

CS 240 with Olga Veksler

Eason Li

2025 W

Contents

1	Introduction and Asymptotic Analysis	6
1.1	What is this course about?	6
1.2	Algorithm Design	6
1.3	Pseudocode	7
1.4	Measuring Efficiency	8
1.4.1	Method 1: Experimental Studies	8
1.4.2	Method 2: Theoretical Analysis	8
1.5	Asymptotic Analysis	10
1.5.1	Asymptotic Upper Bound	10
1.5.2	Asymptotic Lower Bound	11
1.5.3	Tight Asymptotic Bound	11
1.5.4	Common Growth Rate	12
1.5.5	Strictly Smaller Asymptotic Bound	12
1.5.6	Strictly Larger Asymptotic Bound	12
1.5.7	Limit Theorem of Order Notation	13
1.5.8	Algebra of Order Notation	14
1.6	Runtime Analysis	15
1.6.1	Techniques for Algorithm Analysis	15
1.6.2	Worst-case/ Best-case/ Average Time Complexity	16
1.7	Space Analysis	18
1.8	Analysis of Recursive Algorithms	18
1.8.1	Explaining Solution to a Problem	18
1.8.2	Table of Recurrence Relationships	18
2	Priority Queues	19
2.1	Abstract Data Type	19
2.2	ADT Priority Queue	19
2.2.1	Using a Priority Queue to sort	19
2.3	Binary Heap	20
2.3.1	Insertion in Heaps	21
2.3.2	deleteMax in Heaps	21
2.3.3	Fix-down	22

2.4	PQ-Sort and Heapsort	22
2.4.1	PQ-sort with Heaps	23
2.4.2	Heapsort	23
3	Sorting, Average-case and Randomization	26
3.1	Analyzing average-case run-time	26
3.2	Randomized Algorithms	27
3.2.1	Expected Running Time	27
3.3	QuickSelect	31
3.3.1	Two Crucial Subroutines for Quick-Select	31
3.3.2	Partition	32
3.3.3	Average Case Analysis for QuickSelect	33
3.4	QuickSort	36
3.4.1	QuickSort with Tricks	37
3.5	Lower Bound for Comparison-Based Sorting	38
3.5.1	Decision Tree	39
3.5.2	Decision Tree for Concrete Algorithm Sorting 3 items	39
3.6	Non-Comparison-Based Sorting	41
3.6.1	Bucket Sort	41
3.6.2	MSD-Radix-Sort	42
3.6.3	LSD-Radix-Sort	43
4	Dictionaries	44
4.1	ADT Dictionary	44
4.2	Binary Search Trees	44
4.3	AVL Trees	45
4.4	Insertion in AVL Trees	46
4.5	Restructuring a BST: Rotations	46
4.5.1	Right Rotation	46
4.5.2	Double Right Notation	48
4.6	AVL insertion revisited	48
4.6.1	REstructure Pseudocode	49
4.7	Deletion in AVL Trees	49
4.8	AVL Tree Summary	50
5	Other Dictionary Implementations	51
5.1	Skip Lists	51
5.1.1	Search in Skip Lists	52
5.1.2	Insert in Skip List	53
5.1.3	Delete in Skip List	54
5.1.4	Skip List Analysis	54
5.2	Biased Search Requests	56
5.2.1	optimal static ordering	57
5.2.2	dynamic ordering: MTF	57

6	Dictionaries for special keys	58
6.1	Lower Bound for Comparison Based Algorithms	58
6.2	Interpolation Search	58
6.3	Tries	59
6.3.1	Trie: Search	60
6.3.2	Trie: Prefix-Search and Leaf Reference	60
6.3.3	Trie: Insert	61
6.3.4	Trie: Delete	61
6.3.5	Trie Summary	61
6.4	Variations of Tries: Pruned Tries	62
6.5	Variations of Tries: Compressed Tries	62
6.5.1	Compressed Trie: Search	63
6.5.2	Compressed Trie: Summary	64
6.6	Multiway Tries: Larger Alphabet	64
7	Midterm	66
8	Dictionaries via Hashing	69
8.1	Hashing Introduction	69
8.2	Hashing with Chaining	70
8.2.1	Hash with Chaining: Running Time	70
8.2.2	Load Factor and Rehashing	72
8.3	Open Addressing	73
8.3.1	Open Addressing	73
8.3.2	Probe Sequence Operations	74
8.3.3	Double Hashing	75
8.3.4	Cuckoo Hashing	76
8.3.5	Running Time of Open Addressing Strategies	78
8.4	Hash Function Strategies	78
8.4.1	Deterministic Hash Functions	78
8.4.2	Randomized Hash Functions: Carter-Wegman's Universal Hashing	79
8.4.3	Multi-dimensional Data	79
8.5	Hashing vs. Balanced Search Trees	80
9	Range-Searching in Dictionaries for Points	81
9.1	Range Search	81
9.2	Multi-Dimensional Data	81
9.2.1	Multi-Dimensional Range Search	81
9.2.2	d -Dimensional Dictionary via 1-Dimensional Dictionary	82
9.2.3	Multi-Dimensional Range Search (better idea)	82
9.3	Quadtrees	83
9.3.1	<code>search</code>	83
9.3.2	<code>insert</code>	84
9.3.3	<code>delete</code>	84
9.3.4	Quadtree Analysis	84

9.3.5	Quadtree Range Search	85
9.3.6	Quadtree in Other Dimensions	86
9.4	kd-Trees	86
9.4.1	kd-tree Construction Running Time and Space	86
9.4.2	kd-tree Dictionary Operations	87
9.4.3	kd-Tree Range Search	87
9.4.4	kd-tree: Range Search Running Time	88
9.4.5	kd-Tree — Higher Dimension	89
9.5	Range Trees	89
9.5.1	BST Range Search	89
9.5.2	How to Find Top Inside Node	90
9.5.3	BST Range Search Analysis	90
9.5.4	2D Range Tree	91
9.5.5	Range Tree Space and Time Analysis	92
9.6	Conclusion	93
10	String Matching	94
10.1	Introduction	94
10.2	Brute Force Algorithm	95
10.3	Karp-Rabin Algorithm	96
10.4	Knuth-Morris-Pratt algorithm	98
10.4.1	Fast Computation of Failure Array F	101
10.5	Boyer-Moore Algorithm	101
10.5.1	Reverse Searching	101
10.5.2	Boyer-Moore Indexing	103
10.6	Suffix Trees	104
10.6.1	Suffix Tree Summary	105
10.7	Suffix Arrays	106
10.7.1	Suffix Array Construction	106
10.7.2	Suffix Arrays - Algorithm Pseudocode	107
10.8	Conclusion	107
11	Compression	108
11.1	Background	108
11.1.1	Detour: Streams	109
11.2	Single-Character Encodings	109
11.2.1	Fixed Length Character Encoding	109
11.2.2	Variable-Length Encoding	110
11.2.3	Variable-Length — Encoding	110
11.2.4	Variable-Length — Decoding	110
11.3	Huffman Codes — Build the best trie	112
11.3.1	Huffman Tree Construction	112
11.4	Lempel-Ziv-Welch	114
11.4.1	Encoding	114

11.4.2	Decoding	116
11.5	Combining Compression Schemes: bzip2	117
11.5.1	Move-to-Front transform	117
11.5.2	Move-to-Front Transform: Properties	118
11.5.3	0-runs Encoding	118
11.6	Burrows-Wheeler Transform	118
11.6.1	BWT Algorithm and Example	119
11.6.2	BWT Fast Encoding: Efficient Sorting	119
11.6.3	BWT Decoding	119
11.6.4	BWT Summary	120
11.6.5	bzip2 Summary	120
11.7	Compression Summary	121
12	External Memory	122
12.1	Stream based algorithms	122
12.2	External dictionaries	123
12.3	2-4 Trees	123
12.3.1	2-4 Tree, Search	124
12.3.2	2-4 Tree, Insert	124
12.3.3	2-4 Tree, Delete	125
12.4	Red Black Tree	126
12.4.1	2-4 Tree to Red Black Tree	127
12.4.2	Red Black Tree to 2-4 Tree	127
12.4.3	Red Black Trees Summary	128
12.5	(a,b)-tree	129
12.6	B-tree	130
13	Post-Midterm/ Final	131

1 Introduction and Asymptotic Analysis

1.1 What is this course about?

- When first learn to program, we emphasize **correctness**;
- We also concerned with **efficiency**;
 - Processor time;
 - Memory space.
- We study efficient methods of **storing**, **accessing**, and **organizing** large collections of data: Some typical operations are
 - inserting new data items;
 - deleting data items.
- We also cover new ADTs and how to implement ADT efficiently using appropriate data structures.
- We will also cover algorithms solving problems in data management, such as sorting, pattern matching, and compression.

1.2 Algorithm Design

Definition 1.1: Problem

A **problem** is a description of input and required output.

Definition 1.2: Problem Instance

A **problem instance** is one possible input for specific problem.

Definition 1.3: Algorithm

An **algorithm** is a step-by-step process (can be described in finite length) for carrying out a series of computations, given an arbitrary instance I .

Definition 1.4: Solving

An algorithm A **solves** problem Π if for every instance I of Π , A computes a valid output for instance I in finite time.

Definition 1.5: Program

A **program** is an *implementation* of an algorithm using a specific computer language.

Discovery 1.1

In this course, we emphasize on algorithms as opposed to programs.

Theorem 1.1

In general, from problem Π to program that solves it, we care about

1. **Algorithm Design:** design algorithms that solves Π ;
2. **Algorithm Analysis:** assess *correctness* and *efficiency* of algorithms;
3. **Implementation:** If acceptable (correct and efficient), implement algorithms, while multiple implementations are possible;
4. If multiple acceptable algorithms/ implementations are present, run experiments to determine a better solution.

CS240 cares about the first two steps.

1.3 Pseudocode

Definition 1.6: Pseudocode

Pseudocode is a method of communication algorithm to a human.

Comment 1.0: Example of a Pseudocode

Algorithm 1: How to write algorithms

Data: this text

Result: how to write algorithm with L^AT_EX2_ε

```
1 initialization;
2 while not at end of this document do
3   read current;
4   if understand then
5     go to next section;
6   current section becomes this one;
7 else
8   go back to the beginning of current section;
9 end
10 end
```

1.4 Measuring Efficiency

Definition 1.7: Efficiency

Efficiency refers to

- Running Time: aka “Time Complexity”, the amount of time a program takes to run;
- Auxiliary Space: aka “Space Complexity”, the amount of additional memory a program requires.

To determine the running time of an algorithm, there are two methods.

1.4.1 Method 1: Experimental Studies

Write program implementing the algorithm and run program with inputs of *varying size* and *composition*.

Comment 1.1

There are shortcomings about this method:

- implementation may be complicated/costly
- timings are affected by many factors
 - **hardware** (processor, memory)
 - **software environment** (OS, compiler, programming language)
 - **human factors** (programmer)
- cannot test all inputs, hard to select good **sample inputs**

1.4.2 Method 2: Theoretical Analysis

Theorem 1.2

Unlike experimental studies, Theoretical Analysis

- Does not require implementing the algorithm
- Independent of hardware/software environment
- Takes into account all possible input instances

Definition 1.8: Random Access Machine (RAM) Idealized Computer Model

To process theoretical analysis, we assume that we have a RAM Idealized Computer Model, which has the following features:

- Has a set of memory cells (unbounded number), each of which is big enough to store one data item.
- Any access to a memory location takes the same constant time.

- Can run other primitive operations on this machine.

Example 1.1

Memory access, arithmetic, etc.

Comment 1.2

The assumptions may be invalid for real computers.

Algorithm 1.1: Theoretical Framework For Algorithm Analysis

- Write algorithms in pseudo-code
- Run algorithms on idealized computer model
- Time efficiency: count # primitive operations as a function of problem size n
- Space efficiency: count maximum # of memory cells ever in use

Comment 1.3

Pseudocode is essentially a sequence of primitive operations.

Example 1.2: Example of Primitive Operations

Algorithm 2: Find the maximum element in an array

Result: Maximum element of the array A

Input: Array A of n integers

```

1 Algorithm arrayMax( $A$ ,  $n$ )
2  $currentMax \leftarrow A[0]$ ;
3 for  $i \leftarrow 1$  to  $n - 1$  do
4   if  $A[i] > currentMax$  then
5      $currentMax \leftarrow A[i]$ ;
6   end
7 end
8 return  $currentMax$ ;
```

Examples of Primitive Operations:

- **arithmetic:** $-$, $+$, $\%$, $*$, **mod**, **round**
- **assigning a value to a variable**
- **indexing into an array**
- **returning from a method**
- comparisons, calling subroutine, entering a loop, breaking, etc.

Theorem 1.3

To find running time, count the number of primitive operations as a function of n .

Code 1.1

For n is the input size, x^n is not a primitive operation since it depends on n , the input size, while $x^{10000000000000000000000}$ is a primitive operation.

To perform Theoretical Analysis of Running time, we want

- ignore multiplicative constant factors
- focus on behaviour for large n (i.e. ignore lower order terms)

This means focusing on the growth rate of the function.

1.5 Asymptotic Analysis

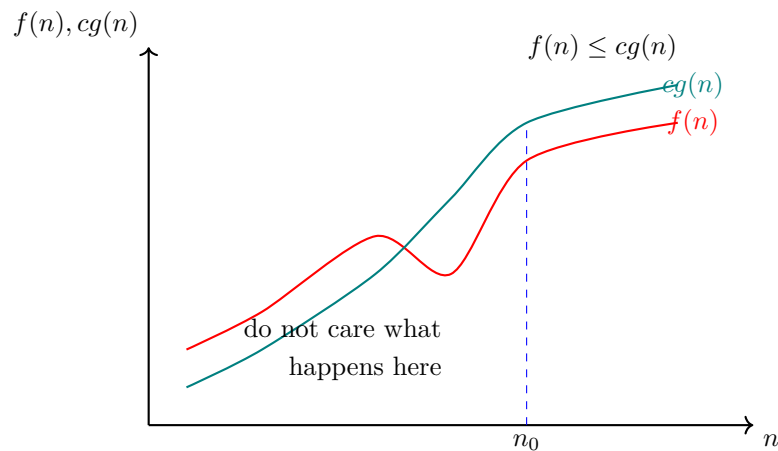
1.5.1 Asymptotic Upper Bound

Definition 1.9: BigO

$f_n \in O(g_n)$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t.

$$|f_n| \leq c|g_n| \quad \text{for all } n \geq n_0$$

Example 1.3



Result 1.1

Big-O gives asymptotic upper bound.

Comment 1.4

Saying “ f_n is $O(g_n)$ ” is equivalent to saying “ $f_n \in O(g_n)$ ”.

Example 1.4

Prove that

$$2n^2 + 3n + 11 \in O(n^2)$$

Need to find $c \geq 0$ and $n_0 \geq 0$ s.t.

$$2n^2 + 3n + 11 \leq cn^2 \text{ for all } n \geq n_0.$$

Notice that $2n^2 + 3n + 11 \leq 2n^2 + 3n^2 + 11n^2 = 16n^2$ for all $n \geq 1$. Hence take $c = 16$, $n_0 = 1$.

Example 1.5

Prove that

$$2n^2 \log n + 3n \in O(n^2 \log n)$$

Need to find $c > 0$ and $n_0 \geq 0$ s.t.

$$2n^2 \log n + 3n \leq cn^2 \log n \text{ for all } n \geq n_0.$$

Notice that $2n^2 \log n + 3n \leq 2n^2 \log n + 3n^2 \log n \leq 5n^2 \log n$ for all $n \geq 1$. Hence take $c = 5$, $n_0 = 2$.

Lecture 2 - Thursday, January 09

1.5.2 Asymptotic Lower Bound**Definition 1.10: Ω -notation**

We say that $f(n) \in \Omega(g(n))$ if there exists $c > 0$ and $n_0 \geq 0$ such that

$$|f(n)| \geq c|g(n)| \quad \text{for all } n \geq n_0$$

Example 1.6

Show that $1/2n^2 - 5n \in \Omega(n^2)$. We notice that

$$\frac{1}{2}n^2 - 5n = \frac{1}{4}n^2 + \left(\frac{1}{4}n^2 - 5n\right) \geq \frac{1}{4}n^2$$

for $n > 20$. Thus we can take $c = 1/4$ and $n_0 = 20$.

1.5.3 Tight Asymptotic Bound**Definition 1.11: Θ -notation**

We say that $f(n) \in \Theta(g(n))$ if there exists $c_1, c_2 > 0$ and $n_0 \geq 0$ such that

$$c_2|g(n)| \leq |f(n)| \leq c_1|g(n)| \quad \text{for all } n \geq n_0$$

Comment 1.5

$f(n) \in \Theta(g(n))$ means that f and g have the same growth rate. Therefore, to show that $f(n) \in \Theta(g(n))$, it is enough to show that

$$f(n) \in O(g(n)) \quad \text{and} \quad f(n) \in \Omega(g(n))$$

Example 1.7

Show that $\log_b(n) \in \Theta(\log(n))$ for $b > 1$.

Proof. We notice that

$$\log_b(n) = \frac{\log n}{\log b} = \frac{1}{\log b} \log n$$

Therefore, we have

$$\frac{1}{\log b} \log n \leq \log_b(n) \leq \frac{1}{\log b} \log n$$

and for $b > 1$, we have $\log b > 0$, thus we take $c_1 = c_2 = 1/\log b$ gives us the desired result. \square

1.5.4 Common Growth Rate

Theorem 1.4

The following common growth rate are listed in increasing order:

$\Theta(1);$	Constant
$\Theta(\log n);$	Logarithmic
$\Theta(n);$	Linear
$\Theta(n \log n);$	Linearithmic
$\Theta(n \log^k n);$	Quasi-linear (k is constant)
$\Theta(n^2);$	Quadratic
$\Theta(n^3);$	Cubic
$\Theta(2^n).$	Exponential

1.5.5 Strictly Smaller Asymptotic Bound

Definition 1.12: o -notation

We say $f(n) \in o(g(n))$ if for all $c > 0$, there exists n_0 such that

$$|f(n)| \leq c|g(n)| \quad \text{for all } n \geq n_0$$

1.5.6 Strictly Larger Asymptotic Bound

Definition 1.13: ω -notation

We say $f(n) \in \omega(g(n))$ if for all $c > 0$, there exists n_0 such that

$$|f(n)| \geq c|g(n)| \quad \text{for all } n \geq n_0$$

Theorem 1.5

We have the following:

$$f(n) \in \omega(g(n)) \Rightarrow g(n) \in o(f(n))$$

Proof. To show that $g(n) \in o(f(n))$, we wish to show that for all $c > 0$, we know that there exists $n_0 \geq 0$ such that

$$g(n) \leq cf(n) \quad \forall n \geq n_0 \quad \equiv \quad 1/c \cdot g(n) \leq f(n) \quad \forall n \geq n_0$$

Since we know that $f(n) \in \omega(g(n))$, so for $c = 1/c$ in the above notation, there exists $m_0 \geq 0$ such that

$$f(n) \geq 1/c \cdot g(n) \quad \forall n \geq m_0$$

Hence we can simply take $n_0 = m_0$. □

Lecture 3 - Tuesday, January 14

1.5.7 Limit Theorem of Order Notation

Theorem 1.6: Limit Theorem of Order Notation

Suppose for all $n \geq n_0$, we have $f(n), g(n) > 0$ and $L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$, 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 1.8

Prove that $(\log n)^a \in o(n^d)$ for any $a > 0$ and $d > 0$.

Proof. Use induction to show that $\lim_n (\ln n)^k / n = 0$ for any integer k . Use this result to show that $\lim_n (\ln n)^a / n^d = 0$. The rest is now easy to see. □

Example 1.9

Show that $f(n) := n(2 + \sin n\pi/2) \in \Theta(n)$.

Proof. We have $n \leq f(n) \leq 3n$. □

Example 1.10

Show that $f(n) := n(1 + \sin n\pi/2) \notin \Omega(n)$.

Proof. f intersects with the horizontal axis infinitely. □

Result 1.2

Relationships between Order Notations:

- $f \in \Theta(g) \iff g \in \Theta(f)$;
- $f \in O(g) \iff g \in \Omega(f)$;
- $f \in o(g) \iff g \in \omega(f)$;
- $f \in o(g) \implies g \in O(f)$;
- $f \in \omega(g) \implies g \in \Omega(f)$;
- $f \in o(g) \implies g \notin \Omega(f)$;
- $f \in \omega(g) \implies g \notin O(f)$;

1.5.8 Algebra of Order Notation

- **Identity:** $f \in \Theta(f)$;
- **Transitivity:**
 - If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$;
 - If $f \in \Omega(g)$ and $g \in \Omega(h)$, then $f \in \Omega(h)$;
 - If $f \in O(g)$ and $g \in o(h)$, then $f \in o(h)$;
- **Maximum Rules:** Suppose that $f(n), g(n) > 0$ for all $n \geq n_0$, then
 1. $f(x) + g(x) \in O(\max\{f(x), g(x)\})$;
 2. $f(x) + g(x) \in \Omega(\max\{f(x), g(x)\})$;

1.6 Runtime Analysis

We wish to use asymptotic notation to simplify run-time analysis. Running time of an algorithm depends on the input size n .

