CS240: Notes Taken From Lectures

by Haocen Jiang
haocen.jiang@edu.uwaterloo.ca
Prof. Arne Storjohann

University of Waterloo — Fall 2018

All contents are from lectures given by Prof. Storjohann and course materials

Contents

Introduction and Asymptotic Analysis

Random Access Machine (RAM) Model

- The `random access machine` has a set of memory cells, each of which stores one item of data.
- Any `access` to a memory location takes constant time
- Any `primitive operation` takes constant time.
- The `running time` of a program can be computed to be the number of memory accesses plus the number of primitive operations

Order Notation

O-notation:

- $f(n) \in O(g(n))$ if there exist constant $c > 0$ and $n_o > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_o$
- $f(n)$ grows "no faster than" $g(n)$

Example 1

Prove that $(n+1)^5 \in O(n^5)$
we need to prove that $\exists c > 0, n_o > 0$ s.t. $0 \le f(n) \le cg(n)\ \forall n \ge n_o$
Proof. Note that $n + 1 \le 2n\ \forall n \ge 1$ Raise both side the the power of 5 gives:

$$(n+1)^5 \le 32n^5$$

Thus we have found $c = 32$ and $n_o = 1$

- **Properties**: Assume that $f(n)$ and $g(n)$ are both ***asymptotically non-negative***
  1. $f(n) \in O(af(n))$ for any constant a
     pf. $0 \le f(n) \le \frac{1}{a}af(n)$ for all $n \ge n_o := N$
  2. if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$
     pf. $f(n) \in O(g(n)) \Rightarrow \exists c_1, n_1 > 0$ s.t. $f(n) \le c_1 g(n)\ \forall n \ge n_1$
     $g(n) \in O(h(n)) \Rightarrow \exists c_2, n_2 > 0$ s.t. $g(n) \le c_2 h(n)\ \forall n \ge n_2$
     $\therefore f(n) \le c_1 c_2 h(n)$ for all $n \ge \max(n_1, n_2)$
  3. a) $\max(f(n), g(n)) \in O(f(n) + g(n))$
        p.f. $0 \le max(f(n), g(n)) \le 1 \cdot [f(n) + g(n)]\ \forall n \ge N$
     b) $f(n) + g(n) \in O(max(f(n), g(n)))$
        p.f. $0 \le f(n) + g(n) \le 2 \cdot [max(f(n), g(n))]\ \forall n \ge N$
  4. a) $a_0 + a_1 n + \cdots + a_d n^d \in O(n^d)$ if $a_d > 0$
     b) $n^d \in O(a_0 + a_1 n + \cdots + a_d n^d)$

Ω-notation:

- $f(n) \in O(g(n))$ if there exist constant $c > 0$ and $n_o > 0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_o$
- $f(n)$ grows "no slower than" $g(n)$

Example 2

$n^3 \log(n) \in \Omega(n^3)$ since $\log(n) \ge 1$ for all $n \ge 3$

Θ-notation:

- $f(n) \in O(g(n))$ if there exist constant $c_1, c_2 > 0$ and $n_o > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_o$
- $f(n)$ grows "at the same rate as" $g(n)$
- **Fact:** $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ adn $f(n) \in \Omega(g(n))$

> **Example 3**
>
> $2n^3 - n^2 \in \Theta(n^3)$

o-notation:

- $f(n) \in o(g(n))$ if **for all** constants $c > 0$, there exist $n_o > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_o$
- $f(n)$ grows "slower than" $g(n)$

> **Example 4**
>
> **Claim:** $2010n^2 + 1388n \in o(n^3)$ ***proof.***
> let $c > 0$ be given, then
>
> $$\begin{aligned} 2010n^2 + 1388n &< 5000n^2 \\ &= \left(\frac{5000}{n}\right) n^3 \\ &\leq cn^3 \quad \forall n \geq \frac{5000}{c} \end{aligned}$$

ω-notation:

- $f(n) \in \omega(g(n))$ if **for all** constants $c > 0$, there exist $n_o > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_o$
- $f(n)$ grows "faster than" $g(n)$
- $f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$

Complexity of Algorithm

Common growth rate

- $\Theta(1)$(*constant complexity*)
- $\Theta(\log n)$(*logarithmic complexity*) e.g. binary search
- $\Theta(n)$(*linear complexity*)
- $\Theta(n\log n))$(*linearithmetic complexity*)e.g. merge sort
- $\Theta(n^2)$(*quadratic complexity*)
- $\Theta(n^3)$(*cubic complexity*)e.g. matrix multiplication
- $\Theta(2^n)$(*quadratic complexity*)

Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_o$. Suppose that

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty \end{cases}$$

Example 5 A1P3

Prove of disprove the following statements

(a) $f(n) \notin o(g(n))$ and $f(n) \notin \omega(g(n)) \Rightarrow f(n) \in \Theta(g(n))$

disprove: Counter example, consider $f(n) := n$ and $g(n) := \begin{cases} 1 & \text{n is odd} \\ n^2 & \text{n is even} \end{cases}$.

- For any odd number $n_1 > c$, we have $f(n_1) = n_1 > c = cg(n_1)$, showing that $f(n) \notin O(g(n))$, and therefore, $f(n) \notin o(g(n))$ - Similarly, for any even number $n_1 > 1/c$ we have $cg(n_1) = cn_1^2 > n_1 = f(n_1)$, showing that $f(n) \notin \Omega(g(n))$ and therefore, $f(n) \notin \omega(g(n))$ - However, since $f(n) \notin \Omega(g(n))$, it has to be the case that $f(n) \notin \Theta(g(n))$

(b) $min(f(n), g(n) \in \Theta\left(\frac{f(n)g(n)}{f(n)+g(n)}\right)$

Proof. We will show that $\frac{f(n)g(n)}{f(n)+g(n)} \leq min(f(n), g(n)) \leq 2\frac{f(n)g(n)}{f(n)+g(n)}$ for all $n \geq 1$. The desired result will then follow from the definition of Θ using $c_1 = 1, c_2 = 2$ and $n_0 = 1//$ By assumption, f and g are positive, so $fg/(f+g) = \min(f,g)\max(f,g)/(f+g)$, which is less than $\min(f,g)$ since $\max(f,g)/(f+g) < 1$. Similarly, $\min(f,g) = 2fg/(2\max(f,g)) \leq 2fg/(f+g)$

Example 6

Prove that $n(2+\sin(n\pi/2)$ is $\Theta(n)$. Note that $\lim_{n\to\infty}(2+\sin n\pi/2)$ does not exist
Proof. $n \leq n(2+\sin\frac{n\pi}{2}) \leq 3n$

Example 7

Compare the growth rates of $\log n$ and n^i (where $i > 0$ is a real number).

$$\lim_{n\to\infty} \frac{\log n}{n^i} = \lim_{n\to\infty} \frac{1/n}{in^{i-1}} = \lim_{n\to\infty} \frac{1}{in^i} = 0$$

This implies that $\log n \in o(n^i)$

Relationships between Order Notations

- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$

- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$
- $f(n) \in o(g(n)) \Rightarrow f(n) \notin \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- $f(n) \in \omega(g(n)) \Rightarrow f(n) \notin O(g(n))$

"Maximum" rules

- $O(f(n) + g(n)) = O(max\{f(n), g(n)\})$
- $\Theta(f(n) + g(n)) = \Theta(max\{f(n), g(n)\})$
- $\Omega(f(n) + g(n)) = \Omega(max\{f(n), g(n)\})$

Transitivity

If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$ If $f(n) \in \Omega(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in \Omega(h(n))$

Techniques for Algorithm Analysis

Two general strategies are as follows.

- Use Θ-bounds throughout the analysis and obtain a Θ-bound for the complexity of the algorithm.
- Prove a O-bound and a matching Ω-bound separately. Use upper bounds (for O-bounds) and lower bounds (for Ω-bound) early and frequently. This may be easier because upper/lower bounds are easier to sum.

Worst-case complexity of an algorithms:

The worst-case running time of an algorithm A is a function $f : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the ***longest*** running time for any input instance of size n:

$$T_A(n) = \max\{T_A(I) : Size(I) = n\}.$$

Average-case complexity of an algorithm:

The average-case running time of an algprith A is a function $f : \mathbb{Z}^+ \to \mathbb{R}$ mapping n (the input size) to the ***average*** running time over all instances of size n:

$$T_A^{avg}(n) = \frac{1}{|\{I : Size(I) = n\}|} \sum_{\{I:Size(I)=n\}} T_A(I).$$

Notes on O-notation

- It is important not to try to make comparisons between algorithms using O-notations.
- For example, suppose algorithm A_1 and A_2 both solve the same problem, A_1 has worst-case run-time $O(n^3)$ and A_2 has worst-case run-time $O(n^2)$. We **cannot** conclude that A_2 is more efficient

> **NOTE**
> 1.The worst-case run-time may only be achieved on some instances.
> 2. O-notation is an upper bound. A_1 may well have worst-case run-time $O(n)$. If we want to be able to compare algorithms, we should always use Θ-notation.

Example 8

Goal: Use asymptotic notation to simplify run-time analysis.

```
Test1(n)
1.    sum ← 0
2.    for i ← 1 to n do
3.        for j ← i to n do
4.            sum ← sum + (i − j)^2
5.    return sum
```

- size of instance is n
- line1 and line5 execute only once: $\Theta(1)$
- running time proportional to: number of iterations if the j-loop

Direct Method:

$$\#\text{ of iteration} = \sum_{i=1}^{n}(n-i+1) = \sum_{i=1}^{n} i = \frac{n(n+1}{2}$$

$\Rightarrow$ # of iterations of j-loop is $\Theta(n^2)$
$\Rightarrow$ Complexity of Test 1 is $\Theta(n^2)$

Sloppy Method:

$$\#\text{ of iteration} = \sum_{i=1}^{n}(n-i+1) \leq \sum_{i=1}^{n} n = n^2$$

$\Rightarrow$ Complexity of Test 1 is $O(n^2)$

Merge Sort

```
MergeSort(A, ℓ ← 0, r ← n − 1)
A: array of size n, 0 ≤ ℓ ≤ r ≤ n − 1
    if (r ≤ ℓ) then
        return
    else
        m = (r + ℓ)/2
        MergeSort(A, ℓ, m)
        MergeSort(A, m + 1, r)
        Merge(A, ℓ, m, r)
```

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + cn \\
&= 2\left(2T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)\right) + cn \\
&= 4T\left(\frac{n}{4}\right) + c\left(2\left(\frac{n}{2}\right) + n\right) \\
&= 4\left(2T\left(\frac{n}{8}\right) + c\left(\frac{n}{4}\right)\right) + c\left(2\left(\frac{n}{2}\right) + n\right) \\
&= 8T\left(\frac{n}{8}\right) + c\left(4\left(\frac{n}{4}\right) + 2\left(\frac{n}{2}\right) + n\right) \\
&= \dots \\
&= nc + c\left(n + 2\left(\frac{n}{2}\right) + 4\left(\frac{n}{4}\right) + \cdots + \left(\frac{n}{2}\right)\left(\frac{n}{\frac{n}{2}}\right)\right) \\
&= nc + cn\log(n)
\end{aligned}$$

Some Recurrence Relations

Recursion	resolves to	example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify ($\rightarrow$ later)
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$	Selection ($\rightarrow$ later)
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search ($\rightarrow$ later)
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpolation Search ($\rightarrow$ later)

Helpful Formulas

Arithmetic Sequence

$$\sum_{i=0}^{n-1}(a + di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2)$$

Geometric Sequence

$$\sum_{i=0}^{n-1} ar^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$$

A few more

$\sum_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^2}{6}$ $\qquad$ $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$

Priority Queues

Abstract Data types

Abstract Data Type(ADT): A description of information and a collection of operations on that information.

- We can say **what is stored**
- We can say **what can be done with it**
- We **Do not** say how it is implemented

Possible Properties of the data

- can check $a = b$ or $a \neq b$
- sets of items may be *totally ordered*
- items may be elements of a ring, e.g. $\{+, -, \times\}$ make sense

Stack ADT

- Stack: an ADT consisting of a collection of items with operations:
  - *push:* inserting an item
  - *pop:* removing the most recently inserted item
- Items are removed in *last-in first-out* order **(LIFO)**. We need no assumptions on items

Realization of Stack ADT: Arrays

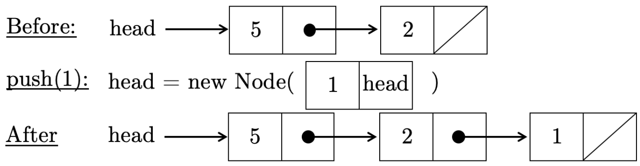
Store the data in an array and keep track of the size of the array. Add the new data to the end of the array every time we insert. Delete the last item in the array when we need to pop an item.

```
pop() //size>0
  temp = A[size-1]
  size--
  return temp
```

Overflow Handling: if the array is full

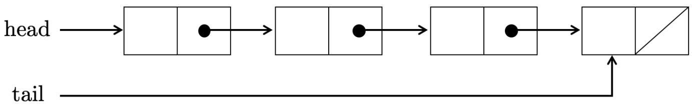
- create new array twice the size
- copy items over
- takes time $\Theta(n)$, but happens rarely
- average over operation costs $\Theta(1)$ overhead.
- In CS240, always assume array has enough space.

