ECE 590.01
C++ Programming, Data structures, and Algorithms

Heaps and Priority Queues

Admin
- Reading
  - Chapters 6 + 7
- Midterm Graded
  - Stats on Piazza
- Homework 3: Due Friday

Talking about recently
- Recently, talking about Maps and Sets
  - Ubiquitous ADTs
  - Implementations
    - Linked Lists
    - Arrays
    - BSTs (possibly balanced)
    - Hash tables
- Before that: Stacks + Queues
  - Easy + efficient to implement with a LinkedList

Talking about recently
- Recently, talking about Maps and Sets
  - Ubiquitous ADTs
  - Implementations
    - Linked Lists
    - Arrays
    - BSTs (possibly balanced)
    - Hash tables
- Before that: Stacks + Queues
  - Easy + efficient to implement with a LinkedList
- Now, new ADT: Priority Queue
  - And how to implement it efficiently

Queue: FIFO
- Recall Queues: First in, First out
  - Enqueue: add to end of queue
  - Dequeue: take from front of queue

Queue: FIFO
- Recall Queues: First in, First out
  - void enqueue(T): add to end of queue
  - T dequeue(void): take from front of queue
  - T peek(void): look at front of queue
- Priority Queue: Best priority first
  - void enqueue(T, int): add with a given priority
  - T dequeue(void): take highest (or lowest) priority item
  - T peek(void): look at highest (or lowest) priority item
**Priority Queue: Why?**

- Why would we want this?
  - Job schedulers
  - Dijkstra's shortest path algorithm (after the break)
  - Huffman compression algorithm (today)
  - ...and many more...

---

**Naive implementation**

- Naive implementation:
  - Sorted Linked List
    - Enqueue = Sorted Insert: O(N)
    - Dequeue = Remove from Front: O(1)
    - Peek = Get Front: O(1)
  - Unsorted Linked List
    - Enqueue = Add to Front: O(1)
    - Dequeue = Remove Max [or Min]: O(N)
    - Peek = Find Max [or Min]: O(N)
- Better implementation:
  - Balanced BST
    - Enqueue = BST Add: O(lg N)
    - Dequeue = BST Remove: O(lg N)
    - Peek = Find Max [or Min]: O(lg N)

---

**Heaps**

- Heaps provide exactly what we want for Priority Queues
  - Many flavors, we will start with Binary Heaps
  - Tree-like structure, different rules than BST
    - Complete tree:
      - All levels (except maybe last) completely full
      - Last level has all nodes at left side
      - Each node is greater than its two children for a Max-Heap
      - Or less than for a Min-Heap

---

**Naive implementation**

- Can we do better?

---

**Heaps**

- Adding to a heap:
  - Place item in next open spot
  - Leftmost un-taken spot on unfilled row
  - Bubble up until heap ordering is restored
  - Swap with parent if greater
- Example: Add 88

---

**Heaps**

- Add 88
  - Place in first available spot
  - Violates ordering with respect to parent 88 > 13

---

**Heaps**

- Add 88
  - Place in first available spot
  - Violates ordering with respect to parent 88 > 13
  - Swap 13 with 88
  - Now 88 < 99, so done
- Now, you all add 75
Heaps

• Add 75
  • Place in first available spot
  • Violates ordering with respect to parent 75 > 1

Heaps

• Add 75
  • Place in first available spot
  • Violates ordering with respect to parent 75 > 1
  • Swap 1 and 75
  • 75 > 42, so still not done

Heaps

• Add 75
  • Place in first available spot
  • Violates ordering with respect to parent 75 > 1
  • Swap 1 and 75
  • 75 > 42, so still not done
  • Swap 75 and 42, done

Heaps

• Delete Max
  • Replace max item (top) with rightmost item in last level
  • Bubble down to fix heap
  • Swap with largest child when 2 children present

Heaps

• Delete Max
  • Remove 99 and replace it with 1 (removing 1 also)

Heaps

• Delete Max
  • Remove 99 and replace it with 1 (removing 1 also)
  • Now need to swap 1 with larger of its two children (88)
Heaps

- Delete Max
  - Remove 99 and replace it with 1 (removing 1 also)
  - Now need to swap 1 with larger of its two children (88)
  - Now swap 1 with 13

- Delete Max again
  - Remove 88 and replace it with 1 (removing 1 also)
  - Need to swap 1 with 75
  - Need to swap 1 with 42
**Array Representation of Heaps**

- Heaps: easily stored in arrays
  - No need for explicit pointers, just use math
  - Scheme 1: Root at index 0
    - Left at $2N + 1$
    - Right at $2N + 2$
  - Scheme 2: Root at index 1
    - Left at $2N$
    - Right at $2N + 1$

**Heap enqueue**

```cpp
class Heap {
private:
    int * data;
    int array_size;
    int last_element;
    void bubbleUp(int index) { ... }
public:
    void enqueue(int prio) {
        last_element++;
        if (last_element == array_size) {
            array_size = array_size * 2;
            data = realloc(array_size * sizeof(*data));
        }
        data[last_element] = prio;
        bubbleUp(last_element);
    }
}
```

**Bubble up: Without Sentinel**

```cpp
void bubbleUp(int index) {
    //check that we aren't at the top already
    if (index <= 1) {
        return;
    }
    //compute parent index
    int parent = index / 2;
    if (data[parent] < data[index]) {
        //swap parent and index
        int temp = data[parent];
        data[parent] = data[index];
        data[index] = temp;
        //bubble up parent
        bubbleUp(parent);
    }
}
```