Algorithm 1.2

1. Identify primitive operations: these require constant time
2. Loop complexity expressed as sum of complexities of each iteration
3. Nested loops: start with the innermost loop and proceed outwards
4. This gives nested summations

Example 1.11

Data: an integer n

Result: how to compute complexity of an algorithm

```
1 sum ← n;  
2 for i ← 1 to n do  
3   for j ← i to n do  
4     sum ← sum + (i + j)2  
5   end  
6 end
```

We have

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

hence the complexity of the algorithm is $\Theta(n^2)$.

Discovery 1.2

We may use Θ -notation earlier, in particular,

$$c + \sum_{i=1}^n \sum_{j=i}^n c \quad \text{is} \quad \Theta \left(c + \sum_{i=1}^n \sum_{j=i}^n c \right)$$

1.6.1 Techniques for Algorithm Analysis

Result 1.3: Techniques for Algorithm Analysis

1. Use Θ -bounds throughout the analysis and obtain Θ bound for the complexity of the algorithm.
2. Prove a O -bound and a matching Ω -bound *separately*.

Example 1.12

Data: an integer n

Result: how to compute complexity of an algorithm

```
1 sum  $\leftarrow$  0;
2  $i = n$ ;
3 while  $i \geq 2$  do
4    $j = 0$ ;
5   while  $j \leq i^2$  do
6     sum  $\leftarrow$  sum + 1;
7      $j = j + i$ 
8   end
9    $i = i/2$ 
10 end
```

- Inner Loop takes $i(c)$ time;
- Outer Loop takes around $\log n$ times, so

$$\sum_{i=1}^{\log n} (1 + n/2^{t-1}) \in O(n)$$

1.6.2 Worst-case/ Best-case/ Average Time Complexity

Definition 1.14: Worst Case Running Time

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

$$T_{worst}(n) = \max_{I \in I_n} T(I)$$

Definition 1.15: Best Case Running Time

Analogous definition as above.

Example 1.13

```
Data: an array of  $n$  integers
1 if  $n = 5$  then
2   | return  $A[0]$ 
3 end
4 if  $n \neq 5$  then
5   | for  $i = 1$  to  $n - 1$  do
6     |   print  $A[i]$ 
7   | end
8 end
```

The best case time complexity for the above program is $\Theta(n)$.

Lecture 4 - Thursday, January 16

Definition 1.16: Average Running Time

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

$$T_{avg}(n) = \frac{1}{|I_n|} \sum_{I \in I_n} T(I)$$

Comment 1.6

We will assume that I_n is finite.

Discovery 1.3

Suppose algorithm A and B both solve the same problem

- A has worst-case runtime $O(n^3)$
- B has worst-case runtime $O(n^2)$

We Cannot conclude that B is more efficient than A because O -notation is only an upper bound. A could have worst case runtime $O(n)$ while for B the bound of $O(n^2)$ could be tight.

Algorithm 1.3

To compare algorithms, it is better to use Θ -notation.

1.7 Space Analysis

We are interested in auxiliary space, space used in addition to the space used by the input data.

Algorithm 1.4

To find space used by an algorithm, count total number of auxiliary memory cells ever accessed (for reading or writing or both) by algorithm

- as a function of input size n ;
- space used must always be initialized, although it may not be stated explicitly in pseudocode.

1.8 Analysis of Recursive Algorithms

1.8.1 Explaining Solution to a Problem

Algorithm 1.5

1. describe the overall idea
2. give pseudocode or detailed description
3. argue correctness
 - key ingredients, no need for a formal proof
 - sometimes obvious enough from idea-description
4. analyze runtime

1.8.2 Table of Recurrence Relationships

Table 1: Time Complexity Analysis for Various Recursions

Recursion	Resolves to	Example
$T(n) \leq T(n/2) + O(1)$	$T(n) \in O(\log n)$	Binary search
$T(n) \leq 2T(n/2) + O(n)$	$T(n) \in O(n \log n)$	Merge sort
$T(n) \leq 2T(n/2) + O(\log n)$	$T(n) \in O(n)$	Heapify (*)
$T(n) \leq cT(n-1) + O(1)$ for some $c < 1$	$T(n) \in O(1)$	Avg-case analysis (*)
$T(n) \leq 2T(n/4) + O(1)$	$T(n) \in O(\sqrt{n})$	Range-search (*)
$T(n) \leq T(\sqrt{n}) + O(\sqrt{n})$	$T(n) \in O(\sqrt{n})$	Interpol. search (*)
$T(n) \leq T(\sqrt{n}) + O(1)$	$T(n) \in O(\log \log n)$	Interpol. search

2 Priority Queues

2.1 Abstract Data Type

Definition 2.1: ADT

ADT is a description of information and a collection of operations on that information.

Comment 2.1

We can have various realizations of an ADT, which specify:

- How the information is stored (*data structure*)
- How the operations are performed (*algorithms*)

Definition 2.2: Stack

An ADT with push and pop, LIFO.

Definition 2.3: Queue

An ADT with enqueue and dequeue, FIFO.

2.2 ADT Priority Queue

Priority Queue generalizes both ADT Stack and ADT Queue.

Definition 2.4: Priority Queue

Priority Queue is a collection of items (each having a priority or key) with operations

- insert: inserting an item tagged with a priority
- delete-max: removing and returning an item of highest priority.

2.2.1 Using a Priority Queue to sort

```
Data: an array of  $n$  integers,  $A[0, \dots, n-1]$ 
1 initialize  $PQ$  to an empty priority queue ;
2 for  $i \leftarrow 0$  to  $n-1$  do
3   |  $PQ.insert(A[i])$ 
4 end
5 for  $i \leftarrow n-1$  to  $0$  do
6   |  $A[i] \leftarrow PQ.deleteMax()$ 
7 end
```

Comment 2.2

The runtime for the above algorithm is $O(\text{initialization} + n \cdot \text{insert} + n \cdot \text{deleteMax})$.

2.3 Binary Heap

Definition 2.5: Binary Tree

A **binary tree** is either empty or contains three parts, a node and two sub binary trees.

Theorem 2.1

A binary tree with n nodes has height h :

$$\log(n + 1) - 1 \leq h \leq n - 1$$

Proof. The upper bound is trivial, for the lower bound, we wish to saturate the lowest levels first and then calculate the possible minimal height. \square

Definition 2.6: Heap

A **max-oriented binary heap** is a binary tree with the following two properties:

- Structural Property:
 - All the levels of a heap are completely filled, except (possibly) for the last level.
 - The filled items in the last level are *left-justified*.
- Heap-order Property: For any node i , the key of the parent of i is larger than or equal to key of i .

Code 2.1

Heaps are ideal for implementing priority queues.

Lemma 2.1

Height of a heap with n nodes is $\Theta(\log n)$.

Proof. Since heap is a binary tree, we know that $h \in \Omega(\log n)$. STP that $h \in O(\log n)$. Since heap has full levels except level h , we know that

$$n \geq 2^0 + 2^1 + \dots + 2^{h-1} + 1$$

Solving for the inequality we obtain that $h \leq \log n$, proving that $h \in O(\log n)$. \square

Result 2.1

We use node and index interchangeably. Here we have some important observations:

- Left child of i , if exists, is $2i + 1$;
- Right child of i , if exists, is $2i + 2$;
- Parent of i , if exists, is $\text{floor}[(i - 1)/2]$.

Proof. Proof by induction. □

Lecture 5 - Tuesday, January 21

2.3.1 Insertion in Heaps

Algorithm 2.1

- Place new key at the first free leaf (Heap-order property might be violated);
- Perform a fix-up.

Algorithm 3: fix-up pseudocode

```
Data:  $i$ : an index corresponding to heap node
1 initialize  $PQ$  to an empty priority queue ;
2 while  $\text{parent}(i)$  exists and  $A[\text{parent}(i)].\text{key} < A[i].\text{key}$  do
3   | swap  $A[i]$  and  $A[\text{parent}(i)]$ 
4   |  $i \leftarrow \text{parent}(i)$  // move to one level up
5 end
```

Discovery 2.1

Time: $O(\text{tree height}) = O(\log n)$.

Insert Pseudocode

Algorithm 4: insertion pseudocode

```
1 increase size
2  $l \leftarrow \text{last}()$ 
3  $A[l] \leftarrow x$ 
4 fix-up ( $A, l$ )
```

2.3.2 deleteMax in Heaps

Algorithm 2.2

- The root has the maximum item

- Replace root by the last leaf and remove last leaf
- perform fix-down

2.3.3 Fix-down

Algorithm 5: fix-down pseudocode

Data: A : array that stores a heap of size n in locations $0, \dots, n-1$
 i : index corresponding to a heap node

```

1 while  $i$  is not a leaf do
2    $j \leftarrow$  left child of  $i$ 
3   if  $i$  has right child and  $A[\text{right child of } i].key > A[j].key$  then
4      $j \leftarrow$  right child of  $i$ 
5   end
6   if  $A[i].key \geq A[j].key$  then
7     break
8   end
9   swap  $A[i]$  and  $A[j]$ 
10   $i \leftarrow j$ 
11 end
```

Discovery 2.2

Time: $O(\text{tree height}) = O(\log n)$.

deleteMax Pseudocode

Algorithm 6: deleteMax pseudocode

```

1  $l \leftarrow \text{last}()$ 
2  $\text{toReturn} = A[\text{root}()]$ 
3  $A[(\text{root}())] = A[l]$ 
4 decrease size
5  $\text{fix-down}(A, \text{root}(), \text{size})$ 
```

2.4 PQ-Sort and Heapsort

Comment 2.3

Priority queue sort is $O(\text{init} + n \cdot \text{insert} + n \cdot \text{deleteMax})$ time.

Theorem 2.2

Heap with n nodes and height h has at least $n/4$ nodes at level $h-1$.

Proof. SFAC that there are less than $n/4$ nodes at level $h - 1$. It is obvious to see that the number of nodes above this level have also less than $n/4$ nodes. Moreover, for level h , because each node in this level must be a child of a node in level $h - 1$, so there are less than $n/4 \cdot 2$ nodes at level h . In total, this gives us less than n nodes, which is a contradiction. \square

2.4.1 PQ-sort with Heaps

Algorithm 2.3

Algorithm 7: PQSortWithHeaps(A)

```

1  $H \leftarrow$  empty heap
2 for  $i \leftarrow 0$  to  $n - 1$  do
3    $H.insert(A[i])$ 
4 end
5 for  $k \leftarrow n - 1$  downto  $0$  do
6    $A[k] \leftarrow H.deleteMax()$ 
7 end
```

Discovery 2.3

PQ-Sort with heap is $\Theta(n \log n)$ and not in place.

2.4.2 Heapsort

Result 2.2

Heapsort: improvement to PQ-Sort with two added tricks

1. use the input array A to store the heap!
2. heap can be built in linear time if know all items in advance.

To introduce heap sort, we first need to solve the following problem:

build a heap from n items in $A[0, \dots, n - 1]$ without using additional space.

We have two choices to create heap order: namely the fix-up and fix-down method. Recall that heap with n nodes and height h has at least $n/4$ nodes at level $h - 1$ (Theorem 2.2). For each of those nodes, we notice that fix-down method takes $O(1)$ time while fix up takes $O(\log n)$ time. Therefore, we would like to establish heap order using fix-down. One big observation is that

Comment 2.4

If both subtrees of node v have correct heap-order, fix-down on v will establish correct order for the whole subtree of v .

Heapify Pseudocode

Algorithm 8: heapify pseudocode

```
Data:  $A$ : an array  
1 for  $i \leftarrow \text{parent}(\text{last}())$  downto 0 do  
2   |  $\text{fix} - \text{down}(A, i, n)$   
3 end
```

Heapify Analysis Note that for depth k , there are 2^k nodes at most, and each of the nodes at this depth is getting moved by fix-down at most $h - k$ times. Therefore, the time is given by

$$\begin{aligned}\sum_{i=0}^{h-1} 2^i (h - i) &= 2^h \sum_{i=0}^{h-1} \frac{h - i}{2^{h-i}} \\ &= 2^h \sum_{i=1}^h \frac{i}{2^i}\end{aligned}$$

where we know that the latter sum is convergent. Thus

$$\text{Time} \leq 2^h c \leq 2^{\log n} c = cn$$

which proves that the time complexity of the given algorithm is $\Theta(n)$.

Therefore, we are now able to implement the heapsort algorithm:

Algorithm 2.4

Algorithm 9: Heapsort

```
1  $n \leftarrow A.\text{size}$   
2 for  $i \leftarrow \text{parent}(\text{last}())$  downto 0 do  
3   |  $\text{fix} - \text{down}(A, i, n)$   
4 end  
5 while  $n > 1$  do  
6   | swap  $A[\text{root}()]$  and  $A[\text{last}()]$   
7   | decrease  $n$   
8   |  $\text{fix} - \text{down}(A, \text{root}(), n)$   
9 end
```

Result 2.3:

The total time is $O(n \log n)$.

Question 2.1. Find the k^{th} smallest item in an array A of n numbers.

Comment 2.5

k^{th} smallest = the item that would be at $A[k]$ if sorted.

Solution: We have in total three solutions so far.

1. Make $k + 1$ passes through the array, deleting the minimum number each time. Complexity: $\theta(kn)$.
2. Sort A , then return $A[k]$. Complexity: $n \log n$.
3. Create a min-heap with $heapify(A)$. Call $delete - min(A)$ $k + 1$ times. Complexity: $n + k \log n$.



3 Sorting, Average-case and Randomization

Lecture 6 - Thursday, January 23

3.1 Analyzing average-case run-time

Example 3.1: An contrived example

Algorithm 10: smallestFirst

Data: A : array storing n distinct integers in range $\{0, 1, \dots, n-1\}$

```
1 if  $A[0]=0$  then
2 for  $j = 1$  to  $n$  do print "first is smallest" ;
3 ;
4 else print "first is not smallest ;
```

Discovery 3.1

Notice that the best case is $O(1)$ and the worst case is $\Theta(n)$.

Solution: In general, there are $n!$ inputs in total: there are $(n-1)!$ of them having $A[0] = 0$ and the rest have the first entry non-zero. Therefore,

$$\begin{aligned} T_{avg}(n) &= \frac{1}{|I_n|} \sum_{I \in I_n} T(I) \\ &= \frac{1}{n!} \left(\underbrace{cn + \dots + cn}_{(n-1)!} + \underbrace{c + \dots + c}_{n! - (n-1)!} \right) \\ &= \frac{1}{n!} (cn(n-1)! + c(n! - (n-1)!)) \\ &= c + c - \frac{c}{n} \in O(1) \end{aligned}$$



Example 3.2: Another example

Algorithm 11: all-0-test

Data: w : bitstring of length n

```
1 if  $n = 0$  then return true;
2 if  $(w[n-1] = 1)$  then return false;
3 all-0-test( $w, n-1$ )
```

Solution: Best case is when the bitstring has a 1 at the end, in which case has a time complexity of $O(1)$. The worst case would be the case when the bit string has all zeros, in which case we have a time complexity of $\Theta(n)$. Let B_n be the set of all bit strings of length n .

Comment 3.1

Note that $|B_n| = 2|B_{n-1}|$.

Hence we obtain that the average run time is

$$T_{avg}(n) = \frac{1}{|B_n|} \sum_{w \in B_n} T(w)$$

where $T(w)$ is defined as

$$T(w) = \begin{cases} 1 & \text{if } w[n-1] = 1 \\ 1 + T(w[0, \dots, n-2]) & \text{otherwise} \end{cases}$$

Therefore, we have

$$\begin{aligned} T_{avg}(n) &= \frac{1}{|B_n|} \sum_{w \in B_n} T(w) \\ &= \frac{1}{|B_n|} \sum_{w \in B_n, w[n-1]=1} 1 + \frac{1}{|B_n|} \sum_{w \in B_n, w[n-1]=0} T(w) \\ &= \frac{1}{|B_n|} \sum_{w \in B_n, w[n-1]=1} 1 + \frac{1}{|B_n|} \sum_{w \in B_n, w[n-1]=0} [1 + T(w[0, \dots, n-2])] \\ &= \frac{1}{2} + \frac{1}{2} + \frac{1}{|B_n|} \sum_{w \in B_n, w[n-1]=0} T(w[0, \dots, n-2]) \\ &= 1 + \frac{|B_{n-1}|}{|B_n|} \cdot \frac{1}{|B_{n-1}|} \sum_{w \in B_{n-1}} T(w) \\ &= 1 + \frac{1}{2} T_{avg}(n-1) \end{aligned}$$

Hence we obtain that $T_{avg}(n)$ is $\Theta(1)$.



3.2 Randomized Algorithms

Definition 3.1: Randomized Algorithm

A **randomized algorithm** is one which relies on random numbers for some steps

3.2.1 Expected Running Time

Discovery 3.2

Define $T(I, R)$ to be running time of randomized algorithm for instance I and R , sequence of random numbers algorithm choses.

Expected Runtime for Instane I : is expected value for $T(I, R)$:

$$T_{exp}(I) = \mathbb{E}[T(I, R)] = \sum_R T(I, R) \cdot Pr(R)$$

Worst-case Expected Runtime: is given by

$$T_{exp}(n) = \max_{I \in I_n} T_{exp}(I)$$

Example 3.3

Algorithm 12: simple

Data: A : array storing n numebrs

```
1  $sum \leftarrow 0$ 
2 if  $random(3) = 0$  then return sum;
3 else
4   for  $i \leftarrow 0$  to  $n - 1$  do
5      $sum \leftarrow sum + A[i]$  ;
6 return sum ;
```

Solution: Notice that the algorithm needs only one random number, and

$$Pr(0) = Pr(1) = Pr(2) = \frac{1}{3}$$

Hence we have

$$\begin{aligned} T_{exp}(I) &= T(I, 0) \cdot Pr(0) + T(I, 1) \cdot Pr(1) + T(I, 2) \cdot Pr(2) \\ &= c \cdot \frac{1}{3} + c \cdot n \cdot \frac{1}{3} + c \cdot n \cdot \frac{1}{3} \in \Theta(n) \end{aligned}$$

as desired.



Example 3.4

Algorithm 13: simple2

Data: A : array storing n numebrs

```
1  $sum \leftarrow 0$ 
2  $r1 \leftarrow random(n), r2 \leftarrow random(n)$ 
3 for  $i \leftarrow 1$  to  $r1$  do
4   for  $j \leftarrow 1$  to  $r2$  do
5      $sum \leftarrow sum + A[j]A[i]$ 
6   end
7 end
```

Solution: We have

$$\begin{aligned}
 T_{exp}(I) &= \sum_{\langle r_1, r_2 \rangle} T(I, \langle r_1, r_2 \rangle) \cdot \left(\frac{1}{n}\right)^2 \\
 &= \left(\frac{1}{n}\right)^2 \sum_{r_1 \in \{0, \dots, n-1\}} c \cdot r_1 \sum_{r_2 \in \{0, \dots, n-1\}} r_2 \\
 &= \left(\frac{1}{n}\right)^2 \sum_{r_1 \in \{0, \dots, n-1\}} c \cdot r_1 \frac{n(n-1)}{2} \\
 &= \left(\frac{1}{n}\right)^2 c \frac{n(n-1)}{2} \frac{n(n-1)}{2}
 \end{aligned}$$

All instances have the same running time, so $T_{exp}(n) \in \Theta(n^2)$. 

Question 3.1. Why do we use Randomized Algorithms

- improved running time;
- improve solution.

Result 3.1

Randomization can shift dependence from what we cannot control (user) to what we can control (random number generation).

The following algorithm essentially does the same thing as the algorithm “all-0-test”, but with randomization:

Algorithm 3.1

Algorithm 14: randomized-all-0-test

```

// test if all entries of bitstring w[0,...,n-1] are 0
1 if (n = 0) then
2   | return true
3 end
4 if (random(2) = 0) then
5   | w[n-1] = 1 - w[n-1]
6 end
7 if (w[n-1] = 1) then
8   | return false
9 end
10 randomized-all-0-test(w,n-1)

```

Solution: Let $T(w, R)$ be the number of bit-comparisons on input w if the random outcomes are R . We have

$$R = \langle x, R' \rangle$$

where x is the first random number and R' are the other random numbers (if any) for the recursions.

Comment 3.2


By random number independence, $Pr(R) = Pr(x) \cdot Pr(R')$.

