CSE 373 Lecture 14: Midterm Review

✦ Today’s Topics:
  ✓ Wrap-up of hashing
  ✓ Review of topics for midterm exam

✦ Midterm details:
  ✓ Chapters 1-6 in the textbook
  ✓ Closed book, closed notes
  ✓ Format: 5 questions, 100 points total
  ✓ Time: Monday, class time 11:30-12:20 (50 minutes)
  ✓ Blank sheets will be provided
  ✓ Bring pens/sharpened pencils (and sharpened minds)

Hashing: Applications

✦ Hash tables are used in many real-word applications:
  ✓ As symbol tables in compilers – store and access info about variables & functions each time their name appears in program being compiled
  ✓ In game programs: Avoid recomputing moves by storing each board configuration encountered with corresponding best move in a hash table
  ✓ In spelling checkers: prehash entire dictionary and check if words in a document are in dictionary in constant time

Summary of Hashing

✦ Main reason to use hashing: speed!
  ✓ O(1) access time (at the cost of using space $O(\text{TableSize})$)
  ✓ Only supports Insert/Find/Delete (no ordering of items)

✦ Components: TableSize (prime), hash function, collision strategy

✦ Chaining collisions allows $\lambda > 1$ but uses space for pointers

✦ Probing requires $\lambda < 1$ but avoids the time and space needed for allocating pointers

Midterm Review: Math Background

✦ Know the definitions of Big-Oh, little-o, big-omega, and theta:
  ✓ $T(N) = O(f(N))$ if there are positive constants $c$ and $n_0$ such that $T(N) \leq cf(N)$ for $N \geq n_0$
  ✓ Think of $O(f(N))$ as “less than or equal to” $f(N)$ ➔ Upper bound
  ✓ Think of $\Omega(f(N))$ as “greater than or equal to” $f(N)$ ➔ Lower bound
  ✓ Think of $\Theta(f(N))$ as “equal to” $f(N)$ ➔ “Tight” bound, same growth rate
  ✓ Think of $o(f(N))$ as “strictly less than” $f(N)$ ➔ Strict upper bound
  ✓ $T(N) = o(f(N))$ means $f(N)$ has faster growth rate than $T(N)$
Summations

\[ \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \quad \text{for (} i = 1; i <= N; i++ \text{)} \]

\[ \sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \quad \text{for large } N \text{ and } k \neq -1 \]

\[ \sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1} \quad \sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \]

Recurrences

✦ Used to analyze run time \( T(N) \) of recursive function for input size \( N \)

✦ Write down cost of each line of function

✦ For recursive calls, write cost in terms of \( T \) and new input size \( N' \)

✦ E.g. \( T(N) = (\text{cost for non-recursive lines}) + T(N-1) \)

```c
int sum ( int v[], int num)
{
    if (num == 0) return 0;
    else return v[num-1] + sum(v,num-1); 
}
```

\[ T(num) = \text{constant} + T(num-1) \]
\[ = 2 \text{constant} + T(num-2) = \ldots = num \text{constant} + \text{constant} = \Theta(num) \]

Lists, Stacks, and Queues

✦ Lists: Insert, Find, Delete

✦ Singly-linked lists with header node

✦ Doubly-linked and Circularly-linked

✦ Run time and space needed for array-based versus pointer-based

✦ Stacks: Push, Pop

✦ Know what push and pop do

✦ Pointer versus array implementation

✦ Use of stacks in balancing symbols and function calls

✦ Queues: Enqueue and Dequeue

✦ Array-based implementation using Rear and Front, and modulo arithmetic for wrap-around

Trees

✦ Terminology: Root, children, parent, path, height, depth, etc.

✦ Height of a node is maximum path length to any leaf

✦ Height of tree is height of root

✦ Single node tree has height and depth 0

✦ Recursive definition of tree

✦ Null or a root node with (sub)trees as children

✦ Preorder, postorder and inorder traversal of a tree

✦ Implementation using recursion or a stack

✦ Minimum and maximum depth of a binary tree
**Binary Search Trees**

- BSTs: What makes a binary tree a BST?
  - Know how to do Find, Insert, and Delete in example BSTs
- AVL tree: What makes a BST an AVL tree?
  - Balanced due to restriction on heights of left/right subtrees
  - Upper bound on height of AVL tree of N nodes
  - Worst case run time for operations
  - Know what happens when you do Inserts into an AVL tree
  - Re-balancing tree using Single or Double rotation
- Splay trees: No explicit balance condition but accessing an item causes splaying (rotations); item moves to root
  - Amortized/worst case running time for operations
  - Know what happens when you do Find/Insert/Delete

**B-Trees**

- Nodes have up to M children, with M-1 keys
  - Children to the right of key k contain values ≥ k
- All leaf nodes at same height
- Know how to do Find, Insert, and Delete in example B-trees
  - Insert may cause leaf node to overflow and split, causing parent to split etc.
  - Deletion may cause leaf to become less than half full, causing a merge with sibling, which may cause parent to merge etc.
- What is the depth of an N-node B-tree?
  - Find: Run time is \(O(\text{depth} \times \log M) = O(\log \left(\frac{M}{2}\right) N \times \log M) = O(\log N)\)
  - Insert and Delete: Run time is \(O(\text{depth} \times M) = O(M \log M) \times \log N)\)

**Priority Queues: Binary Heaps**

- What is a binary heap?
  - Understand array implementation – parent and children in array
  - d-heaps: d children per node
- Main operations: FindMin, Insert, DeleteMin
  - Know how to Insert/DeleteMin in example binary heaps
  - Insert – add item to end of array, then percolate up
  - DeleteMin – move item at end of array to top, then percolate down
- Other operations: DecreaseKey, IncreaseKey, Merge
- Depth and running time of operations for binary heap of N nodes
- No need to know details of leftist or skew heaps

**Binomial Queues**

- Recursive definition of binomial trees
  - Contains one or more trees \(B_i\), each containing exactly \(2^i\) nodes
- Binomial queue = forest of binomial trees, each obeying heap property
- Main operation: Merge two binomial queues
  - Start from \(i = 0\) and attach pairs of \(B_i\) creating \(B_{i+1}\)
- Insert item: Merge original BQ with new one-item BQ
- DeleteMin: Delete smallest root node and merge its subtrees with original BQ
- First Child/Next Sibling implementation and run time analysis
Hashing

- Know how hash functions work:
  - Hash(X) = X mod TableSize
  - TableSize is chosen to be a prime number in real-world applications
- Know how the different collision resolution methods work:
  - Chaining: colliding values are stored in a linked list
  - Open addressing with linear probing: look linearly (F(i) = i) for empty slot starting from initial hash value; clustering problem
  - Open addressing with quadratic probing: look using squares (F(i) = i^2) for empty slot starting from initial hash value; theorem guarantees a slot if TableSize prime and array less than half full
  - Rehashing: when probing is used and the table starts to get full
- Know what the load factor $\lambda$ of a hash table means and how the run time of Find/Insert is related to $\lambda$

Next Class: Midterm exam

To Do:
1. Hash everything into brain but minimize collisions
2. Ace the midterm