

CS 240 — Data Structures and Data Management

Kevin James

Spring 2014

Contents

1	Algorithms	2
2	Data Types	4
2.1	Stacks	4
2.1.1	Dynamic Arrays	5
2.2	Queues	5
2.2.1	Priority Queues	5
2.3	Heaps	5
2.3.1	Heap Insert	6
2.3.2	Heap Delete	7
2.3.3	Heap Creation	7
3	Sorting	8
3.1	PQ Sort	9
3.2	Heap Sort	9
3.3	Quick Sort	9
3.3.1	Selecting	9
3.3.2	Sorting	11
3.4	Counting Sort	11
3.5	Radix Sort	11
4	Dictionary	12
5	Search Trees	12
5.1	Binary Search Trees	12
5.2	AVL Trees	12
5.3	Red-Black Trees	14
5.4	Leftist Trees	14
5.5	AA-Trees	14
5.6	2-3 Trees	14

1 Algorithms

An **algorithm** is a step-by-step process for carrying out a set of operations given an arbitrary problem instance. An algorithm **solves** a problem if, for every instance of the problem, the algorithm finds a valid solution in finite time.

A **program** is an implementation of an algorithm using a specified programming language.

For each problem we can have several algorithms and for each algorithm we can have several programs (implementations).

In practice, given a problem:

1. Design an algorithm.
2. Assess the correctness of that algorithm.
3. If the algorithm is acceptable, implement it. Otherwise, return to step 1.

When determining the efficiency of algorithms, we tend to be primarily concerned with either the runtime or the memory requirements. In this course, we will focus mostly on the runtime.

To perform runtime analysis, we may simply implement the algorithm and use some method to determine the end-to-end time of the program. Unfortunately, this approach has many variables: test system, programming language, programmer skill, compiler choice, input selection, This, of course, makes manual implementation a bad approach.

An idealized implementation uses a **Random Access Machine (RAM)** model. RAM systems have constant time access to memory locations and constant time primitive operations, thus the running time is determinable (as the number of memory operations plus the number of primitive operations).

We can also generally use **order notation** to compare multiple algorithms. For the most part, we compare assuming n is very large, since for small values of n the runtime will be miniscule regardless of algorithm.

A timing function is a function T_A such that $T_A : \{Q\} \rightarrow \mathbb{R} > 0$. We denote the runtime of a function as $T(f(x))$, for example: $T(3 \times 4)$ may be equal to $0.8ns = 8ops$. The return value is the number of operations required in the worst-case scenario.

Example: given $T_A(n) = 1000000n + 2000000000$ and $T_B(n) = 0.01n^2$, which is ‘better’? For $n < 100000000$, algorithm B is better. Since we only care about large inputs, though, we say A is better overall.

Example 1.1. Prove that $2010n^2 + 1388 = O(n^3)$.

Proof. We want to prove that for all positive $c \in \mathbb{Z}$, there exists some n_0 such that for all $n > n_0$, $2010n^2 + 1388 \leq cn^3$. So for $n > 1388$ we have $2010n^2 + 1388 \leq 2011n^2 \leq cn^3$ which in turn gives us $2011n^2 \leq cn^3 \iff 2011 \leq cn$. Then this holds given $n_0 = 2010$. \square

Definition 1.1. $f(n) = O(g(n))$ if there exists a positive real number c and an integer $n_0 > 0$ such that $\forall n \geq n_0$, $f(n) \leq cg(n)$.

More concretely, we can say that $f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$. It's also worth noting that order notation is transitive (e.g. $f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$).

We use five different symbols to denote order notation:

- o denotes a function *always less* than a given order
- O denotes a function *less than or equal* to a given order
- Θ denotes a function *exactly equal* to a given order
- Ω denotes a function *greater than or equal* to a given order
- ω denotes a function *always greater* than a given order

More concretely, we have the formulae

Definition 1.2 (o). $f(n) = o(g(n))$ if for all $c > 0$ there exists a constant $n_0 > 0$ such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Definition 1.3 (O). $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Definition 1.4 (Θ). $f(n) = \Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Definition 1.5 (Ω). $f(n) = \Omega(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

Definition 1.6 (ω). $f(n) = \omega(g(n))$ if for all $c > 0$ there exists a constant $n_0 > 0$ such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$.