Hence we have

$$\begin{aligned}
 T_{exp}(w) &= \sum_R Pr(R) \cdot T(w, R) \\
 &= \sum_{\langle x, R' \rangle} Pr(R') Pr(x) \cdot T(w, \langle x, R' \rangle) \\
 &= \frac{1}{2} \sum_{\langle x, R' \rangle} Pr(R') \cdot T(w, \langle x, R' \rangle) \\
 &= \frac{1}{2} \sum_{R'} Pr(R') \cdot T(w, \langle x = w[n-1], R' \rangle) + \frac{1}{2} \sum_{R'} Pr(R') \cdot T(w, \langle x \neq w[n-1], R' \rangle) \\
 &= \frac{1}{2} \sum_{R'} 1 + \frac{1}{2} \sum_{R'} Pr(R') \cdot (1 + T(w[0..n-2], R')) \\
 &= \frac{1}{2} + \frac{1}{2} \sum_{R'} Pr(R') + \frac{1}{2} \sum_{R'} Pr(R') \cdot T(w[0..n-2], R') \\
 &= \frac{1}{2} + \frac{1}{2} + \frac{1}{2} \sum_{R'} Pr(R') \cdot T(w[0..n-2], R') \\
 &= 1 + \frac{1}{2} T_{exp}(\text{some instance of size } n-1) \\
 &\leq 1 + \max_{v \in B_{n-1}} T_{exp}(v) \\
 &= 1 + \frac{1}{2} T_{exp}(n-1)
 \end{aligned}$$

Since we know that the inequality is true for all w , hence

$$T_{exp}(n) = \max_{w \in B_n} T_{exp}(w) \leq 1 + \frac{1}{2} T_{exp}(n-1)$$

which is a recurrence relation resolves to $\Theta(1)$. 

Discovery 3.3

Average case runtime and expected runtime are different concepts!

Result 3.2

average case	expected
$T^{avg}(n) = \frac{\sum_{I \in I_n} T(I)}{ I_n }$	$T^{exp}(I) = \sum_R T(I, R) \cdot Pr(R)$
sum is over instances	sum is over random outcomes
	applied only to a randomized algorithm

Lecture 7 - Tuesday, January 28

3.3 QuickSelect

Given array A of n numbers, and $0 \leq k < n$, find the element that would be at position k if A was sorted.

Definition 3.2: Rank

k is also known as **rank**.

Comment 3.3

A special case is when $k = \lfloor \frac{n}{2} \rfloor$, which is known as **MedianFinding**.

Theorem 3.1

Selection can be done with heaps in $\Theta(b + k \log n)$ time.

Question 3.2. Can we do selection in linear time?

Yes, with quick-select (average case analysis).

3.3.1 Two Crucial Subroutines for Quick-Select**Code 3.1**

- `choose-pivot(A)` : return an index p in A , $v = A[p]$ is called **pivot value**;
- `partition(A, p)` : uses $v = A[p]$ to rearranges A so that

$$A[m] \leq v \quad \forall m \in \{0, \dots, i-1\},$$

$$A[i] = v,$$

$$A[n] \geq v \quad \forall n \in \{i+1, \dots, n-1\}$$

3.3.2 Partition

Algorithm 15: partition

Input: Array A of size n , p is an integer such that $0 \leq p < n$

```
1 partition( $A$ ,  $p$ )
2 create empty lists small, equal and large
3  $v \leftarrow A[p]$ 
4 for each element  $x$  in  $A$  do
5     if  $x < v$  then small.append( $x$ )
6     else if  $x > v$  then large.append( $x$ )
7     else equal.append( $x$ )
8 end
9  $i \leftarrow \text{small.size}$ 
10  $j \leftarrow \text{equal.size}$ 
11 overwrite  $A[0, \dots, i - 1]$  by elements in small
12 overwrite  $A[i, \dots, i + j - 1]$  by elements in equal
13 overwrite  $A[i + j, \dots, n - 1]$  by elements in large
14 return  $i$ ;
```

Comment 3.4

The above is an easy linear-time implementation using extra (auxiliary) $\Theta(n)$ space, we wish to have $\Theta(1)$ space.

Algorithm 16: Partition the array using a pivot

Input: Array A of size n , pivot index p such that $0 \leq p < n$

```
1 swap( $A[n - 1]$ ,  $A[p]$ )                                // Put pivot at the end
2  $i \leftarrow -1$ ,  $j \leftarrow n - 1$ ,  $v \leftarrow A[n - 1]$ 
3 Loop:    do  $i \leftarrow i + 1$  while  $A[i] < v$ 
4           do  $j \leftarrow j - 1$  while  $j \geq i$  and  $A[j] > v$ 
5           if  $i \geq j$  then break
6           else swap( $A[i]$ ,  $A[j]$ )
7 End loop
8 swap( $A[n - 1]$ ,  $A[i]$ )                                // Put pivot in the correct position
9 return  $i$ 
```

Algorithm 3.2

The partition algorithm below is an efficient in-place partition whose running time is $\Theta(n)$.

Now we may introduce the quickselect algorithm:

Code 3.2: QuickSelect

Algorithm 17: QuickSelect

Input: Array A of size n , k is an integer such that $0 \leq k < n$

```
1 quickSelect( $A, k$ )
2  $p \leftarrow \text{choose-pivot}(A)$ 
3  $i \leftarrow \text{partition}(A, p)$ 
4 if  $i = k$  then return  $A[i]$ ;
5 else if  $i > k$  then return  $\text{quickSelect}(A[0, \dots, i-1], k)$  else if  $i < k$  then return
    $\text{quickSelect}(A[i+1, \dots, n-1], k - (i+1))$ 
```

Result 3.3

Let $T(n, k)$ be # of comparisons in array of size n with parameter k .

- *Best Case:* First chosen pivot could have pivot index k , hence no recursive call, total cost is $\Theta(n)$
- *Worst Case:* Pivot value is always the largest and $k = 0$, in which case we have a recursive relation

$$T(n, 0) = \begin{cases} T(n-1, 0) & n > 1 \\ 1 & n = 1 \end{cases}$$

which resolves to $\Theta(n^2)$.

3.3.3 Average Case Analysis for QuickSelect

the expected time of randomized QuickSelect is the same as average case runtime of nonrandomized QuickSelect.

Theorem 3.2

We can show (complicated) that the average case runtime for QuickSelect is $\Theta(n)$.

Discovery 3.4

Instead, we will randomize QuickSelect, so that the expected time of randomized QuickSelect is the same as average case runtime of nonrandomized QuickSelect.

Randomization

Idea 1: Shuffle the input then run QuickSelect

Algorithm 18: quickSelectShuffled

Input: Array A of size n

```

1 quickSelectShuffled( $A$ ,  $k$ )
2 for  $i \leftarrow 1$  to  $n - 1$  do
3   | swap( $A[i]$ ,  $A[random(1 + i)]$ )
4 end

```

Comment 3.5

Can show that every permutation of A is equally likely after shuffle.

Idea 2: Change pivot selection

Algorithm 19: randomizedQuickSelect

Input: Array A of size n , k : integer such that $0 \leq k < n$

```

1 randomizedQuickSelect( $A$ ,  $k$ )
2  $p \leftarrow random(A.size)$ 
3  $i \leftarrow partition(A, p)$ 
4 if  $i = k$  then return  $A[i]$ 
5 else if  $i > k$  then
6   return randomizedQuickSelect( $A[0, \dots, i - 1]$ ,  $k$ )
7 else if  $i < k$  then
8   return randomizedQuickSelect( $A[i + 1, \dots, n - 1]$ ,  $k - (i + 1)$ )

```

Now we perform the runtime analysis. Let $T(A, k, R)$ be the number of key-comparisons on array A of size n , selecting k th element, using random numbers R . Hence R is a sequence of randomly generated pivot indexes,

$$R = \langle first, rest \rangle = \langle i, R' \rangle$$

Structure of array A after partition is in the form of

B	v	C
-----	-----	-----

where B has a size of i and C has a size of $n - i - 1$. Recurse in array B or C or algorithms stops:

$$T(A, k, \langle i, R' \rangle) = \begin{cases} T(B, k, R') & i > k \\ T(C, k - i - 1, R') & i < k \\ 0 & otherwise \end{cases}$$

Notice that the runtime of `randomizedQuickSelect(A , k)` also depends on k :

$$T^{exp}(n) = \max_{A \in I_n} \max_{k \in \{0, \dots, n-1\}} \sum_R T(A, k, R) Pr(R)$$

Hence

$$\begin{aligned}
\sum_R T(A, k, R) \Pr(R) &= \sum_{\langle i, R' \rangle} T(A, k, \langle i, R' \rangle) \Pr(\langle i, R' \rangle) \\
&= \sum_{R=\langle i, R' \rangle} T(A, k, \langle i, R' \rangle) \Pr(i) \Pr(R') \\
&= \frac{1}{n} \sum_{i=0}^{k-1} \sum_{R'} T(A, k, \langle i, R' \rangle) \Pr(R') + \frac{1}{n} \sum_{i=k+1}^{n-1} \sum_{R'} T(A, k, \langle i, R' \rangle) \Pr(R') \\
&= \frac{1}{n} \sum_{i=0}^{k-1} \sum_{R'} [n + T(C, k - i - 1, R')] \Pr(R') + 1 + \frac{1}{n} \sum_{i=k+1}^{n-1} \sum_{R'} [n + T(B, k, R')] \Pr(R') \\
&\leq n + \frac{1}{n} \sum_{i=0}^{k-1} \max_{D \in I_{n-i-1}, w \in \{0, \dots, k-1\}} \sum_{R'} T(D, w, R') \Pr(R') \\
&\quad + \frac{1}{n} \sum_{i=k+1}^{n-1} \max_{D \in I_i, w \in \{k+1, \dots, n-1\}} \sum_{R'} T(D, w, R') \Pr(R') \\
&= n + \frac{1}{n} \sum_{i=0}^{k-1} T^{\text{exp}}(n - i - 1) + \frac{1}{n} \sum_{i=k+1}^{n-1} T^{\text{exp}}(i)
\end{aligned}$$

Since above bound works for any A and k , it will work for the worst A and k .

$$\begin{aligned}
T^{\text{exp}}(n) &= \max_{A \in I_n} \max_{k \in \{0, \dots, n-1\}} \sum_R T(A, k, R) \Pr(R) \\
&\leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T^{\text{exp}}(i), T^{\text{exp}}(n - i - 1)\}
\end{aligned}$$

Theorem 3.3

For the above $T(n)$, we have

$$T(n) \in O(n)$$

Proof. We will prove $T(n) \leq 4n$ by induction on n .

- **Base case:** when $n = 1$, we have $T(n) = 1 \leq 4$.
- **Induction Hypethesis:** We have

$$\begin{aligned}
T(n) &\leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T(i), T(n - i - 1)\} \\
&\leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{4i, 4(n - i - 1)\} \\
&\leq n + \frac{4}{n} \sum_{i=0}^{n-1} \max\{i, n - i - 1\}
\end{aligned}$$

Continue on next page:

$$\begin{aligned}
\sum_{i=0}^{n-1} \max(i, n-i-1) &= \left(\sum_{i=0}^{\frac{n}{2}-1} \max(i, n-i-1) \right) + \left(\sum_{i=\frac{n}{2}}^{n-1} \max(i, n-i-1) \right) \\
&= \left(\max(0, n-1) + \max(1, n-2) + \max(2, n-3) + \dots + \max\left(\frac{n}{2}-1, n-\left(\frac{n}{2}-1\right)-1\right) \right) \\
&\quad + \left(\max\left(\frac{n}{2}, n-\frac{n}{2}-1\right) + \dots + \max(n-1, 0) \right) \\
&= \left((n-1) + (n-2) + \dots + \frac{n}{2} \right) + \left(\frac{n}{2} + \left(\frac{n}{2}+1\right) + \dots + (n-1) \right) \\
&= \left(\frac{3n}{2} - 1 \right) \frac{n}{4} + \left(\frac{3n}{2} - 1 \right) \frac{n}{4} \\
&\leq \frac{3}{4}n^2
\end{aligned}$$

as desired. □

3.4 QuickSort

Algorithm 20: QuickSort Algorithm

Input: Array A of size n
Output: Sorted Array A

```

1 Function QuickSort( $A$ ):
2   if  $n \leq 1$  then
3     return
4   end
5    $p \leftarrow \text{choose-pivot}(A)$ 
6    $i \leftarrow \text{partition}(A, p)$ 
7   QuickSort( $A[0, \dots, i-1]$ )
8   QuickSort( $A[i+1, \dots, n-1]$ )

```

Discovery 3.5

- *Worst case:* $T(n) = T(n-1) + n$, which is a recurrence solved in the same way as quickSelect, $O(n^2)$;
- *Best case* $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ which is solved in the same way as mergeSort, $\Theta(n \log n)$.

Comment 3.6

We find the average case through randomized version of QuickSort by changing the line of initializing p to

$$p \leftarrow \text{random}(A.\text{size})$$

The expected running time for `RandomizedQuickSort`, derived similarly to `RandomizedQuickSelect`, is

$$T^{\text{exp}}(n) \leq \frac{1}{n} \sum_{i=0}^{n-1} (n + T^{\text{exp}}(i) + T^{\text{exp}}(n-i-1))$$

hence

$$\begin{aligned} T^{\text{exp}}(n) &\leq \frac{1}{n} \sum_{i=0}^{n-1} (n + T^{\text{exp}}(i) + T^{\text{exp}}(n-i-1)) \\ &= n + \frac{1}{n} \left(\sum_{i=0}^{n-1} T^{\text{exp}}(i) + \sum_{i=0}^{n-1} T^{\text{exp}}(n-i-1) \right) \\ &= n + \frac{2}{n} \sum_{i=0}^{n-1} T^{\text{exp}}(i) \end{aligned}$$

Thus

$$T^{\text{exp}}(n) \leq n + \frac{2}{n} \sum_{i=0}^{n-1} T^{\text{exp}}(i)$$

Theorem 3.4

For the above $T(n)$, we have

$$T(n) \leq 2n \ln n \quad \text{for all } n > 0$$

Proof. The proof is using induction. $T(1) = 0$ (no comparisons). Hence for $n \geq 2$

$$T(n) \leq n + \frac{2}{n} \sum_{i=2}^{n-1} T(i) \leq n + \frac{2}{n} \sum_{i=2}^{n-1} 2i \ln i = n + \frac{4}{n} \sum_{i=2}^{n-1} i \ln i$$

In particular,

$$\begin{aligned} \sum_{i=2}^{n-1} i \ln i &\leq \int_2^n x \ln x \, dx = \frac{1}{2} n^2 \ln n - \frac{1}{4} n^2 \underbrace{-2 \ln 2 + 1}_{\leq 0} \\ &\leq \frac{1}{2} n^2 \ln n - \frac{1}{4} n^2 \end{aligned}$$

This is tight since best-case run-time is $\Omega(n \log n)$. □

Lecture 8 - Thursday, January 30

3.4.1 QuickSort with Tricks

We have a several ways to improve our algorithm:

- We can reduce that auxiliary space to $\Theta(\log n)$ worst-case by recurse in smaller sub-array first and replace the other recursion by a while-loop (tail call elimination);

- Stop recursion when, say $n \leq 10$. At this point, array is not completely sorted, but almost sorted, hence we can run insertionSort, which sorts in just $O(n)$ time since all items are within 10 units of the required position;
- In terms of implementation, instead of passing full arrays, pass only the range of indices could avoid recursion altogether by keeping an explicit stack.

Algorithm 21: Improved QuickSort with Insertion Sort

```

input  : An array  $A$  of size  $n$ 
output: A sorted array  $A$ 

1 Function QuickSortImproves( $A, n$ ):
2   initialize a stack  $S$  of index-pairs with  $\{(0, n - 1)\}$ 
3   while  $S$  is not empty do
4      $(\text{left}, \text{right}) \leftarrow S.\text{pop}()$  // get the next subproblem
5     while  $\text{right} - \text{left} + 1 > 10$  // work on it if it's larger than 10
6       do
7          $\text{pivot} \leftarrow \text{choose-pivot}(A, \text{left}, \text{right})$ 
8          $\text{index} \leftarrow \text{partition}(A, \text{left}, \text{right}, \text{pivot})$ 
9         if  $\text{index} - \text{left} > \text{right} - \text{index}$  // is left side larger than right?
10          then
11             $S.\text{push}((\text{left}, \text{index} - 1))$  // store larger problem in  $S$  for later
12             $\text{left} \leftarrow \text{index} + 1$  // next work on the right side
13          end
14          else
15             $S.\text{push}((\text{index} + 1, \text{right}))$  // store larger problem in  $S$  for later
16             $\text{right} \leftarrow \text{index} - 1$  // next work on the left side
17          end
18        end
19    end
20    InsertionSort( $A$ )


```

3.5 Lower Bound for Comparison-Based Sorting

We have seen many sorting algorithms

Question 3.3.

Can one do better than $\Theta(n \log n)$ running time?

Solution: It depends on what we allow. For comparison-based sorting, the lower bound is $\Omega(n \log n)$ and there is no restriction on input. For non-comparison-based sorting, they can achieve $O(n)$ but there are restrictions on input. 

Definition 3.3: Comparison Model

In the **comparison model** data can only be accessed in two ways

- comparing two elements:

$$A[i] \leq A[j]$$

- moving elements around (e.g. copying, swapping).

3.5.1 Decision Tree

Definition 3.4: Decision Tree

Decision tree succinctly describes all decisions that are taken during the execution of an algorithm and the resulting outcome

Comment 3.7

For simplicity, will assume array A stores unique numbers.

Definition 3.5: Sorting Permutation

Sorting permutation tells us how to sort the array. It stores array indexes in the order corresponding to the sorted array.

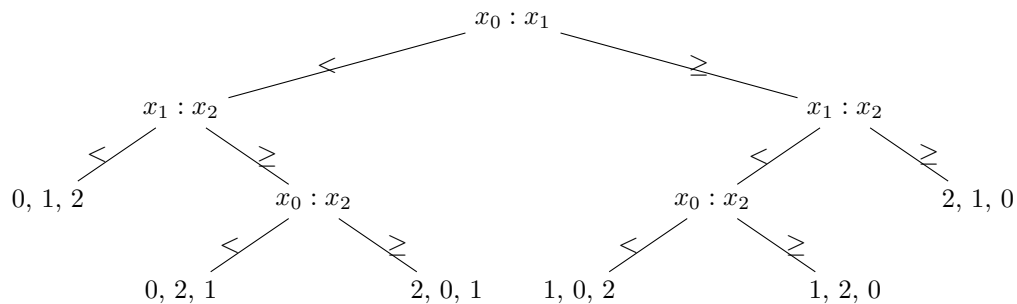
Example 3.5

For the array A :

$$A = \begin{bmatrix} 14 & 2 & 3 & 5 & 1 & 11 & 7 \end{bmatrix}$$

we have $\pi = (4, 1, 2, 3, 6, 5, 0)$.

3.5.2 Decision Tree for Concrete Algorithm Sorting 3 items



Algorithm 3.3**Algorithm 22:** Nested Decision Structure for Printing Sorted Values

```

1 if  $x_0 < x_1$  then
2   | If  $x_1 < x_2$ , then print( $x_0, x_1, x_2$ )
3   | ElseIf  $x_0 < x_2$ , then print( $x_0, x_2, x_1$ )
4   | Else, print( $x_2, x_0, x_1$ )
5 end
6 else
7   | if  $x_1 < x_2$  then
8   |   | If  $x_0 < x_2$ , then print( $x_1, x_0, x_2$ )
9   |   | Else, print( $x_1, x_2, x_0$ )
10  | Else, print( $x_2, x_1, x_0$ )
11 end

```

Discovery 3.6

We Can prove that the height of any decision tree is at least $c \cdot n \log n$.

Theorem 3.5

Under comparison model, **any** sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof. Let **SortAlg** be any comparison based sorting algorithm. Since **SortAlg** is comparison based, it has a decision tree. **SortAlg** must sort correctly any array of n elements. Let

$S_n = \text{set of arrays storing not-repeating integers } 1, \dots, n$

Notice that $|S_n| = n!$. Let $\pi(x)$ denote the sorting permutation of $x \in S_n$. When we run x through T , we must end up at a leaf labeled with $\pi(x)$, and $x, y \in S_n$ with $x \neq y$ have sorting permutations $\pi(x) \neq \pi(y)$. Because there are $n!$ instances in S_n which must go to distinct leaves, so the tree must have at least $n!$ leaves. Observe that binary tree with height h has at most 2^h leaves, so our height h must be at least such that $2^h \geq n!$. Hence

$$\begin{aligned}
h &\geq \log(n!) = \log(n(n-1) \cdots 1) \\
&= \log n + \log(n-1) + \cdots + \log\left(\frac{n}{2} + 1\right) + \log\left(\frac{n}{2}\right) + \cdots + \log 1 \\
&\geq \underbrace{\log \frac{n}{2} + \cdots + \log \frac{n}{2}}_{\frac{n}{2} \text{ terms}} \\
&= \frac{n}{2} \log \frac{n}{2} \\
&= \frac{n}{2} \log n - \frac{n}{2} \in \Omega(n \log n)
\end{aligned}$$

as desired. □

3.6 Non-Comparison-Based Sorting

Algorithm 3.4

Non-comparison based sorting is less general than comparison based sorting.

