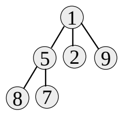
## CSE 373 Lecture 12: Hashing

- Today's Topics:
  - Summary of Priority Queues
    - d-heaps, leftist, and skew heaps
  - Hashing
    - Hash functions
    - Collision Resolution
      - Separate Chaining
      - Open Addressing (or probing)
- Covered in Chapters 5 and 6 in the text

## Other Priority Queues: d-Heaps

- Similar to a binary heap, except we allow more than 2 children per node
- d-heap has d children per node
- Example: 3-heap – root is A[1] and children of node A[i] are A[3i-1], A[3i], A[3i+1]
- Just as in B-tree, more children means shallower heap
  - Depth is $O(\log_d N)$ instead of $O(\log_2 N)$
  - But, d-1 comparisons to find smallest child
  - Tradeoff between depth and "breadth"
  - Optimal d value is application dependent



## Other Priority Queues: Leftist and Skew Heaps

- Leftist Heaps: Binary heap-ordered trees with left subtrees always "longer" than right subtrees
  - Null path length (NPL) = shortest path length to a leaf or 1-child node
  - Leftist heap: NPL(left child) ≥ NPL(right child) for all nodes
  - Right path is always short → has O(log N) nodes
  - Main idea: Recursively work on right path for Merge/Insert/DeleteMin
  - Merge, Insert, DeleteMin all have O(log N) running time (see text)
- Skew Heaps: Self-adjusting version of leftist heaps (*a la* splay trees)
  - No restriction on NPL
  - Adjust tree by swapping left and right children during each merge
  - O(log N) amortized time per operation for a sequence of M operations
- We will skip details for now (may return to these later in the quarter)

## Summary of Priority Queues

- Balanced binary search trees: Find in O(log N) time
  - Priority Queues: FindMin in O(1) time
- Most common PQ: array-based Binary Heap
  - FindMin is O(1) while Insert and DeleteMin are O(log N)
- Merging is inefficient for binary heaps (O(N) time)
  - Pointer-based alternatives such as Binomial Queues allow merging in O(log N) time
- Priority queues are used in applications (such as job schedulers in OS) where repeated searches are made to find and delete the minimum (highest priority) items

## Hashing: Motivation

- Data structures we have looked at so far (Lists and BSTs):
  - Used comparison operation to find items
  - Needed O(N) or O(log N) time for Find and Insert
- In real world applications, N is typically between 100 and 100,000 (or more)
  - log N is between 6.6 and 16.6
- What if we could do Find and Insert in O(1) time?
  - Could speed up our application by a factor of over 16
- Hash tables are designed for O(1) Find and Inserts…

## Hash Tables: Motivation

- Data records can be stored in arrays. E.g.
  - A[0] = {“CSE 143”, Size 151, Avg. Grade 3.4}
  - A[3] = {“CHEM 110”, Size 89, Avg. Grade 3.1}
  - A[17] = {“CSE 373”, Size 77, Avg. Grade 3.9}
  - (note the high expectations for CSE 373)
- Suppose you want to know the class size for CSE 373
  - Need to search the array – O(N) worst case time
- What if we could directly index into the array using the key?
  - A[“CSE 373”] = {Size 77, Avg. Grade 3.9}
- Main idea behind hash tables: Use a key (string or number) to index directly into an array – O(1) time to access records

## Hash Tables: How?

- Problem: Need a *hash function* to convert the key (string or number) to an integer (*hash value*)
  - Can then use this value to index into an array
  - E.g. Hash(“CSE 373”) = 155, Hash(“CSE 143”) = 22, etc.
- Constraint: Output of hash function should always be less than size of array (stored in the variable *TableSize*)
- Solution: Use modulo arithmetic
  - Recall: A mod B = remainder when A is divided by B (= A % B)
  - E.g. If TableSize = 100, 155 mod 100 = 55 and 22 mod 100 = 22.

## Hash Functions

- If keys are integers, we can use the hash function:
  - Hash(*key*) = *key* mod *TableSize*
- Problem 1: What if *TableSize* is 10 and all keys end in 0?

## Hash Functions

- If keys are integers, we can use the hash function:
  - Hash(*key*) = *key* mod *TableSize*
- Problem 1: What if *TableSize* is 10 and all keys end in 0?
  - Need to pick *TableSize* carefully: typically, a prime number

## Hash Functions

- If keys are strings (in the form `char *key`), can get an integer by adding up ASCII values of characters in *key*

  -
    ```
    While (*key != '\0')
      StringValue += *key++;
    ```
- Problem 2: What if *TableSize* is 10,000 and all keys are 8 or less characters long? (chars have values between 0 and 127)
  - Keys will hash only to positions 0 through 8*127 = 1016
  - Need to evenly distribute keys

## Hashing Strings

- Problems with adding up character values for string keys
  1. If string keys are short, will not hash to all of the hash table
  2. Different character combinations hash to same value
     - "abc", "bca", and "cab" all add up to 6
- Suppose keys can use any of 26 characters plus blank
- A good hash function for strings: treat characters as digits in base 27 (using "a" = 1, "b" = 2, "c" = 3, "d" = 4 etc.)
  - "abc" = $1*27^2 + 2*27^1 + 3 = 786$
  - "bca" = $2*27^2 + 3*27^1 + 1 = 1540$
  - "cab" = $3*27^2 + 1*27^1 + 2 = 2216$
- Can use 32 instead of 27 and shift left by 5 bits for faster multiplication (as in textbook)

## Properties of Good Hash Functions

- Should be efficiently computable – O(1) time
- Should hash evenly throughout hash table
- Should utilize all slots in the table
- Should minimize collisions

## Collisions and their Resolution

- A collision occurs when two different keys hash to the same value
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1
- Cannot store both data records in the same slot in array!
- Two different methods for collision resolution:
  - **Separate Chaining:** Use data structure (such as a linked list) to store multiple items that hash to the same slot
  - **Open addressing (or probing):** search for empty slots using a second function and store item in first empty slot that is found

Next Class:

Chaining, Probing, and Hashing

To Do:

Read Chapter 5

Programming Assignment #1 (due April 27)

Have a great weekend!