**Heap dequeue**

```cpp
class Heap {
private:
    int * data;
    int array_size;
    int last_element;
    void bubbleDown(int index) { ... }
public:
    int dequeue() {
        int ans = data[1];
        data[1] = data[last_element];
        last_element--;
        bubbleDown(1);
        return ans;
    }
}
```

**Bubble up: With Sentinel**

```cpp
void bubbleUp(int index) {
    //Using Sentinel?  Don't need this check!
    if (index <= 1) {
        return;
    }
    //compute parent index
    int parent = index / 2;
    if (data[parent] < data[index]) {
        //swap parent and index
        int temp = data[parent];
        data[parent] = data[index];
        data[index] = temp;
        //bubble up parent
        bubbleUp(parent);
    }
}
```
Heap dequeue

```c
void bubbleDown(int index) {
    //How about you all try this one?
}
```

Heap dequeue

```c
void bubbleDown(int index) {
    int left = 2 * index;
    int right = 2 * index + 1;
    if (left > last_element) {
        return;
    }
    if (left == last_element) {
        //corner case: only have left child
        if (data[left] > data[index]) {
            swap(left, index);
        }
    }
    int maxidx = data[left] > data[right] ? left : right;
    if (data[maxidx] > data[index]) {
        swap(maxidx, index);
        bubbleDown(maxidx);
    }
}
```

Generality

- Can make templated PriorityQueue/Heap...
- Design choices and considerations
  - PQ of Ts: is priority part of T, or explicit/separate?
  - enqueue(T): use overloaded < on Ts to order
  - enqueue(T,int): order by ints, T does not need <
  - Priority is not an int? Using sentinel may be difficult
  - Ints are finite for computers:
    - INT_MAX = largest signed int
    - UINT_MAX = largest unsigned int
  - Strings are comparable, but not finite
  - Thing you have the largest string?
  - Add one letter to the end
  - Min heap may be easier, but requires programmer to know min value for type

Binary Heap vs Fancier Heaps

- We did “binary heap”
  - O(1) find min
  - O(lg N) insert
  - O(lg N) remove min
  - Fancier heaps exist
    - Binomial Heaps
    - Fibonacci Heaps
  - We aren’t going to do them, good to know they exist

Application: Compression

- Huffman coding (compression) easily implemented with PQ
- Algorithm works with (symbol, frequency) pairs
  - Symbol = letter = char
  - Frequency = count of how many = int
- Builds binary tree of symbols from bottom up
  - Not BST, just binary tree: no ordering
  - Built from min frequency first: use PQ for efficiency
  - Resulting tree tells encoding: path from root to symbol is its code
    - Left = 0
    - Right = 1

Huffman Motivation/Example

- Frequencies of symbols drastically different
  - a 5985
  - b 1276
  - c 4322
  - d 3046
  - e 13663
  - f 1901
  - g 1764
  - h 3631
  - i 6788
  - j 6788
  - k 16
  - l 76
  - m 166
  - n 3
  - o 53168
- Use short bit sequences for common syms, long for uncommon
Huffman Example

- Initial Heap (using . for space)
- Level 0: Z
- Level 1: j J
- Level 2: M Q X B
- Level 3: F H O U Y z C D
- Level 4: k q K G v P w X V A y n f o E
- Level 5: a p h I i r s L e u t N d S T b c g m W I

Tree building algorithm

- Then we run this algorithm:
  - P is a priority queue which orders Nodes by frequency
  - Leaf Nodes (just a symbol): that sym’s frequency
  - Internal Nodes: sum of leafs under it

while (p->count() >= 2) {
  Node * l = p->dequeue();
  Node * r = p->dequeue();
  Node * x = new Node(l, r);
  p->enqueue(x);
}

- At the end, p->dequeue() has the root of the tree

After one step

- Level 0: (Z, J)
- Level 1: Q
- Level 2: M U X B
- Level 3: F H O R Y z C D
- Level 4: k q K G v P w X V A y n f o E
- Level 5: a p h I i r s L e u t N d S T b c g m W I

After two steps

- Level 0: ((Z, J), j)
- Level 1: Q X
- Level 2: M U z B
- Level 3: F H O R Y A C D
- Level 4: k q K G v P w X V A y n f o E
- Level 5: a p h I i r s L e u t N d S T b c g m W

Final tree

- (Part of the) Final tree:
Reading off the encoding

- Now we build a map from symbols to bit strings
- Read the encoding off the binary tree by traversal

Final tree

- (Part of the) Final tree:

Huffman Coding continued

- Our less common symbols get very long strings
  - U = 0111101001
  - V = 1010000001
  - W = 1010000110
  - X = 00111110001
  - Y = 0111101101
  - Z = 00111110000000
  - J = 001111110000001
  - j = 001111100001
- For frequencies I used,
  - Average bits/symbol = 4.01446
  - Uncompressed, best we can do with ~64 symbols is 6 bits/sym
  - Saved 33% of the space

Huffman Decompression

- Decompression relies on the fact nothing is a prefix of anything else
  - Z = 00111111000000
  - J = 001111110000001
  - j = 001111110000001
- May share a common prefix (path through tree) with other symbols
  - But one symbol’s encoding may not be a prefix of another
  - J, j, and Z all start the same, but none is a prefix of the other
  - Easy to see this is true:
    - Symbols are at the leaves of the tree
    - Unique paths through the tree produce unique bit strings
    - Leaves are not on a prefix of any other path through the tree
Heaps and Sorting

- Heaps are also good for sorting
  - Maps and Sets: most ubiquitous ADTs
  - Searching and Sorting: most ubiquitous algorithms
- Heapsort: efficient sorting algorithm, using heaps
  - Next time: all about sorting