Example 1.2. For the psuedo-function

```
function(n):
    sum = 0
    for i=1 to n:
        for j=i to n:
            sum = sum + (i-j)^2
        sum = sum^2
    return sum
```

we find the order equation

$$\begin{aligned}
 &= \Theta(1) + \sum_{i=1}^n \sum_{j=i}^n \Theta(1) + \Theta(1) \\
 &= \Theta(1) \sum_{i=1}^n \sum_{j=1}^n 1 \\
 &= \Theta(1) \sum_{i=1}^n (n - i + 1) \\
 &= \Theta(1) \left(\sum_{i=1}^n n - \sum_{i=1}^n i + \sum_{i=1}^n 1 \right) \\
 &= \Theta(1) (n^2 + i^n + n) \\
 &= \Theta(n^2) + \Theta(i^n) + \Theta(n)
 \end{aligned}$$

$$= \Theta(i^n)$$

Example 1.3. For the psuedo-function

```
function(n):
    sum = 0
    for i=1 to n:
        j = i
        while j >= 1:
            sum = sum + i/j
            j = j/2
    return sum
```

we find the order equation

$$\begin{aligned} \sum_{i=1}^n \sum_{j=1}^{\log_2 i} c &= \sum_{i=1}^n (c \log_2 i) \\ &= c(\log 1 + \log 2 + \log 3 + \dots + \log n) \\ \text{all } n \text{ of our terms are below } \log n &\quad \text{half of our } n \text{ terms are above } \frac{n}{2} \\ &= O(n \log n) \quad = \Omega\left(\frac{n}{2} \log \frac{n}{2}\right) \\ &\quad = \Omega(n \log n) \\ &= \Theta(n \log n) \end{aligned}$$

Since it is too hard to describe the runtime for every possible input, we decide to only describe the worst case behaviour. The worst case gives us a *guarantee* for required completion time and tends to describe most cases.

More formally, we take

$$\max_I \{T_A(I)\}$$

2 Data Types

2.1 Stacks

A **stack** is a collection of items which supports the operations **push** (insert an item), **pop** (remove the most recently inserted item), **peek** (view the last item), and **isEmpty**.

The most common ways to implement a stack are as a

- **linked list**: a pointer to the top of the stack is maintained and is moved whenever an item is inserted or removed, or an
- **array**: the last item is easy to access, though sometimes resizing the array will be necessary

2.1.1 Dynamic Arrays

Linked lists support $O(1)$ insertion and deletion, $O(n)$ accessing. Arrays are vice-versa. **Dynamic arrays** offer a compromise: $O(1)$ for both, but can only insert or delete from the end of the list.

2.2 Queues

A **queue** is a data structure where you **insert** at the end of the list and **dequeue** from the front. Implementations are the same as a stack: either an array or a linked list may be used, though a pointer or reference to the first item is necessary.

2.2.1 Priority Queues

A **priority queue** is similar to a queue, but each element has a priority attached to it (a numerical “score”). The elements are dequeue’d in order of priority. A priority queue supports

- **insert**(x , p): inserts an element x with priority p
- **delete**(): deletes the element with the highest priority p
- **peek**(): views the top element

One of the most useful applications of priority queues is for sorting: by inserting all elements from an array and then deleting them, we will have the elements returned in a correctly sorted order. Then we simply need to examine the runtime efficiency of **insert** and **delete** to determine the speed of our sorting algorithm.

There are two common implementations of priority queues: unsorted arrays and heaps.

In an **unsorted array**, insertion takes $O(1)$ time, since the element is simply placed at the end of the array. Deletion must walk the array, then replace the deleted (read: smallest it has seen as it walked) with the last element in the array. Thus, this is $O(n) + O(1) = O(n)$ time. Sorting, then, takes $O(n^2)$ time.

A better method is to use heaps.

