Heap

Two types of binary heaps

- In both kinds, the values in the nodes satisfy a ***heap property***,
- In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root,
  - $A[\text{PARENT}(i)] \geq A[i]$
  - the value of a node is at most the value of its parent.
  - Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.
- ***min-heap property*** is that for every node i other than the root,
  - $A[\text{PARENT}(i)] \leq A[i]$.
- For the heapsort algorithm, we use max-heaps

View a heap as a tree

- The ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf,
- The height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$
- The basic operations on heaps take (lgn)
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in (lgn)
- MAX-HEAPIFY procedure, which runs in (lgn)
- BUILD-MAX-HEAP → linear time
- HEAPSORT → nlgn

Heap Building

- Top Down
  - Induction base: $A[1]$ is a heap
  - Induction hypothesis: the array $A[1, \ldots, i]$ is a heap
  - How to add $A[i+1]$ to the heap $A[1, \ldots, i]$?
    Compare to parent $A[\lfloor (i+1)/2 \rfloor]$
    Exchange if parent is bigger
    Until new parent is smaller
  - Now $A[1, \ldots, i, i+1]$ is a heap.
    $\lceil \log_2(i) \rceil$ possible comparisons and exchanges.
  - Time: $\sum_{i=2}^{n} \log_2 i = O(n \log n)$
- Bottom up
  - convert an array AŒ1 : : n, where n D A: *length*, into a max-heap.
  - the elements in the subarray AŒ.bn=2cC1/ : : n are all leaves of the tree, and so each is a 1-element heap to begin with
  - BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

  - Each call to MAX-HEAPIFY costs lgn time, and BUILD- MAX-HEAP makes n such calls. Thus, the running time is nlgn (upper bound)
    - Bounded by O(h)

Sorting

Bucket sort (Mailroom sort)

- allocate a sufficient number of boxes – buckets – and put each element in the corresponding bucket.
- Works very well only for elements from a small, simple range that is known in advance
  - e.g. sorting letters by state (by province)
  - e.g. sorting letters by zip code – we need 26**3 · 10**3 buckets
- Input $x_1, x_2, \ldots, x_n$, $1 \le x_i \le m$ and x_i are distinct integers. Allocate m buckets. For each i, we put x_i in the bucket corresponding to its value. Finally, we scan the buckets in order and collect all elements.
- Time and space complexity:
  - time: $O(n + m)$: n for sort n elements and m for final scan
  - space: $O(m)$: 1 unit for each bucket
  - If $m = O(n)$ then this is linear sorting

Radix Sort

- Natural extension of bucket sort.
- We want to reduce the number of buckets (we need more passes).
- We use induction to show the algorithm.
  - Induction Hypothesis: We know how to sort elements of $< k$ digits
  - Given elements with k digits, we first ignore the most significant digit (left-most digit) and sort the elements according to the rest of the digits by induction!
  - Scan all the elements again and use bucket sort on the most significant digit with d buckets.
  - Collect all the buckets in order.
- Why does it work?
  - Two elements that are put in different buckets in the LAST step are in the right order
    - do not need induction
    - most significant digits determine the order
  - Two elements having the same most significant digit
    - By induction, they are in right order before the last step.
    - Make sure that elements put in the same bucket REMAIN in the same order
      - using a queue for each bucket
      - appending the d queues at the end of a stage to form one global queue of all elements
- Time complexity: $O(kn)$
  - Initialize the queues: $O(d)$
  - Put n elements into buckets: $O(n)$
  - Append d queues: $O(d)$
  - Therefore for one pass, total time is $O(n)$
- Counting sort can be used to implement Radix Sort.

