Advanced data structures

William Hendrix

Lecture 11

Outline

- Project overview
- Review
- Other dictionary implementations
  - Bit vectors
- Heaps
- Union-find
- Prefix and suffix trees (not going to cover)

Project 1

- Implement 4 sorting algorithms
  - SelectionSort
  - InsertionSort
  - MergeSort
  - QuickSort
  - Submit your code (C++ or Java)
- Run these algorithms with 4 kinds of input
  - Increasing, decreasing, random, and constant arrays
  - Array size 10k-100k
  - Submit table of results (10 x 16 + headers)
- Apply OLS regression to estimate time complexity
- Compare empirical complexity to theoretical (given in assignment)
  - Submit PDF with equations and analysis

Hash tables

- Apply hash function to map value into an allocated array
- Use one of two strategies to handle collisions
- Separate chaining
  - Each location is the head of a linked list
  - Append new element to list
  - Never "need" to reallocate
- Open addressing
  - Find the next open location, insert there
    - Can scan quadratically to avoid "congestion"
  - No links, so table can be larger with same memory
  - Linear scanning benefits from caching
  - Deleting an element requires reinserting everything that follows
- Expected-case complexity
  - Collisions are relatively infrequent (O(1))
  - Array size (*m*) is *O(n)*

Hash table complexity

Operation	Hash tables worst-case	Hash tables expected-case	Balanced BST
Search(x)	*O(n)*	*O(1)*	*O(lg n)*
Delete(x)	*O(n)*/***O(n²)***	*O(1)*	*O(lg n)*
Insert(x)	*O(n)*	*O(1)*	*O(lg n)*
Build	***O(m + n²)***	*O(n)*	*O(n lg n)*
Resize	***O(m + n²)***	*O(1), amortized*	*n/a*
Min()	*O(m + n)/O(m)*	*O(n)*	*O(lg n)*
Max()	*O(m + n)/O(m)*	*O(n)*	*O(lg n)*
Predecessor(x)	*O(m + n)/O(m)*	*O(n)*	*O(lg n)*
Successor(x)	*O(m + n)/O(m)*	*O(n)*	*O(lg n)*

- This is amazing!
- The three most important techniques are hashing, hashing, and hashing.

-Udi Manber, Chief Scientist, Yahoo! (2001)

Hash tables: summary

- Poor worst-case complexity
- Excellent expected-case complexity
- Often fastest data structure in practice
  - Not as space-efficient as array
- **However:** be careful about expected-case assumptions
  - Hash function matters *intensely* for good performance
    - If too many values are mapped to the same location, we get worst-case performance
    - If data distribution includes lots of values that hash function collides, we get worst-case performance
    - Must be fast (impacts all operations)
  - Data distribution is also important
    - Some datasets are very skewed in frequency
    - Another data structure might be more appropriate
  - Need to ensure that we don't insert too many elements into table
    - Load factor: n / m
    - Usually resize above a certain threshold (e.g., 0.5, 0.75)

Bit vectors

- Represents a set as a sequence of Boolean values (bits)
  – One bit per value the set could contain
- **Example:**
  – Set of 1-100

00000000	00000000	00000000	00000000	00000000	00000000	00000000
8	16	24	32	40	48	56

00000000	00000000	00000000	00000000	00000000	0000
64	72	80	88	96	100

  – Insert 3, 15, 25, 27, 82, 96, 100

Bit vectors

- Represents a set as a sequence of Boolean values (bits)
  - One bit per value the set could contain
- **Example:**
  - Set of 1-100

00100000	00000010	00000000	10100000	00000000	00000000	00000000
8	16	24	32	40	48	56

00000000	00000000	00000000	01000000	00000001	0001
64	72	80	88	96	100

  - Insert 3, 15, 25, 27, 82, 96, 100
  - Byte = floor((x - min) / 8)
  - Bit = (x-min) mod 8
- Implemented as `char` array (C, C++, Java)
- Use bitwise operations
  - Left shift and right shift (`<<`, `>>`)
  - Bitwise and, or, xor, not (`&`, `|`, `^`, `~`)

Bit vector operations

- **Search(x)**
  - Test bit with bitwise and
  - `arr[byte] & (1 << bit)`
  - *O(1)* time
- **Insert(x)**
  - Set bit with bitwise or
  - `arr[byte] |= (1 << bit)`
  - *O(1)* time
- **Delete(x)**
  - Unset bit with bitwise and
  - `arr[byte] &= ~(1 << bit)`
  - *O(1)* time

Summary: bit vector

Operation	Bit vector	Hash table (expected)	Balanced BST
Search(x)	*O(1)*	*O(1)*	*O(lg n)*
Delete(x)	*O(1)*	*O(1)*	*O(lg n)*
Insert(x)	*O(1)*	*O(1)*	*O(lg n)*