2.3 Heaps

A **heap** is a *complete* binary tree with the *heap property*. To have a complete binary tree, the tree must be structured such that for each node in the tree that node must have zero or two children, unless that node is the rightmost leaf in the bottom level (in which case it may have only one). Furthermore, deeper leaves must be leftmost in the tree and all the leaves in the bottom two levels must be consecutive.

The **heap property** is the principle such that all nodes have a lower priority than that of their parent. In the case of multiple elements with the same priority, arbitrary element positioning is possible.

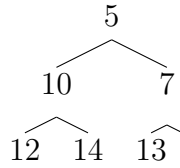


Figure 1: A min-PQ tree that satisfies the heap properties

This is not a binary search tree, since the larger element does not need to be on the right. In fact, there is no relationship of any kind with a node's siblings—the only relationship is with its parent.

When we remove an item from a heap, we must replace that item with the current lowest element. This will guarantee that the tree remains in a complete form (that all holes are filled). After replacing it with this element, the newly moved element must be sorted: recursively compare its priority with that of its children; if any children have a higher priority, it should swap with its parent. Otherwise, stop recursing.

When inserting, we use the opposite process: the new element is placed at the base of the tree, then it is recursively compared and potentially swapped with its parent until it is in a valid position.

Note that we do not implement tree-like data structures as such, but rather as arrays. The array follows a format such that the tree represented in figure 1 would be placed in an array as $[5, 10, 7, 12, 14, 13]$. More concretely: the parent of a node at index n is at index $\lfloor \frac{n-1}{2} \rfloor$. Its children are located at indices $2n + 1$ and $2n + 2$.

Heap functions have worst case running times bounded as follows

- Heap insertion: $\Theta(\log n)$
- Heap deletion: $\Theta(\log n)$
- Top-down heap creation: $\Theta(n \log n)$
- Bottom-up heap creation: $O(n)$

2.3.1 Heap Insert

The item to-be-inserted must be placed in the only possible location to preserve integrity: i.e. the bottom level in the first available position. However, this may violate heap ordering procedure; we must perform bottom-up heap swaps until the ordering is satisfied.

To perform a bottom-up heap swap, we must

- compare node with parent
- if the node is larger than the parent
 - swap
 - recurse
- else
 - quit

These swaps can be performed in $\Theta(1)$ time and since the maximum number of swaps is equal to the height of the heap we have an overall time complexity of $\Theta(\log n)$. This worst case would occur when the item to-be-inserted is larger than anything else on the heap, since it would need to swap all the way to the top.

2.3.2 Heap Delete

The item to-be-deleted can simply be removed from our heap; however, we must replace this vacated position with the lowest item in our heap. This satisfies the structural integrity of the heap. Afterward, we must follow top-down heap swapping in order to satisfy ordering integrity.

To perform a top-down heap swap, we must

- compare node with children
- if the node is smaller than at least one child
 - swap with the largest child
 - recurse
- else
 - quit

Again, these swaps can be performed in $\Theta(1)$ time and we can perform no more than $\lfloor \log n \rfloor$ swaps. As such, we have worst case time complexity $\Theta(\log n)$ when the deleted item is at the top of the heap and the replacement item is the smallest in the heap.

2.3.3 Heap Creation

Given an array in no particular order, our goal is to adjust the ordering such that we have a valid heap.

The obvious approach is to call the heap insertion method once for each item in our input array. This is the **top-down** approach to heap creation.

This can generally be done in place, as we can treat the input array as a structurally valid but orderingly incorrect binary tree. As such, each insertion operation simply ‘ignores’ values to its right. Since we are calling out $\Theta(\log n)$ insertion method n times, we see that top-down heap creation has runtime complexity $\Theta(n \log n)$.

More formally, we see we have an upper bound such that the cost of the swaps is proportional to the depth of the node (i.e. which level it is on) and the number of nodes to process. This depth is given by $O(\log n)$ and the number of nodes by n , so we have $O(n \log n)$ as our upper bound. Our lower bound is given as such: the worst case occurs when the input array is sorted in ascending order (i.e. requires we perform the maximal number of swaps for each insertion). In this case, we would be required to swap $\lfloor \log n \rfloor$ times for each of our n insertions.