Counting Sorting

- Counting sort assumes that each of the n input is an integer in the range of 0 to k, for some k.
- The input is in A[1..n] and the output will be in B[1..n].
- We also use an array C[0..k] for temporary space.

```
Counting-Sort(A, B, k)
1.  for i = 0 to k
2.      do C[i] = 0
3.  for j = 1 to n
4.      do C[A[j]] = C[A[j]] + 1
5.  for i = 1 to k
6.      do C[i] = C[i] + C[i − 1]
7.  for j = n downto 1
8.      do B[C[A[j]]] = A[j]
8.         C[A[j]] = C[A[j]] − 1
```

- An important property of counting sort is that it is stable.
    - Integers with the same value appear in the output array exactly the same order as they do in the input array.
    - This is important in the application of counting sort.
- When k = O(n), the sort runs in Θ(n) time

Insertion Sort

- Assume we can sort $n-1$ elements, then we can find the right place for the n'th element and insert it there
- worst case:
    - data movement: $i-1$ for the i'th iteration $\Longrightarrow \Omega(n^2)$
    - comparisons: $\Omega(n\log n)$
- average case:
    - There are i positions where x (ith element) can go.
    - The probability that x belongs to any position is $1/i$.

To improve insertion sort:
Use a data structure that supports search and also insertion, for example AVL trees or red-black trees. These methods require extra space though.

Selection Sort

- find the maximum element, swap it with the last element.
- data movement: $O(n)$, 1 for each iteration
- comparisons: $O(n^2)$, i for ith largest

Insertion and Selection

Insertion	Data movement	$O(n^2)$
	Comparisons	$O(n\log n)$
Selection	Data movement	$O(n)$
	Comparisons	$O(n^2)$

To improve selection sort:
Use a data structure that supports find max and also deletions (e.g. heap sort).

Merge Sort

- The merge process can be considered as an improvement of insertion sort.
- Idea: With the time to insert one element, we can insert many elements.

- We want to insert B into A
- We scan A from the left for the right position for b_1
- We can then continue, without going back, to scan for the right position for b_2 and so on.
- *Data movement:* copy them into a temporary array. Each element moves only once.
- $O(n+m)$ time

Mergesort: Divide-and-conquer sorting

Divide by half	$O(1)$
Solve each half recursively	$2T(n/2)$
Merge two sorted halves	$O(n)$

Time: $T(n) = 2T(n/2) + O(n) \Longrightarrow O(n\log n)$

Quick Sort

- choose a pivot, x1
- use two pointers (indices) L and R
  - Initially, L points to the left end of the array and R points to the right end of the array.
  - The pointers move in opposite directions.

```
Procedure Q-Sort(X, Left, Right)
begin
    if Left < Right then
        Middle = Partition(X, Left, Right);
        Q-Sort(X, Left, Middle-1);
        Q-Sort(X, Middle + 1, Right);
end
```

- How to choose a pivot?
  - Choose a random element from the sequence is a good choice
  - If the sequence is random, we can just choose the first element
  - If we choose another element to be pivot, we can exchange it with the first element, then use our partition algorithm
- Induction proof
- Worst case
- Average case
- Space complexity
  - From the appearance of the algorithm, it seems that we do not need any extra space
  - However, recursions are implemented by using run-time stacks
    Each call: a pair of indices of the array has to be stacked.
  - There are at most $n-1$ calls, there are at most $(n-1)$ pair of indices to be stacked.
  - Space complexity: $O(n)$ (extra space)
  - If we use explicit stack, we can guarantee $O(\log n)$ extra space
- Improvement
  - Improve the selection of the pivot
    - choose a random index between L and R.

        - choose the median of x_L, x_R and $x_{(L+R)/2}$
        - (We need to do extra work, but it is worth it.)
    - Use a simple algorithm for small size.
        - e.g. when size is less than 15, use insertion sort
        - avoid problem of stacking overhead (”choose the base of induction wisely”)
    - Use explicit stacking : avoid overhead of system (run-time) stack
    - Minimize the size of the stack: always stack the larger part first (solve smaller part first).
    - Put pivot into register, for each comparison only one data movement from memory.

