inputs result in different outputs
  – collision: having many keys map to same index

• collisions eventually will happen, need to be solved
  – collisions should be balanced (uniformly distributed) per output indices; same as saying simple uniform hashing (approx) is desirable, even if not exact.

• quality TWO: the set of returned indices must be manageable
  – for example returns integers from 1 to 100000
  – or returns integers in range (0, MAXHASH)
Hash Function – division method

- map key to integer $k$ (key=$k$ if key is already integer)
- $h(k) = k \mod m \ (m=\text{MAXHASH})$
  - this equation guarantees that $h(k)$ is one of $\{0,1,2,\ldots, \text{MAXHASH}-1\}$

- bad choices for $m$ : close to powers of 2
  - $m=2^p$
  - $m=2^p-1$

- good choice for $m$ : prime numbers far away from powers of 2
  - example: $m=701$
Hash Function - multiplication method

- fractional(x) = fractional part of x, or x - ⌊x⌋
  - example fractional(3.1472) = 0.1472

- h(k) = ⌊m * fractional(kA)⌋

- typically m is a power of 2

- A is a fractional of form s/2^w where s<2^w
  - for example A = 2654435769 / 2^{32}
Hash Function – Universal

• if the hash function is known, an adversary can attack the hashing schema by using many keys that all collide to the same index
  
  – $h(\text{key1}) = h(\text{key2}) = h(\text{key3})$...

• to prevent this, we can use set $H$ of hash functions
  
  – universal set $H$: for each pair of keys $(k,l)$ the number of hash functions $h \in H$ that collide $k$ and $l$ $h(k) = h(l)$ is no more than $|H|/m$
  
  – each time we build a hash (run the code), a random hash function is selected from the set

• building a universal set $H$ of hash functions relies on number theory – see book
Red-Black Trees

further reading necessary from textbook
Binary Search Trees – Recap

- each node has at most two children
- any node value is
  - not smaller than any value in the left subtree
  - not larger than any value in the right subtree
  - $h =$ height of tree

Operations:
- search, min, max, successor, predecessor, insert, delete
- runtime $O(h)$
Binary Search Trees – Recap

- each node has at most two children
- any node value is
  - not smaller than any value in the left subtree
  - not larger than than any value in the right subtree
  - \( h = \) height of tree

- Operations:
  - search, min, max, successor, predecessor, insert, delete
  - runtime \( O(h) \)

left subtree values \( \leq 15 \)
Binary Search Trees – Recap

• each node has at most two children

• any node value is
  - not smaller than any value in the left subtree
  - not larger than any value in the right subtree
  - $h$ = height of tree

• Operations:
  - search, min, max, successor, predecessor, insert, delete
  - runtime $O(h)$
Balanced Trees

- a) balanced tree: depth is about \( \log(n) \) – logarithmic
- b) unbalanced tree: depth is about \( n \) – linear
Red-Black Trees

- **binary search tree**
- **want to enforce** balancing **of the tree**
  - height logarithmic in \( n=\)number of nodes in the tree
  - height = longest path root->leaf
- **extra:** each node stores a color
  - color can be either red or black
  - color can change during operations
- **red-black properties**
  - root is black
  - leafs (terminals) are black
  - if a node is red, then both children are black
  - for any given node, all paths to leaves (node->leaf) have the same number of black nodes
Red-Black Trees

Theorem: a red-black tree with n nodes has height at most $2 \log(n+1)$

- or logarithmic height
- thus enforcing the balancing of the tree
- and so all operations can be implemented in $O(\log n)$ time.
Tree operations

- insert, delete - need to account for colors
  - rest of the lecture: insert and delete in red-black trees

- search, min, max, successor, predecessor - same as for regular binary search trees
Red-Black Trees – Rotation

- **Rotation** is a utility operation that facilitates maintenance of red-black properties
  - during insert and delete, the tree might temporarily violate the red-black properties
  - using rotation we can fix the tree so it satisfies red-black.

- Rotate-left at node \( x \)
  - \( x \) is replaced by its right child \( y \)
  - \( \beta = \text{left subtree of } y \text{ becomes right subtree of } x \)
  - \( x \) becomes the left child of \( y \)

- Rotate-right at \( y \) symmetric
Red-Black Trees – Rotation

Example
Red-Black Trees – Insertion

• add node “z” as a leaf
  – like usual in a binary search tree

• color z red, add terminal “NIL” nodes

• check red-black conditions
  – most conditions are still satisfied or easy to fix
  – the real problem might be the condition that requires children of red nodes to be black.
  – start fixing at the new node z, and as we proceed more fixes might be necessary
  – three “fixing cases”
  – overall still $O(\log n)$ time.

• RB-INSERT-FIXUP procedure in the textbook
Fixing insertion case 1

- $z.p = z.parent$ and $y = z.uncle$ are red

- **fix:**
  - make $z.p$ and $y$ black
  - make $z.p.p$ red
  - advance $z$ to $z.p.p$
Fixing insertion case 2

- z.p is red, y is black, z is the right child

- fix:
  - rotate left at z.p
  - z advances to its old parent (now his left child)
Fixing insertion case 3

- z.p red, y black, z is left child

- fix:
  - rotate right at z.p.p
  - color z.p black
  - color old z.p.p (now z brother) red
Red-Black Trees – Deletion

• delete “z” as we usually delete from a binary search tree
  – maintain search property: left values ≤ node value ≤ right values

• additionally keep track of
  – y = the node to replace z
  – y original color (its color might change in the process)

• Fix-up the tree red-black properties, if they are violated
  – a procedure with 4 cases
  – RB-DELETE-FIXUP procedure in the textbook
Fixing deletion case 1

- case 1: x is black, brother w red
- fix:
  - rotate left at x.p;
  - color x.p red;
  - color w (now x.p.p) black
Fixing deletion case 2

- case2: brother w is black, and w children also black

- fix:
  - color w red
  - advance x to its parent
Fixing deletion case 3

- case3: brother w is black; w's left child is red; w's right child is black

- fix:
  - rotate right at w
  - color the new brother from red to black
  - color the old brother from black to red
Fixing deletion case 4

- **case4:** brother w is black, w’s right child is red
- **fix:**
  - rotate left at x.p
  - color old w’s right child from red to black
  - color x.p from red to black
  - color old w from black to red
Running time

- most BST operations same running time as BST trees
  - search, min, max, successor, predecessor
  - these dont affect RB colors
- Insertion including fixup $O(\log n)$
- Deletion including fixup $O(\log n)$