Lemma 2.1. *At least half of the nodes in a heap are in the last two levels (have depth $d \geq h - 1$)*

Based on lemma 2.1, we can ignore all levels above the bottom two when computing the lower bound. The worst case number of swaps at level $h - 1$ or h is at least $h - 1$, thus the overall number of swaps $s \geq \frac{n}{2}(h - 1)$ which is $\Omega(n \log n)$.

We can also creation a heap in the **bottom-up** fashion as such: the entire array is transformed into a heap in one iteration, by finding the elements from largest to smallest and inserting them in that order.

We begin in the same way as for top-down creation, by placing our array into a incomplete binary tree (or associating our already-made array with a binary tree). Now, we start with the element in the last position and perform a top-down swap operation for each element in the array (recurring backwards).

This is very similar to the top-down heap creation method, but has different boundaries.

Since our swaps can now be made in the opposite direction, our lowest level of the tree no longer needs to swap and our second lowest level can only possible swap once. As such, over half of our tree has a maximal swapping runtime of $O(1)$ and the remainder of the tree has $O(\log n)$, whereas the top-down approach uses the constant boundary $O(\log n)$.

More concretely, the total number of swaps we can perform is equal to

$$\begin{aligned} \sum_{i=0}^h i 2^{h-i} &< n \sum_{i=0}^{\infty} \frac{i}{2^i} \\ &< 2n \end{aligned}$$

and thus we have the upper bound of $O(n)$.

The lower bound is at least $\Omega(n)$ (since we must run our heap creation on n elements), thus we have an overall runtime complexity of $\Theta(n)$.

3 Sorting

There exist many sorting algorithms:

Algorithm	Complexity
Selection Sort	$\Theta(n^2)$
Insertion Sort	$\Theta(n^2)$
Merge Sort	$\Theta(n \log n)$
Heap Sort	$\Theta(n \log n)$
Quick Sort	$\Theta(n \log n)$

For a comparison-based algorithm, we have a lower bound of $\Omega(n \log n)$. For non-comparison-based algorithms, we can achieve $O(n)$.

3.1 PQ Sort

As mentioned above, we can sort data using a priority queue. Both **insert** and **delete** take $O(\log n)$ time, as does recursively sorting any newly moved elements. Thus, for sorting n elements we see that PQ sort will take $O(n \log n)$ time, which is the same as mergesort.

3.2 Heap Sort

Heap sort is a specialized PQ sort.

The heap sort algorithm is

```
heapSort(A, n):  
    H = new Heap(n)  
    for i in range(0, n - 1):  
        heapInsert(H, A[i])  
    for i in range(0, n - 1):  
        A[n-1 - i] = heapDeleteMax(H, n-1)
```

or given our “heapify” algorithm

```
heapSort(A, n):  
    H = heapify(A, n)  
    for i in range(0, n - 1):  
        A[n-1 - i] = heapDeleteMax(H, n-1)
```

which gives us a complexity of

$$\begin{aligned} O(n) + \sum_{i=0}^{n-1} \Theta(\log n) &= O(n) + \Theta(n \log n) \\ &= \Theta(n \log n) \end{aligned}$$

3.3 Quick Sort

The quick sort algorithm works by having one function which selects the optimal pivot point, moving elements such that all items left of the pivot are smaller and vice-versa, then recursing on each side.

3.3.1 Selecting

Given an array A , we want to find the k th smallest (or largest) element. We do this by recursively taking the first element (the **pivot**), then sorting the elements such that any elements smaller than the pivot are on its left (and vice versa). This continues until there are fewer than k elements on the requested side.

```

quickSelect(A, k):
    p = randomPivot(A)
    i = partition(A, p)
    if k < i:
        quickSelect(A[0, i-1], k)
    elif k > i:
        quickSelect(A[i+1, n-1], k - i - 1)
    else:
        return p

```

When selecting, it is only required that we recurse on one side of the pivot (i.e. the side which contains the element we are searching for).

