CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC05 SORTING II : HEAPSORT / PRIORITY QUEUES CONT. (PG. 103-120)

Lecture slide courtesy of Prof. Steven Skiena

PROBLEM OF THE DAY

Take as input a sequence of 2n real numbers. Design an O(n log n) algorithm that partitions the numbers into n pairs, with the property that the partition minimizes the maximum sum of a pair.

For example, say we are given the numbers (1,3,5,9). The possible partitions are ((1,3),(5,9)), ((1,5),(3,9)), and ((1,9),(3,5)). The pair sums for these partitions are (4,14), (6,12), and (10,8). Thus the third partition has 10 as its maximum sum, which is the minimum over the three partitions.

HEAP DEFINITION

A *binary heap* is defined to be a binary tree with a key in each node such that:

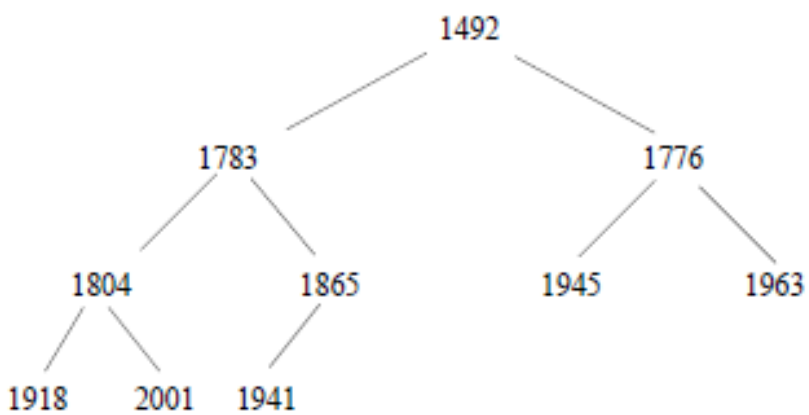
- 1. All leaves are on, at most, two adjacent levels.
- 2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
- 3. The key in root is $\leq$ all its children (min heap), and the left and right subtrees are again binary heaps.

Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.

* *heap property*: If A is a parent node of B then the *key* (the *value*) of node A is ordered with respect to the key of node B with the same ordering applying across the heap

BINARY HEAPS

- Heaps maintain a partial order on the set of elements which is weaker than the sorted order (so it can be efficient to maintain) yet stronger than random order (so the minimum element can be quickly identified).



1	1492
2	1783
3	1776
4	1804
5	1865
6	1945
7	1963
8	1918
9	2001
10	1941

A heap-labeled tree of important years (l), with the corresponding implicit heap representation (r)

ARRAY-BASED HEAPS

- The most natural representation of this binary tree would involve storing each key in a node with pointers to its two children.

- However, we can store a tree as an array of keys, using the position of the keys to *implicitly* satisfy the role of the pointers.
  - The *left* child of k sits in position $2k$ and
  - The *right* child in $2k + 1$.
  - The *parent* of k is in position floor(k/2).

CAN WE IMPLICITLY REPRESENT ANY BINARY TREE?

- The implicit representation (array-based) is not efficient if the tree is sparse, meaning that the number of nodes $n < 2^h$.
  - All missing internal nodes still take up space in our structure.
- Space efficiency demands that the heap be balanced/full at each level as possible.

- The array-based representation is also not as flexible to arbitrary modifications as a pointer-based tree. (we don't use it for binary search trees)

CONSTRUCTING HEAPS

- Heaps can be constructed incrementally, by inserting new elements into the left-most open spot (n+1 position) in the array.
- If the new element is greater than its parent, swap their positions and recur.
- Since all but the last level is always filled, the height *h* of an *n* element heap is bounded because:

$$\sum_{i=1}^{h} 2^i = 2^{h+1} - 1 \geq n$$

so $h = \lfloor \lg n \rfloor$

- Doing *n* such insertions takes $\theta(n \log n)$, since the last *n/2* insertions require $O(\log n)$ time each

HEAP INSERTION

```
typedef struct {
        item_type q[PQ_SIZE+1];        /* body of queue */
        int n;                         /* number of queue elements */
} priority_queue;
```

```
pq_insert(priority_queue *q, item_type x)
{
    if (q->n >= PQ_SIZE)
        printf("Warning: overflow insert");
    else {
        q->n = (q->n) + 1;
        q->q[ q->n ] = x;
        bubble_up(q, q->n);
    }
}
```

bubbling up the new key to its proper position in the hierarchy.