Heap sort

- Like selection sort, heapsort is in place
- Like mergesort, heapsort is $O(n \log(n))$
- heapsort combines the better features of the two sorting algorithms.
- fast sorting algorithm
- not quite as fast as quicksort but not much slower
- unlike quicksort, its performance is guaranteed
    - *Build Heap*
    - consider the largest element
        - Swap $A[1], A[n]$
        - $A[n]$ now has correct element
        - rearrange $A[1, \ldots n-1]$ to form a heap (push $A[1]$ down the tree).
    - Assume $A[1, \ldots i+1]$ is a heap and $A[i+2], \ldots, A[n]$ have correct elements.
        - swap $A[i+1]$ and $A[1]$
        - $A[i+1]$ has now correct element
        - rearrange $A[1, \ldots i]$ to form a heap (push $A[1]$ down the heap).
        - time: $\sum_{i=1}^{n} \log(i) = O(n \log n)$
          (time for transforming a heap to a sorted sequence.)
- Time complexity
    - Heap building
        - $2n$ comparisons
        - n data movements
    - Heap sort
        - $2\sum_{i=2}^{n} \log i$ comparisons
        - $\sum_{i=2}^{n} \log i$ data movements
    - $\sum_{i=2}^{n} \log i \leq n \log n - n$
        - $2n + 2\sum_{i=2}^{n} \log i \leq 2n \log n$ comparisons.
        - $n + \sum_{i=2}^{n} \log i \leq n \log n$ data movements.

Lower Bounds for sorting Problem

- Insertion sort, Selection sort: $O(n^2)$ Mergesort, heapsort, (quicksort): $O(n \log n)$
- Lower bound for a Problem:

- A proof that NO Algorithm can solve the problem better.
- Much harder to prove a lower bound for a problem since we have to consider ALL possible algorithms, not just one particular approach.
- We need a model corresponding to an arbitrary (unspecified) algorithm
  - And a proof that ANY algorithm that fits the model will has a running time higher than the lower bound.
- Example:
  - We cannot say we will use a special data structure for this problem. Because there may be an algorithm that do not use this data structure and runs faster.

Decision tree model

- binary trees with two types of nodes:
  internal nodes: two children, *leaves:* no child.
  (Also called two-trees or 2-trees)
- Internal node: associated with a query, the outcome is one of two possibilities. Each one is associated to one of the branches.
- Leaf: associated with a possible output
- Input is a sequence of numbers: $x_1, x_2, \ldots, x_n$
  Computation starts at the root of the tree.
  In each internal node, the query is applied.
  Either go left or go right depending on the result of the query.
- When reaching a leaf, the output associated with the leaf is the output of the computation.
- The worst-case running time of a tree T is the height of T. That is the maximum number of queries required by an input.

Union-Find

Problem:

- Given a set X of n elements x1, x2, . . . , xn. We would like to maintain a collection of disjoint subsets (groups) of X
- 3 operations
  - Make set(i): makes x_i a subset and assigns a name for the subset.
  - Find(i): returns the name of the subset that contains x_i.
  - Union(i, j): combines subsets that contain x_i and x_j, say S_i and S_j, into a new subset with a unique name.

Naïve solution (array)

- Make set(i): we just set A[i] to i.
- Find(i): we just look at A[i] and find out the name for the subset.
- Union(i, j): (Assume the name of the resulting subset is Si's name) Change the subset name for all elements in Sj
- Time: O(n**2)

Solution 2 – linkedlist

- Each set is represented by a linked list.
- The first node in each list serves as its set's representative.

- Each node of the list contains a set member, a pointer to the next node, and a pointer back to the representative.
- Each list maintains a pointer, head, to the first node and a pointer, tail, to the last node.
- For the Union(i,j), we will append the smaller list onto the longer list and update representative pointers of the smaller list.
- $O(m + n \log n)$
- Could still be implemented with an array, but each element will be a record with four members: next, head, tail, and size
    - Nodes in each list will maintain next and head.
    - Only the first node of each list needs to maintain tail and size.
    - Make set(i): A[i].next=i, A[i].head=i, A[i].tail=i, and A[i].size=1. O91)
    - Find(i): A[i].head. O(1)
    - Union(i, j): you can add details here → O(min(A,B)) and worst case O(n)

Solution 3 – Tree

- make Union operation simple.
- Each set is a tree and each node in a tree is a record: one field for element name (not really necessary), one field for a pointer (parent pointer) to another node.
- Find(i): from entry i, follow parent pointer until we find a node with a nil pointer, or a pointer pointing to itself (root). Return the name in that node.
    - O(n)
- Union(i, j): we change the pointer of the root of set S_j to pointing to the root of set S_i, or vice-versa.
    - O(1)