The partition function is defined thusly

```

partition(A, p):
    swap(A[0], A[p])
    i = 1
    j = n - 1
    while true:
        while i < n and A[i] < A[0]:
            i = i + 1
        while j >= 1 and A[j] > A[0]:
            j = j - 1
        if j < i:
            break
        else:
            swap(A[i], A[j])

```

In the *worst case*, we have $\Theta(n^2)$ time. This would occur if the selected pivot is always $\max(A)$ and we are searching for the smallest element, for example in the array $[n, n-1, n-2, \dots, 3, 2, 1]$.

In the *average case*, the probability of selecting a “good” pivot is 50%. After r recursive calls, this means the array was halved approximately $\frac{r}{2}$ times. After $4 \log n$ calls, we must be left with an array of size 1. Thus we have

$$\begin{aligned}
 f(n) &\leq \frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn \\
 &\leq 2cn + 2cn\left(\frac{3n}{4}\right) + 2c\left(\frac{9n}{16}\right) + \dots + T(1) \\
 &\leq \frac{1}{2}\left[\frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn\right] + \frac{1}{2}\left[\frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}T\left(\frac{9n}{16}\right) + c\frac{3n}{4}\right] \\
 &\leq \frac{1}{4}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}cn + \frac{1}{4}T\left(\frac{9n}{16}\right) + c\frac{3n}{8} + cn \\
 &\leq T(1) + 2cn \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i \\
 &= \Theta(n)
 \end{aligned}$$

3.3.2 Sorting

The general quick sort algorithm is

```
quickSort(A, n):  
    if n == 0:  
        return  
    p = randomPivot(A)  
    i = partition(A, p)  
    quickSort(A[0, i-1])  
    quickSort(A[i+1, n])
```

The worst case is $\Theta(n^2)$ and the best case is $\Theta(n \log n)$. Fortunately, the worst case is extremely unlikely and the average case is also $\Theta(n \log n)$.

3.4 Counting Sort

Assume each element e in an array satisfies $0 \leq e \leq k - 1$ for some known value k . Then we can use the **counting sort** algorithm on this array, implemented as follows:

```
countingSort(A[1..n], k):  
    C = [0]*k  
    for i = 0 to n-1:  
        C[A[i]]++  
    for i = 1 to k-1:  
        C[i] += C[i-1]  
    B = copy(A)  
    for i = reverse(0 to n-1):  
        C[B[i]]--  
        A[C[B[i]]] = B[i]
```

Counting sort has time complexity $\Theta(n+k)$, space complexity $\Theta(n+k)$, and is a **stable** algorithm (equal items are not reordered).

3.5 Radix Sort

Assume we have a set of (probably) unique integers. Then we can use the **radix sort** algorithm on this set, implemented as follows:

```
radixSort(A[1..n], d, k):  
    for i = 0 to d - 1:  
        countingSort(A, k) //with each x_i as the key
```

We can perform radix-sort from either the left or right sides (LSD vs. MSD). The MSD-Radix sort, though, can only be performed if all integers have the same number of digits (or are padded with zeroes).

Radix sort has time complexity $\Theta(d(n+k))$ and space complexity $\Theta(n+k)$.

4 Dictionary

A **dictionary** is a collection of items, each of which contain a **key** and some **data** and is referred to as a **key-value pair**. Keys can be compared and are (typically) unique.

We define the following functions for a dictionary:

- `search(k)`
- `insert(k, v)`
- `delete(k)`
- and optionally: `join`, `isEmpty`, `size`, ...

We can implement a dictionary in several ways

- as an unordered array: has $\Theta(n)$ search and deletion time and $\Theta(1)$ insertion time
- as a sorted array: has $\Theta(\log n)$ search time and $\Theta(n)$ insertion and deletion time
- as a linked list: has $\Theta(n)$ search and deletion time and $\Theta(1)$ insertion time

5 Search Trees

5.1 Binary Search Trees

A **BST** is either empty or contains a KVP, left-child BST, and right-child BST. Every key k_L in $T.left$ is less than the root key, every key k_R in $T.right$ is greater. These properties are recursive.