BUBBLE UP

```
pq_parent(int n)
{
        if (n == 1) return(-1);
        else return((int) n/2);
}
```

```
bubble_up(priority_queue *q, int p)
{
    /* at root of heap, no parent */
    if (pq_parent(p) == -1) return;

    if (q->q[pq_parent(p)] > q->q[p]) {
        pq_swap(q,p,pq parent(p));
        bubble_up(q, pq parent(p));
    }
}
```

BUILDING HEAP

- Thus an initial heap of *n* elements can be constructed in $O(n \log n)$ time through *n* such insertions:

```
pq_init(priority_queue *q)
{
        q->n = 0;
}

make_heap(priority_queue *q, item_type s[], int n)
{
        int i;                          /* counter */

        pq_init(q);
        for (i=0; i<n; i++)
                pq_insert(q, s[i]);
}
```

EXTRACTING THE MINIMUM

- Delete_Min by removing the top element (first element in the array).
- This leaves a hole in the array.
- This can be filled by moving the element from the *right-most* leaf (sitting in the *n*th position of the array) into the first position.
- This may violate the heap property.
- We need to move the dissatisfied element *bubbles down* the heap until it dominates all its children, perhaps by becoming a leaf node and ceasing to have any.
- This percolate-down operation is also called *heapify*, because it merges two heaps (the subtrees below the original root) with a new key.

```
pq_young_child(int n)
{
        return(2 * n);
}
```

```
item_type extract_min(priority_queue *q)
{
        int min = -1;                    /* minimum value */

        if (q->n <= 0) printf("Warning: empty priority queue.\n");
        else {
                min = q->q[1];

                q->q[1] = q->q[ q->n ];
                q->n = q->n - 1;
                bubble_down(q,1);
        }

        return(min);
}
```

We will reach a leaf after $\lfloor \lg n \rfloor$ bubble_down steps, each constant time. Thus root deletion is completed in $O(\log n)$ time.

BUBBLE DOWN IMPLEMENTATION

```
pq_young_child(int n)
{
        return(2 * n);
}
```

```
bubble_down(priority_queue *q, int p)
{
    int c;                  /* child index */
    int i;                  /* counter */
    int min_index;          /* index of lightest child */

    c = pq_young_child(p);
    min_index = p;

    for (i=0; i<=1; i++){
            if ((c+i) <= q->n) {
                    if (q->q[min_index] > q->q[c+i])  min_index = c+i;
            }
     }
     if (min_index ! = p) {
            pq_swap(q,p,min_index);
            bubble_down(q, min_index);
      }
}
```

BUILDING HEAP FASTER

- Robert Floyd found a better way to build a heap, using *heapify*.
- Given two heaps and a fresh element, they can be merged into one by making the new one the root and bubbling down.

```
make_heap(priority_queue *q, item_type s[], int n)
{
        int i;                          /* counter */

        q->n = n;
        for (i=0; i<n; i++) q->q[i+1] = s[i];

        for (i=q->n; i>=1; i--) bubble_down(q,i);
}
```

Worst case running time analysis of $O(n \log n)$. Is this tight?

IS OUR ANALYSIS TIGHT?

- "Are we doing a careful analysis? Might our algorithm be faster than it seems?"
- Doing at most x operations of at most y time each takes total time O(xy).
- However, if we overestimate too much, our bound may not be as tight as it should be!

TIGHTER ANALYSIS OF BUILD HEAP WITH HEAPIFY

- The identify for the sum of a geometric series is

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

- If we take the derivative of both sides, . . .

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

- Multiplying both sides of the equation by x gives:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

- Substituting x = 1/2 gives a sum of 2, so Build-heap uses at most 2n comparisons and thus linear time.

HEAPSORT

Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

```
heapsort(item_type s[], int n)
{
        int i;                  /* counters */
        priority_queue q;       /* heap for heapsort */

        make_heap(&q,s,n);

        for (i=0; i<n; i++)
                s[i] = extract_min(&q);
}
```

Exchanging the min element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm. Why is it not $O(n)$?