Realization of Stack ADT: Linked List



Queue ADT

- Queue: an ADT consisting of a collection of items with operations:
  - *enqueue*: inserting an item
  - *dequeue:* removing the least recent inserted item
- Items are removed in first-in first-out (**FIFO**) order.
- Items enter the queue at the *rear* and are removed from the *front*
- Realizations of Queue ADT
  - using (circular) arrays(partially filled)
  - using linked lists



Priority Queue ADT

Priority Queue ADT

- Priority Queue: An ADT consists of items (each having a *priority*) with operations:
  - *insert*: inserting an item tagged with a priority
  - *deleteMax*: removing the item of *highest priority*
- the priority is also called *key*
- the above definition is for a **maximum-oriented** priority queue. A **minimum-oriented** priority queue is defined in the natural way, by replacing the operation *deleteMax* by *deleteMin*

PQ-sort

```
PQ-Sort(A[0..n − 1])
    initialize PQ to an empty priority queue
    for k ← 0 to n − 1 do
        PQ.insert(A[k])
    for k ← n − 1 down to 0 do
        A[k] ← PQ.deleteMax()
```

Realizations of Priority Queue: Unsorted Array

- *insert*: $\Theta(1)$
  - insert at position n, increment n
- *deleteMax:* $\Theta(n)$
  - find maximum priority of element $A[i]$
  - swap $A[i]$ with $A[n-1]$
  - return $A[n-1]$ and decrement n

Realizations of Priority Queue: Sorted Array

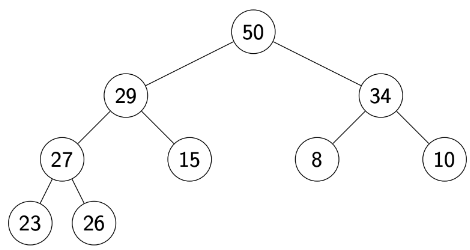
- *insert*: $\Theta(n)$
  - find the correct position to insert
  - shift all the elements after to make room
- *deleteMax:* $O(1)$
  - delete the last element and decrement n

Goal: achieve $O(\log(n))$ run-time for both *insert* and *deleteMax*
Solution: use heap: max stores two possible candidates for the next biggest item

Binary Heaps

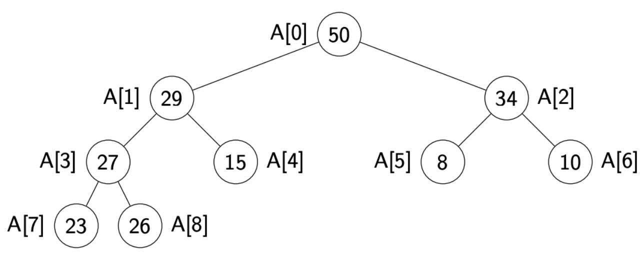
A *(binary)heap* is a certain type of binary tree such that

1) **Structural properties**
   - all levels are full except for the last level
   - last level is left justified
2) **heap-order properties**
   - for any node i, key of the parent of i is larger than to equal to the key of i



(In our examples we only show the priorities, and we show them directly in the node. A more accurate picture would be priority = 50, <other info>)

Lemma: The height of a heap with n nodes is $\Theta(\log n$.
Heap in Array: Heaps should not be stored as binary trees.
Let H be a heap of n items and let$A$ be an array of size n. Store root in $A[0]$ and continue with elements level-by-level from top to bottom, in each level left-to-right.



It is easy to navigate the heap using this array representation:

- the *root* node is $A[0]$,
- the *left child* of $A[i]$ (if it exists) is $A[2i+1]$,
- the *right child* of $A[i]$ (if it exists) is $A[2i+2]$,
- the *parent* of $A[i]$ $(i \neq 0)$ is $A[\lfloor \frac{i-1}{2} \rfloor]$,
- the *last* node is $A[n-1]$

Operations in Binary Heaps

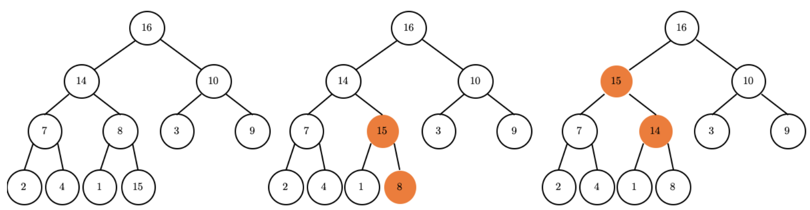
Insertion in Heaps

- Place the new key at the first free leaf
- since we have a array-representation, increase the *size* of the array and insert the new `last(size)`: (bottom level, left most free spot)
- the **heap order property** may be violated: perform a `fix-up`
  - compare the key of the node with its parent
  - if the key bigger than its parent, swap with the parent
  - The new item bubbles up until it reaches its correct place in the heap.
  - **Time:** $O(\text{height of heap}) = O(\log n)$

```
fix-up(A, k)
k: an index corresponding to a node of the heap
    while parent(k) exists and A[parent(k)] < A[k] do
        swap A[k] and A[parent(k)]
        k ← parent(k)
```

Example 9 fix-up

insert(15) on $A = [16, 14, 10, 7, 8, 3, 9, 2, 4, 1]$



$A = [16, 15, 10, 7, 14, 3, 9, 2, 4, 1, 8]$

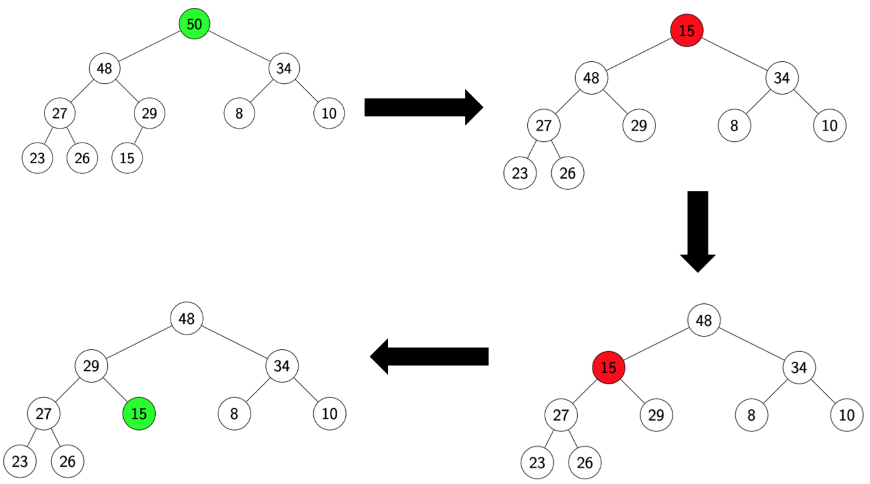
DeleteMax in Heaps

- The maximum item of a heap is just the root node.
- We replace root by the last leaf, decrease the size
- the **heap order property** may be violated: perform a `fix-down` on the root:
  - compare the node with its children
  - if it is larger than both children, we are done
  - otherwise swap the node with the larger children, then perform `fix-down` on it again
  - **Time:** $O(\text{height of heap}) = O(\log n)$

```
fix-down(A, n, k)
A: an array that stores a heap of size n
k: an index corresponding to a node of the heap
    while k is not a leaf do
        // Find the child with the larger key
        j ← left child of k
        if (j is not last(n) and A[j + 1] > A[j])
            j ← j + 1
        if A[k] ≥ A[j] break
        swap A[j] and A[k]
        k ← j
```

Example 10 fix-down

perform `deleteMax` on $A = [16, 15, 10, 7, 14, 3, 9, 2, 4, 1, 8]$



Proprity queue using max heaps

- Use a partially filled array
- keep track of size of the heap
- Keep heap-order satisfied using
  - `fix-up` on `insert`
  - `fix-done` on `deleteMax`
- Goal achieved: $O(\log n)$ for `insert` and `deleteMax`

PQ-sort and HeapSort

Recall Priority Queue Sort

1) for $i = 0$ to $n - 1$, insert $A[i]$ into the priority queue
2) for $i = n - 1$ to 0, `deleteMax()` from the PQ and insert the key into $A[i]$

```
PQ-SortWithHeaps(A)
    initialize H to an empty heap
    for k ← 0 to n − 1 do
        H.insert(A[k])
    for k ← n − 1 down to 0 do
        A[k] ← H.deleteMax()
```

⇒ using heap for PQ: PQ sort takes:

- $O(n \log n)$ time
- $O(n)$ auxiliary space

Improvements

1: Use the same array for input/output
⇒ need only $O(1)$ auxiliary space

2: ***Heapify:*** Do line one faster

- We know all items to insert beforehand
- can atually build heap in $O(n)$ time!

Heapofy: Bottom up creation of heap

Put items in nearly complete binary tree, for $i = n - 1$ down to 1, do `fix-down` position i

```
heapify(A)
A: an array
    n ← A.size()
    for i ← parent(last(n)) downto 0 do
        fix-down(A, n, i)
```

Example 11 Heapify

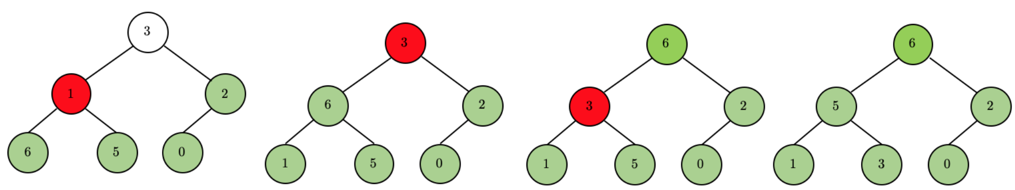
Run `Heapify` on $A = [3, 1, 2, 6, 5, 0]$

for $i = 5, 4, 3$, nothing to do

for $i = 2$, key is OK

for $i = 1$, sway with 6, we are done

for $i = 0$, swap with 6, swap with 5, we are done



Note:

- `fix-down` does nothing for keys at the last level h
- code could start at the key $A[\lfloor \frac{n-1}{2} \rfloor]$
- Heapify using `fix-up` also works, but this might take more time when the heap gets large, because we need to call `fix-up` on all the nodes at level h, which will iterate through its branch to all the way to the root.

HeapSort

- Idea: *PQ-Sort* with heaps
- But use the same input array A for (in-place) storing the heap.

```
HeapSort(A, n)
    // heapify
    n ← A.size()
    for i ← parent(last(n)) downto 0 do
        fix-down(A, n, i)
    // repeatedly find maximum
    while n > 1
        // do deleteMax
        swap items at A[root()] and A[last(n)])
        decrease n
        fix-down(A, n, root())
```

- The `for-loop` takes $\Theta(n)$ time and the `while-loop` tkaes $O(n \log n)$ time.
- number of swaps is bounded by

$$\sum_{i=0}^{h} i 2^{h-i} \leq \sum_{i=0}^{h} \frac{in}{2^i} \leq n \sum_{i=0}^{\infty} \frac{i}{2^i} = 2n$$

Intro for the Selection Problem

Problem

- **Given:** array $A[0, 1, \ldots, n-1]$, index k with $0 \leq k \leq n-1$
- **Want:** item that would be at $A[k]$ if A were sorted

Possible solutions

1. Make k passes through the array, deleting the minimum number each time. $\Rightarrow \Theta(kn)$
2. Sort the array first, the return $A[k] \Rightarrow \Theta(n \log n)$
3. Build a maxHeap from A and call `deleteMax` $(n-k+1)$ times $\Rightarrow \Theta(n + (n-k+1)\log n)$
4. Build a minHeap from A and call `deleteMin` k times $\Rightarrow \Theta(n + k \log n)$

Sorting and Randomized Algorithms

QuictSelect

Selection and sorting

The *selection problem*: Given an array A of n numbers, and $0 \leq k < n$, find the element that would be at position k of the sorted array.

- The best heap-based algorithm had running time $\Theta(n + k \log n)$, and for *median finding*, this is $\Theta(n \log n)$
- Question: Can we do selection in linear time?
- The **quick-select** answers this question in the affirmative

Partition and choose-pivot

quick-select and related algorithm *quick-sort* rely on two subroutines

- `choose-pivot`(A): Choose an index p. We will use the *pivot value* $v < -A[p]$ to rearrange the array. The simplest idea is to **return the last element**
- `partition`(A, p): Rearrange A and return *pivot-index* i so that

- the pivot value v is in $A[i]$,
- all items in $A[0, \ldots, i-1]$ are $\leq v$,
- all items in $A[i+1, \ldots, n-1]$ are $\geq v$.

0				p			n - 1
?	$\cdots$	$\cdots$	?	v	$\cdots$	$\cdots$	?