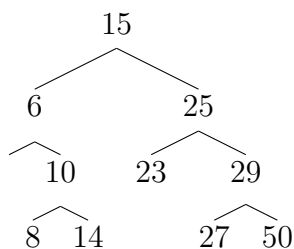


Figure 2: A sample binary search tree

`search`, `insert` and `delete` all have time complexities $\Theta(h)$, where h is the height of the tree. h has a best-case value of $\log n$ and a worst-case value of n .

5.2 AVL Trees

An **AVL tree** is a BST with an additional structural property: the heights of the left and right subtree must differ by no more than one (where an empty tree is defined to have a height of -1).

At each non-empty node, we store $\text{height}(R) - \text{height}(L) \in \{-1, 0, 1\}$: -1 means the tree is left-heavy, 1 means the tree is right-heavy, and 0 means the tree is balanced. We could store the actual height, but storing the balances is simpler and more convenient.

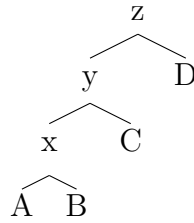


Figure 3: An inbalanced AVL tree.

An AVL tree can perform **rotations** to rebalance itself.

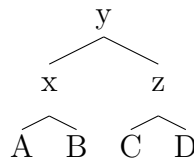


Figure 4: A balanced AVL tree (after rotating right).

This function can be represented as

```
rotate_right(T):
    newroot = T.left
    T.left = newroot.right
    newroot.right = T
    return newroot
```

```
rotate_left(T):
    newroot = T.right
    T.right = newroot.left
    newroot.left = T
```

We can also perform **double rotations**, which may be either two of the same rotation operation or one of each.

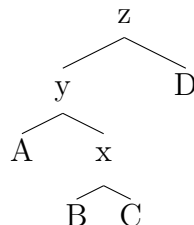


Figure 5: An inbalanced AVL tree requiring two rotations (left then right).

The **search** operation is the same as a BST and is in $\Theta(h)$ time. The **insert** uses rotations and will thus be $\Theta(h)$. Finally, the **delete** operation swaps then applies fixes (rotations) and is thus $\Theta(h)$.

We define $N(h)$ to be the least number of nodes in an AVL with height h . One subtree must have height of at least $h - 1$, the other at least $h - 2$. Then we have

$$N(h) = \begin{cases} 1 + N(h - 1) + N(h - 2) & h \geq 1 \\ 1 & h = 0 \\ 0 & h = -1 \end{cases}$$

5.3 Red-Black Trees

5.4 Leftist Trees

5.5 AA-Trees

5.6 2-3 Trees

A **2-3 tree** is a BST with additional structural properties:

- every node either contains one KVP and two children or two KVPs and three children.
- all the leaves (nodes without children) are at the same level

Searching through a 1-node is the same as a BST, but we must examine both keys in a 2-node to determine the most accurate path.

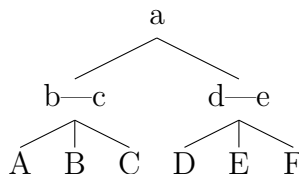


Figure 6: A balanced 2-3 tree.

To insert into a 2-3 tree, we first search to find the leaf where the key belongs. If the leaf has a single KVP, we simply add the new key and make a 3-node. Otherwise, we sort the keys, split the left-most and right-most into two 1-nodes, and recursively insert the middle key into the parent (along with the link).

To delete, we first swap the KVP with its successor so we are always deleting from a leaf. If the target node is a 2-node, we simply delete the key. Otherwise, if the target node has an immediate 2-node sibling we perform a **transfer** (put the intermediate KVP in the parent between the two nodes into the target node and replace it with the adjacent KVP in the sibling node). Otherwise, we **merge** the target node and a 1-node sibling by removing the target node and recursively deleting the intermediate KVP in the parent and adding it to the sibling node.

5.7 B Trees

A 2-3 tree is a specific type of **B tree**.