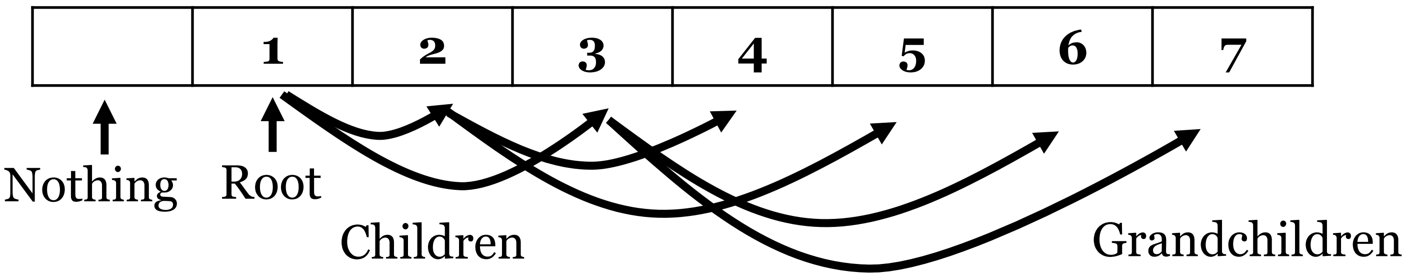
- Constant time operations, even in worst case
- Very low coefficients
- Set operations (union, intersection, difference) can be implemented with bitwise ops
- Can only store integer data
- Cannot store duplicate values
- Space (and initialization/set op cost) determined by data *range*, not by number of values (n)
  - Inefficient if range not bounded

Priority queues

- Data structure for finding maxima or minima
  - E.g., in a greedy algorithm
- Main operations
  - **Insert(x)**: adds a value to the heap
  - **Max()/Min()**: returns max/min value
  - **DeleteMax()/DeleteMin()**: deletes the max/min value from the heap
- Primary implementation: heap
  - Complete binary tree (possibly incomplete bottom level)
  - Heap property
    - Every child has value ≤ its parent's value (max-heap)
    - Every child has value ≥ its parent's value (min-heap)
    - Consequence: root node is max (min)
  - Extra op:
    - **Heapify(arr)**: builds a heap from an unsorted array

Heap implementation

- Implemented as array
- Root stored at index 1
- Children of element *i* stored at 2*i* and 2*i*+1
  - Parent at floor(*i*/2)



- Because heaps are complete, array has no "gaps"

Priority queue operations

- Descriptions assume max-heap
- **Insert(x):**
  - Append to array
  - Swap with parents until parent is larger (or at root)
  - $O(\lg n)$ time
- **Max():**
  - Return root
  - $O(1)$ time

Priority queue deletion

- **DeleteMax()**
  - Swap max with last element and call PercolateDown(1)
  - **PercolateDown(i)**
    - If *heap[i]* is smaller than its max child, swap
    - If so, repeat on that element until it is larger than max child or a leaf
    - $O(\lg n)$ time
  - $O(\lg n)$ time
- **Heapify()**
  - Call PercolateDown(i) from end to beginning
  - Half have no children, half of rest have 1 child, etc
  - $O(n)$ time total

Algorithm: Delete(x)
Swap $heap[1]$ and $heap[n]$;
$n = n - 1$;
PercolateDown(1);

Algorithm: PercolateDown(i)
if $2i \leq n$ **then**
    $mc = 2i$;
    if $mc + 1 \leq n$ and $heap[mc + 1] > heap[mc]$ **then**
        $mc = mc + 1$;
    end
    if $heap[i] < heap[mc]$ **then**
        Swap $heap[i]$ and $heap[mc]$;
        PercolateDown(mc);
    end
end

Algorithm: Heapify(i)
for $i = \lfloor n/2 \rfloor$ to 1 **step** -1 **do**
    PercolateDown(i);
end

Priority queue implementations

Operation	Heap	Unsorted array	Sorted array	Balanced BST	Fibonacci heap
Insert(x)	*O(lg n)*	*O(1)*	*O(n)*	*O(lg n)*	*O(1)*
Max()	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*
DeleteMax()	*O(lg n)*	*O(n)*	*O(1)*	*O(lg n)*	*O(lg n), amort.*
Build	*O(n)*	*O(n)*	*O(n lg n)*	*O(n lg n)*	*O(n)*

- BST has similar complexity, but higher coefficients
- Great at finding max (or min)
- Other operations (min/max, search, predecessor, etc.) are not good
  - Min-max heap can do either, but is more complex
- Fibonacci heap has even better complexity
  - More complex, higher coefficients, less space efficient
  - Fairly slow unless data is quite large

Union-Find data structure

- A.k.a., disjoint set data structure
- Used to represent a *partition*
  - Larger set split into smaller sets with no overlap
  - E.g., clusters, connected components
- Primary operations
  - **Find(x)**
    - Return the partition ID for element x
    - All elements in the same partition must return the same value
    - IDs might not be consecutive
  - **Union(a, b)**
    - Join partitions containing *a* and *b*

Union-Find implementation

- Array contains element IDs
- Partition IDs are elements pointing to themselves
- Initially, all elements are isolated:

0	1	2	3	4	5	6	7
{0}	{1}	{2}	{3}	{4}	{5}	{6}	{7}

- First try:
  - Find(x)
    - Follow links until you hit a partition ID
    - Return partition ID
    - *O(n)* time
  - Union(a, b)
    - Point Find(a) to b
    - *O(n)* time

```
Algorithm: Find(x)
if unionfind[x] = x then
    return x;
else
    id = Find(unionfind[x]))
    return id;
end
```

```
Algorithm: Union(a, b)
id = Find(a);
unionfind[id] = b;
```

Coming up

- Finish union-find
- Sorting algorithms

- **Project 1** is posted on Canvas
  – Sorting algorithms, Big-Oh analysis

- **Recommended readings:** Sections 4.2 and 4.5
- **Practice problems:** p. 100: 1-2 problems from "Applications of Tree Structures", attempt a problem from "Interview Problems"