Comment 3.8

We will assume we are sorting non-negative integers.

3.6.1 Bucket Sort

Suppose all keys in A of size n are integers in range $[0, \dots, L - 1]$, we use an auxiliary bucket array

$$B[0, \dots, L - 1]$$

to sort, i.e. array of linked lists, where initialization is $\Theta(L)$.

Theorem 3.6

Running time for Bucket Sort is $\Theta(L + n)$.

Single Digit BucketSort

Algorithm 23: Bucket Sort Algorithm

```
input :  $A$ : An array  $A$  of size  $n$ , containing numbers with digits in  $\{0, \dots, R - 1\}$ 
input :  $d$ : index of digit by which we wish to sort

1 Bucket-sort( $A, d$ )
2 initialize array  $B[0, \dots, R - 1]$  of empty lists (buckets)
3 for  $i \leftarrow 0$  to  $n - 1$  do
4    $next \leftarrow A[i]$ 
5   append  $next$  at the end of  $B[d\text{th digit of } next]$ 
6 end
7  $i \leftarrow 0$ 
8 for  $j \leftarrow 0$  to  $R - 1$  do
9   while  $B[j]$  is non-empty do
10    move first element of  $B[j]$  to  $A[i++]$ 
11  end
12 end
```

Discovery 3.7

- Sorting is stable: equal items stay in original order
- Run-time $\Theta(n + R)$;

- Auxiliary space $\Theta(n + R)$;

3.6.2 MSD-Radix-Sort

Algorithm 3.5

Sorts multi-digit numbers from the most significant to the least significant.

Comment 3.9

We cannot sort the whole array by the second digit since it will mess up the order. Have to break down in groups by the first digit.

MSD-Radix-Sort Space Analysis

Question 3.4.

MSD-Radix-Sort Space Analysis.

Solution: Bucket-sort has auxiliary space $\Theta(n + R)$, and our recursion depth is $m - 1$, which has auxiliary space $\Theta(m)$.

Therefore, the total auxiliary space $\Theta(n + R + m)$.



MSD-Radix-Sort Time Analysis

Question 3.5.

MSD-Radix-Sort Time Analysis.

Solution: Time spent for each recursion depth

- Depth $d = 0$: one bucket sort on n items, yields $O(n + R)$
- At depth $d > 0$
 - Let k be number of bucket sorts where, $k \leq n$. We can index bucketsorts as $1, \dots, i, \dots, k$. Hence bucketsort i involves n_i keys, which implies that bucket sort i takes $n_i + R$ time.

$$\sum_{i=1}^k (n_i + R) = \sum_{i=1}^k n_i + \sum_{i=1}^k R \leq n + kR \leq n + nR$$

Hence total time at depth d is $O(nR)$.

Notice that number of depths is at most $m - 1$.

Therefore, the total time is $O(mnR)$.



Pseudocode

Algorithm 24: MSD-Radix-Sort using bucket sort for each digit

Input : An array A , left index l , right index r , digit index d

Output: Sorted array A based on MSD

```

1 Function MSD-Radix-Sort( $A, l, r, d$ ):
2   if  $l < r$  then
3     Bucket-Sort( $A[l \dots r], d$ )
4     if there are digits left then
5        $l' \leftarrow l$ 
6       while  $l' < r$  do
7          $r'$  is the maximal s.t  $A[l' \dots r']$  have the same  $d$ th digit
8         MSD-Radix-Sort( $A, l', r', d + 1$ )
9          $l' \leftarrow r' + 1$ 
10      end
11    end
12  end

```

3.6.3 LSD-Radix-Sort

Example 3.6

Prepare to sort by last digit	Sorted by last digit	Prepare to sort by middle digit	Sorted by last two digits	Prepare to sort by first digit	Sorted by all three digits
123	230	101	101	101	121
230	320	210	210	121	123
121	210	320	320	210	210
320	121	121	121	123	230
210	101	123	123	230	232
232	232	230	230	232	320
101	123	232	232	320	101

m bucket sorts, on n items each, one bucket sort is $\Theta(n + R)$.

Total time cost $\Theta(m(n + R))$.

4 Dictionaries

4.1 ADT Dictionary

Definition 4.1: Dictionary

A **dictionary** is a collection of items, each of which contains a key some data (the “value”) and is called a **key-value pair** (KVP). Keys can be compared and are (typically) unique.

Code 4.1

Operations:

1. `search(k)` (also called `lookup(k)`)
2. `insert(k, v)`
3. `delete(k)` (also called `remove(k)`) optional: `successor`, `merge`, `is-empty`, `size`, etc.

Example 4.1

Examples are symbol table, license plate database.

Result 4.1

	search	insert	delete
unsorted list/array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
binary search tree	$\Theta(\text{height})$	$\Theta(\text{height})$	$\Theta(\text{height})$

4.2 Binary Search Trees

Definition 4.2: Binary Tree

(Structure) all nodes have two (possibly empty) subtrees. Every node stores a KVP.

Comment 4.1

Empty subtrees usually not shown.

(Ordering) Every key k in $T.\text{left}$ is less than the root key. Every key k in $T.\text{right}$ is greater than the root key.

Code 4.2

`search(k)` Start at root, compare k to current node’s key. Stop if found or subtree is empty, else recurse at subtree.

`insert(k, v)` Search for k , then insert (k, v) as new node.

- `delete(x)`
- First search for the node x that contains the key.
 - If x is a leaf (both subtrees are empty), delete it.
 - If x has one non-empty subtree, move child up
 - Else, swap key at x with key at successor node and then delete that node.

Definition 4.3: Successor

Successor is the next-smallest among all keys in the dictionary.

`BST::search`, `BST::insert`, `BST::delete` all have cost $\Theta(h)$, where

h = height of the tree = max.path length from root to leaf

Question 4.1.

If n items are inserted one-at-a-time, how big is h ?

Solution: We have

- Worst-case: $n - 1 = \Theta(n)$
- Best-case: $\Theta(\log n)$. Any binary tree with n nodes has height $h \geq \log(n + 1) - 1$.

This is because layer i has at most 2^i nodes, so

$$n \leq \sum_{i=0}^h 2^i = 2^{h+1} - 1$$



4.3 AVL Trees

Definition 4.4: AVL Trees

an **AVL Tree** is a BST with an additional height-balance property at every node:

The heights of the left and right subtree differ by at most 1

Definition 4.5: Balance

We define

$$\text{balance}(v) := \text{height}(R) - \text{height}(L)$$

Code 4.3

Need to store at each node v the height of the subtree rooted at it. (Note that we would also store balance (instead of height) at each node.)

Theorem 4.1

An AVL tree on n nodes has $\Theta(\log n)$ height.

Proof. • Define $N(h)$ to be the least number of nodes in a height- h AVL tree.

- What is a recurrence relation for $N(h)$?
- What does this recurrence relation resolve to?

□

4.4 Insertion in AVL Trees

Algorithm 4.1

To perform $AVL :: insert(k, v)$:

1. First, $insert(k, v)$ with the usual BST insertion.
2. We assume that this returns the new leaf z where the key was stored.
3. Then, move up the tree from z . Update height:

Algorithm 25: *set_height_from_subtrees(u)*

```
1  $u.height \leftarrow 1 + \max\{u.left.height, u.right.height\}$ 
```

Comment 4.2

If the height difference becomes ± 2 at node z , then z is unbalanced. Must re-structure the tree to rebalance.

4.5 Restructuring a BST: Rotations

Discovery 4.1

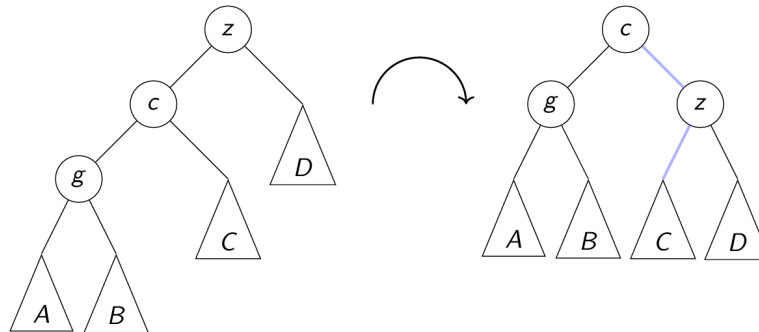
There are many different BSTs with the same keys.

Our goal is to change the structure locally nodes without changing the order. Our long term goal is to restructure such that the subtree becomes balanced.

4.5.1 Right Rotation

Example 4.2

This is a **right rotation** on node z :



Algorithm 26: Right Rotation Pseudocode

```
1  $c \leftarrow z.left$ 
  // fix links connecting to above
2  $c.parent \leftarrow (p \leftarrow z.parent)$ 
3 if  $p = NULL$  then
4    $root \leftarrow c$ 
5 end
6 else
7   if  $p.left = z$  then  $p.left \leftarrow c$  else  $p.right \leftarrow c$ .
8 end
  // actual rotation
9  $z.left \leftarrow c.right, c.right.parent \leftarrow z$ 
10  $c.right \leftarrow z, z.parent \leftarrow c$ 
11  $set\text{-}height\text{-}from\text{-}subtrees(z), set\text{-}height\text{-}from\text{-}subtrees(c)$ 
12 return  $c$  // returns new root of subtree
```

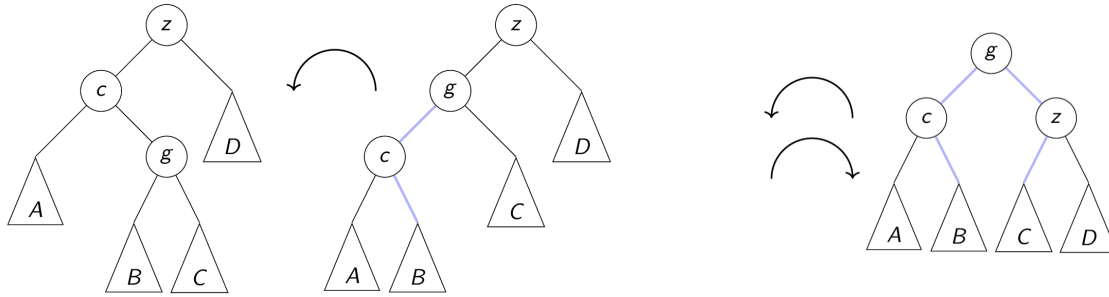
Discovery 4.2

Only $O(1)$ links are changed. Useful to fix left-left imbalance.

Result 4.2

Symmetrically, there is a left rotation on node z .

4.5.2 Double Right Notation



Definition 4.6: Double Right Rotation

A **double right rotation** on node z is a left rotation at c followed by a right rotation at z .

4.6 AVL insertion revisited

Code 4.4

- Imbalance at z : do (single or double) rotation;
- Choose c as child where subtree has bigger height.

Algorithm 27: AVL Tree Insertion

Input: k, v

Output: Updated tree with the node inserted maintaining the AVL property

```

1  $z \leftarrow \text{BST}::\text{insert}(k, v)$                                 // leaf where k is now stored
2 while  $z \neq \text{NULL}$  do
3   if  $|z.\text{left}.\text{height} - z.\text{right}.\text{height}| > 1$  then
4     Let  $c$  be taller child of  $z$ 
5     Let  $g$  be taller child of  $c$  (so grandchild of  $z$ )
6      $z \leftarrow \text{restructure}(g, c, z)$                         // see later
7     break                                                  // can argue that we are done
8   end
9    $\text{set-height-from-subtrees}(z)$ 
10   $z \leftarrow z.\text{parent}$ 
11 end

```

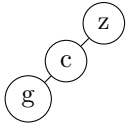
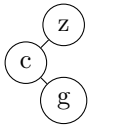
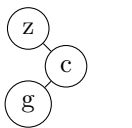
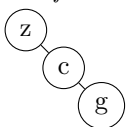
Algorithm 4.2

Rule of thumb: The middle key of g, c, z becomes the new root.

4.6.1 REstructure Pseudocode

Algorithm 28: Structure Function for AVL Tree

```

1 Function restructure( $g, c, z$ ):
    // node  $g$  is child of  $c$  which is child of  $z$ 
2   switch do
3     case Right rotation do
        
         $u \leftarrow \text{rotate-right}(z)$ 
        return  $u$ 
4     end
5     case Double-right rotation do
        
         $\text{rotate-left}(c); \quad u \leftarrow \text{rotate-right}(z)$ 
        return  $u$ 
6     end
7     case Double-left rotation do
        
         $\text{rotate-right}(c); \quad u \leftarrow \text{rotate-left}(z)$ 
        return  $u$ 
8     end
9     case Left rotation do
        
         $u \leftarrow \text{rotate-left}(z)$ 
        return  $u$ 
10    end
11  end
12 end

```

4.7 Deletion in AVL Trees

Algorithm 4.3

1. Remove the key k with `BST::delete`.
2. Find node where structural change happened. (This is not necessarily near the node that had k .)
3. Go back up to root, update heights, and rotate if needed.

Code 4.5

Algorithm 29: AVL Tree Deletion

Input: Key k to delete

Output: Updated AVL tree after deletion

```
1  $z \leftarrow \text{BST}::\text{delete}(k)$     // leaf where  $k$  was removed, assume  $z$  is the parent of this
   node
2 while  $z \neq \text{NULL}$  do
3   if  $|z.\text{left}.\text{height} - z.\text{right}.\text{height}| > 1$  then
4     Let  $c$  be the taller child of  $z$ 
5     Let  $g$  be the taller child of  $c$            // break ties to avoid double rotation
6      $z \leftarrow \text{restructure}(g, c, z)$ 
7     break                                   // can argue that we are done
8   end
9    $\text{set-height-from-subtrees}(z)$ 
10   $z \leftarrow z.\text{parent}$ 
11 end
```

4.8 AVL Tree Summary

1. **search:** Just like in BSTs, costs $\Theta(\text{height})$;
2. **insert:** `BST::insert`, then check & update along path to new leaf.
 - total cost: $\Theta(\text{height})$;
 - restructure will be called at most once.
3. **delete:** `BST::delete`, then check & update along path to deleted node
 - total cost: $\Theta(\text{height})$;
 - restructure may be called $\Theta(\text{height})$ times.

5 Other Dictionary Implementations

Theorem 5.1

We can show: If the KVPs were inserted in random order, then the expected height of the binary search tree would be $O(\log n)$.

5.1 Skip Lists

Definition 5.1: Skip List

A hierarchy L of ordered linked lists (levels) L_0, L_1, \dots, L_h ,

- L_0 contains the KVPs of some S in non-decreasing order;
- For each of the subsequent levels, choose each item from previous level with probability $1/2$;
- each L_i also contains special keys (sentinels) $-\infty$ and $+\infty$;
- L_h contains only sentinels, the left sentinel is the root.

Discovery 5.1

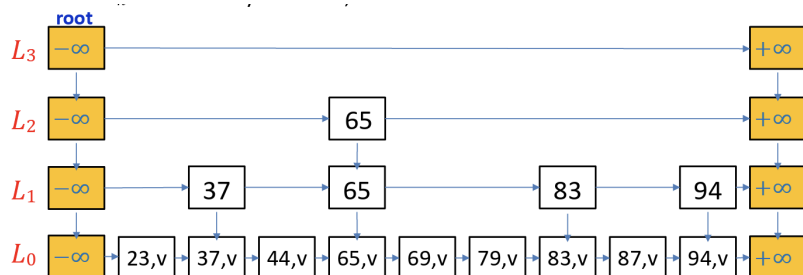
Each list is a subsequence of previous one, i.e.

$$L_0 \subseteq L_1 \subseteq \dots \subseteq L_h$$

Discovery 5.2

- i^{th} list is expected to have $n/2^i$ nodes;
- Expect about $\log(n)$ lists in total.

Example 5.1



Definition 5.2: Tower

Each key k belongs to a tower of nodes, and we define the **height of tower** for k is the largest i s.t. $k \in L_i$.

Definition 5.3: Height of Skip List

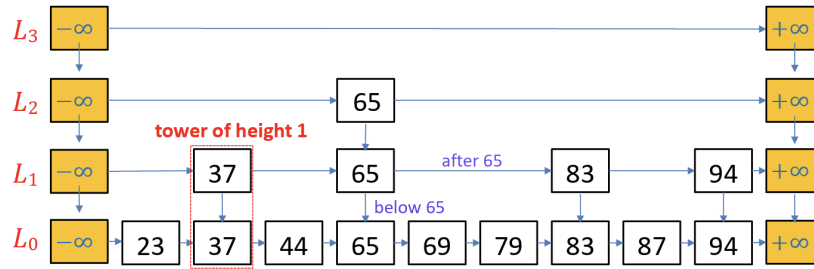
Height of the skip list is the maximum height of any tower.

Comment 5.1

Height is 3 in the above example.

Code 5.1

Each node p has references to `after(p)` and `below(p)`.

Example 5.2**5.1.1 Search in Skip Lists****Algorithm 30: getPredecessors**

Input: Key k

Output: Stack of nodes P containing predecessors

```

1  $p \leftarrow \text{root};$ 
2  $P \leftarrow$  stack of nodes, initially containing  $p$ ;
3 while  $p.\text{below} \neq \text{NULL}$  do
4    $p \leftarrow p.\text{below};$ 
5   while  $p.\text{after.key} < k$  do
6      $p \leftarrow p.\text{after};$ 
7   end
8    $P.\text{push}(p);$ 
9 end
10 return  $P$ 

```

Definition 5.4: Predecessor

For each level, predecessor of key k is

- If key k is present at the level: node before node with key k ;
- if key k is not present at the level: node before node where k would have been.

Algorithm 31: skipList::search

Input: Key k

Output: Node corresponding to key k or location where k would be

```

1  $P \leftarrow \text{getPredecessors}(k)$ ;
2  $q \leftarrow P.\text{top}()$ ;
3 if  $q.\text{after.key} = k$  then
4   | return  $q.\text{after}$ 
5 else
6   | return “not found, but would be after  $q$ ”
7 end
```

5.1.2 Insert in Skip List**Code 5.2****Algorithm 32:** Insertion in a Skip List

Input : Key k , Value v

Output: Inserts the key-value pair (k, v) into the skip list

```

1  $i \leftarrow 0$  ;                                     // initialize level counter
2 while  $\text{random}(2) = 1$  do  $i \leftarrow i + 1$  ;      // random tower height
3  $h \leftarrow 0$ ;  $p \leftarrow \text{root.below}$ ; ;          // start from the base level
4 for (  $h \leftarrow 0$ ,  $p \leftarrow \text{root.below}$ ;  $p \neq \text{NIL}$ ;  $p \leftarrow p.\text{below}$  ) do  $h++$  ;
5  $P \leftarrow \text{getPredecessors}(k)$ ;
6  $p \leftarrow P.\text{pop}()$ ;
7  $z\text{Below} \leftarrow$  new node with  $(k, v)$  inserted after  $p$  ;      // insert  $(k, v)$  in  $L_0$ 
8 while  $i > 0$  do
9   |  $p \leftarrow P.\text{pop}()$ ;
10  |  $z \leftarrow$  new node with  $k$  added after  $p$ ;
11  |  $z.\text{below} \leftarrow z\text{Below}$ ;
12  |  $z\text{Below} \leftarrow z$ ;
13  |  $i \leftarrow i - 1$ ;
14 end
```

5.1.3 Delete in Skip List

Code 5.3

Algorithm 33: Deletion from a Skip List

```

Input : Key  $k$  to be deleted
Output: Deletes the key  $k$  from the skip list

1  $P \leftarrow \text{getPredecessors}(k)$  ; // Find predecessors of  $k$ 
2 while  $P$  is non-empty do
3    $p \leftarrow P.\text{pop}()$  ; // Get a predecessor of  $k$  in some layer
4   if  $p.\text{after}.key = k$  do  $p.\text{after} \leftarrow p.\text{after}.\text{after}$  ; // Remove node
5   else break
6 end
7  $p \leftarrow$  left sentinel of the root-list ; // Move to root-list
8 while  $p.\text{below}.\text{after}$  is the  $\infty$  sentinel do
9   // The two top lists are both only sentinels, remove one
9    $p.\text{below} \leftarrow p.\text{below}.\text{below}$  ;
10   $p.\text{after}.\text{below} \leftarrow p.\text{after}.\text{below}.\text{below}$  ;
11 end

```

5.1.4 Skip List Analysis

Let X_k be the height of tower for key k , we know $P(X_k \geq i) = \frac{1}{2^i}$. If $X_k \geq i$ then list L_i includes key k . Let $|L_i|$ be the number of keys in list L_i . Define $I_{i,k}$ as follows:

$$I_{i,k} = \begin{cases} 0 & \text{if } X_k < i \\ 1 & \text{if } X_k \geq i \end{cases}$$

Then, the number of keys in list L_i can be expressed as:

$$|L_i| = \sum_{\text{key } k} I_{i,k}$$

The expected length of list L_i is:

$$E[|L_i|] = E\left[\sum_{\text{key } k} I_{i,k}\right] = \sum_{\text{key } k} E[I_{i,k}] = \sum_{\text{key } k} P(I_{i,k} = 1) = \sum_{\text{key } k} \frac{1}{2^i} = \frac{n}{2^i}$$

Comment 5.2

Sentinels do not count towards the length; L_0 always contains all n keys.

