31251 – Data Structures and Algorithms

Week 7, Autumn 2020

Xianzhi Wang

This time, in 31251 DSA:

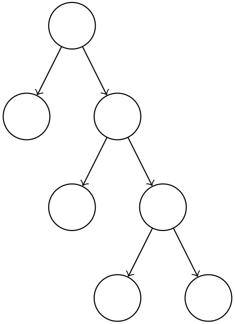
- Heaps
- Sorting Algorithms

Heaps

Another Use for Binary Trees

We have seen Binary Search Trees:

- Keeps things in a complete order.
- Mostly easier to maintain the ordering than an array or linked list ($O(\log n)$ insertion and search).
- May not perform well in all case (e.g., the tree on the right).



Alternatives to BSTs

We can ameliorate the worst-case behavior with more a complicated data structure (AVL Tree, Red Black Tree, etc.).

What if we don't really need a complete order?

Heaps

A Heap uses a binary tree to maintain a weak ordering:

- In a min-heap, the value stored at a vertex is smaller than the value at both its children.
- In a max-heap, the value stored at a vertex is greater than the value at both its children.

This property then holds for the entire subtree under any given vertex.
This means the thing at the root is the min/max.

Why is this useful?

Why not just keep everything fully ordered?

- Sometimes all we need is the min or max element.
- It can be done with less effort than something like an AVL-tree.
- Time complxity:
  - Look up: $O(1)$
  - Insert: $O(\log n)$
  - Remove: $O(\log n)$

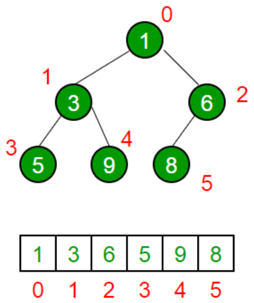
How do we do this?

Remember the array embedding of a binary tree?

- It's perfect for a heap!
- We know exactly where the last leaf/next available leaf is (it's just the next spot in the array).
- We can take an array and turn it into a heap without using any extra space!

What do we need to make this work?

- Keep track of next available leaf.
- Maintain order upon insert/remove.
- Restrict removal to the root only.



Reference: `https://www.geeksforgeeks.org/binary-heap/`

Insertion

1. Insert new element at next available leaf.
2. Keep swapping with the parent to maintain the ordering.

```
void insert(int i) {
    if (next_leaf >= capacity){
        double_size();
    }
    array[next_leaf++] = i;
    bubble_up(next_leaf-1);
}

void bubble_up(int p){
    if (has_parent(p) && array[p]<array[parent(p)]) {
        swap(p, parent(p));
        bubble_up(parent(p));
    }
}
```

Removal

1. Move last leaf value to root, and delete the last leaf.
2. Swap the root value down to maintain ordering.

```
int remove() {
    if (next_leaf > 0){
        int temp = array[0];
        array[0] = array[--next_leaf];
        bubble_down(0);
        return temp;
    }
    return -1;
}
void bubble_down(int p){
    if (has_left(p)) {
        int the_smaller = (has_right(p) && array[right(p)] < array[left(p)])
                                    ? right(p) : left(p);
        if (array[p] > array[the_smaller]) {
            swap(p, the_smaller);
            bubble_down(the_smaller);
        }
    }
}
```

Sorting

Sorting as a Problem

- Sorting is one of the basic algorithmic problems.
    - Given a list of elements, we want a permutation that's in order according to some comparator.
    - E.g., in ascending, descending order
- Despite being an obvious problem, sorting efficiently is not necessarily as easy.
- It even still attracts attention as a problem to solve!

Comparison-based Sorting

Comparison-based Sorting

- Without knowing anything special about the input, we must compare elements to each other.
  - Hence the name "Comparison-based Sorting".

- If we are limited to comparing elements to each other, we have a definite lower bound on performance:
  1. For a list of n distinct items, there are $n!$ arrangments, only one of which is sorted.
  2. If the algorithm takes $f(n)$ steps, and each step distinguishes two cases, it can distinguish at most $2^{f(n)}$ cases.
  3. So we need $2^{f(n)} \geq n!$, then doing some algebra, $f(n) \geq \log(n!) \Rightarrow f(n) \geq n \log n - n \log_2 e + O(\log n) \Rightarrow$ $f(n) \in \Omega(n \log n)$

Some Examples

Many of the algorithms you may know are comparison-based sorts:

- Bubble sort.
- Insertion sort.
- Selection sort.
- Merge sort.
- Quick sort.
- Heap sort.

Bubble Sort

```
bubbleSort(int a[]) {
//assume array is length n
    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < n-1; i++){
            if (a[i] > a[i+1]){
            swap(a[i], a[i+1]);
            swapped = true;
            }
        }
    }while(swapped);
}
```

Bubble Sort

- Easy to code yet bad at almost everything else.
- Running time:
  - Best case time: $O(n)$.
  - Average case: $O(n^2)$.
  - Worst case: $O(n^2)$.
- Running Space:
  - Only $O(1)$ extra space need though!