$\leq v$	$\leq v$	$\cdots$	$\leq v$	v	$\geq v$	$\cdots$	$\geq v$
0				i			n - 1

Implementations

```
partition(A, p)
A: array of size n,   p: integer s.t. 0 ≤ p < n
      Create empty lists small and large.
      v ← A[p]
      for each element x in A[0, ..., p−1] or A[p+1 ... n−1]
            if x < v append x to small
            else append x to large
      i ← size(small)
      Overwrite A[0 ... i−1] by elements in small
      Overwrite A[i] by v
      Overwrite A[i+1 ... n−1] by elements in large
      return i
```

```
partition(A, p)
A: array of size n,   p: integer s.t. 0 ≤ p < n
1.    swap(A[n − 1], A[p])
2.    i ← −1,   j ← n − 1,   v ← A[n − 1]
3.    loop
4.          do i ← i + 1 while i < n and A[i] < v
5.          do j ← j − 1 while j > 0 and A[j] > v
6.          if i ≥ j then break    (goto 9)
7.          else swap(A[i], A[j])
8.    end loop
9.    swap(A[n − 1], A[i])
10.   return i
```

Quick-Select1

```
quick-select1(A, k)
A: array of size n,   k: integer s.t. 0 ≤ k < n
      p ← choose-pivot1(A)
      i ← partition(A, p)
      if i = k then
            return A[i]
      else if i > k then
            return quick-select1(A[0, 1, ..., i − 1], k)
      else if i < k then
            return quick-select1(A[i + 1, i + 2, ..., n − 1], k − i − 1)
```

- Recall that i is returned by partition, thus we have no control
- if $i = k$, then return v as the solution
- if $i > k$
  - There are $i > k$ items in A that are $\leq v$
  - Therefore the desired return value m is $\leq v$
  - Find m by searching recursively on the left
- if $i < k$
  - There are $i < k$ items in A that are $\leq v$
  - Therefore the desired return value m is $\geq v$
  - Find m by searching recursively on the right

Runtime of Quick-Select1

Let $T(n)$ be the run time, if we select from n elements. Since partition takes $O(n)$ if $n \geq 2$, for some constant c, we have

$$T(n) = \begin{cases} c & n = 1 \\ T(\text{size of the subarray}) + cn & n \geq 2 \end{cases}$$

Worse-case: size of the subarray is $n-1$, therefore

$$\begin{aligned}
T^{\text{worst}}(n) &\le cn + T^{\text{worst}}(n-1) \\
&\le cn + c(n-1) + T^{\text{worst}}(n-2) \\
&\le cn + c(n-1) + c(n-2) + T^{\text{worst}}(n-3) \\
&\le \dots \\
&\le cn + c(n-1) + \dots + c(2) + c(1) \\
&= c\frac{n(n+1)}{2} \in O(n^2)
\end{aligned}$$

Best-case: One single partition: $\Theta(n)$

Average-case: There are infinitely many instances of size n, how to calculate the average? ***Sorting Permutation***

Sorting Permutation

Observation: `quickSelect` is comparison based: It doesn't care what actual input numbers are: it only cares if $A[i] \le A[j]$

For example: It will act the same on inputs $A = [4, 8, 2]$ and $A = [5, 7, 3]$

Simplifying assumption: All input numbers are distinct, since only their relative order matter, we can characterize type of inputs by sorting permutation π

- We assume all $n!$ permutations are *equally likely*
- Average cost is the sum of costs for all permutations, divided by $n!$

Therefore, we an analyze the average case of `quick-select1`:

- If we know the pivot-index i, then the subarrays have sizes i and $n-i-1$
- $T(n) \le cn + \max(T(i), T(n-i-1))$
- How many *sorting permutations* π lead to index i?
  * Let's sat $p = n-1$
  * Pivot-index = i $\iff$ the pivot element $A[p]$ is $i'th$ smallest $\iff$ $\pi(i) = p$
  * All other $n-1$ elements of π could be arbitrary
- Thus there are $(n-1)!$ sorting permutations has pivot index i

$$T^{avg}(n) \le \frac{1}{n!} \underbrace{\sum_{i=0}^{n-1}}_{\text{split by i}} \underbrace{(n-1)!}_{\text{have pivot-index}i} \underbrace{(cn + \max(T^{avg}(i), T^{avg}(n-i-1)))}_{\text{run-time bound if pivot index is}i}$$

- **Lemma:** $\sum_{i=0}^{n-1} \max(i, n-i-1) \le \frac{3}{4}n^2$

  Proof. **If n is even:**

$$\begin{aligned}
\sum_{i=0}^{n-1} \max(i, n-i-1) &= 2\sum_{i=\frac{n}{2}}^{n-1} i \\
&= \frac{3}{4}n^2 - \frac{1}{2}n \\
&\le \frac{3}{4}n^2
\end{aligned}$$

  if n is odd:

$$\begin{aligned}
\sum_{i=\lfloor\frac{n}{2}\rfloor}^{n-1} \max(i, n-i-1) &= \lfloor\frac{n}{2}\rfloor + 2\sum_{i=0}^{n-1} i \\
&= \lfloor\frac{n}{2}\rfloor + n^2 - \lceil\frac{n}{2}\rceil^2 + \lceil\frac{n}{2}\rceil - n \\
&< \frac{3}{4}n^2
\end{aligned}$$

□

- **Theorem:** $T^{avg}(n) \le 4cn$ (prove by Induction)
  * *Base case:* n = 1
    $$T^{avg}(1) = c \le 4c$$
  * *Induction hypothesis*: Assume that $T^{avg}(N) \le 4cn$ for all $N < n, n > 1$
  * *Inductive Step*

    Proof.
    $$\begin{aligned}
    T^{avg}(n) &\le cn + \frac{1}{n}\sum_{i=0}^{n-1} \max(T(i), T(n-i-1)) \\
    &\le cn + \frac{1}{n}\sum_{i=0}^{n-1} \max(4ci, 4c(n-i-1)) \\
    &= cn + \frac{4c}{n}\sum_{i=0}^{n-1} \max(i, n-i-1) \\
    &\le cn + \frac{4c}{n}\frac{3}{4}n^2, \text{ by Lemma} \\
    &= cn + 3cn \\
    &= 4cn
    \end{aligned}$$
    □
- i.e. $T^{avg}(n) \in O(n)$, which is a tight upper bound

Randomized Algorithms

Expected Running Time

- A *randomized algorithm* is one which relies on some random numbers in addition to the input
- The cost will depend on the input and the random numbers used
- Define $T(I, R)$ to be the running time of the randomized algorithm for instance I and the sequence of random numbers R.
- The *expected running time* $T^{exp}(I)$ for instance I is the *expected value* for $T(I, R)$:
  $$T^{exp}(I) = E[T(I, R)] = \sum_R T(I, R) \cdot P(R)$$
- The *worse-case expected running time* is
  $$T^{exp}_{avg}(n) = \max_{\{I:size(I)=n\}} T^{exp}(I)$$
-
- The *average-case expected running time* is
  $$T^{exp}_{avg}(n) = \frac{1}{|\{I : size(I) = n\}|} \sum_{\{I:size(I)=n\}} T^{exp}(I)$$

Randomized Quick Select

```
choose-pivot2(A)
1.    return random(n)
```

```
quick-select2(A, k)
1.    p ← choose-pivot2(A)
2.    . . .
```

- To achieve average-case run-time, we randomly permute inputs.
- **simple Idea:** pick pivot-index p randomly in $\{0, \ldots, n-1\}$
- **key insight:** $P(\text{index of pivot is } i) = \frac{1}{n}$
- **Detour 1:** How to choose random index?
  Use language provided `random(max)`
- **Detour 2:** How to analyze an algorithm that use `random`?
  - Measure expected running time of randomized algorithm A,
  - For one instance I

$$T^{exp}(I) = \sum_{r} T(A \text{ on } I \text{ with } r \text{ chosen}) \cdot P(\text{r was chosen})$$

  - For `quick-select`

$$\begin{aligned} T^{exp}(n) &= cn + \sum_{i=0}^{n-1} P(\text{pivot index is i}) \cdot (\text{run-time if index is i}) \\ &\le cn + \sum_{i=0}^{n-1} \frac{1}{n} \max(T^{exp}(i), T^{exp}(n-i-1)) \end{aligned}$$

> **Note**
> 1. The message is that randomized `quick-select` has $O(n)$ expected run time.
> 2. This expression is the same is the running time of non-randomized quick-select. For average-case of non-randomized quick-select, $\frac{1}{n}$ represents the proportion of the permutation with i chosen as pivot index the over all the possible permutation. While here , $\frac{1}{n}$ is a probability

QuickSort

Hoare developed *partition* and *quick-select* in 1960; together with a *sorting* method based on partitioning:

```
quick-sort1(A)
A: array of size n
    if n ≤ 1 then return
    p ← choose-pivot1(A)
    i ← partition(A, p)
    quick-sort1(A[0, 1, ..., i − 1])
    quick-sort1(A[i + 1, ..., n − 1])
```

Worst case

$T^{\text{worst}}(n) = T^{\text{worst}}(n-1) + \Theta(n)$ Same as quick-select: $T^{\text{worst}}(n) \in \Theta(n^2)$

Best case

$T^{\text{best}}(n) = T^{\text{best}}(\lfloor \frac{n-1}{2} \rfloor) + T^{\text{best}}(\lfloor \frac{n-1}{2} \rfloor) + \Theta(n)$

Average case

- Rather than analyze run-time, can simply count comparisons.
- Observe: partition uses $\le n$ comparisons.

- Recurrence relation (if we know pivot-index)

$$T(n) \leq \begin{cases} 0 & n \leq 1 \\ n + T(i) + T(n-i-1) & n > 1, \text{pivot index } i \end{cases}$$

$$\begin{aligned}
T^{\text{avg}}(n) &= \frac{1}{n!} \sum_{\text{perm } \pi} (\#\text{ of comparisons if input has sorting permutation } \pi) \\
&= \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{\text{perm } \pi} (\#\text{ of comparisons if input has sorting permutation } \pi) \\
&= \frac{1}{n!} \sum_{i=0}^{n-1} (\#\text{of permutations with pivot index } i)(n + T^{\text{avg}}(i) + T^{\text{avg}}(n-i-1)) \\
&\leq \frac{(n-1)!}{n!} \sum_{i=0}^{n-1} (n + T^{\text{avg}}(i) + T^{\text{avg}}(n-i-1)) \\
&= n + \frac{1}{n} \sum_{i=1}^{n-1} T^{\text{avg}}(i) + \frac{1}{n} \sum_{i=0}^{n-1} T^{\text{avg}}(n-i-1) \\
&= n + \frac{1}{n} \sum_{i=0}^{n-1} T^{\text{avg}}(i) \text{ (and can forget } i = 0 \text{ since } T(0) = 0)
\end{aligned}$$

Theorem: $T^{\text{avg}}(n) \leq 2n \log_{4/3} n$ for all $n \geq 1$.

Proof. Proof by induction on n.

Base case: $n = 1$, ok, since $T(1) = 0 \leq 2 \cdot 1 \cdot \log 1$.

Inductive Hypothesis: Assume $T^{\text{avg}}(N) \leq 2N \log_{4/3} N$ for all $N < n, n \geq 2$.

Step ($n \geq 2$):

$$\begin{aligned}
T^{\text{avg}}(n) &\leq n + \frac{2}{n} \sum_{i=1}^{n-1} T^{\text{avg}}(i) \\
&\leq n + \frac{2}{n} \sum_{i=1}^{n-1} (2 \cdot i \cdot \log_{4/3} i) \\
&\leq n + \frac{4}{n} \sum_{i=1}^{\frac{3}{4}n} i \underbrace{\log_{4/3} i}_{\leq \log_{4/3} \frac{3}{4}n} + \frac{4}{n} \sum_{i=\frac{3}{4}n+1}^{n-1} i \underbrace{\log_{4/3} i}_{\leq \log_{4/3} n}
\end{aligned}$$

Recall:

$$\begin{aligned}
\log_{4/3} \frac{3}{4} n &= \log_{4/3} \frac{3}{4} + \log_{4/3} n \\
&= (\log_{4/3} n) - 1 \\
&\leq n + \frac{4}{n} \sum_{i=1}^{\frac{3}{4}n} i(\log_{4/3} n - 1) + \frac{4}{n} \sum_{i=\frac{3}{4}n+1}^{n-1} i \log_{4/3} n \\
&\leq n + \frac{4}{n} \sum i = 1^{n-1} i \log_{4/3} n - \frac{4}{n} \underbrace{\sum i = 1^{\frac{3}{4}n} i}_{\leq \frac{1}{2} \frac{9}{16} n^2} \\
&\leq n + \frac{4}{n} \frac{(n-1)n}{2} \log_{4/3} n - n \\
&\leq 2n \log_{4/3} n
\end{aligned}$$

□

Message: Quicksort is fast ($\Theta(n \log n)$) on average, but not in worst case.

Tips and tricks for Quick Sort

- **Choosing pivots**
  - Simplest idea: use $A[n-1]$ as pivot.
  - Better idea: pick middle element $A\left[\lfloor \frac{n}{2} \rfloor\right]$
  - Even better idea: median of 3. Look at $A[0]$, $A\left[\lfloor \frac{n}{2} \rfloor\right]$, $A[n-1]$. Sort them, put min/max at $A[0], A[n-1]$. Use middle as pivot.
  - Weird idea: use the median. Use faster version of Quick Select. Theoretically good runtime, but horribly slow in practice.
  - Another good idea: use a random pivot. Can argue: get same recurrence as for average case, so expected runtime $\Theta(n \log n)$
- **Reduce auxilliary space**
  - QuickSort uses auxilliary space for recursion stack, this could be $\Theta(n)$
  - Improve to $\Theta(\log n)$ by recursing on smaller side first
  - Do not recurse on bigger side. Instead, keep markers of what needs sorting and loop.
- **End recursion early**
  - Orginal code had if $(n \leq 1)$...
  - Replace by if $(n \leq 20)$
  - Find array not sorted, but items close to correct position.
  - On this input, insertion sort takes $O(n)$ time.

Lower bound for comparison sorting

- Have seen: sorting can be done in $\Theta(n \log n)$ time.
- Can we sort in $o(n \log n)$?
- Answer depends on what we allow.

We have seen many sorting algorithms:

Sort	Running time	Analysis
Selection Sort	$\Theta(n^2)$	worst-case
Insertion Sort	$\Theta(n^2)$	worst-case
Merge Sort	$\Theta(n \log n)$	worst-case
Heap Sort	$\Theta(n \log n)$	worst-case
quick-sort1	$\Theta(n \log n)$	average-case
quick-sort2	$\Theta(n \log n)$	expected
quick-sort3	$\Theta(n \log n)$	worst-case

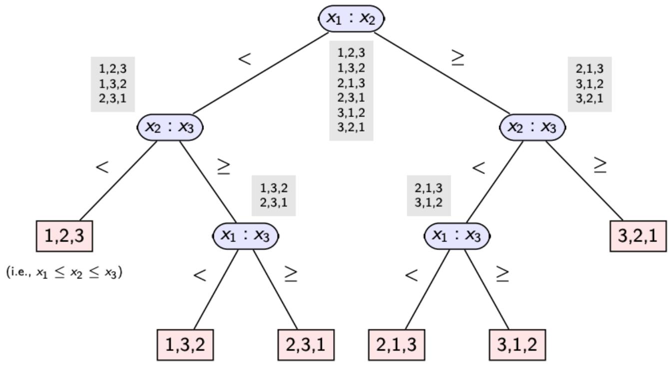
Theorem

Any comparison based sorting algorithm A use $\Omega(n \log n)$ comparisons in worst case.