Result 5.1

Thus, the expected length of list S_i is $\frac{n}{2^i}$.

Also define

$$I_i = \begin{cases} 0 & \text{if } |L_i| = 0 \\ 1 & \text{if } |L_i| \geq 1 \end{cases}$$

Let $h = 1 + \sum_{i \geq 1} I_i$ (here +1 is for the sentinel-only level).

Discovery 5.3

- Since $I_i \leq 1$ we have that $E[I_i] \leq 1$.
- Since $I_i \leq |L_i|$ we have that $E[I_i] \leq E[|L_i|] = \frac{n}{2^i}$.

Comment 5.3

For ease of derivation, assume n is a power of 2.

$$\begin{aligned} E[h] &= E \left[1 + \sum_{i \geq 1} I_i \right] = 1 + \sum_{i \geq 1} E[I_i] = 1 + \sum_{i=1}^{\log n} E[I_i] + \sum_{i=1+\log n}^{\infty} E[I_i] \\ &\leq 1 + \sum_{i=1}^{\log n} 1 + \sum_{i=1+\log n}^{\infty} \frac{n}{2^i} \\ &= 1 + \log n + \frac{n}{2 \cdot 2^{\log n}} \sum_{i=0}^{\infty} \left(\frac{1}{2} \right)^i \\ &= 1 + \log n + 1 \end{aligned}$$

Result 5.2

Expected skip list height $\leq 2 + \log n$.

Skip List Analysis: Expected Space We need space for nodes storing sentinels and nodes storing keys:

1. *Space for nodes storing sentinels:* There are $2h + 2$ sentinels, where h be the skip list height, recall that $E[h] \leq 2 + \log n$. Hence expected space for sentinels is at most:

$$E[2h + 2] = 2E[h] + 2 \leq 6 + 2 \log n$$

2. *Space for nodes storing keys:* Let $|L_i|$ be the number of keys in list L_i , recall that $E[|L_i|] = \frac{n}{2^i}$. Hence expected space for keys is:

$$E \left[\sum_{i \geq 0} |L_i| \right] = \sum_{i \geq 0} E[|L_i|] = \sum_{i \geq 0} \frac{n}{2^i} = 2n$$

Result 5.3

Total expected space is $\Theta(n)$.

Skip List Analysis: Expected Running Time**Theorem 5.2**

Total expected running time is $O(\log n)$.

Proof. `search`, `insert`, and `delete` are dominated by the runtime of `getPredecessors`. So we analyze the expected time of `getPredecessors`. We ‘drop-down’ h times, where h is skip list height, so the total expected time spent on ‘drop-down’ operations is $O(\log n)$. It now suffices to show that expected number of ‘scan-forward’ is also $O(\log n)$.

In general, in a random level i , the probability of scan-forward at least l times is at most $(1/2)^l$.

Comment 5.4

‘at most’ because we could scan-down down.

Hence

$$E[\# \text{ scan-forward at level } i] = \sum_{l \geq 1} l \cdot P(\text{scans} = l) = \sum_{l \geq 1} P(\text{scans} \geq l) \leq \sum_{l \geq 1} \frac{1}{2^l} = 1$$

At level $i < h$: $E[\text{number of scan-forward}] \leq 1$. Also, expected number of scan-forward at level $i < \text{number of keys at level } L_i$. Therefore

$$\begin{aligned} \sum_{i \geq 0} E[\# \text{ of scan-forward at level } i] &= \sum_{i=1}^{\log n} E[\# \text{ of scan-forward at level } i] + \sum_{i=1+\log n}^{\infty} E[\# \text{ of scan-forward at level } i] \\ &\leq \sum_{i=1}^{\log n} 1 + \sum_{i=1+\log n}^{\infty} \frac{n}{2^i} \\ &= \log n + 1 \end{aligned}$$

Expected number of scan-forwards is $O(\log n)$. □

5.2 Biased Search Requests

Unordered lists/arrays are among simplest data structures to implement but `search` is inefficient ($\Theta(n)$).

Question 5.1.

Can we make search in unordered lists/arrays more effective in practice?

Solution: No if items are accessed equally likely. Yes if the search requests are biased. 🎵

5.2.1 optimal static ordering

Question 5.2.

Scenario: We know access distribution, and want to find the best list order:

<i>Key</i>	A	B	C	D	E
<i>Frequency of Access</i>	2	8	1	10	5
<i>Access Probability</i>	2/26	8/26	1/26	10/26	5/26

Definition 5.5:

Let the cost of search for key located at position i be i . The expected cost of search, denoted as $T_{\text{exp}}(n)$, is given by:

$$\begin{aligned} T_{\text{exp}}(n) &= \sum_{I \in I_n} T(I) \cdot \Pr(\text{randomly chosen instance } I) \\ &= \sum_i i \cdot \Pr(\text{search for key at position } i) \\ &= \sum_i i \cdot (\text{access probability for key at position } i) \end{aligned}$$

Ordering items by non-increasing access-probability minimizes expected access cost, i.e. best static ordering.

5.2.2 dynamic ordering: MTF

This time, we do not know the access probabilities ahead of time, so we modify the order dynamically, i.e. while we are accessing.

Comment 5.5

Rule of thumb: recently accessed item is likely to be accessed soon again

Definition 5.6: MTF Heuristic

Move-To-Front heuristic (MTF): after search, move the accessed item to the front.

Code 5.4

Additionally, in list: always insert at the front.

Theorem 5.3

Can show: MTF is “2-competitive”.

6 Dictionaries for special keys

Realizations we have seen so far:

- Balanced Binary Search trees (AVL trees): $\Theta(\log n)$ `search`, `insert`, and `delete` (worst-case);
- Skip lists: $\Theta(\log n)$ `search`, `insert`, and `delete` (expected).

Various other realizations sometimes faster on insert, but search always takes $\Omega(\log n)$ time.

Question 6.1.

Can one do better than $\Theta(\log n)$ time for search?

Solution: It depends on what we allow:

- No: Comparison-based searching lower bound is $\Omega(\log n)$.
- Yes: Non-comparison-based searching can achieve $o(\log n)$ (under restrictions!).



6.1 Lower Bound for Comparison Based Algorithms

Theorem 6.1

Any comparison-based algorithm requires in the worst case $\Omega(\log n)$ comparisons to search among n distinct items.

Proof. Via decision tree for items x_0, \dots, x_{n-1} and search for k . □

6.2 Interpolation Search

Recall Binary Search, we always choose the middle point as our pivot point. But now, we have a more specific way of choosing such index:

Discovery 6.1

$$\ell + \left\lceil \frac{\overbrace{k - A[\ell]}^{\text{distance from left key}}}{\underbrace{A[r] - A[\ell]}_{\text{distance between left and right keys}}} \times \underbrace{(r - \ell - 1)}_{\# \text{ unknown keys in range}} \right\rceil$$

Code 6.1

The following is the pseudocode for interpolating search algorithm.

Algorithm 34: Interpolation Search Algorithm

```

1  $\ell \leftarrow 0, \quad r \leftarrow n - 1$ 
2 while  $\ell \leq r$  do
3   if  $(k < A[\ell] \text{ or } k > A[r])$ , then return “not found”
4   if  $A[r] == k$ , then return “found at  $A[r]$ ”
5    $m \leftarrow \ell + \left\lceil \frac{k - A[\ell]}{A[r] - A[\ell]} \cdot (r - \ell - 1) \right\rceil$ 
6   if  $(A[m] == k)$ , then return “found at  $A[m]$ ”
7   else if  $(A[m] < k)$  then  $\ell \leftarrow m + 1$ 
8   else  $r \leftarrow m - 1$ 
9 end

```

Theorem 6.2

We can show that $T^{avg}(n) \leq T^{avg}(\sqrt{n}) + \Theta(1)$, which resolves to

$$T^{avg}(n) \in O(\log \log n)$$

Proof. This is difficult, specifically the recurrence relation part. □

6.3 Tries

Now we could have our keys as words instead of integers.

Definition 6.1: Words (Strings)

Words are sequences of characters over alphabet Σ .

Code 6.2

Conventionally, words have end-sentinel \$ (but it is sometimes not shown).

Definition 6.2: Size (of a word)

The **size** of a word is the number of non-sentinel characters. For instance,

$$|helloworld\$| = 10$$

Comment 6.1

Words are sorted lexicographically.

Definition 6.3: Trie

Comment 6.2

Pronounced as “try”.

Trie is also known as redix tree, it is a tree of bistrings based on bitwise comparisons whose edge are labelled with corresponding bit.

Discovery 6.2

Due to end-sentinels, all key-value pairs are at leaves.

6.3.1 Trie: Search

Algorithm 6.1

1. Follow links that corresponds to current bits in w . 2. Repeat until no such link or w found at a leaf.

Algorithm 35: `Trie::get-path-to(w)`

Output: Stack P with all ancestors of where w would be stored

```
1  $P \leftarrow$  empty stack;  $z \leftarrow$  root;  $d \leftarrow 0$ ;  $P.\text{push}(z)$ 
2 while  $d \leq |w|$  do
3   if  $z$  has a child-link labelled with  $w[d]$  then
4      $z \leftarrow$  child at this link;  $d \leftarrow d + 1$ ;  $P.\text{push}(z)$ 
5   else break
6 end
7 return  $P$ 
```

Algorithm 36: `Trie::search(w)`

```
1  $P \leftarrow$  get-path-to( $w$ );  $z \leftarrow P.\text{top}$ 
2 if ( $z$  is not a leaf) then
3   return “not found, would be in sub-trie of  $z$ ”
4 end
5 return key-value pair at  $z$ 
```

6.3.2 Trie: Prefix-Search and Leaf Reference

We want another search-operation:

`prefix-search(w)`

which find word w' in trie for which w is a prefix.

Algorithm 6.2

To find w' quickly, we need leaf-references

- Every node z stores reference $z.leaf$ to a leaf in subtree
- **Convention: store leaf with longest word.**

Code 6.3

Algorithm 37: `Trie::prefix-search(w)`

```
1  $P \leftarrow \text{get\_path\_to}(w)$ 
2 if (number of nodes on  $P$  is  $w.size$  or less) then
3   | return "no extension of  $w$  found"
4 end
5 return  $P.top().leaf$ 
```

6.3.3 Trie: Insert

Algorithm 6.3

- $P \leftarrow \text{get_path_to}(w)$ gives ancestors that exist already,
- Expand the trie from $P.top()$ by adding necessary nodes that correspond to extra bits of w .
- Update leaf-references (also at P if w is longer than previous leaves)

6.3.4 Trie: Delete

Algorithm 6.4

- $P \leftarrow \text{get_path_to}(w)$ gives all ancestors.
- Let ℓ be the leaf where w is stored
- Delete ℓ and nodes on P until ancestor has two or more children.
- Update leaf-references on rest of P . (If $z \in P$ referred to ℓ , find new $z.leaf$ from other children.)

6.3.5 Trie Summary

Discovery 6.3

`search(w)`, `prefix-search(w)`, `insert(w)`, `delete(w)` all take time $\Theta(|w|)$.

Comment 6.3

Search-time is independent of number n of words stored in the trie!

Result 6.1

The trie for a given set of words is unique (except for order of children and ties among leaf-references).

Discovery 6.4

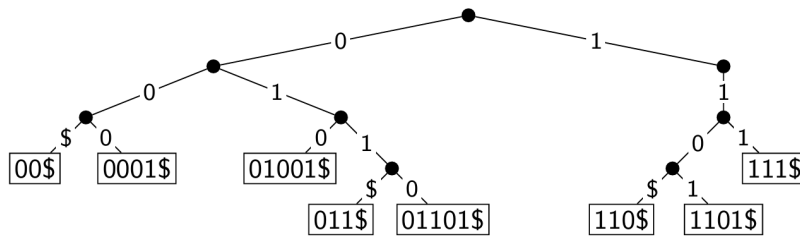
Disadvantages: Tries can be wasteful with respect to space. Worst-case space is $\Theta(n \cdot (\text{maximum length of a word}))$.

6.4 Variations of Tries: Pruned Tries

Definition 6.4: Pruned Trie

Stop adding nodes to trie as soon as the key is unique. In other words, a node has a child only if it has at least two descendants.

Example 6.1



Discovery 6.5

We have (implicitly) seen pruned tries before: For equal-length bitstrings: Pruned trie equals recursion tree of MSD radix-sort.

6.5 Variations of Tries: Compressed Tries

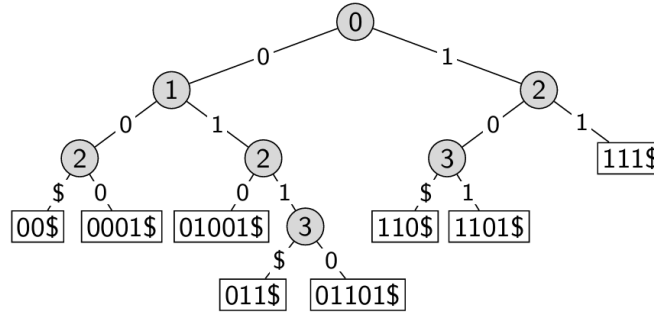
Definition 6.5: Compress Trie

Compress paths of nodes with only one child. Each node stores an index, corresponding to the level of the node in the uncompressed trie. (On level d , we searched for link with $w[d]$.)

Comment 6.4

Compressed Tries are also known as **Patricia-Tries**: Praticeal Algorithm to Retrieve Information Coded in Alphanumeric.

Example 6.2



Theorem 6.3

Compressed trie with n keys has at most $n - 1$ internal (non-leaf) nodes.

6.5.1 Compressed Trie: Search

Comment 6.5

Stored indices say which bits to compare; We must compare w to word found at the leaf.

Code 6.4

Algorithm 38: `CompressedTrie::get_path_to(w)`

```
1  $P \leftarrow \text{emptystack}; \quad z \leftarrow \text{root}; \quad P.\text{push}(z)$ 
2 while  $z$  is not a leaf and  $(d \leftarrow z.\text{index} \leq w.\text{size})$  do
3   if  $z$  has a child-link labelled with  $w[d]$  then
4      $z \leftarrow$  child at this link;  $P.\text{push}(z)$ 
5   end
6   else break;
7 end
8 return  $P$ 
```

Algorithm 39: `CompressedTrie::search(w)`

```
1  $P \leftarrow \text{get\_path\_to}(w), \quad z \leftarrow P.\text{top}()$ 
2 if ( $z$  is not a leaf or word stored at  $z$  is not  $w$ ) then
3   | return “not found”
4 end
5 return key-value pair at  $z$ 
```

6.5.2 Compressed Trie: Summary

Code 6.5

`search` and `prefix-search` are easy. `insert(w)` and `delete(w)` are conceptually simple.

Result 6.2

All operations take $O(|w|)$ time for word w . Use $O(n)$ space.

Comment 6.6

More complicated than standard tries, but space savings are worth it if words are unevenly distributed.

6.6 Multiway Tries: Larger Alphabet

Definition 6.6: Multiway Trie

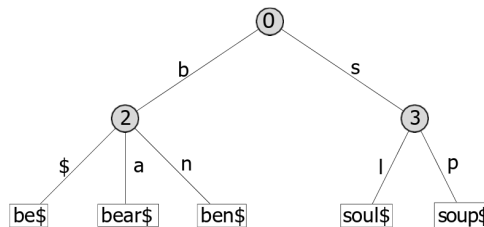
Multiway Trie represents Strings over any fixed alphabet Σ .

Discovery 6.6

Any node has at most $|\Sigma| + 1$ children, where $+1$ is the one child for the end-of-word character $\$$.

Example 6.3

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



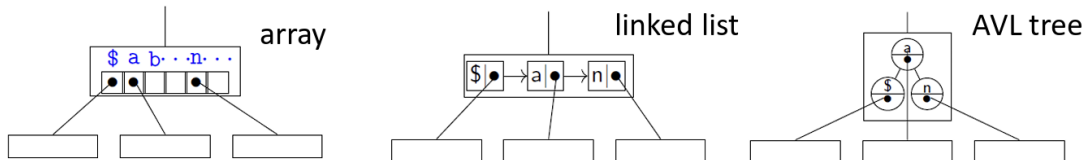
Discovery 6.7

Run-time $O(|w| \cdot (\text{time to find the appropriate child}))$.

Question 6.2.

How should children be stored?

Solution: We have several options:



However,

Result 6.3

Arrays are fast, lists are space efficient, AVL tree is best in theory, but not worth it in practice unless $|\Sigma|$ is huge.



Comment 6.7

In practice, use hashing (next section).

7 Midterm

Exercise 7.1

Order Notation

Prove from first principle that $n \in \omega(2^{\sqrt{\log n}})$.

Solution: Prove that

$$2^{\sqrt{\log n}} \leq \sqrt{n} \leq n$$

the rest follows easily.



Exercise 7.2

Order Notation

Disprove the following statement: if $f(n) \in o(n \log n)$, then $f(n) \in O(n)$.

Solution: Take $f(n) = n \log \log n$.



Exercise 7.3

Pseudocode Analysis

Given a tight bound on the running time as a function on n of the following pseudocode:

```
1      i = 2
2      x = 0
3      while (i < n):
4          for j = 1 to n:
5              for k = 1 to j:
6                  x = x + 1
7          i = i * i
```

Solution: We have $\lfloor \log \log n \rfloor$ iterations for the while loop, hence we have the tight bound:

$$\sum_{y=1}^{\log \log n} \sum_{j=1}^n \sum_{k=1}^j c \in \Theta(n^2 \log \log n)$$

as desired.



Exercise 7.4

Expected Runtime

Give a right big- O bound for the expected runtime of the following algorithm:

```
1      ArrayAlg(A, n, k)
```


```

2      // n = A.size()
3      // A is a permutation of [0, ..., n-1]
4      // k is in the set {0, ..., n-1}
5      i = random(n)
6      if A[i] == k then
7          return i
8      for j = 0 to n-1
9          print("a")
10     return ArrayAlg(A, n, k)

```

Solution: Define $R = \langle i, R' \rangle$ and so we have

$$T(A, R) = \begin{cases} c & \text{if } A[i] = k \\ cn + T(A, R') & \text{if } A[i] \neq k \end{cases}$$


Hence solving for the expected runtime we get $T^{exp}(n) \in O(n^2)$. 

Exercise 7.5

Priority Queue

Given a family k sorted arrays A_1, A_2, \dots, A_k , where the combination of the k arrays has n elements, provide an $O(n \log k)$ time algorithm that produces a single sorted array containing all n elements.

Hint: use a priority queue.

Solution: Create a min heap start with the first elements of all the A_i 's, keep deleting min and reinsert with the subsequent element in the corresponding A_i to obtain the sorted array of length n . Each `deleteMin` and `insert` takes $\log k$ and there are n of them so the total runtime is in $O(n \log k)$. 

Exercise 7.6

Epsilon

Let $0 < \epsilon < 1$. Suppose that we have an array A of n items such that the first $n - n^\epsilon$ items are sorted. Describe an $O(n)$ time algorithm to sort A .

Solution: Merge sort the unsorted part, and then merge the originally sorted with the newly sorted. Merge sort the unsorted part takes $O(n^\epsilon \log(n^\epsilon))$, which can be shown to be in $O(n)$ using the fact that

$$(\log n)^c \in o(n^d)$$

for any c and d constants. Then it follows easily that merge takes linear time as well.


Comment 7.1

The auxiliary space is also in $O(n)$. 

Exercise 7.7

Numbers in Range

We have an array A of n non-negative integers such that each integer is less than k . Provide an $O(n+k)$ time preprocessing algorithm such that queries of the form "how many integers are there in A that are in the range $[a, b]$?" can be answered in $O(1)$ time. Note that a and b are not fixed; they are parameters given to the query algorithm.