Insertion Sort

```
insertionSort(int a[]){
    for (int i = 1; i < n; i++){
        int x = a[i];
        int pos = i-1;
        while (pos >= 0 && a[pos] > x){
           a[pos + 1] = a[pos];
           pos--;
        }
        a[pos + 1] = x;
    }
}
```

Insertion Sort

- Same complexity profile as Bubble sort:
  - $O(n^2)$ worst case.
  - $O(1)$ extra space.
- Turns out to be observably faster.
  - Minimizing the number of comparisons.
- This is actually a Divide-and-Conquer algorithm.
  - Just a really bad use of one.

Selection Sort

```
selectionSort(int a[]) {
    for (int i = 0; i < n-1; ++i) {
        int min_pos = i;
        for (int j = i + 1; j < n; ++j){
            if (a[j] < a[min_pos]){
                min_pos = j;
            }
        }
        swap(a[i], a[min_pos]);
    }
}
```

Selection Sort

- $O(n^2)$ time and $O(1)$ space again, between Bubble Sort and Insertion Sort (usually).
- Sort of the inverse of Insertion Sort - finds the next smallest thing, rather than the place for the next thing.
- Pretty simple to implement.

Merge Sort

- Also a divide and conquer algorithm.
- Very fast.
- Also more complicated code.

Merge Sort, part 1

```
int[] mergesort(int a[]){
    if (a.length = 1) return a;

    int left[] = a[0..n/2];
    int right[] = a[n/2+1 ..n-1];

    return merge(mergsort(left), mergesort(right));
}
```

Merge Sort, part 2

```
int[] merge(int left[], int right[]){
    int merged[left.lenght + right.length];
    int i = 0;
    int lpos = 0;
    int rpos = 0;
    while (i < merged.length){
        if (left[lpos] < right[rpos]){
            merged[i] = left[lpos++];
        } else {
            merged[i] = right[rpos++];
        }
        i++;
    }
    return merged;
}
```

Merge Sort

- We finally achieve $O(n \log n)$ worst-case running time.
- Uses $O(n)$ extra space though.
    - But by complicating the code, we can make this $O(1)$ space!
- Also happens to be highly parallelisable.

Quick Sort

- This is actually a decent sorting algorithm!
- It's a Divide-and-Conquer algorithm.
- But it has more complex code (it's that trade-off again!).

Quick Sort, part 1

```
quicksort(int a[], int low, int high){
    if (low  < high){
        p = partition(a, low, high);
        quicksort(a, low, p);
        quicksort(a, p+1, high);
    }
}
```

Quick Sort, part 2

```
int partition(int a[], int low, int high){
    int pivot = a[high];
    int part = low - 1;
    for (int j = low; j < high; j++){
        if (a[j] < pivot){
            part++;
            swap(a[part], a[j]);
        }
    }
    swap(a[part+1], a[high]);
    return part+1;
}
```

Quick Sort

- $O(n \log n)$ average-case time!
- But still $O(n^2)$ worst case...
  - It all depends on the pivot value in partion. In the worst case, this is just Insertion sort.
- Typically fast in practice. Lots of heuristics for picking a pivot value well.

Heap Sort

- Uses a Heap data structure.
- $O(n \log n)$ worst-case time, and $O(1)$ extra space usage!
- Just turn the array into a min-heap, and remove things until it's empty! (Not so parallelisable)

Non-Comparison Based Sorting (optional)

Sorting without directly comparing elements

- If we know something more about the data, we can sometimes do better.
- For example, if the data has hierarchical structure.
  - Like digits in a number, characters in a string for lexicographic ordering...

Bucket Sort

1. Split your data range into sub-ranges (buckets).
2. Go through the array and put each element in the proper bucket.
3. Recursively sort the buckets (using bucket sort, or something else).
4. Merge the buckets back together.

Bucket Sort

- If the bucket's range is 1 value, this becomes Counting Sort.
- If you use two buckets, it's a version of Quick Sort.
- Worst cast time: $O(n^2)$, but average case: $O(n + k)$ where k is the number of buckets.
    - So if $k \approx n$... $O(n)$ sorting! (If everything comes out right...)
- $O(nk)$ space though...
- If you do it with the digits of the number, you get...

MSD Radix Sort

1. Start with the most significant digit, and work down to the least significant.
2. For the current digit split the elements into b buckets, where b is the base of the number system you are using.
3. Recursively sort each bucket.
4. Merge the buckets.

Radix Sort

- Suuuuuper fast for small integers.
- Worst case complexity is $O(dn)$ where d is the number of digits in your numbers.
- If you have a lot of different numbers, $d > \log n$ (maybe a loooot bigger), so we don't beat comparison based sorting really, but...
- If all your numbers are small (maybe many repeats if you have a lot of numbers), this is $O(n)$.
- It's space is "good" too $O(d + n)$.

Hardware Based Sorting (optional)

Sorting Networks

- One last little example – we can trade off processors for speed.
- If we don't mind using lots of "CPUs", we can sort really fast in parallel.
- These are called sorting networks, their speed is governed by their depth – $O((\log n)^2)$ depth is achievable, but you need polynomially many "CPUs".