"Comparison based" uses key comparisons. (i.e., questions like $A[i] \leq A[j]$ and nothing else)

We study the decision tree of A

Comparison-based algorithms can be expressed as *decision tree*.
To sort $\{x_1, x_2, x_3\}$:



- interior nodes: comparisons
- children labeled by outcome
- leaves: result returned
- depth of leaf $\equiv$ number of comparisons to get there
- worst case number of comparisons $\equiv$ length of tree

Proof of the theorem:

- there are $n!$ permutations, each gives a different result
- so at least $n!$ leaves in tree
- at least $n!$ nodes
- height $\geq \log n! \in \Omega(n \log n)$

Non-Comparison-Based Sorting

Previously, we looked at comparison based sorting that needs $\Omega(n \log n)$ comparisons. We will non look at **digits sorting**

Assumptions

- Given numbers with digits in $\{0, 1, 2, \ldots, R-1\}$
  - R is called the radix. $R = 2, 10, 16, 128, \ldots$ are most common
  - Example: $R = 4,\ A = [123, 230, 21, 320, 210, 232, 101]$
- All keys have the same number of m digits
  - In computer, $m = 32$ or $m = 64$
  - can achieve after padding with leading 0s.
  - Example :$R = 4,\ A = [123, 230, 021, 320, 210, 232, 101]$
- Therefore, all numbers are in range $\{0, 1, \ldots, R^m - 1\}$

Bucket Sort

- We sort the numbers by a single digit
- Create a “bucket” for each possible digit. Array $B[0 \ldots R-1]$ of the lists
- Copy item with digit i into bucket $B[i]$
- At the end, copy buckets in order into A

```
Bucket-sort(A, d)
A: array of size n, contains numbers with digits in {0, ..., R − 1}
d: index of digit by which we wish to sort
    Initialize an array B[0...R − 1] of empty lists
    for i ← 0 to n − 1 do
        Append A[i] at end of B[d^th digit of A[i]]
    i ← 0
    for j ← 0 to R − 1 do
        while B[j] is non-empty do
            move first element of B[j] to A[i++]
```

Example 12

Sort array A by last digit:

A
123
230
021
320
210
232
101

$\Longrightarrow$

B						
$B[0]$	$\rightarrow$ 230	$\rightarrow$ 320	$\rightarrow$ 210
$B[1]$	$\rightarrow$ 021	$\rightarrow$ 101	
$B[2]$	$\rightarrow$ 232		
$B[3]$	$\rightarrow$ 123		

$\Longrightarrow$

A
230
320
210
021
101
232
123

- This is **Stable**: equal items stay in original order.
- Run-time of sorting one digit is $\Theta(n + R)$, space $\Theta(n)$

Count Sort

- Bucket sort wastes space for `linked lists`
- **Observe:** we know exactly where numbers in $B[j]$ goes!
  - The first of them is at index $|B[0]| + \cdots + |B[j-1]|$
  - The others follows
- So compute $|B[j]|$ then copy A directly to the new array.
- count $C[j] = |B[j]|$, index $idx[j]$ = first index to put $B[j]$ into.

```
key-indexed-count-sort(A, d)
A: array of size n, contains numbers with digits in {0, ..., R − 1}
d: index of digit by which we wish to sort
// count how many of each kind there are
    count ← array of size R, filled with zeros
    for i ← 0 to n − 1 do
        increment count[d^th digit of A[i]]
// find left boundary for each kind
    idx ← array of size R, idx[0] = 0
    for i ← 1 to R − 1 do
        idx[i] ← idx[i − 1] + count[i − 1]
// move to new array in sorted order, then copy back
    aux ← array of size n
    for i ← 0 to n − 1 do
        aux[idx[A[i]]] ← A[i]
        increment idx[A[i]]
    A ← copy(aux)
```

Example 13

A			count	idx			aux
123		0	3	0		0	
230		1	2	3		1	
021		2	1	5		2	
320	$\Longrightarrow$	3	1	6	$\Longrightarrow$	3	
210						4	
232						5	
101						6	

A			count	idx			aux
123		0	3	3		0	230
230		1	2	5		1	320
021		2	1	6		2	210
320	$\Longrightarrow$	3	1	7	$\Longrightarrow$	3	021
210						4	101
232						5	232
101						6	123

Sorting multidigit numbers

- **MSD-Radix-Sort**
  - To sort large numbers, we compare leading digit, then each group by next digit, etc.

```
MSD-Radix-sort(A, l, r, d)
A: array of size n, contains m-digit radix-R numbers
l, r, d: integers, 0 ≤ l, r ≤ n − 1, 1 ≤ d ≤ m
    if l < r
        partition A[l..r] into bins according to dth digit
        if d < m
            for i ← 0 to R − 1 do
                let l_i and r_i be boundaries of ith bin
                MSD-Radix-sort(A, l_i, r_i, d + 1)
```

  - Partition using count-sort
  - **Drawback:** Too many recursions
  - Runtime: $O(m \cdot (n + R))$
- **LSD-Radix-Sort**
  - Key Insight: when $d = i$, the array is sorted w.r.t. the last $m - i$ digits
  - for $i < m$, we change order of 2 items $A[k]$ and $A[j]$ only if they have different i^{th} digit

```
LSD-radix-sort(A)
A: array of size n, contains m-digit radix-R numbers
    for d ← m down to 1 do
        key-indexed-count-sort(A, d)
```

  - Run-time for both MSD and LSD are $O(m(n + R))$
  - But LSD has cleaner code, no recursion
  - LSD looks at all digits, MSD only looks at those it needs to.

Summary

Sort	Run-time	Analysis	Comments
Insertion Sort	$\Theta(n^2)$	worst-case	good if mostly sorted; **stable**
Merge Sort	$\Theta(n \log n)$	worst-case	flexible; merge runtime useful; **stable**
Heap Sort	$\Theta(n \log n)$	worst-case	clean code; **in-place**
Quick Sort	$\Theta(n \log n)$	worst-case	**in-place**; fastest in practice
Randomized QuickSort	$\Theta(n \log n)$ $\Theta(n^2)$ $\Theta(n \log n)$	average-case worse-case expected-case	
Key-Indexed	$\Theta(n + R)$	worst-case	**stable**; need integers in $[0, R)$
Radix Sort	$\Theta(m(n + R))$	worst-case	**stable**; needs m-digit radix-R numbers

Dictionaries

ADT Dictionaries

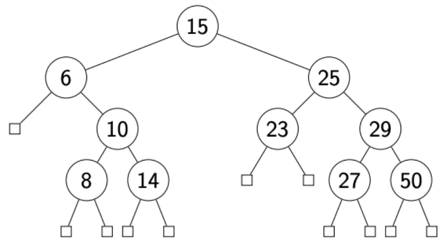
Dictionary

- A `dictionary` is a collection of items, each of which contains a `key` and some data, and is called a `key-value pair` (KVP). Keys can be compared and are (typically) unique.
- **Operations**
  - `insert(key, value)`: inserts a KVP
  - `search(key)`: returns the KVP with this `key`
  - `delete(key)`: delete the KVP from dictionary
- **Common Assumptions:**
  - All keys are distinct
  - keys can be compared in $O(1)$ time
  - KVP takes $O(1)$ space.
- **Implementations we may have seen:**
  - **Unsorted array or linked list:**
    * $\Theta(n)$ search
    * $\Theta(1)$ insert
    * $\Theta(n)$ delete
  - **Sorted array:**
    * $\Theta(\log n)$ binary search
    * $\Theta(n)$ insert
    * $\Theta(n)$ delete

Review: BST

- Either empty
- of KVP at root with left, right subtrees
  - keys in left subtree are smaller than the key at root
  - keys in right subtree are larger than the key at root
- **Insert and Search**
  - run time O(max number of level)=O(height)
  - unfortunately, height $\in \Omega(n)$ for some BSTs
- **Delete**: run time is O(height)
  - if x is a leaf, just delete it
  - if x has one child, delete it and move the child up
  - Else, swap key at x with the key at successor node and then delete that node (i.e. go right once and then go all the way left)

Example 14 BST



AVL Trees

Balanced BST

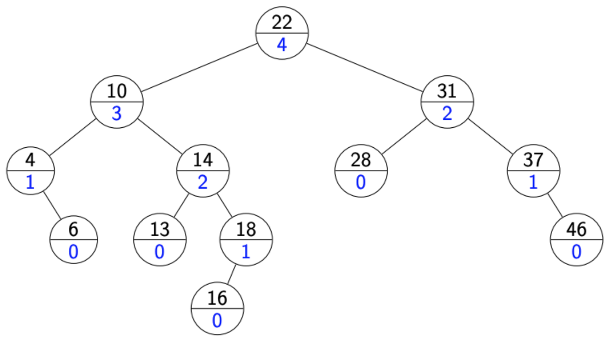
- impose some conditions on BST
- show that these guarantee the height of $O(\log n)$
- modify `insert/delete` so that they maintain these conditions

AVL Trees

- The AVL conditions: The heights of the left subtree L and right subtree R differ by at most 1.
- i.e. every node has **balance** $\in \{-1, 0, 1\}$, where balance:=height(R)-height(L)
- Note that the height of a tree is the length of the longest path from the root to any leaf, and the height of an empty tree is defined to be -1

Example 15 AVL Tree

The lower numbers indicate the height of the subtree



Theorem

Any AVL Tree has height $O(\log n)$

Proof. It's enough to show that In any AVL tree wit height h and n nodes, $h \leq \log_c n$ for some c

- **rephrase:** In any AVL tree with height h and n nodes: $c^h \leq n$
- or equivalently, It the height is h, then there must be at least c^h nodes
- Define $N(h)$ = smallest number of nodes in an AVL tree of height h. The by induction $N(h) \geq (\sqrt{2})^h$

Base case: $N(0) = 1, (\sqrt{2})^0 = 1$, $N(1) = \sqrt{2} \geq \sqrt{2}$
Inductive step

$$\begin{aligned} N(h) &= N(h-1) + N(h-2) + 1 \\ &\geq 2N(h-2) + 1 \\ &\geq (\sqrt{2})^2 \cdot (\sqrt{2})^{h-2} \\ &= (\sqrt{2})^h \end{aligned}$$

□

Insertion in AVL Trees

Insert

- do a BST insert
- move up the tree from the new node, updating heights
- as soon as we find a unbalanced node, fix via **Rotation**

```
AVL-insert(r, k, v)
    z ← BST-insert(r, k, v)
    z.height ← 0
    while (z is not null)
        setHeightFromChildren(z)
        if (|z.left.height − z.right.height| = 2) then
            AVL-fix(z) // see later
            break      // can argue that we are done
        else
            z ← parent of z
```

```
setHeightFromChildren(u)
    u.height ← 1 + max{u.left.height, u.right.height}
```

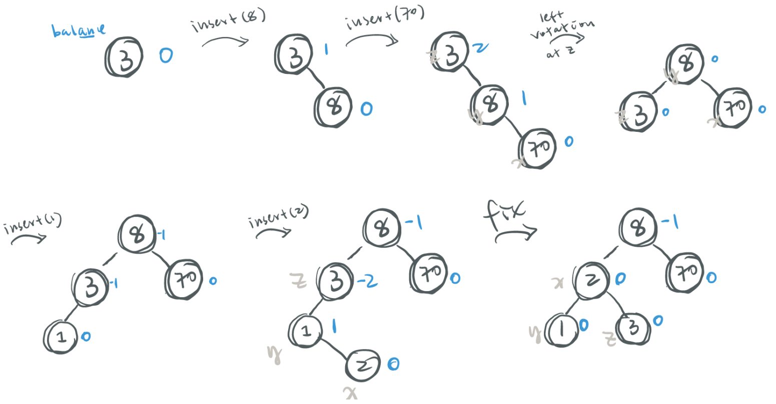
```
AVL-fix(z)
// Find child and grand-child that go deepest.
    if (z.right.height > z.left.height) then
        y ← z.right
        if (y.left.height > y.right.height) then
            x ← y.left
        else x ← y.right
    else
        y ← z.left
        if (y.right.height > y.left.height) then
            x ← y.right
        else x ← y.left
    Apply appropriate rotation to restructure at x, y, z
```