Solution: First bucket sort to count the number of each integers, which takes $O(n+k)$, and then for each of the integer ℓ , sum the number of integers in the range $[0, \ell]$, which takes $O(k)$, so that to find the number of integers in $[a, b]$, we can just calculate $B[b] - B[a-1]$ in $O(1)$ times. 

Exercise 7.8

Binary Search and Interpolation Search

T or F: Consider an algorithm that runs both Binary Search and Interpolation Search simultaneously until one of them stops. The worst-case runtime will be bounded by $O(\log n)$ and if the keys are uniformly distributed we would expect a runtime of $O(\log \log n)$.

Solution: True. 

Exercise 7.9

Heap

Suppose heaps were implemented as trees (instead of arrays). Given two heaps H_1 and H_2 of the same height, a new heap can be created by building a new node with infinite priority and adding the root nodes of H_1 and H_2 as the left and right children, and then calling `deleteMax`.

Solution: False. 

8 Dictionaries via Hashing

Recall *Direct Addressing*: Consider a special case where every key k is integer with $0 \leq k < M$ for some $M \in \mathbb{N}$. The implementation for it is:

Algorithm 8.1

- **store**(k, v) : in array A of size M via $A[k] \leftarrow v$;
- **search**(k) : check if $A[k]$ is empty;
- **insert**(k, v) : $A[k] \leftarrow v$;
- **delete**(k) : $A[k] \leftarrow \text{empty}$;

Discovery 8.1

Notice that all operations are $O(1)$, but total storage is $\Theta(M)$.

8.1 Hashing Introduction

Definition 8.1: Hashing (idea)

First map keys to small integer range and then use direct addressing.

Comment 8.1

Assumption: keys come from some universe U . (Typically $U = \{0, 1, \dots\}$, sometimes U is finite.)

Definition 8.2: Hash Function

We define **hash function** to be a function

$$h : U \rightarrow \{0, 1, \dots, M-1\}$$

and $h(k)$ is called **hash value** of k . Generally, hashing functions are not injective.

Example 8.1

For an instance: $h(k) = k \pmod{M}$.

We store dictionary in array T of size M called **hash table**.

Result 8.1

Hashing functions are fast ($O(1)$) to compute.

Discovery 8.2

However, as you might have noticed, collisions might happen.

Definition 8.3: Collision

Collision happens when we want to insert (k, v) , but $T[h(k)]$ is occupied.

Algorithm 8.2

When collisions happen, there are two main strategies to deal with it:

- **Chaining**: allow multiple items at each table location;
- **Open addressing**: alternative slots in array:
 - probe sequence: many alternative locations
 - * linear probing
 - * double hashing
 - cuckoo hashing: just one alternative location

8.2 Hashing with Chaining

Definition 8.4: Bucket

Each slot is a **bucket** containing 0 or more KVPs.

Definition 8.5: Chaining

Chaining refers to the approach that makes each bucket an unsorted linked list.

Code 8.1: Operations

- `search(k)` : look for key k in the list at $T[h(k)]$;

Comment 8.2

apply MTF heuristic.

- `insert(k, v)` : add (k, v) to the front of list at $T[h(k)]$;
- `delete(k)` : search and delete from the list at $T[h(k)]$.

8.2.1 Hash with Chaining: Running Time

Discovery 8.3

We have

- **insert** : $\Theta(1)$, essentially insertion of unordered linked list;
- **search** , **delete** : $\Theta(1 + \text{length at } T[h(k)])$.

Comment 8.3

We don't say $\Theta(\text{length at } T[h(k)])$ because the length could be 0.

Result 8.2

However, notice that the worst case could happen: n keys are hashed at the same slot.


Comment 8.4

To get meaningful average-case bounds, we need some assumptions on hash function and key, but it's hard to make realistic assumptions. On the contrary, it's easier to switch to randomized hashing.

Hashing with Chaining: Randomization

Question 8.1.

How can we randomize?

Solution: Notice that we cannot insert at a random location, as key k must hash to the hash value $h(k)$. Hence we assume hash-function is chosen randomly from a set of all hash functions. 

Theorem 8.1: Uniform Hashing Assumption (UHA)

Any possible hash-function is equally likely to be chosen.
(This is not realistic, but makes the analysis possible.)

Comment 8.5

In practice: chose a random hash function from a certain family of hash functions.

Example 8.2

Pick prime number $p > M$ and random $a, b \in \{0, \dots, p-1\}$ with $a \neq 0$, define

$$h(k) = (ak + b \pmod{p}) \pmod{M}$$

Notice that under UHA (any hash-function is chosen equally likely), we have

1. $P(h(k) = i) = \frac{1}{M}$ for any key k and slot i .

Proof. Let k, i be some key and slot. Let \mathcal{H}_j (for $j = 0, \dots, M-1$) be set of hash-functions h s.t. $h(k) = j$. Hence for $j \neq i$, there exists an one-to-one map between \mathcal{H}_j and \mathcal{H}_i . As a result, size of \mathcal{H}_j is equal to $1/M$ of all hash functions. \square

2. hash-values of any two keys are independent of each other.

Result 8.3

(1) and (2) mean that the distribution of keys is not important, chance of two keys colliding is $1/M$.

Definition 8.6: Load Factor (Probe Sequence)

We define load factor, α , to be n/M , where n is the number of items and M is the size of the hash table.

Theorem 8.2

For any key k , the expected size of bucket $T[h(k)]$ is at most $1 + \alpha$.

Proof. Suppose $h(k) = i$, so we have two cases:

1. *k is not in the dictionary:* then each of the n items in dictionary hashes to i with probability of $1/M$. Let $I_q^i = 1$ if key q hashes to i and $I_q^i = 0$ otherwise, so

$$\mathbb{E}[|T[i]|] = \mathbb{E}\left[\sum_{\text{key } q} I_q^i\right] = \sum_{\text{key } q} \mathbb{E}[I_q^i] = \frac{n}{M} \leq 1 + \alpha$$

2. *k is in the dictionary:* $T[i]$ definitely has key k , and the remaining $n-1$ dictionary items hash to i with probability $1/M$. Hence

$$\mathbb{E}[|T[i]|] = 1 + \frac{n-1}{M} \leq 1 + \alpha$$

thus we have completed our proof. \square

Result 8.4

Expected runtime of **search** and **delete** is $\Theta(1 + \alpha)$, **insert** is $\Theta(1)$. Space is $\Theta(M + n)$.

Discovery 8.4

Therefore, if we could maintain $\alpha \in \Theta(1)$, we can have **search** and **delete** in constant time as well.

8.2.2 Load Factor and Rehashing

Algorithm 8.3

- start with small M
- during `insert` / `delete`, update n
- if load factor becomes too big, rehash
 - chose new $M' \approx 2M$
 - find a new random hash function h' that maps U into $\{0, 1, \dots, M' - 1\}$
 - create new hash table T' of size M'
 - reinsert each KVP from T into T'
 - update $T \leftarrow T'$, $h \leftarrow h'$
- if load factor becomes too small, rehash with smaller M' .

Result 8.5

Rehashing costs $\Theta(M + n)$ but happens rarely, cost amortized over all operations.

8.3 Open Addressing

8.3.1 Open Addressing

Notice how chaining wastes space on links.

Question 8.2.

Can we resolve collisions in the array H ?

Solution: The idea is: each hash table entry holds only one item, but key k can go in multiple locations. 🎵

Definition 8.7: Probe Sequence

`search` and `insert` follow a **probe sequence** of possible locations for key k :

$$h(k, 0), h(k, 1), h(k, 2), \dots$$

until an empty spot is found.

Definition 8.8: Linear Probing

Linear probing is the simplest method for probe sequence: If $h(k)$ is occupied, place item in the next available location.

Comment 8.6

We assume circular array. i.e. modular arithmetic

$$h(k, i) = h(k) + i \pmod{M}$$

Discovery 8.5

Notice that if so, **delete** becomes problematic, because it would leave an empty spot which would make the next search not go far enough.

Solution: To solve this, one approach is *lazy deletion*: mark spot as deleted (rather than empty), and when perform searching, continue past deleted spots. Similarly, insert in empty or deleted spot.

We also need to keep track of how many items are deleted and rehash if there are too many.



8.3.2 Probe Sequence Operations

Code 8.2: Insert

```
1 probe-sequence::insert( $T, (k, v)$ )
2 for  $i = 0$  to  $M - 1$  do
3   if  $T[h(k, i)]$  is empty or deleted then
4      $T[h(k, i)] = (k, v)$  ;
5     return success ;
6   end
7 end
8 return failure to insert
```

Code 8.3: Delete

```
1 probe-sequence::search( $T, k$ )
2 for  $i = 0$  to  $M - 1$  do
3   if  $T[h(k, i)]$  is empty then return item-not-found ;
4   if  $T[h(k, i)]$  has key  $k$  then return  $T[h(k, i)]$  ;
   //  $T[h(k, i)] = \text{deleted}$  or not in the data structure therefore keep searching
5 end
6 return item not found
```

Result 8.6: Drawbacks of linear probing

Entries tend to cluster into contiguous regions; Many probes for each `search`, `insert`, and `delete`.

8.3.3 Double Hashing

Definition 8.9: Double Hashing

Double hashing: open addressing with probe sequence:

$$h(k, i) = h_0(k) + i \cdot h_1(k) \pmod{M} \quad \text{for } i = 0, 1, \dots$$

Where

- h_1 is a secondary hash function (step size) s.t. $h_1(k) \neq 0$;
- h_1 is relative prime with M for all keys k – because otherwise probe-sequence does not explore the entire hash table.

Comment 8.7

Easiest is to choose M prime.

Theorem 8.3

Double hashing with a good secondary hash function does not cause the bad clustering produced by linear probing.

Discovery 8.6

We use multiplicative method for second hash function:

- let $0 < A < 1$;
- $h(k) = M \underbrace{(kA - \lfloor kA \rfloor)}_{\in [0,1)}$.

Comment 8.8

Multiplying with A scrambles the keys. We should use at least $\log |U| + \log |M|$ bits of A .

Theorem 8.4

Taking $A = \varphi = \frac{\sqrt{5}-1}{2}$ works well. For double hashing, to ensure $0 < h(k) < M$, use

$$h_1(k) = \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor + 1$$

Example 8.3

Consider $M = 11$, $h_0(k) = k \bmod 11$, $h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$, and

$$h(k, i) = (h_0(k) + i \cdot h_1(k)) \bmod M$$

for sequence $i = 0, 1, \dots$. The table to the right demonstrate the cells we check when we try to perform

`insert(194)`

aside: $h_0(194) = 7$ and $h_1(194) = 9$, and

$$h(194, 0) = 7 + 0 \cdot 9 \bmod 11 = 7$$

$$h(194, 1) = 7 + 1 \cdot 9 \bmod 11 = 5$$

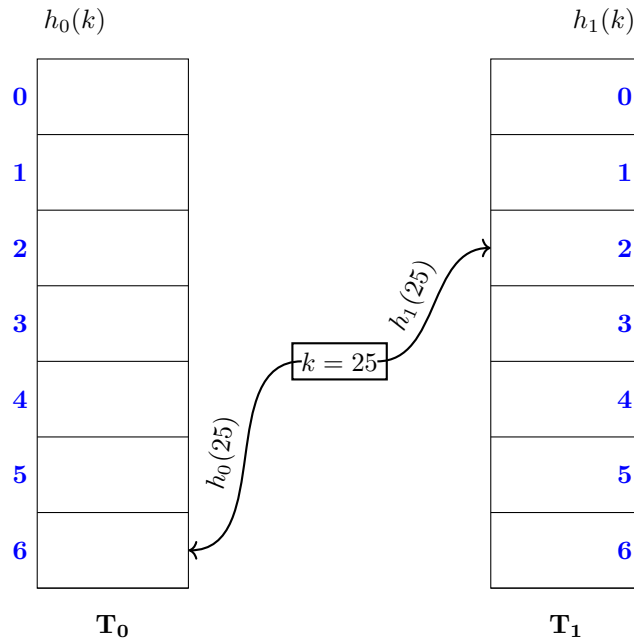
$$h(194, 2) = 7 + 2 \cdot 9 \bmod 11 = 3$$

Index	Value
0	
1	45
2	13
3	194
4	92
5	49
6	
7	7
8	41
9	
10	43

8.3.4 Cuckoo Hashing

The main idea of cuckoo hashing is that

An item with key k can be only at $T_0[h_0(k)]$ or $T_1[h_1(k)]$.



Discovery 8.7

`search` and `delete` take $O(1)$ time.

Algorithm 8.4

insert always initially puts key k into $T_0[h_0(k)]$.

- evict item that may have been there already
- if so, evicted item k' is inserted at $T_1[h_1(k')]$
- this may lead to a loop of evictions
- we can show that if insertion is possible, then there are at most $2n$ evictions
- so we abort after too many attempts

Code 8.4

Algorithm 40: cuckoo::insert(k, v)

```
1  $i \leftarrow 0$ ;  
2 do at most  $2n$  times  
3   if  $T_i[h_i(k)]$  is empty then  
4   |    $T_i[h_i(k)] \leftarrow (k, v)$ ;  
5   |   return "success";  
6   end  
7   // insert  $T_i[h_i(k)]$  into the other table;  
8   swap(( $k, v$ ),  $T_i[h_i(k)]$ ) // kick out current occupant;  
9    $i \leftarrow 1 - i$  // alternate between 0 and 1;  
10 return failure // re-hash;
```

Comment 8.9

Practical tip: we do not wait for $2n$ unsuccessful tries to declare failure. In practice, declare failure much earlier than $2n$.

Definition 8.10: Load Factor (cuckoo Hashing)

Load factor in cuckoo hashing is defined as

$$\alpha = n / (\text{size of } T_0 + \text{size of } T_1)$$

Theorem 8.5

Can show that if the load factor is small enough, $\alpha < 1/2$, then insertion has $O(1)$ expected time. but this wastes space.

Theorem 8.6

Can show expected space is $O(n)$.

8.3.5 Running Time of Open Addressing Strategies

Result 8.7

Under the restrictions of load factors and the Universal Hashing Assumption

- All strategies have $O(1)$ expected time for `search`, `insert`, `delete`
- Cuckoo hashing has $O(1)$ worst case for `search`, `delete`
- Probe sequence use $O(n)$ worst case space
- Cuckoo hashing uses $O(n)$ expected space

For any hashing, the worst case runtime is $\Theta(n)$ for `insert`.

Comment 8.10

In practice, double hashing is the most popular, or cuckoo hashing if there are many more searches than insertions.

8.4 Hash Function Strategies

Since satisfying the UHA is impossible, there are two ways to compromise:

- *Deterministic*: hope for a good performance by choosing a hash function that is
 - unrelated to any possible patterns in the data
 - Depends on all parts of the key
- *Randomized*: choose randomly among a limited set of functions, but aim for $P(\text{two keys collide}) = 1/M$.

8.4.1 Deterministic Hash Functions

Algorithm 8.5

Randomized Hash Functions: Carter-Wegman's Universal Hashing

- Modular method;
- Multiplicative method.

8.4.2 Randomized Hash Functions: Carter-Wegman's Universal Hashing

Algorithm 8.6

Requires: all keys are in $\{0, \dots, p-1\}$ for some (big) prime p . At initialization and whenever rehash

- choose number $M < p$, M equal to some power of 2 is ok.
- choose (and store) two **random** numbers $a, b \in \{0, \dots, p-1\}$
 - $b = \text{random}(p)$
 - $a = 1 + \text{random}(p-1)$, so that $a \neq 0$
- Use them as hash function

$$h(k) = ((ak + b) \mod p) \mod M$$

- can be computed quickly

Theorem 8.7

We can prove that two keys collide with probability at most $\frac{1}{M}$

Comment 8.11

Enough to prove the expected runtime bounds for chaining, although uniform hashing assumption is not satisfied.

8.4.3 Multi-dimensional Data

Question 8.3.

What if the keys are multi-dimensional, such as strings?

Solution: Standard approach is to **flatten** string w to integer $f(w) \in \mathbb{N}$, e.g.

$$\begin{aligned} A \cdot P \cdot P \cdot L \cdot E &\rightarrow (65, 80, 80, 76, 69) \quad (\text{ASCII}) \\ &\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R^1 + 69R^0 \\ &(\text{for some radix } R, \text{ e.g. } R = 255) \end{aligned}$$

We combine this with a modular hash function:

$$h(w) = f(w) \mod M$$



Code 8.5

To compute this in $O(|w|)$ time without overflow, use Horner's rule and apply mod early. For example,

$h(APPLE)$ is

$$(((((((65R + 80) \bmod M) R + 80) \bmod M) R + 76) \bmod M) R + 69) \bmod M$$

8.5 Hashing vs. Balanced Search Trees

Advantages of Balanced Search Trees

- $O(\log n)$ worst-case operation cost
- does not require any assumptions, special functions, or known properties of input distribution
- predictable space usage (exactly n nodes)
- never need to rebuild the entire structure
- supports ordered dictionary operations (rank, select, etc.)

Advantages of Hash Tables

- $O(1)$ expected time operations (if hashes are well-spread and load factor is small)
- can choose space-time trade-off via load factor
- cuckoo hashing achieves $O(1)$ worst-case for search & delete

9 Range-Searching in Dictionaries for Points

9.1 Range Search

Definition 9.1: Range Search

Operation `RangeSearch(x, x')` look for all items that fall within given range (interval) $Q = (x, x')$.

Comment 9.1

Q may have open or closed intervals.

As usual, n is the number of input items, and let s be the *output-size*, i.e. the number of items in the range. We need $\Omega(s)$ time just to report the items in the range, where s could be anything in $[0, n]$. Therefore, running time depends both on s and n . It is easy to get $O(n)$ times.

Question 9.1.

Can we achieve $O(\log n + s)$?

Solution: For unsorted list/array/hash table, we must check for each item explicitly if it is in the range, so range search requires $\omega(n)$ time.

For sorted array, we have:

Algorithm 9.1

$O(\log n)$: use binary search to find i s.t. x is at (or would be at) $A[i]$
 $O(\log n)$: use binary search to find i' s.t. x' is at (or would be at) $A[i']$
 $O(s)$: report all items in $A[i + 1 \dots i' - 1]$
 $O(1)$: report $A[i]$ and $A[i']$ if they are in the range

hence range search can be done in $O(\log n + s)$ time.

For BST, range search can be done in $O(\text{height} + s)$ time, which will be proven later.



9.2 Multi-Dimensional Data

Comment 9.2

Range searches are of special interest for multidimensional data: flights that leave between 9am and noon, and cost between \$400 and \$600.

9.2.1 Multi-Dimensional Range Search

Definition 9.2: Multi-Dimensional Range Search

(Orthogonal) d -dimensional range search is an operation such that given a query rectangle Q , it finds all points that lie within Q .

9.2.2 d -Dimensional Dictionary via 1-Dimensional Dictionary

We have two options:

Algorithm 9.2

1. Reduce to one-dimensional dictionary;
2. Use several dictionaries, one for each dimension.

Code 9.1: Reduce to one-dimensional dictionary

We combine d -dimensional key into one dimensional key: i.e.

$$(x, y) \mapsto x + y \cdot n^2$$

and hence two distinct (x, y) map to a distinct one dimensional key. This way, we can search for a specific key, but no efficient range search.

Code 9.2: Use several dictionaries, one for each dimension

The problem with this is that it wastes space and leads to inefficient search.

Example 9.1: Worst case example

- Insert all points into two dictionaries based on their x and y coordinates.
- Perform 1D range searches in both dictionaries, each returning half of the points.
- To find the intersecting points from both searches, insert the results from one dimension into an AVL tree and check for intersections with the other dimension.
- This method's complexity can approach $O(n \log n)$, worse than an exhaustive search ($O(s + \log n)$).

9.2.3 Multi-Dimensional Range Search (better idea)

Discovery 9.1

We design new data structures specifically for points.

Comment 9.3

We assume that points are in general position: no two x -coordinates or y -coordinates are the same. i.e, no two points on a horizontal lines, no two points on a vertical line.