Efficient Union-Find

- Idea: balance and collapse the trees.
- Balancing: when union operation is performed, the root pointer of the smaller tree is set to point to the root of larger tree.
    - Rather than explicitly keeping the size of the subtree rooted at each node, we use another approach.
    - For each node, we maintain a rank that is an upper bound on the height of that node.
    - In union by rank, the root with smaller rank is made to point to the root with larger rank during an Union(i,j) operation.
        - If two roots have equal ranks, we arbitrarily choose one of the roots as the parent, increase its rank by 1, and reset the other root.
        - With Make set(i), the rank is set to 0.
- If union by rank is used, then for any node, its height is bounded by its rank.
- If union by rank is used, then for any node i, its rank is bounded by log(size(i)).
- Any find operation is at most O(log n)
- Any sequence of m ≥ n operations will be bounded by O(m log n).
- Union: constant time.
- Find: O(log n) time.
- Path compression (collapse the tree)

- If both balancing and path comparisons are used, then the total number of steps in the worst case for any sequence of $m \geq n$ operations, n of which are Make set operations, is $O(m \log n)$.

Huffman Codes

- Data compression is an important technique for saving storage
- Given a file,
  - We can consider it as a string of characters
  - We want to find a compressed file
  - The compressed file should be as small as possible
  - The original file can be reconstructed from the compressed file
- This is useful when access to the file is infrequent (most files are this kind of files!)
  - So we do not care about the work of compressing and decompressing.
- It is important in data communication where the cost of sending information is greater than the cost of processing them.
- Fixed length encoding → ASCII

Variable length encoding

- We use fewer bits to represent letters that appear very often (for example A).
- We use more bits to represent letters that appear less often (for example Z).
- can still decode without the delimiters – as long as we know where to start.
- the prefix constraint
  - the prefixes of a code must not equal the complete code of another code
  - With this constraint, we can scan the bits from left to right and determine the codes for letters.
  - In general when we shorten the code for one letter, we may have to lengthen the code for another letter.
  - Our goal is to find the best balance

The Problem: Given a text (a sequence of characters), find an encoding for all the characters that satisfies the prefix constraint and that minimizes the total number of bits needed to encode the text.

First we compute the number of times each character appears in the text. We call this value the frequency of the character. (In many cases we can use a standard frequency table.)

- Let $c_1, c_2, \ldots, c_n$ be the characters
- Let $f(c_1), f(c_2), \ldots, f(c_n)$ be their frequencies.
- Let T be an encoding method.
- Let $d_T(c_i)$ be the length of the code for c_i under T.
- The length, in bits, of the file compressed by using T is:

$$B(T) = \sum_{i=1}^{n} f(c_i) d_T(c_i)$$

- Our goal is to find an encoding T that satisfies prefix constraint and minimizes $B(T)$.

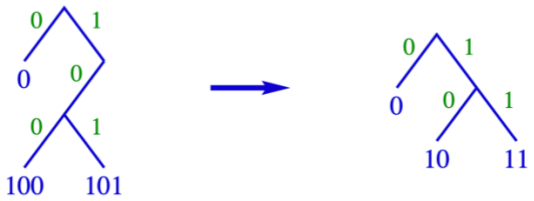
Tree representation of encoding

- Given an encoding of n characters satisfying the prefix constraint, we can construct a binary tree to represent this encoding.
- Each code is a path from root to a leaf. 0 means go left, 1 means go right. Each leaf represents a character.
- The number of leaves is n.
- Given an n-leave binary tree, it represents an encoding of n characters satisfying prefix constraint.
- We label edge to the left with 0. We label edge to the right with 1.
- Each leaf corresponds to one code for a character.
- An encoding of n characters satisfying prefix constraint is equivalent to an n-leave binary tree.

An optimal encoding is always represented by a 2-tree.

Why?

If one internal node has only one child, we can delete this internal node, therefore shortening the encoding for some characters.



In an optimal encoding T, there are x and y such that $d_T(x) = d_T(y)$ is maximal and their codes only differ in the last bit.

E.G. Consider the binary 2-tree of this optimal encoding. Let its height be h. Let N be an internal node of level $h-1$. Then the children of N have the above property.