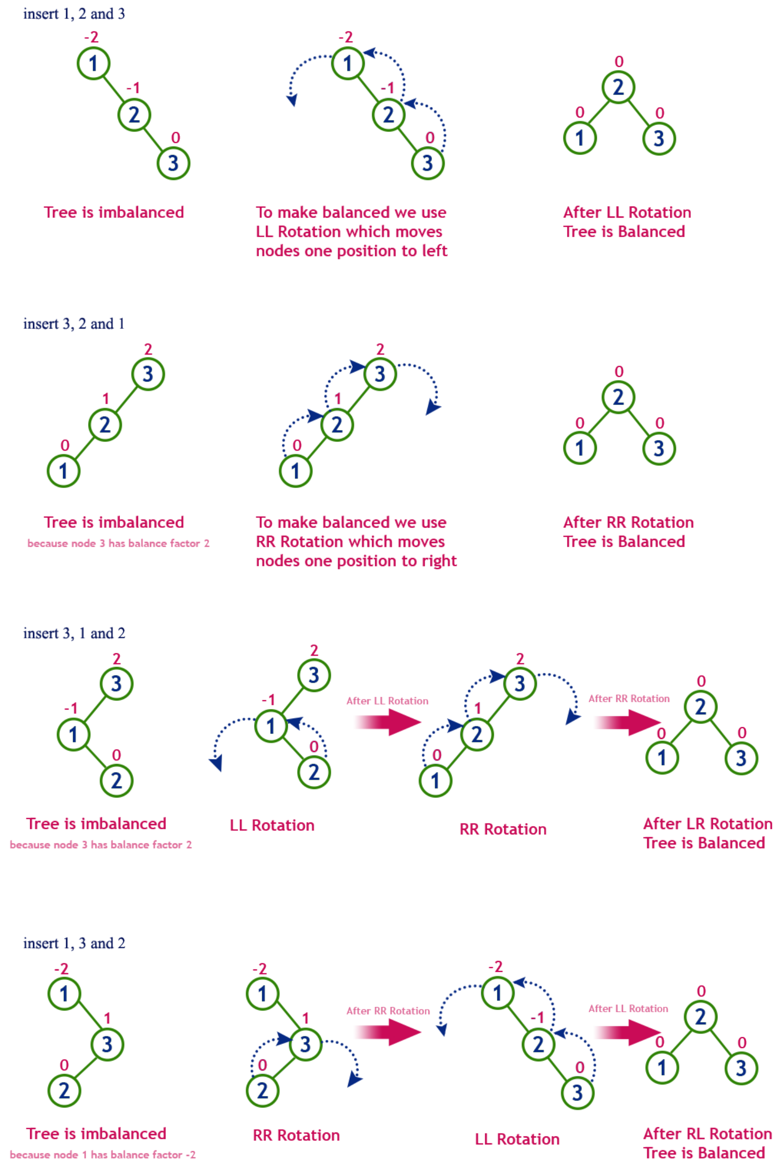
Rotations in BST

- Observe: There are many BSTs with the same set of keys
- Goad: rearrange the tree so that
  - keep ordering-property intact
  - move "bigger subtree" up
  - do only local changes $O(1)$

AVL Rotations

Example 16





$http://btechsmartclass.com/DS/U5_T2.html$,All balance factor in above pictures are inverse of the ones we define

```
AVL-fix(z)
    ... // identify y and x as before
    case
        : // Right rotation
            rotate-right(z)
        : // Double-right rotation
            rotate-left(y)
            rotate-right(z)
        : // Double-left rotation
            rotate-right(y)
            rotate-left(z)
        : // Left rotation
            rotate-left(z)
```

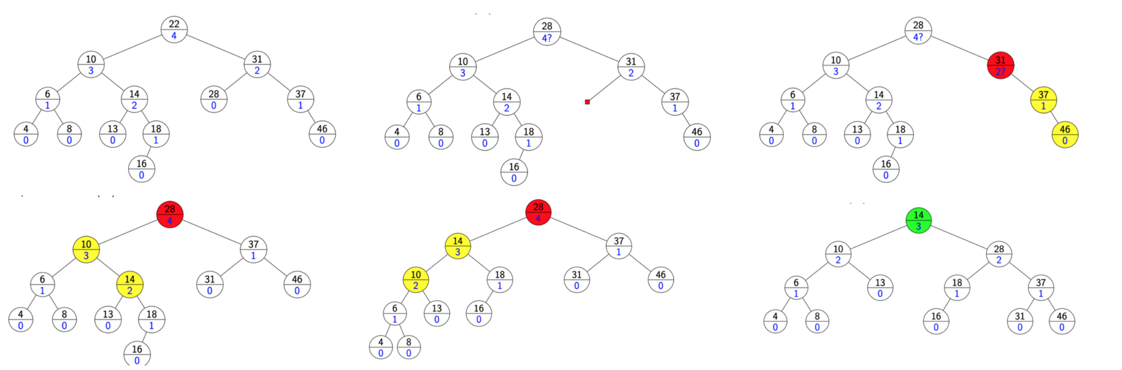
Deletion in AVL Trees

Remove the key k with `BST-delete`. We assume that `BST-delete` returns the place where structural change happened, i.e., the parent z of the node that got deleted. (This is not necessarily near the one that had k.) Now go back up to root, update heights, and rotate if needed

```
AVL-delete(r, k)
    z ← BST-delete(r, k)
    while (z is not null)
        setHeightFromChildren(z)
        if (|z.left.height − z.right.height| = 2) then
            AVL-fix(z)
        // Always continue up the path and fix if needed.
        z ← parent of z
```

Example 17 AVL-Delete

$T.Delete(22)$



AVL Tree Operations Runtime

- All of BST operations take $O(height)$
- It takes $O(height)$ to trace back up to the root updating balances
- **Calling** `AVL-fix`
  - `insert`: $O(1)$ rotations, in fact *at most once*
  - `delete`: $O(height)$ rotations

Other Dictionary Implementations

Skip List

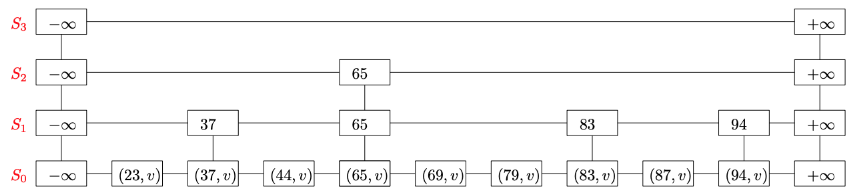
Skip List

- discovered in 1987
- randomized data structure for dictionary ADT
- competes with and always beats AVL Trees

A hierarchy S of ordered linked lists (levels) $S_0, S_1,$ ůůů, S_h:

- Each list S_i contains the special keys $-\infty$ and $+\infty$ (sentinels)
- List S_0 contains the KVPs of S in non-decreasing order. (The other lists store only keys, or links to nodes in S_0.)
- Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$
- List S_h contains only the sentinels

Example 18 Skip List



- The skip list consists of a reference to the topmost left node.
- Each node p has a reference to *after*(p), *below*(p),
- Each KVP belongs to a *tower* of nodes
- **Intuition**: $|S_i| \cong 2|S_{i+1}| \Rightarrow$ height $\in O(\log n)$
- also, we use randomization to satisfy with high probabilities

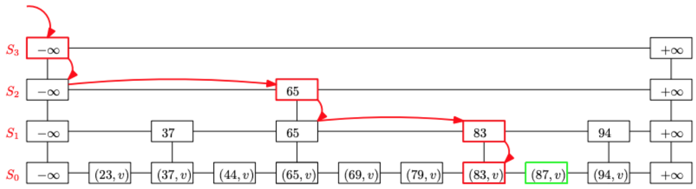
Search

- Start at the top left and move right/down as needed
- keep track of nodes at drop down location
- return a stack s
- next key of $top(s)$ is the searched key if in dictionary

```
skip-search(L, k)
    p ← topmost left node of L
    P ← stack of nodes, initially containing p
    while below(p) ≠ null do
        p ← below(p)
        while key(after(p)) < k do
            p ← after(p)
        push p onto P
    return P
```

Example 19 Skip List Search

`Skip-Search(S,87)`



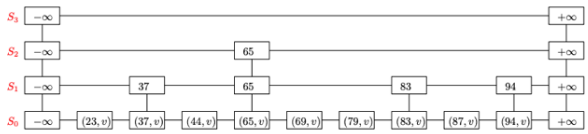
The stack contains $s = [(S_0, 83), (S_1, 83), (S_2, 65), (S_3, -\infty)]$, and the key 87 is $next(top(s))$

Insert

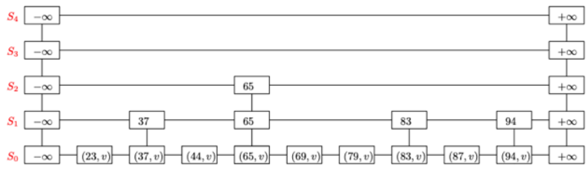
- Determine the tower height by randomly flipping a coin until get a tails
- Increase the height of the skip list if needed
- Search for key, which returns the stack of predecessors $s = [(S_0, p_0), (S_1, p_1), \ldots, (S_i, p_i)]$
- Insert the KVP(k, v) after p_0 in S_0 and insert the key k after p_j in S_j for $1 \le j \le i$

Example 20

`insert(`S`,100,`v`)` with Coin tosses: $H, H, H, T \Rightarrow i = 3$

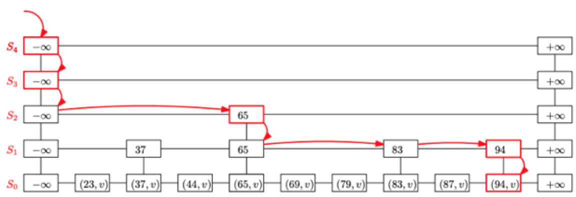


Height Increase



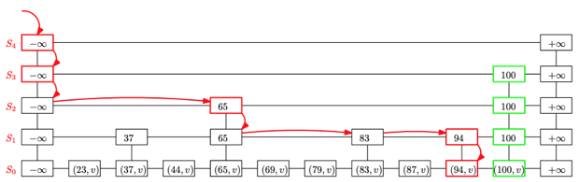
Search(S, 100)

returns



$(S_0, -\infty)$
$(S_1, -\infty)$
$(S_2, 65)$
$(S_3, 94)$
$(S_4, 94)$

Insert the KVP



Delete

- Search for the key, which returns the stack of predecessors
- Remove the items after predecessors, if they store the key
- remove duplicated layers that only have sentinels

Analysis

- Questions to ask
  1. What is the expected height?
  2. What is the expected space?
  3. How long does `search` take?
- Here, only do height bound
- Let x_k = height of the tower k = the max level that contains k, we have

$$P(x_k \geq 0) = 1, P(x_k \geq 1) = 1/2, P(x_k \geq 2) = 1/4, \ldots, P(x_k \geq i) = 1/2^i$$

$$P(\text{height } \geq i = P(\max_k\{x_k\} \geq i) \leq \sum_k P(x_k \geq i) = n\frac{1}{2^i}$$

- Therefore, we have $P(h \geq 3\log n) \leq \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2}$
- So, $P(h \leq 3\log n) \geq 1 - 1/n^2$

Summary

- Expected space usage: $O(n)$
- Expected height: $O(\log n)$
- A skip list with n items has height at most $3\log n$
- Skip-Search: $O(\log n)$ expected time
- Skip-Insert: $O(\log n)$ expected time
- Skip-Delete: $O(\log n)$ expected time
- Skip lists are fast and simple to implement in practice

Reordering Items

Dictionaries with based search-request

- Recall that *unordered array* implementation of ADT Dictionary *search:* $\Theta(n)$, *insert:* $\Theta(1)$, *delete:* $\Theta(1)$
- Arrays are a very simple and popular implementation. We can do something to make search more effective
- **80/20 rule:** 20% of items are searched for 80% of time
- Should put frequently-searched-for items in easy-to-find places
- Two scenarios:
  - We know the probabilities that keys will be accessed
  - We don't know the probabilities beforehand, but still want to adjust.

Optimal Static Ordering

- Set up: *unsorted array*: search means scan from left to right
- Intuition: Frequently searched items should be placed in front

Example 21

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

- We can measure quality via expected search cost:

$$\sum_{key\ k} \underbrace{P(\text{access } k)}_{\text{given}} \times \underbrace{(\text{cost of access } k)}_{\text{proportional to the index } k}$$

- Order **A,B,C,D,E** has expected access cost:

$$\frac{2}{26}\cdot 1+\frac{8}{26}\cdot 2+\frac{1}{26}\cdot 3+\frac{10}{26}\cdot 4+\frac{5}{26}\cdot 5=\frac{86}{26}\approx 3.31$$

- Order **D,B,E,A,C** has expected access cost:

$$\frac{10}{26}\cdot 1+\frac{8}{26}\cdot 2+\frac{5}{26}\cdot 3+\frac{2}{26}\cdot 4+\frac{1}{26}\cdot 5=\frac{66}{26}\approx 2.54$$

- colorblue Claim: Expected search-cost is minimized if sorted by **non-increasing probability**
- Proof idea: For any other ordering, exchanging two items that are out-of-order according to their access probabilities makes the total cost decrease
  - Whenever we exchanged at increasing pair, we improve
  - optimum occurs when probabilities are non-increasing

Dynamic Ordering: Move-To-Front(MTF) heuristic

- When searching or inserting a key, move it to the first position, and shuffle the rest back

A	B	C	D	E			

↓ Search(D)

D	A	B	C	E			

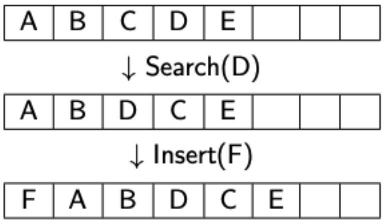
↓ Insert(F)

F	D	A	B	C	E		

- Very simple, should always do when using unsorted array/list
- **Downside:** Double runtime. (search/shuffle $O(1 + idx(k))$), but usually worth it in practice because of biases.
- Theoretical analysis for MTF is "**2-competitive**", meaning at most twice as costlt as the best possible ordering

Dynamic Ordering: Transpose

- Upon a successful search, swap the accessed item with the item immediately preceding it



- Neither strategy is better than $\Theta(n)$ in worst case so far

Dictionaries for special keys

Lower Bound

Lower bound for search

- The fastest implementations of dictionary ADT require $\Theta(\log n)$ time to search a dictionary containing n items.
- **Theorem:** Any comparison-based implementation of Dictionaries requires $\Omega(\log n)$ for search in n elements
- *Proof.* (use a decision tree argument.)
  - assume we are searching for k in items $a_1, \ldots, a_n$
  - we only use comparisons, thus can draw decision tree T
    * leaves correspond to answers returned by algorithms
    * have as lease $n+1$ possible answers ("not found" of key at position i, for $i = 1, \ldots, n$)
    * $\geq$ n+1 leaves in decision tree
    * height is as least $\log(n+1)$
    * some leaf is at level $\log(n+1)$ or lower
    * the input that leads to this answer requires $\geq \log(n+1)$ comparisons.