Here are the new data structure we design to help multidimensional search:

Result 9.1

1. **Partition trees:** We will be studying two types of partition trees
 - (a) quadtrees: does not use general points position assumption
 - (b) kd-trees: uses general points position assumption
2. **Multi-dimensional range trees:** a tree that generalizes BST to support multidimensional search whose both internal and leaf nodes store points just like similar to one dimensional BST. It uses general points position assumption

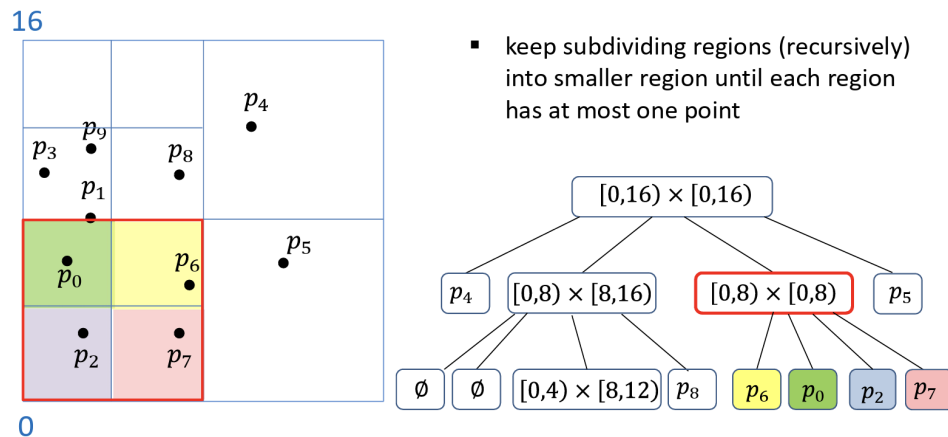
9.3 Quadtrees

Suppose we have a set of points in the plane, we need to find a *bounding box* $R = [0, 2^k) \times [0, 2^k)$ such that R is the smallest $2^k \times 2^k$ square covers all the points.

Code 9.3

keep subdividing regions (recursively) into smaller region until each region has at most one point.

Example 9.2



Comment 9.4

Convention: points on split lines belong to region on the right (or top).

9.3.1 search

Analogous to trie or BST.

Code 9.4

Three possibilities for where search ends

1. leaf storing point we search for (found)
2. leaf storing point different from search point (not found)
3. empty leaf (not found)

9.3.2 insert

Code 9.5

1. First perform search
2. Now we would have two cases:
 - (a) search finds a leaf storing one point, then repeatedly split the leaf **while** there are two points in one region
 - (b) else if search finds an empty leaf, we then simply insert the point into leaf
3. If we insert point outside the bounding box, no need to rebuild the part corresponding to the old tree, it becomes subtree in the new tree.

9.3.3 delete

Code 9.6

search will find a leaf containing the point, so we remove the point leaving the leaf empty, if parent now stores only one point in its region, we make parent node into a leaf storing its only child, and we check up the tree, repeating making any parent with only 1 point into a leaf. We do not make parent into a leaf as it stores multiple points.

9.3.4 Quadtree Analysis

Definition 9.3: Spread Factor

We define **spread factor** $\rho(S)$ of points S to be

$$\rho(S) = \frac{L}{d_{min}}$$

where L is the side length of R and d_{min} is the minimum distance between two points in S .

Theorem 9.1

In the worst case

$$h \in \Omega(\log \rho(S))$$

Proof. While the smallest region diagonal is $\geq d_{min}$, there are two points in the same region, and if the height is h , we need to perform h rounds of subdivisions. Notice that after this many subdivisions, the

smallest region has side length of $L/2^h$, and hence its diagonal is given by

$$\sqrt{2} \cdot \frac{L}{2^h}$$

recall that we want all the region to contain at most one points, hence we wish to have

$$\sqrt{2} \cdot \frac{L}{2^h} < d_{min}$$

which solves to be $h > \log(\sqrt{2}\rho(S))$. □

Theorem 9.2

In **any** case

$$h \in O(\log \rho(S))$$

Proof. let be v an internal node at depth $h - 1$, so there are at least two points p and q in this region. Remember that $d_{min} \leq d(p, q)$, and the the corresponding region has side length $L/2^{h-1}$. Recall taht the maximum distance between 2 points in such region is $\sqrt{2} \cdot L/2^{h-1}$. Hence we have

$$d_{min} \leq d(p, q) \leq \sqrt{2} \cdot \frac{L}{2^{h-1}}$$

which gives us $h \leq 1 + \log(\sqrt{2}\rho(S))$ □

Result 9.2

The complexity to build the initial tree is $\Theta(nh)$ worst case.

9.3.5 Quadtree Range Search

Code 9.7

Algorithm 41: Quadtree Range Search Algorithm

Input: $r \leftarrow \text{root}$, $Q \leftarrow \text{query rectangle}$

```

1 Function Qtree::RangeSearch( $r$ ,  $Q$ ):
2   let  $R$  be the region associated with  $r$ 
3   if  $R \subseteq Q$  then
4     report all points below  $r$  return
5   else if  $R \cap Q = \emptyset$  then
6     return                                     // outside node, stop search
7   if  $r$  is a leaf then
8      $p \leftarrow$  point stored at  $r$ 
9     if  $p$  is not NULL and in  $Q$  return  $p$  else return
10  for each child  $v$  of  $r$  do Qtree::RangeSearch( $v$ ,  $Q$ )

```

Discovery 9.2

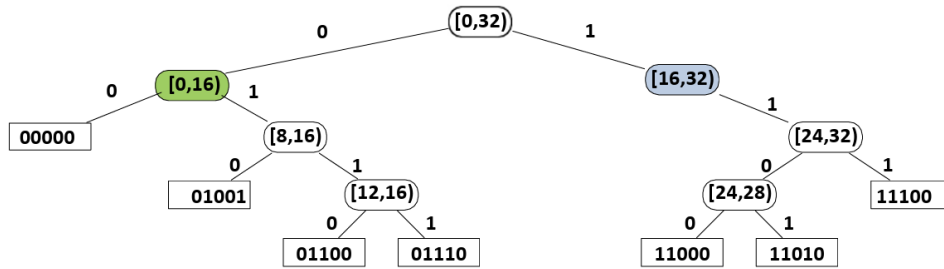
Running time is number of visited nodes output size. We don't have a good bound on number of visited nodes. In the worst case, we may have to visit nearly all nodes in the worst case, hence $\Theta(nh)$ is the worst-case.

Comment 9.5

In practice it is usually much faster.

9.3.6 Quadtree in Other Dimensions

points	0	9	12	14	24	26	28
base 2	00000	01001	01100	01110	11000	11010	11100



9.4 kd-Trees

Since quad trees can be extremely unbalanced, so we wish to introduce something new: kd-tree.

Definition 9.4: kd-Tree

For kd-Trees, we construct by splitting into regions with equal number of points, note that we can split either vertically or horizontally.

Comment 9.6

Alternating vertical and horizontal splits gives range search efficiency.

9.4.1 kd-tree Construction Running Time and Space

The expected time for partitioning S is $\Theta(n)$ with **QuickSelect**. Both subtrees have $\approx n/2$ points. Hence we have the following sloppy recurrence:

Theorem 9.3

We have

$$T^{exp}(n) = 2T^{exp}\left(\frac{n}{2}\right) + O(n)$$

which resolves to be $\Theta(n \log n)$ expected time.

Comment 9.7

We can improve the above to $\Theta(n \log n)$ worst-case runtime by pre-sorting coordinates.

Theorem 9.4

In terms of the height, we have the following recurrence relation:

$$h(0) = 1 \quad \text{and} \quad h(n) \leq h\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$$

which resolves to $O(\log n)$, specifically, $\lceil \log n \rceil$.

Comment 9.8

The above bound is tight.

Theorem 9.5

In terms of the space, all interior nodes have exactly 2 children, therefore there are $n - 1$ interior nodes, and hence the total number of nodes is $2n - 1$. Hence space is $\Theta(n)$.

9.4.2 kd-tree Dictionary Operations

Code 9.8

search as in binary search tree using indicated coordinate

insert first search, insert as new leaf

delete first search, remove leaf and any parent with one child

Question 9.2.

Notice that there are several problems with kd-Tree operations:

- after insert or delete, split might no longer be at exact median
- height is no longer guaranteed to be
- kd-tree do not handle insertion/deletion well

Solution: Remedy: allow a certain imbalance; re-building the entire tree when it becomes too unbalanced; However, **rangeSearch** will be slower. 

9.4.3 kd-Tree Range Search

Code 9.9**Algorithm 42:** $kdTree::RangeSearch(r \leftarrow root, Q)$ **Input:** r : root of kd-tree, Q : query rectangle

```

1  $R \leftarrow$  region associated with node  $r$  ;
2 if  $R \subseteq Q$  then
3   | report all points below  $r$  ;
4   | return ;
5 end
6 if  $R \cap Q = \emptyset$  return;
7 if  $r$  is a leaf then
8   |  $p \leftarrow$  point stored at  $r$  ;
9   | if  $p \in Q$  return  $p$  ;
10  | else return
11 end
12 foreach child  $v$  of  $r$  do
13   |  $kdTree::RangeSearch(v, Q)$  ;
14 end

```

9.4.4 kd-tree: Range Search Running Time

We visit blue, red, and green nodes, where there is constant work at each node. Hence runtime is proportional to the number of blue, red, green nodes.

Green Nodes: Recall that s is the number of nodes in the output of range search, so there are at most s leaves over all green subtrees, and, therefore, at most $2s$ nodes over all green subtrees. Hence

Theorem 9.6

Number of green nodes is $O(s)$.

Discovery 9.3

Notice that red nodes $\leq 2 \cdot$ blue nodes since each red node has a blue parent. Hence it is enough to count the number of blue nodes.

Blue Nodes: Let $B(n)$ to be the number of blue nodes.

Theorem 9.7

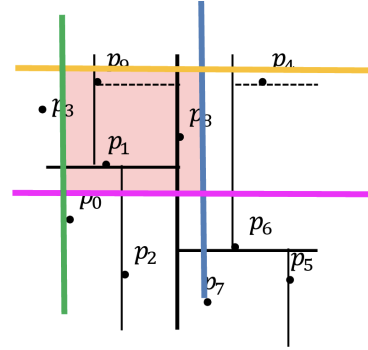
We can show that

$$B(n) \leq 2B\left(\frac{n}{4}\right) + O(1)$$

which can be resolved to $B(n) \in O(\sqrt{n})$.

Proof. We have

$$\begin{aligned} B(n) &\leq \# \text{ regions intersecting } \text{---} \\ &\quad \# \text{ regions intersecting } \text{---} \\ &\quad \# \text{ regions intersecting } \text{---} \\ &\quad \# \text{ regions intersecting } \text{---} \end{aligned}$$



We consider $\#$ regions intersecting --- , and so the other cases can be handled similarly. Define $B^x(n)$ to be

$$B^x(n) := \# \text{ of regions intersecting } \text{---}$$

if tree root is splitted by x coordinate. Hence we have

$$B^x(n) = 1 + B^y\left(\frac{n}{4}\right)$$

where $B^y\left(\frac{n}{2}\right) = 1 + 2 \cdot B^x\left(\frac{n}{4}\right)$. Combining them we get our desired result. \square

Result 9.3

Therefore, the running time for range search is

$$O(s + \sqrt{n})$$

9.4.5 kd-Tree — Higher Dimension

Consider a kd-Tree for d -dimensional space, we have

$$\begin{aligned} \text{storage} &: O(n); \\ \text{height} &: O(\log n); \\ \text{Construction Time} &: O(n \log n); \\ \text{Range query time} &: O\left(s + n^{1-1/d}\right), \quad \text{where we assume } d \text{ is a constant.} \end{aligned}$$

9.5 Range Trees

Question 9.3.

Our question is: can we use ideas from BST/AVL trees for multi dimensional dictionaries?

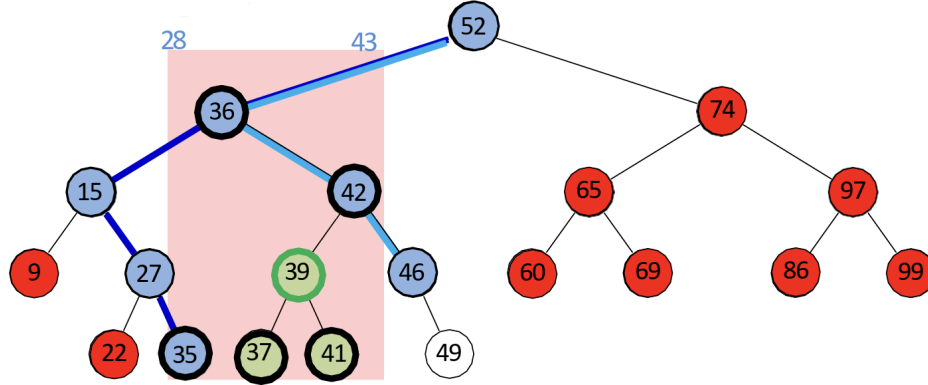
9.5.1 BST Range Search

Example 9.3

First let us consider range search in BST:

`BST::RangeSearch(T, 28, 43)`

where T is:



Notice that we have the following three types of nodes:

- **Boundary nodes:** nodes on P_1 and P_2 . We need to check if boundary nodes are in the search range.
- **Outside nodes:** nodes that are left of P_1 or right of P_2 , in this case, nodes are not in search range so range search never examines them.
- **Inside nodes:** nodes that are right of P_1 and left of P_2 , so we keep a list of topmost inside nodes, and all descendants of topmost inside node are in the range, so we just report them.

9.5.2 How to Find Top Inside Node

Theorem 9.8

A node v is a **top inside node** if:

- v is not in P_1 or P_2 ;
- parent of v is in P_1 or P_2 (but not both)
 - If parent is in P_1 , then v should be the right child of it, otherwise left child.

Comment 9.9

Top inside nodes are important for efficient 2D range search, they are also important if need to just count the number of points in the search range.

9.5.3 BST Range Search Analysis

Comment 9.10

We assumed that we have balanced BST.

Running time consists of

1. search for path P_1 is $O(\log n)$
2. search for path P_2 is $O(\log n)$
3. check if boundary nodes in the range
 - $O(1)$ at each **boundary node**, there are $O(\log n)$ of them, $O(\log n)$ total time
4. spend $O(1)$ at each topmost inside node
 - since each topmost inside node is a child of boundary node, there are at most $O(\log n)$ topmost inside nodes, so total time $O(\log n)$
5. report descendants in subtrees of all topmost inside nodes
 - topmost nodes are disjoint, so #descendants for inside topmost nodes is at most s , output size

Result 9.4

Hence total time is $O(s + \log n)$.

9.5.4 2D Range Tree

Definition 9.5: 2D Range Tree

Suppose we have points: $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$. **Range tree** is a **tree of trees** (a *multi-level* data structure)

- **Primary structure**
 - balanced BST T storing S and uses **x -coordinates** as keys
 - assume T is balanced, so height is $O(\log n)$
- Each node v of T stores an **associated tree** $T(v)$, which is a balanced BST
 - let $S(v)$ be all descendants of v in T , including v
 - $T(v)$ stores $S(v)$ in BST, using **y -coordinates** as key
 - * note that v is not necessarily the root of $T(v)$

Algorithm 9.3: Range search in 2D Range Tree Overview

Suppose we want to perform `RangeTree::RangeSearch(T, 5, 14, 5, 9)`, we wish to perform the following:

1. Perform *modified* **BST-RangeSearch**($T, 5, 14$)

- find boundary and topmost inside nodes, but **do not** go through the inside subtrees
 - modified version takes $O(\log n)$ time
 - does not visit all the nodes in valid range for $BST\text{-}RangeSearch(T, 5, 14)$
2. Check if boundary nodes have valid x -coordinate **and** valid y -coordinate
 3. For every topmost inside node v , search in associated tree $BST::RangeSearch(T(v), 5, 9)$

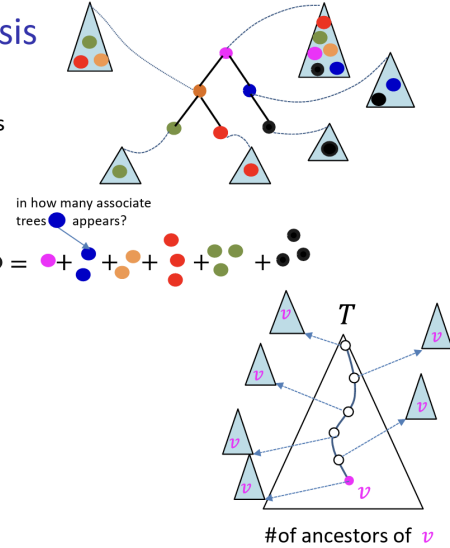
9.5.5 Range Tree Space and Time Analysis

Range Tree Space Analysis

- Primary tree T uses $O(n)$ space
- For each v , associated tree $T(v)$ uses $O(|T(v)|)$ space
- Space for all associated trees is

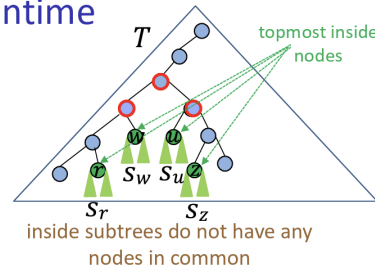
$$\begin{aligned}
 \sum_{v \in T} |T(v)| &= \text{[diagram showing sum of node counts for each node's subtree]} \\
 &= \sum_{v \in T} \underbrace{\text{\#of ancestors of } v}_{\leq \log n} \\
 &\leq \sum_{v \in T} \log n = cn \log n
 \end{aligned}$$

- Space is $O(n \log n)$
 - in the worst case, have $n/2$ leaves at the last level, and space needed is $\Theta(n \log n)$



Range Trees: Range Search Runtime

- Find boundary nodes in the primary tree and check if keys are in the range
 - $O(\log n)$
- Find topmost inside nodes in primary tree
 - $O(\log n)$
- For each topmost inside node v , perform range search for y -range in associate tree
 - $O(\log n)$ topmost inside nodes
 - let s_v be #items returned for the subtree of topmost node v
 - running time for one search is $O(\log n + s_v)$



$$\sum_{\text{topmost inside node } v} c(\log n + s_v) = \underbrace{\sum_{\text{topmost inside node } v} c \log n}_{O(\log^2 n)} + \underbrace{\sum_{\text{topmost inside node } v} c s_v}_{\leq cs}$$

- Time for range search in range tree: $O(s + \log^2 n)$
 - can make this even more efficient, but this is beyond the scope of the course

Result 9.5

Range trees can be generalized to d -dimensional space:

$$\begin{array}{ll}
 \text{storage} & : \quad O(n(\log n)^{d-1}); \\
 \text{Construction Time} & : \quad O(n(\log n)^d); \\
 \text{Range query time} & : \quad O(s + (\log n)^d)
 \end{array}$$

9.6 Conclusion

Quadtrees

- simple, easy to implement insert/delete (i.e. dynamic set of points)
- work well only if points evenly distributed
- wastes space, especially for higher than two dimensions

kd-trees

- linear space
- range search is $O(s + \sqrt{n})$
- inserts/deletes destroy balance and range search time
 - fix with occasional rebuilt

Range trees

- fastest range search $O(\log^2 n + s)$
- wastes some space
- insert and delete destroy balance, but can fix this with occasional rebuilt

10 String Matching

Definition 10.1: String Matching

The problem is about searching for a string (pattern) in a large body of text. Here are some of the notations:

1. $T[0, \dots, n-1]$: text (or haystack) being searched;
2. $P[0, \dots, m-1]$: pattern (or needle) being searched for.

Comment 10.1

Convention: return the first occurrence of P in T .

antidisestablishmentarianism

Definition 10.2: Substring

- **Substring** $T[i \dots j]$ $0 \leq i \leq j+1 \leq n$ is a string $T[i], T[i+1], \dots, T[j]$
 - length is $j-i+1$
 - empty string included: $T[i \dots i-1]$

Definition 10.3: Prefix

- **Prefix** of T is a substring $T[0 \dots i-1]$ of T for some $0 \leq i \leq n$
 - empty prefix included: $T[0 \dots -1]$

Definition 10.4: Suffix

- **Suffix** of T is a substring $T[i \dots n-1]$ of T for some $0 \leq i \leq n$
 - empty suffix included: $T[n \dots n-1]$

Definition 10.5: Empty String

- The **empty substring** is usually denoted by Λ

10.1 Introduction

Algorithm 10.1

Pattern matching algorithms consist of *guesses* and *checks*.

Definition 10.6: Guess

A **guess** is a position i such that P might start at $T[i]$.

Definition 10.7: Check

A **check** of a guess is a single position j with $0 \leq j < m$ where we compare $T[i + j]$ to $P[j]$.

Comment 10.2

Note that

- must perform m checks of a single correct guess;
- may make fewer checks of an incorrect guess.

10.2 Brute Force Algorithm

Code 10.1

```

1  Bruteforce::PatternMatching(T [0..n - 1], P[0..m - 1])
2  T : String of length n (text), P : String of length m (pattern)
3
4  for i <- 0 to n - m do
5      if strcmp(T, P, i, m) = 0
6          return "found at guess i"
7  return FAIL

```

where `strcmp` takes $\Theta(m)$ time.

```

1  strcmp(T, P, i <- 0, m <- P.size())
2  // compare m chars of T and P, starting at T[i]
3
4  for j <- 0 to m - 1 do
5      if T[i + j] is before P[j] in  $\Sigma$  then return -1
6      if T[i + j] is after P[j] in  $\Sigma$  then return 1
7  return 0

```

Example 10.1: Worst Possible Input

Worst possible input is

$$P = \underbrace{a \dots a}_{m-1 \text{ terms}} b, \quad T = \underbrace{aaa \dots aaa}_n$$

Since we perform $(n - m + 1)m$ checks, the time is $\Theta((n - m)m)$.