□

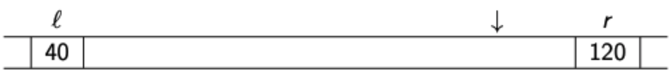
Interpolation search

Motivation

- **Requires:** diction stores integers
- Idea: use a sorted array
- **Binary search**: Compare the key with KVP at middle index:

$$m = \lfloor \frac{l+r}{2} \rfloor = l + \lfloor \frac{1}{2}(r-l) \rfloor$$

- **Interpolation search:**
  - skew where you search based on values at the dictionaries.



  - For the example above, where would be the expected key $k = 100$?
  - from left to right covers 90 integers
  - key 100 is 60 units bigger than key at l

- Should be about 3/4 down from left

- Interpolation search, like binary search, but using

$$m = l + \left\lfloor \frac{k - A[l]}{A[r] - A[l]} \dot{(}r - l) \right\rfloor$$

```
Interpolation-search(A, n, k)
A: Array of size n, k: key
    ℓ ← 0
    r ← n − 1
    while ((A[r]! = A[ℓ])&&(k ≥ A[ℓ])&&(k ≤ A[r]))
        m ← ℓ + (k−A[ℓ])/(A[r]−A[ℓ]) · (r − ℓ)
        if (A[m] < k)   ℓ = m + 1
        elsif (k < A[m])   r = m − 1
        else return m
    if (k = A[ℓ]) return ℓ
    else return "not found"
```

Runtime of interpolation search

- can be bad: $O(n) if and only id numbers are bad distributed$
- can argue that: if numbers are well-distributed then *expected size of subarray to recursion is* $O(\sqrt(n))$
- This implies expected runtime is

$$T(n) = \Theta(1) + T(\sqrt{n})$$

, and this resolves to $\Theta(\log \log n)$

Example 22 Interpolation Search

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	449	450	600	800	1000	1200	1500

`Search(449)`

- Initially $l = 0, r = 10, m = l + \left\lfloor \frac{449-0}{1500-0}(10-0) \right\rfloor = l + 2 = 2$
- $l = 3, r = 10, m = l + \left\lfloor \frac{449-3}{1500-3}(10-3) \right\rfloor = l + 2 = 5$
- $l = 3, r = 4, m = l + \left\lfloor \frac{449-3}{449-3}(4-3) \right\rfloor = l + 1 = 4$, found at $A[4]$

Tries

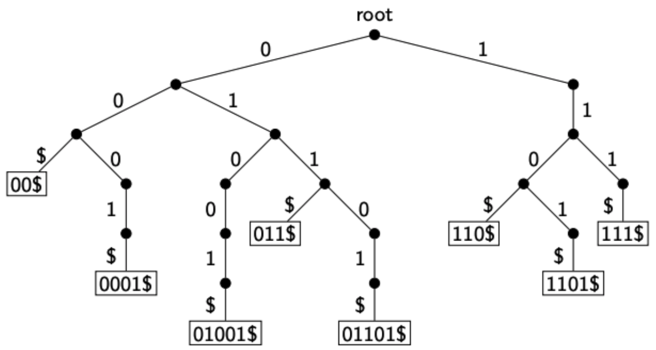
Tries: Introduction

- **Require:** keys are words (array of characters in alphabet Σ)
- study first: keys are bit string ($\Sigma = \{0, 1\}$)
- **Assumption:** Dictionary is *prefix-free*(there is on pair of binary strings in the dictionary where one is the prefix of the other)
  - **Prefix** of string $S[0 \ldots n-1]$: a substring $S[0 \ldots i]$ of S for some $0 \leq ilen - 1$
  - This is always satisfied if all strings have the same length.
  - This is always satisfied if all strings end with special 'end-of-word' character $

- Name comes form ‘retrieval’, but pronounced “try”
- **Structure of trie:**
  - Items(keys) are stored only in the leaf nodes
  - Edge to child is labeled with corresponding bit or \$

Example 23 trie

Example: A trie for
$S = \{00\$, 0001\$, 01001\$, 011\$, 01101\$, 110\$, 1101\$, 111\$\}$



Search(w)

- “**Follow character down**”
- go to child that’s labeled with next character of the word
- repeat until you reach a leaf (success), or until you get “no such child”

```
Trie-search(v ← root, d ← 0, x)
v: node of trie; d: level of v, x: word
    if v is a leaf
        return v
    else
        let c be child of v labelled with x[d]
        if there is no such child
            return "not found"
        else Trie-search(c, d + 1, x)
```

Insert(w)

- Search for w. This should give “no such child $w[i]$’ for some i’
- For $j = i + 1, \ldots, |w|$, create child with edge labeled $w[j]$
- place w(and the KVP) in leaf
- **Runtime for search and insert**
  - we are handling $|w|$ nodes, thus $O(|w|)$ time
  - this is amazing as search/insert are independent of the number of stored words

Delete(w)

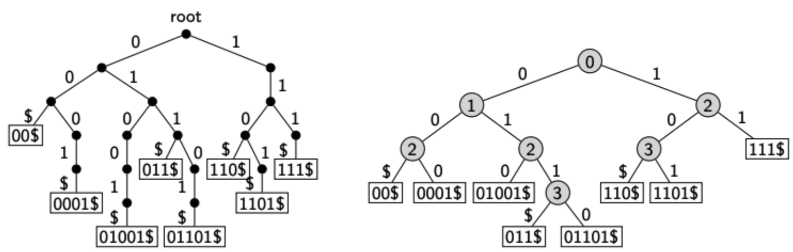
- First search of w, this gets us to the leave l_w
- delete l_w, go back up and delete nodes until we get one node with ≥ 2 children
- runtime: $O(|w|)$

Compressed Tries(Patricia Tries)

Compressed Tries

- Idea: omit all comparisons that did not split the subtree
  - stop splitting if there is only on key left in the subtree
  - skip comparisons if answer is the same for all keys in the subtree. This is same as omitting nodes with only one child
- **previously,** node on level compares by $w[i]$,
- **now,** nodes must store index of character to compare with
- this is hard for human to read, but is **space saving:** have at most $n-1$ internal nodes for n words in dictionary
- This is called a **PATRICIA Trie**: Practical Algorithm to Retrieve Information Coded in Alphanumeric
- Patricia Trie is unique for given set of keys

Example 24 Patricia trie



Compressed Tries: Search

- Much like for normal tries, but with 2 changes
- when choosing child, use the character indicated by node
- if we reach leaf l, must do string-comparison with stored words on the leaf

```
Patricia-Trie-search(v ← root, x)
v: node of trie; x: word
    if v is a leaf
        return strcmp(x, key(v))
    else
        let d be the bit stored at v
        let c be child of v labelled with x[d]
        if there is no such child
            return "not found"
        else Patricia-Trie-search(c, x)
```

Compressed Tries: Insert and Delete

- **Delete(w)**
  - Perform Search(w)
  - Remove the node v that stored x
  - compress along path to v whenever possible

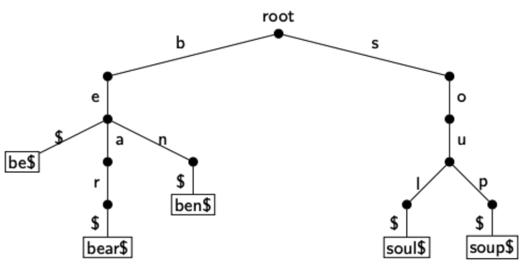
- **Delete(w)**
  - Perform Search(w)
  - Let v be the node where the search ended. item Uncompress oath from root to v, insert w and then compress paths from root to v and from root to x
- **All operations take $O(|w|)$ time.**

Multiway Tries: Larger Alphabet

- **Main Question:** how to store the references to children
- each node has up to $|\Sigma| + 1$ children
- *Simplest:* store array child $[0, \ldots, |\Sigma|]$. This takes $O(1)$ time to find, but wastes space
- *Good for place method:* store a list of children. Constant space overhead, but slow to find (**Use MTF**)

Example 25 multiway tries

A trie holding strings $\{bear\$, ben\$, be\$, soul\$, soup\$\}$



Dictionaries via Hashing

Range-Searching in Dictionaries for Points

String Matching

Introduction

Motivation

- **How does `Grasp` or `Ctrl-f` work?**
- Given text $T[0 \ldots n-1]$(*haystack*) of chars and pattern $P[0 \ldots m-1]$(*needles*) of chars. Does P occur as a substring of T?
- Our goal is to find the first occurrence of P in T or `FAIL`

Terminology

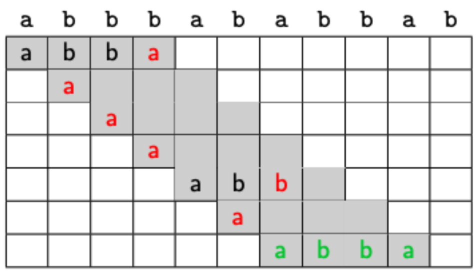
- **substring** of T: Any string that equals $T[i \ldots j]$ for some $0 \leq i \leq j \leq n-1$
- **prefix** of T: Any string that equals $T[0 \ldots j]$ for some $0 \leq j \leq n-1$
- **suffix** of T: Any string that equals $T[i \ldots n-1]$ for some $0 \leq i \leq n-1$
- **empty string** Λ is considered a substring, prefix and suffix of T
- A **guess** is a position such that $T[i \ldots i+m=1]$ could be P
- Guess i is an **occurrence** if $T[i \ldots i+m-1] = P[0 \ldots m-1]$

Brute Force Algorithm

- **Simplest idea:** for any possible guess, check whether this is an occurrence. i.e. *try guess* $i = 0, 1, 2 \ldots, n-m$
- example of worst case: $P = aaab, T = aaaaaaaaaaa$

Example 26 Brute Force

$T = abbbababbab, P = abba$



- Brute force algorithm has run-time $(n-m+1) \cdot m$ comparisons$\Theta(mn) \in \Theta(n^2)$ if $m \approx n/2$

KMP Algorithm

Algorithm

- Knuth-Morris-Pratt algorithm (1977)
- achieves $O(n)$ runtime in worst-case
- compares pattern from left to right
  - we call the check index i, checking $T[i]$)
  - j is the index of $P[j]$ corresponds to i on T
  - then the **guess index** is $i-j$
  - in fact, both $i-j$ and i are monotonically non-decreasing
- shifts the pattern more intelligently than the brute-force algorithm
- *When a mismatch occurs, what is the most we can shift the pattern by reusing knowledge from the previous match*
- **ANS:** the largest *prefix* of $P[0...j]$ that is a suffix of $P[1..j]$

KMP Failure array

- **Main ideal of KMP:** if we keep i the same, then *decrementing* j is the same as *shifting the pattern to the right*
- **Failure Array:** $F[j]$ for pattern $P[0..m-1]$ is defined as
  - $F[0] = 0$
  - for > 0, $F[j]$:=length of the largest prefix of $P[0..j]$ that is suffix of $P[1..j]$

Example 27 failure array

$P = abacaba$

j	$P[1..j]$	P	$F[j]$
0	—	abacaba	0
1	b	abacaba	0
2	ba	abacaba	1
3	bac	abacaba	0
4	baca	abacaba	1
5	bacab	abacaba	2
6	bacaba	abacaba	3

Example 28 usage of failure array

							i	
T =	...	a	b	a	c	a	a	...
P =		a	b	a	c	a	b	a
							j	

* Now, $T[i]$ doesn't match with $P[j = 5]$,
* from the fact that $P[0 \ldots 4]$ is matched with text T and the failure array $F[4]$, we know that the longest suffix of $P[1..4] = baca$ that is also a prefix of $P[0..4] = abaca$ is the substring a, i.e. length = 1.
* as the pattern $P[0..4]$ matched with the text, the new shift will be a potential match only if the right shift allows the overlapping part of pattern P before and after the shift to be matched.
* In this case, $F[j-1] = F[4] = 1$ tells us we can shift the pattern so that the suffix a matches with the prefix a of $abaca$. i.e. shift the pattern by $5 - F[4] = 4$, which is the same as decrementing j by 4, which is the same as setting $j = F[4]$

							i					
T =	...	a	b	a	c	a	a	...	...	...	...	...
		a	b	a	c	a	b	a				
P =						(a)	b	a	c	a	b	a
							j = 1					

```
KMP(T, P)
T: String of length n (text), P: String of length m (pattern)
    F ← failureArray(P)
    i ← 0
    j ← 0
    while i < n do
        if T[i] = P[j] then
            if j = m − 1 then
                return i − j //match
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0 then
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 // no match
```

Boyer-Moore Algorithm

Three ideas

- **Reverse-order searching:** Compare P with a subsequence of T in reverse order
- **Bad character jumps:** When a mismatch occurs at $T[i] = c$
  - if P contains c, we can shift P to align the last occurrence of c in P with $T[i]$
  - otherwise, if the pattern doesn't contain c, we can shift P to align $P[0]$ with $T[i+1]$ ($P[m-1]$ with $T[i+m]$)
- **Good suffix jumps:** If we have already matched a suffix of P, then get a mismatch, we can shift P forward to align with the previous occurrence of that suffix, similar to *failure array*

Example 29 Bad Character(1)

P = a l d o
T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								

- starting from the first possible guess index $i - j = 0$, we have $i = 3, j = 3$, $T[i] \neq P[j]$ tells us that we get a dismatch at $T[i] = r$
- However, as there is no occurrence of r in P, the bad character jump tells us there will never be a match if any character of P aligns with $T[i]$. i.e we can shift the pattern by $|P|$ and start over.
- This can be done by $i := i + m - 1 - (-1) = m, j := m - 1$

P = a l d o
T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
											o

P = a l d o
T = w h e r e i s w a l d o

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

6 comparisons in total

Example 30 Bad Character(2)

P = m o o r e

b	o	y	e	r	m	o	o	r	e
				e					

- Now we have a dismatch at $i = 4, T[i] = r$
- The character r appears in P, and the last occurrence is $P[3] = r$, therefore we can align $P[3]$ with $T[i]$
- This can be done by $i := i + m - 1 - 3 = i + 1, j := m - 1$

P = m o o r e

b	o	y	e	r	m	o	o	r	e
				e					
				[r]	e				

- Now check from the back again, we have $T[i = 5] = m \neq P[j = 4]$
- The character m does occur in P, and the last occurrence is $P[0] = m$, therefore we can align $P[0]$ with $T[i]$
- This can be done by $i := i + m - 1 - 0 = 9, j := m - 1$

P = m o o r e

b	o	y	e	r	m	o	o	r	e
				e					
				[r]	e				
					[m]	o	o	r	e

7 comparisons in total

Example 31 Good Suffix(1)

P = sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							

Here we have 4 characters matched, and find a dismatch at $i = 7, j = 7, T[i] = s, P[j] = h$

- To utilize the fact that the 4 characters have been matched with the text T, we can notice that the suffix $P[j+1..m-1] = P[8..11] = ells$ matches with $P[0+1..0+11-7] = P[1..4]$, also, since $P[0] \neq P[7]$, we can shift the patter right so that $P[0]$ aligns with $T[i = 7]$
- This can be done by $i := i + m - 1 - 0 = 18, j := m - 1$

P = sells␣shells

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s							
							×	(e)	(l)	(l)	(s)							

Last-Occurrence Function

- *Preprocess* the patter P and the alphabet Σ

- Build the *last-occurrence function* L mapping Σ to integers
- $L(c)$ is defined as
  - the largest index i such that $P[i] = c$, or
  - -1 if no occurrence of c in P
- *Example:* $\Sigma = \{1, b, c, d\}, P = abacab$

c	a	b	c	d
L(c)	4	5	3	-1

- It can be computed in $O(m + |\Sigma|)$

Suffix skip array

- *Preprocess* the patter P and the alphabet Σ
- *Suffix skip array* S: When we have a dismatch at $P[i]$ whereas $P[i+1..m-1]$ matches, we shift the pattern right by positive but minimum amount so that:
  (A) any pattern characters after $P[i]$ still matches up
  (B) $P[j]$ doesn't match with $P[i]$
- then $S[i] = j = i-$shift amount

Example 32 Suffix Skip Array

Compute $S[0\ldots7]$ for $P = bonobobo$

- $S[6]$

i=6

b	o	n	o	b	o	b	o						
	b	o	n	o	b	o	b	o					
		b	o	n	o	b	o	b	o				
			b	o	n	o	b	o	b	o			
				b	o	n	o	b	o	b	o		

  - shift 1: (A) not satisfied
  - shift 2: (B) not satisfied
  - shift 3: (A) not satisfied
  - shift 4: both satisfied!!
  - return $S[6] = i - 4 = 2$
- $S[3]$

i=3

b	o	n	o	b	o	b	o						
	b	o	n	o	b	o	b	o					
		b	o	n	o	b	o	b	o				
			b	o	n	o	b	o	b	o			
				b	o	n	o	b	o	b	o		
					b	o	n	o	b	o	b	o	
						b	o	n	o	b	o	b	o

  - shift 1/2/3/4/5: (A) not satisfied
  - shift 6: both satisfied!!
  - return $S[3] = i - 6 = -3$

i	0	1	2	3	4	5	6	7
P[i]	b	o	n	o	b	o	b	o
S[i]	−6	−5	−4	−3	2	−1	2	6

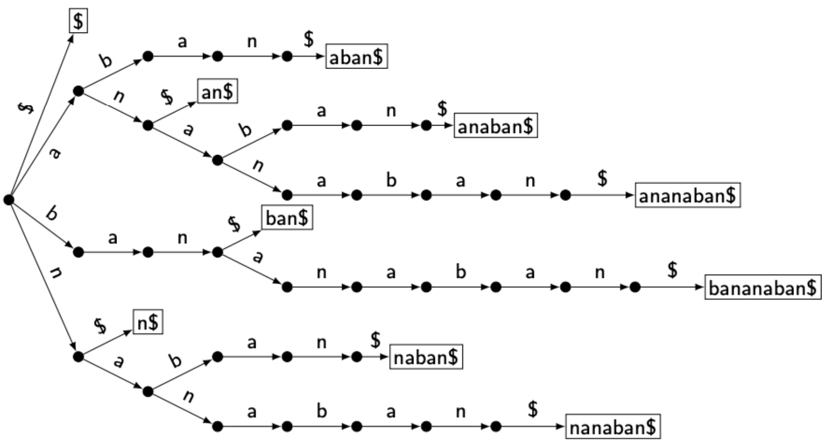
Suffix Tree

- KMP and Boyer-Moore: Preprocess P then work on T
- **Suffix tree:** Preprocess T, the work on P
- **Observation:** P is a substring of$T$ iff P is a prefix of a suffix of T
- **Idea:** Store all suffixes of T in a trie that allow us to search for P as a prefix. *P is in T $\Leftrightarrow$ search for P in trie ends up at a node*
- **Better idea:** to save space, store all suffixes of T in a compressed trie where leaf labels use indices of T

Example 33 suffix tree

T = bananaban has suffixes

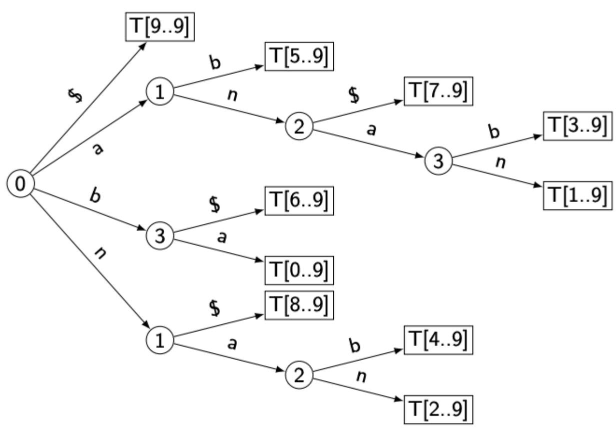
{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, Λ}



Example 34 suffix tree compressed

Suffix tree: Compressed trie of suffixes

	0	1	2	3	4	5	6	7	8	9
$T =$	b	a	n	a	n	a	b	a	n	$



Construction

- fairly easy in $O(n^2)$ time: $n \times \text{insert} \in O(n)$
- feasible in $O(n)$, but complicated

String Matching with Suffix Trees

- Search (as in compressed trie) until P runs out of character or reach a leaf.
  - **if** we get "no child" the P is not in T
  - **else**, go from the node that we stopped to any leaf l in subtree
  - leaf l tells index i, where P may in T, then compare $T[i, \ldots, i + m + 1] == P$ explicitly

Conclusion

	Brute-Force	KMP	Boyer-Moore	Suffix trees
Preprocessing:	–	$O(m)$	$O(m + \|\Sigma\|)$	$O(n^2)$
Search time:	$O(nm)$	$O(n)$	$O(n)$ (often better)	$O(m)$
Extra space:	–	$O(m)$	$O(m + \|\Sigma\|)$	$O(n)$

Compression

Encoding Basics

Data Storage and Transmission

- **Source text**: The original data, string S of characters from the source alphabet Σ_S
- **Coded text**: The encoded data, string C of characters from the coded alphabet Σ_C, *Typically,* $\Sigma_C = \{0, 1\}$
- ***Goal***: Want to compress (make C smaller than S)
- Encoding schemes that try to minimize the size of the coded text perform *data compression*. We will measure the *compression ratio:*
$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$
- Compression must be ***lossless***: can recover S without error from C.
- there fore we need two algorithms:
  - **compress** (encoding)
  - **decompress** (decoding)
  - Main objective: coded text should be short
  - Second objective: fast encoding/decoding

Character-by-Character encodings

- simplest method: assign each $c \in \Sigma_S$ to a code word in Σ_C^*
- **Definition:** Map each character from the source alphabet to a sting in coded alphabet
$$E : \Sigma_S \to \Sigma_C^*$$
- we call $E(c)$ the codeword of c
- **Example:** ASCII(American Standard Code for Information Interchange)

char	null	start of heading	start of text	end of text	...	0	1	...	A	B	...	$\sim$	delete
code	0	1	2	3	...	48	49	...	65	66	...	126	127

  - ASCII uses 7 bits to encode 128 characters, but not enough
  - **ISO-8859**: 8 bits = 1 byte, handles most western languages.
  - **Unicode UCS-2**: 16 bits, handles a lot more, but huge encoding and still not enough

Fixed-length encoding

- All codewords have the same length
- e.g. ASCII, ISO8859, Unicode, etc.
- k bits can represent 2^k different items
- *rephrase:* we need $\lceil \log n \rceil$ bits to represent n different items
- *Fact:* Fix-length encoding of test S of length n over Σ_S needs $n \lceil \log |\Sigma_S| \rceil$ bits

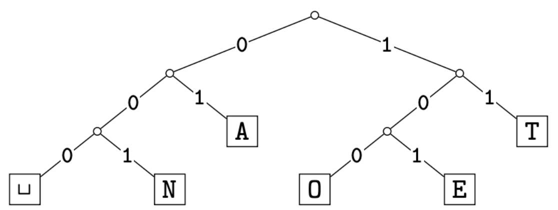
Variable-length encoding

- Different codewords have different lengths

> **Example 35 variable-length encoding**
>
> Consider $\Sigma_S = \{S, Y, N, E, O\}, \Sigma_C = \{0, 1\}$, and encoding
>
> $$S = 11, Y = 01, N = 0110, E = 1010, O = 1011$$
>
> - all codewords are distinct
> - Answer to question: should we attack: 01101011
>   * $\overbrace{01}^{Y}\overbrace{1010}^{E}\overbrace{11}^{S}$ = YES
>   * $\underbrace{0110}_{N}\underbrace{1011}_{O}$ = NO
> - Therefore, we really need ***prefix code***$\equiv$ no code word is prefix of any other (should really be called *prefix-free code*)

Prefix codes:

- no codword is prefix of another $\Leftrightarrow$ encoding trie has no data at interior nodes
- **Encoding:** Loop up codewords in dictionary and append to output
- **Decoding:** Parse bit-string while going down trie. At leaf output characters, the restart from the root

> **Example 36 prefix-free**
>
> Code as table:
>
> | $c \in \Sigma_S$ | ␣ | A | E | N | O | T |
> |---|---|---|---|---|---|---|
> | $E(c)$ | 000 | 01 | 101 | 001 | 100 | 11 |
>
> Code as trie:
>
> 
>
>
> - Encode $AN\_ANT \rightarrow 010010000100111$
> - Decode $111000001010111 \rightarrow TO\_EAT$

Huffman's algorithm

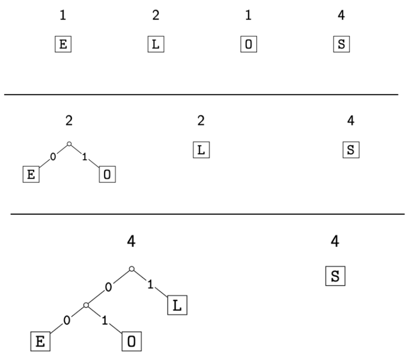
- *Which prefix code is the best?*
- **Overall goal**: Find an encoding that is short
- **Idea:** Frequent characters should have shorter codewords
- **Rephrase:** Infrequent characters should have large depth in trie.

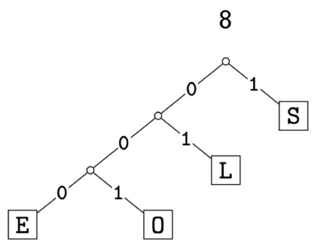
Huffman's algorithm

- For each character $c \in \Sigma_S$
  - compute frequency $= f(c) =$ # of times c appears in S
  - create a 1-node trie containing only c with weight $f(c)$
- Find and remove 2 tries with minimum weight.
- Merge these tries with new interior node; new wight is the sum.
- Repeat until only 1 trie left

Example 37

text S= LOSSLESS, $\Sigma_S = \{L, O, S, E\}$ frequencies: $E : 1$, $L : 2$, $O : 1$, $S : 4$





$LOSSLESS \rightarrow 01001110100011$
compression ratio:

$$\frac{14}{8 \cdot \log 4} = \frac{14}{16} \approx 88\%$$

```
Huffman-Encoding(S[0..n−1])
S: text over some alphabet Σ_S
    f ← array indexed by Σ_S, initially all-0
    for i = 0 to n − 1 do f[S[i]]++
    Q ← min-oriented priority queue that stores tries
    for all c ∈ Σ_S with f[c] > 0
        Q.insert(single-node trie for c with weight f[c])
    while Q.size() > 1
        T_1 ← Q.deleteMin(), T_2 ← Q.deleteMin()
        Q.insert(trie with T_1, T_2 as subtries and weight w(T_1)+w(T_2))
    D ← Q.deleteMin // decoding trie
    C ← PrefixFreeEncodingFromTrie(D, S)
    return C and D
```

Runtime of Huffman's Algorithm

- use min-oriented priority queue to store tries
- Time to compute frequency $O(n)$
- Time to compute D is $O(\Sigma_S \cdot \log |\Sigma_S|)$
  - $|\Sigma_S|$ insertions, so D has size $\leq |\Sigma_S|$
  - while loop executes $|\Sigma_s - 1|$ times
- Time to encode text is $O(|S| +$ length of encoding$)$
- overall: $O(n + |\Sigma_S| \log |\Sigma + S| +$ length of encoding$)$

Discussion on Huffman:

- **Thm:** Any trie created by Huffman's algorithm gives the shortest possible encoding for S
- trie is not unique, even for fixed frequencies.
- must include trie with encoded text
- for typical English text, Huffman reduces by about 60%
- Two pass algorithm
- Huffman gives the best possible encoding when *frequencies are known and independent.* That is best we can do when encoding one character at a time

Run-Length Encoding

Recall: Huffman

- *Variable-length prefix encoding*
- Huffman is optimal if:
  - Frequencies are known and independent
  - Encodes only **one** character at a time

Multi-character encoding

- **Main Idea:**Encode multiple characters with one codeword.
- Examples:
  1. **Run-length encoding:** Use substrings that use only one character
  2. **Lempel-Ziv:** Use longest substring we've seen and extend

Run-length encoding

- Input S must be a bit string
- a "**run**" $\equiv$ maximal substring that has the same bits
- **Encoding Idea:**
  - Give the first bit of S (either 0 or 1)
  - Then gives a sequence of integers indicating run lengths
  - We don't have to git the bits for runs since they alternate
  - *For example,* if $S = \underbrace{00000}_{5}\underbrace{111}_{3}\underbrace{0000}_{4}$, then we can represent S as $0, 5, 3, 4$
  - **The problem is:** We want a **bit string** as output. **Question:**How to encode the integers as output?
  - Recall: binary length of positive integer k is $\lfloor \log k \rfloor + 1$
  - Use *Elias gamma code* to encode k, it allows us to encode length k of run unambiguously with $2\lfloor \log k \rfloor + 1$ bits.
- **For prefix free encoding:**
  - Write $\lfloor \log k \rfloor$ 0's
  - the write down the binary encoding of k
- **For decoding**
  - Don't forget to treat the first bit separately.
  - Count how many 0's are there $\to l$
  - Then parse the next $(l + 1)$ bits, decode the bits as binary

Example 38 encoding of k

Encoding example

k	$\lfloor \log k \rfloor$	k in binary	encoding
1	0	1	1
2	1	10	010
3	1	11	011
4	2	100	00100
5	2	101	00101
6	2	110	00110
⋮	⋮	⋮	⋮

```
RLE-Encoding(S[0...n−1])
S: bitstring
    initialize output string C ← S[0]
    i ← 0
    while i < n
        k ← 1
        while (i + k < n and S[i + k] = S[i])
            k++
        for (ℓ = 1 to ⌊log k⌋)
            C.append(0)
        C.append(binary encoding of k)
        i ← i + k
    return C
```

```
RLE-Decoding(C)
C: stream of bits
    initialize output string S
    b ← C.pop() // bit-value for the current run
    while C has bits left
        ℓ ← 0
        while C.pop() = 0   ℓ++
        k ← 1
        for (j = 1 to ℓ)   k ← k ∗ 2 + C.pop()
            // if C runs out of bits then encoding was invalid
        for (j = 1 to k)   S.append(b)
        b ← 1 − b
    return S
```

Example 39 RLE

- **Encode example** $S = 11111110010000000000000000000011111111111$

$$C = 1\underbrace{00111}_{7}\underbrace{010}_{2}\underbrace{1}_{1}\underbrace{000010100}_{20}\underbrace{0001011}_{11}$$

Compression Ration: $26/41 \approx 63\%$

- **Decoding example** $C = 00001101001001010$

$$C = 0\underbrace{0001101}_{8\text{ 0's}}\underbrace{00100}_{4\text{ 1's}}\underbrace{1}_{1\text{ 0's}}\underbrace{010}_{2\text{ 1's}}$$

$$S = 000000001111011$$

RLE Properties:

- An all-0 string of length n would be compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits
- Usually, we are not that lucky
  - No compression until run-length $k \geq 6$
  - **Expansion** when run-length k =2 or 4
- Methods can be adapted to larger alphabet sizes
- Used in some image formats (e.g. TIFF)

Lempel-Ziv-Welch

- Huffman and RLE mostly take advantage of frequent or repeated single characters.
- **Observation:** Certain *substrings* are much more frequent than others

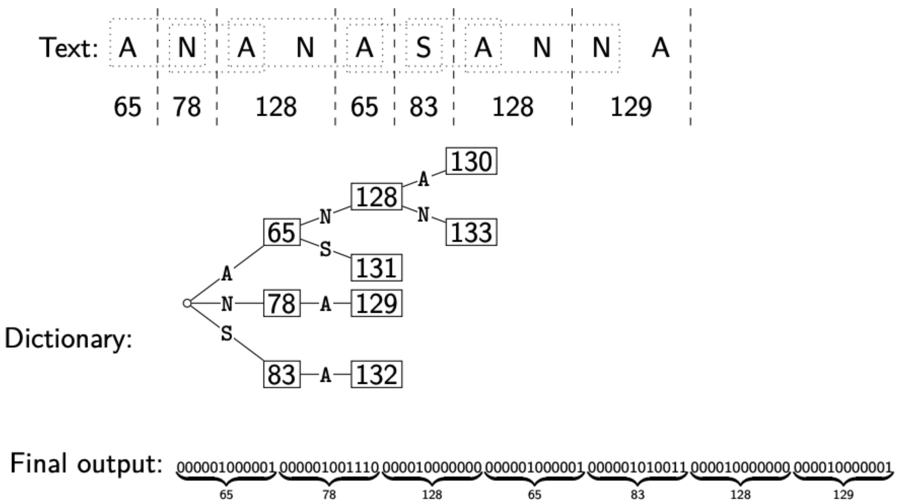
Adaptive encoding

- non-static encoding
- assign fixed length codes to common sequence of > 1 ASCII characters
- automatically detects frequently used character combinations
- **Advantages of Lempel-Ziv-Welch**
  1) only one pass requires (streaming algorithm e.g. “zgrep” in Linux)
  2) typically better compression than Huffman

LZW encoding

- Start with dictionary D as a trie that stores ASCII(usually)
- **Main idea:** always encode longest substring that is already in the dictionary, then add one-longer to D, using the codeword 128,129, ...
- Encoding
  > parse text along until we find the longest prefix in D, i.e. get stuck at a node l
  > output codeword stored at l
  > add child at l with next character in the text
  > repeat until end of input
  > This gives a list of numbers. This is usually converted into bit-string with fixed-width encoding using 12 bits. (*This limits the codewords to 4096*)
- **overall time:** O(length of the input string)
  - time to find l is proportional to encoded substring S
  - inserting new item at l takes $O(1)$

Example 40 LZW Encode



LZW decoding

- Decoder builds some dictionary while decoding.
- First few codes are ASCII
- To save space, store string as **code of prefix + one character**
- Example: 67, 65:
  - From ASCII, 67 corresponds to C, thus 67 is decoded into C.
  - From ASCII, 65 corresponds to A, thus 65 is decoded into A.
  - Then, insert next KVP $(128, CA)$ into the dictionary(in the form 67, A)
- It is possible that we are given a code that we don't know yet. Notice that it only happens when the decoder is **"one step behind"**
  - This problem only occurs if we want to use a code that we are about to build.
  - But then we know what is going on!
    * Input: codeword k at the time when we are about to assign k
    * Decoder knows s_{prev} = string decoded in the previous step.
    * Let s be the next string that the decoder will insert, i.e. the next string that will occur in the source text
    * We know that decoder is about to assign next key to $s_{prev} + s[0]$
    * Since $s[0]$ represents the first character of what k decodes to
    * We have $s = s_{prev} + s_{prev}[0]$

```
LZW-encode(S)
S : stream of characters
1.    Initialize dictionary D with ASCII in a trie
2.    idx ← 128
3.    while there is input in S do {
4.        v ← root of trie D
5.        K ← S.peek()
6.        while (v has a child c labelled K)
7.            v ← c; S.pop()
8.            if there is no more input in S break    (goto 10)
9.            K ← S.peek()
10.       output codenumber stored at v
11.       if there is more input in S
12.           create child of v labelled K with codenumber idx
13.           idx++
14.   }
```

```
LZW-decode(C)
C: stream of integers
1.    D ← dictionary that maps {0, ..., 127} to ASCII
2.    idx ← 128
3.    S ← empty string
4.    code ← first code from C
5.    s ← D(code); S.append(s)
6.    while there are more codes in C do
7.        s_prev ← s
8.        code ← next code of C
9.        if code = idx
10.           s ← s_prev + s_prev[0]
11.       else
12.           s ← D(code)
13.       S.append(s)
14.       D.insert(idx, s_prev + s[0])
15.       idx++
16.   return S
```

Example 41 LZW decoding

Example:67 65 78 32 66 129 133 83

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	???	133		

- Now we encounter 133 as a codeword that we have never seen before
- we know that 133 is the KVP that we are about to insert.
- the string that we are about to insert is s_{prev} plus one more character of the rest
- Also, this means in the current step, we want to use this code $s = s_{prev} + c$, meaning the first character of the rest is $s[0] = s_{prev}[0] = A$
- Therefore, 133 corresponds to the string ANA

$D =$

Code #	String
...	
32	␣
...	
...	
65	A
66	B
67	C
...	
78	N
...	
83	S
...	

input	decodes to	Code #	String (human)	String (computer)
67	C			
65	A	128	CA	67, A
78	N	129	AN	65, N
32	␣	130	N␣	78, ␣
66	B	131	␣B	32, B
129	AN	132	BA	66, A
133	ANA	133	ANA	129, A
83	S	134	ANAS	133, S

LZW summary

- Can be very bad (when no repeated substring), in this case encoded text may get bigger
- best case in theory $O(\sqrt{n})$ codes for the text of length n
- In practice, compression ratio $\approx 50\%$

Compression Summary

Huffman	Run-length encoding	Lempel-Ziv-Welch
variable-length	variable-length	fixed-length
single-character	multi-character	multi-character
2-pass	1-pass	1-pass
60% compression on English text	bad on text	45% compression on English text
optimal 01-prefix-code	good on long runs (e.g., pictures)	good on English text
must send dictionary	can be worse than ASCII	can be worse than ASCII
rarely used directly	rarely used directly	frequently used
part of pkzip, JPEG, MP3	fax machines, old picture-formats	GIF, some variants of PDF, Unix compress

bzip2

bzip2 overview

- To achieve even better compression, bzip2 uses **text transformation:** Change input into a different text so that is not necessarily shorted, but that has other desirable qualities

text T_0		
↓	Burrows-Wheeler transform	If T_0 has repeated longer substrings, then T_1 has long runs of characters.
text T_1		
↓	Move-to-front transform	If T_1 has long runs of characters, then T_2 has long runs of zeros.
text T_2		
↓	Modified RLE	If T_2 has long runs of zeroes, then T_3 has chars $0'$ and $1'$ very frequently
text T_3		
↓	Huffman encoding	Compresses well since input-chars unevenly distributed
text T_4		

Move-to-front transformation

- have adaptive dictionary D throughout encoding.
- do not add to dictionary, but change order with MTF heuristic
- **Idea:** transform text so that we have long runs of chars

```
MTF-encode(S)
    L ← array with Σ_S in some pre-agreed, fixed order
    while S has more characters do
        c ← next character of S
        output index i such that L[i] = c
        for j = i − 1 down to 0
            swap L[j] and L[j + 1]
```

```
MTF-decode(C)
    L ← array with Σ_S in some pre-agreed, fixed order
    while C has more characters do
        i ← next integer from C
        output L[i]
        for j = i − 1 down to 0
            swap L[j] and L[j + 1]
```

Burrows-Wheeler Transformation

- BWT also permutes input text, and is **reversible**
- Require the source text S ends with end-of-word character \$ that occurs nowhere else in S

Encoding

1. write down all cyclic shifts
2. Sort cyclic shifts
3. Extracts last characters form sorted shifts

Fast Decoding

- **Idea:** Given C, we can reconstruct the first and last column of the array of cyclic shifts by sorting
  1. Last Column: C
  2. First Column: C sorted
  3. Disambiguate by row index
  4. Starting from \$, recover S

Example 42 BWT decoding

C=$ard\$rcaaaabb$

```
$.........a          $,3.........a,0
a.........r          a,0.........r,1
a.........d          a,6.........d,2
a.........$          a,7.........$,3
a.........r          a,8.........r,4
a.........c          a,9.........c,5
b.........a         b,10.........a,6
b.........a         b,11.........a,7
c.........a          c,5.........a,8
d.........a          d,2.........a,9
r.........b          r,1.........b,10
r.........b    ⇒     r,4.........b,11
```

$S = abracadabra$

```
BWT-decoding(C[0..n − 1])
C : string of characters over alphabet Σ_C
    A ← array of size n
    for i = 0 to n − 1
        A[i] ← (C[i], i)
    Stably sort A by first entry
    S ← empty string
    for j = 0 to n
        if C[j] = $ break
    repeat
        j ← second entry of A[j]
        append C[j] to S
    until C[j] = $
    return S
```