Question 10.1.

How can we improve?

Solution: Improvement via Preprocessing. There are two preprocessing options for pattern matching

1. Do preprocessing on pattern P , eliminate guesses based on preprocessing:
 - Karp-Rabin
 - KMP
 - Boyer-Moore
2. Do preprocessing on text T , create a data structure to find matches easily:
 - Suffix-tree
 - Suffix-arrays



10.3 Karp-Rabin Algorithm

The idea is

use hash values (called **fingerprints**) to eliminate guesses

where hash function $h : \{\text{string of length } m\} \rightarrow \{0, \dots, M-1\}$.

Comment 10.3

if $h(P) \neq h(guess)$ then guess cannot work; if $h(P) = h(guess)$ verify with `strcmp` if pattern matches text.

Example 10.2

Example: $\Sigma = \{0-9\}$, $P = 92653$, $T = 31415926535$: Precompute:

$$h(P) = h(92653) = 18$$

and we have the following table showing the process:

	3	1	4	1	5	9	2	6	5	3	5	
no strcmp	fingerprint 84											$h(31415) = 84$
no strcmp		fingerprint 94										$h(14159) = 94$
no strcmp			fingerprint 76									$h(41592) = 76$
do strcmp, false				fingerprint 18								$h(15926) = 18$
no strcmp					fingerprint 95							$h(59265) = 95$
do strcmp, true						fingerprint 18						$h(92653) = 18$

Discovery 10.1

For each guess, we need $\Theta(m)$ time to compute hash value. Note that this method is thus worse than brute force, as the total time now is

$$\Theta(mn)$$

How can we improve this?

Solution: compute next hash from previous one in $O(1)$ time.



Example 10.3

Consider

$$T = 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5$$

The initialization of the algorithm is:

1. compute first fingerprint: $h(41592) = 41592 \bmod 97 = 76$
2. also pre-compute $T^{m-1} \bmod M$ (here is $10000 \bmod 97 = 9$).

Down below is an example showing how we can get $h(15926)$ from $h(41592)$ in $O(1)$ time.

$$41592 \xrightarrow{-4 \cdot 10000} 1592 \xrightarrow{\times 10} 15920 \xrightarrow{+6} 15926$$

Algebraically,

$$(41592 - (4 \cdot 10000)) \cdot 10 + 6 = 15926$$

Hence

$$(41592 - (4 \cdot 10000)) \cdot 10 + 6 = 15926$$

$$((41592 - (4 \cdot 10000)) \cdot 10 + 6) \bmod 97 = 15926 \bmod 97$$

$$(((41592 \bmod 97) - (4 \cdot (10000 \bmod 97))) \cdot 10 + 6) \bmod 97 = 15926 \bmod 97$$

$$((76 - (4 \cdot 9)) \cdot 10 + 6) \bmod 97 = 15926 \bmod 97$$

Code 10.2

```

1  Karp-Rabin-RollingHash::PatternMatching(T, P)
2      M ← suitable prime number
3      hP ← h(P[0...m − 1])
4      hT ← h(T[0...m − 1])
5      s ← Rm−1 mod M
6
7      for i ← 0 to n − m
8          if hT = hP
9              if strcmp(T, P, i, m) = 0
10                 return "found at guess i"
11             if i < n − m // compute fingerprint for next guess

```

```

12          $h_T \leftarrow ((h_T - T[i] \cdot s) \cdot R + T[i + m]) \bmod M$ 
13     return FAIL

```

Comment 10.4

Choose “table size” M at random to be prime in $\{2, \dots, mn^2\}$.

Result 10.1

We can show that expected running time is $O(m + n)$. Although $\Theta(mn)$ is the worst-case, but this is extremely unlikely.

10.4 Knuth-Morris-Pratt algorithm

Algorithm 10.2

We first preprocess the pattern: initialize a **failure array** F with the same length as P , and we

Store the length of the longest valid suffix of $P[1, \dots, j]$ in $F[j]$

Example 10.4

Suppose we have $P = ababaca$,

- $j = 0$, $P[1 \dots 0] = \text{“”}$, $P = abacaba$, longest valid suffix is “”

Comment 10.5

$F[0] = 0$ for any pattern.

- $j = 1$, $P[1 \dots 1] = b$, $P = abacaba$, longest valid suffix is “”
- $j = 2$, $P[1 \dots 2] = ba$, $P = abacaba$, longest valid suffix is **a**
- $j = 3$, $P[1 \dots 3] = bab$, $P = abacaba$, longest valid suffix is **ab**
- $j = 4$, $P[1 \dots 4] = baba$, $P = abacaba$, longest valid suffix is **aba**
- $j = 5$, $P[1 \dots 5] = babac$, $P = abacaba$, longest valid suffix is “”
- $j = 6$, $P[1 \dots 6] = babaca$, $P = ababaca$, longest valid suffix is **a**

and hence the **failure array** F is:

F	0	1	2	3	4	5	6
	0	0	1	2	3	0	1

Discovery 10.2

Failure array is precomputed before matching starts. A straightforward computation is $O(m^3)$ time, but we can show that there is a way of computing it in $O(m)$ time later.

Now we introduce to you the actual algorithm:

Algorithm 10.3

Comment 10.6

KMP starts similar to brute force pattern matching:

- it maintains variables i and j such that j is the position in the pattern and i is the position in the text where we do the check. Check is performed by determining if

$$T[i] = P[j]$$

hence **current guess is** $i - j$.

1. Begin matching with $i = 0, j = 0$;
2. If $T[i] \neq P[j]$ and $j = 0$, we shift pattern by 1, which same action as in brute-force:

$$i = i + 1 \quad \text{and} \quad j \text{ is unchanged}$$

current guess is $(i + 1) - j$.

3. If $T[i] = P[j]$ and $j = 0$, we shift pattern by 1, which same action as in brute-force:

$$i = i + 1 \quad \text{and} \quad j = j + 1$$

current guess is also unchanged: $(i + 1) - (j + 1)$.

4. When failure is at pattern position $j > 0$, do something smarter than brute force:

$$i \text{ stays the same} \quad \text{and} \quad j = F[j - 1]$$

Discovery 10.3

The idea is that since some subpattern has already been read, so we start from the next position in the subpattern so that we can skip the part that has been read to save time.

Code 10.3

```

1  KMP::pattern-matching(T, P)
2  F ← compute-failure-array(P)
3  i ← 0 // current character of T
4  j ← 0 // current character of P
5  while i < n do
6      if P[j] = T[i]
7          if j = m - 1
8              return "found at guess  $i - m + 1$ "
9              // guess is equal to  $i - j$ 
10         else // rule 1
11             i ← i + 1
12             j ← j + 1
13         else //  $P[j] \neq T[i]$ 
14             if j > 0
15                 j ← F[j - 1] // rule 2
16             else
17                 i ← i + 1 // rule 3
18 return FAIL

```

Example 10.5

Here is an exmaple:

String matching with KMP: Example

- $T = cabababcababaca, P = ababaca$

 F

0	1	2	3	4	5	6
0	0	1	2	3	0	1

Diagram illustrating the execution of the Longest Common Subsequence (LCS) algorithm for the strings $T = \text{cababab}$ and $P = \text{abcaba}$. The diagram shows the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

The top part shows the initial state where $j=0$ and $j=2$ are highlighted, indicating the start of the iteration.

The middle part shows the state after the first iteration, where $j=1$ and $j=3$ are highlighted, indicating the start of the next iteration.

The bottom part shows the state after the second iteration, where $j=2$ and $j=4$ are highlighted, indicating the start of the next iteration.

The diagram also includes a table showing the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
T	c	a	b	a	b	a	b
P	a						
		a	b	a	b	a	c
				(a)	(b)	(a)	b
					(a)	(b)	a
						a	
							a

The diagram also includes a table showing the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
T	c	a	b	a	b	a	b
P	a						
		a	b	a	b	a	c
				(a)	(b)	(a)	b
					(a)	(b)	a
						a	
							a

The diagram also includes a table showing the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
T	c	a	b	a	b	a	b
P	a						
		a	b	a	b	a	c
				(a)	(b)	(a)	b
					(a)	(b)	a
						a	
							a

The diagram also includes a table showing the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
T	c	a	b	a	b	a	b
P	a						
		a	b	a	b	a	c
				(a)	(b)	(a)	b
					(a)	(b)	a
						a	
							a

The diagram also includes a table showing the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
T	c	a	b	a	b	a	b
P	a						
		a	b	a	b	a	c
				(a)	(b)	(a)	b
					(a)	(b)	a
						a	
							a

The diagram also includes a table showing the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
T	c	a	b	a	b	a	b
P	a						
		a	b	a	b	a	c
				(a)	(b)	(a)	b
					(a)	(b)	a
						a	
							a

The diagram also includes a table showing the state of the algorithm at various points, with indices i and j indicating the current position in the strings.

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
T	c	a	b	a	b	a	b
P	a						
		a	b	a	b	a	c
				(a)	(b)	(a)	b

Result 10.2

KMP Running Time is $O(n)$.

10.4.1 Fast Computation of Failure Array F

Code 10.4

```
1  // compute-failure-array(P)
2  // P: string of length m (pattern)
3
4  F[0] <- 0
5  j <- 1 // matching P[1 ... j]
6  l <- 0
7
8  while j < m do
9      if P[j] = P[l] then // rule 1
10         l <- l + 1
11         F[j] <- l
12         j <- j + 1
13     else if l > 0 then // rule 2
14         l <- F[l - 1]
15     else // rule 3
16         F[j] <- 0 // l = 0
17         j <- j + 1
```

10.5 Boyer-Moore Algorithm

Comment 10.7

Fastest pattern matching in practice on English Text.

Algorithm 10.4

Important components: **Reverse-order searching**, which compare P with a guess moving backwards. When a mismatch occurs choose the better option among the two below:

- Bad character heuristic – eliminate shifts based on mismatched character of T .
- Good suffix heuristic – eliminate shifts based on the matched part (i.e.) suffix of P .

10.5.1 Reverse Searching

Example 10.6: Reverse Searching

Consider

$T = \text{whereiswaldo},$ and $P = \text{aldo}$

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

r does not occur in $P = \text{aldo}$, so we move pattern past **r**. **w** does not occur in $P = \text{aldo}$, so again we move pattern past **w**.

Discovery 10.4

Bad character heuristic can rule out many guesses with reverse searching.

Question 10.2.

What if Mismatched Text Character Occurs in P ?

Solution: Find last occurrence of the mismatched character in P and move the pattern to the right until the character's last occurrence in P aligns with that in text. This is because if we moved until the first of that character in P aligns with that in text, we could miss an earlier guess which is also possible:

Example 10.7

$T = \text{acranapple},$ and $P = \text{aaron}$

a	c	r	a	n	a	p	p	l	e	
			o	n						
		a	a	r	o	n				missed valid guess
			a	a	r	o	n			also a valid guess

After aligning, we keep matching the pattern in reverse.



This means that we need to precompute our *last occurrence array*:

Definition 10.8: Last Occurrence Array

Compute the **last occurrence array** $L(c)$ of any character in the alphabet:

1. $L(c)$ if character c does not occur in P , otherwise;
2. $L(c)$ is the largest index j such that $P[j] = c$.

Result 10.3

Total time for computing the Last Occurrence Array is $O(m + |\Sigma|)$.

10.5.2 Boyer-Moore Indexing

Comment 10.8

Same as KMP.

Algorithm 10.5

1. maintain variables i and j
2. j is the position in the pattern
3. i is the position in the text where we do the next check
4. check is performed by determining if $T[i] = P[j]$
5. current guess is $i - j$

When mismatch occurs at text position i , pattern position j , update

- $j = m - 1$;
- $i = i + m - 1 - L(c)$

(this is not completed yet... see more in the discovery below).

Comment 10.9

Notice that formula also works if $L(c) = -1$, which is when the mismatched character does not appear in the pattern.

Discovery 10.5

Notice that the old guess was $i - j$ and the new guess is $i - L(c)$, so we are actually going backward if $L(c) > j$. Therefore, if we have $L(c) > j$, we perform the brute force rule: i.e., update

$$\begin{aligned}j &= m - 1 \\i &= i - j + m\end{aligned}$$

Theorem 10.1

Unified formula for Bad Character Heuristic:

$$i = i + m - 1 - \min\{L(c), j - 1\}$$

Code 10.5

```
1  BoyerMoore(T, P)
2  L ← last occurrence array computed from P
3  j ← m - 1
4  i ← m - 1
5
6  while i < n and j ≥ 0 do // current guess begins at index i-j
7      if T[i] = P[j] then
8          i ← i - 1
9          j ← j - 1
10     else
11         i ← i + m - 1 - min{L(c), j - 1}
12         j ← m - 1
13
14     if j = -1 return "found at guess i + 1"
15     else return FAIL
```

Comment 10.10

We will not study the precise way to do Good Suffix Heuristic.

10.6 Suffix Trees

Question 10.3.

What if we search for many patterns P within the same fixed text T ?

Solution: Preprocess the text T rather than pattern P .



Discovery 10.6

P is a substring of T if and only if P is a prefix of some suffix of T .

The naive idea is: store all suffixes of T in a trie. If $|T| = n$, then $n + 1$ suffixes together have $0 + 1 + 2 + \dots + n \in \Theta(n^2)$ characters, which wastes space.

Definition 10.9: Suffix Tree

Suffix tree saves space in multiple ways

1. store suffixes implicitly via indices into T
2. use compressed trie

This is only $O(n)$ space since we store $n + 1$ suffixes (words).

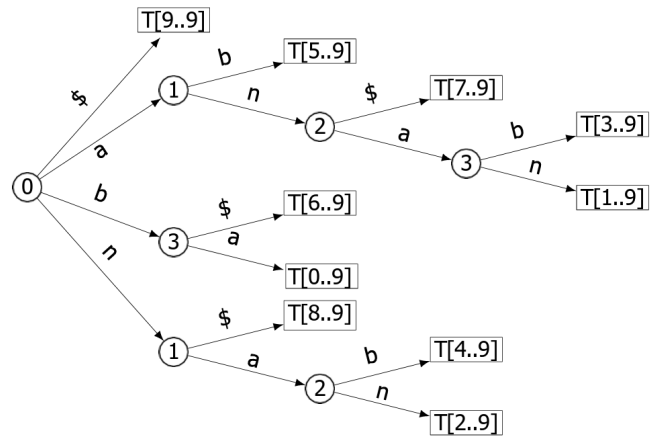
Example 10.8

Suffix tree

$$T =$$

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

- Compress trie of suffixes to get **suffix tree**



10.6.1 Suffix Tree Summary

- Building**
 - text T has n characters and $n + 1$ suffixes
 - can build suffix tree by inserting each suffix of T into compressed trie
 - $\Theta(|\Sigma|n^2)$ time
 - there is a way to build a suffix tree of T in $\Theta(|\Sigma|n)$ time
 - beyond the course scope
- Pattern Matching**
 - prefix-search* for P in compressed trie
 - run-time is
 - $O(|\Sigma|m)$, assuming a node stores children in a linked list
 - $O(m)$, assuming a node stores children in an array
- Summary**
 - theoretically good, but construction is slow or complicated and lots of space-overhead
 - rarely used in practice

10.7 Suffix Arrays

The idea is

- store suffixes implicitly, by storing start indices;
- store sorting permutation of the suffixes of T .

Example 10.9

We have

	1	2	3	4	5	6	7	8	9	10
T =	b	a	n	a	n	a	b	a	n	\$
Suffix Array =	9	5	7	3	1	6	0	8	4	2
	0	1	2	3	4	5	6	7	8	9

i	suffix $T[i \dots n]$
0	bananaban\$
1	ananaban\$
2	nanaban\$
3	anaban\$
4	naban\$
5	aban\$
6	ban\$
7	an\$
8	n\$
9	\$

→ sort lexicographically

j	$A^s[j]$	
0	9	\$
1	5	aban\$
2	7	an\$
3	3	anaban\$
4	1	ananaban\$
5	6	ban\$
6	0	bananaban\$
7	8	n\$
8	4	naban\$
9	2	nanaban\$

10.7.1 Suffix Array Construction

Theorem 10.2

We do not need n rounds. We can prove that $\Theta(\log n)$ rounds is enough, which produces $\Theta(n \log n)$ run time. $\Theta((n + \Sigma) \log n)$ if accounting for alphabet size.

Comment 10.11

Construction-algorithm: MSD-radix sort plus some bookkeeping, which needs only one extra array and is easy to implement (details are covered in an algorithms course).

10.7.2 Suffix Arrays - Algorithm Pseudocode

Code 10.6

```

1  // SuffixArray-Search(T, P, As)
2  // As: suffix array of T, P: pattern
3  l ← 0, r ← last index of As
4  while l ≤ r do
5      v ← floor((l + r) / 2)
6      i ← As[v]
7
8      // assume strcmp handles out of bounds suitably
9      s ← strcmp(T, P, i, m)
10
11     if (s < 0) do l ← v + 1
12     else if (s > 0) do r ← v - 1
13     else return "found at guess i"
14 return FAIL

```

10.8 Conclusion

	Brute Force	KR	BM	KMP	Suffix Trees	Suffix Array
preproc.	—	$O(m)$	$O(m + \Sigma)$	$O(m)$	$O(\Sigma n^2)$ → $O(\Sigma n)$	$O(n \log n)$ → $O(n)$
search time (preproc excluded)	$O(nm)$	$O(n + m)$ expected	$O(n + \Sigma)$ with good suffix often better	$O(n)$	$O(m(\Sigma))$	$O(m \log n)$ → $O(m + \log n)$
extra space	—	$O(1)$	$O(m + \Sigma)$	$O(m)$	$O(n)$	$O(n)$

Comment 10.12

Algorithms stop once they found one occurrence. However, most of them can be adapted to find all occurrences within the same worst-case run-time.

11 Compression

11.1 Background

Question 11.1.

How to store and transmit data efficiently?

Definition 11.1: Source Text

Source text is the original data, string S of characters from source alphabet Σ_S .

Definition 11.2: Coded Text

Coded text is the encoded data, string C of characters from coded alphabet Σ_C .

Comment 11.1

We consider *lossless compression*.

Question 11.2.

Main objective: for data compression, we want to minimize the size of the coded text.

Definition 11.3: Compression Ratio

Compression Ratio is defined as:

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$$

Discovery 11.1

We also measure efficiency of encoding/decoding algorithms, as for any usual algorithm, we always need time $\Omega(|S| + |C|)$, and sometimes we need even more time.

Other possible goals include *,

- reliability (e.g. error-correcting codes)
- security (e.g. encryption)

Discovery 11.2

No lossless encoding scheme can have compression ratio < 1 for all input strings.

*These are not studied in this course

11.1.1 Detour: Streams

Usually texts are huge and do not fit into computer memory, so we usually store S and C as streams.

Code 11.1

Input stream: (`std::cin`)

- read one character at a time
- `pop()`, `top()`, `isEmpty()`
- sometimes need `reset()` to start processing from the start

Output stream: (`std::cout`)

- write one character at a time
- `append()`, `isEmpty()`

Comment 11.2

Advantage of stream include:

- can start processing text while it is still being loaded
- avoids needing to hold the entire text in memory at once

11.2 Single-Character Encodings

Definition 11.4: Character Encoding

A **character encoding** E (or single-character encoding) maps each character in the source alphabet to a string in code alphabet:

$$\begin{aligned} E : \Sigma_S &\rightarrow \Sigma_C^* \\ c &\mapsto E(c) \end{aligned}$$

Definition 11.5: Fixed-Length Code

Fixed-length code: all codewords have the same length.

Definition 11.6: Variable-Length Code

Variable-length code: codewords may have different lengths.

11.2.1 Fixed Length Character Encoding

Example 11.1

ASCII (American Standard Code for Information Interchange), 1963, where each codeword $E(c)$ has length 7 bits.

Example 11.2

Other (earlier) fixed-length codes: Baudot code, Murray code.

Discovery 11.3

Fixed-length codes do not compress.

Proof. let $E(c) = b$ and assume binary code alphabet, then

$$\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot |\Sigma_S|} = \frac{b \cdot |S|}{|S| \cdot \log 2^b} = 1$$

□

11.2.2 Variable-Length Encoding

Comment 11.3

Some alphabet letters occur more often than others. Hence the idea is to use shorter codes for more frequent characters.

Example 11.3

Morse code and UTF-8 encoding of Unicode.

11.2.3 Variable-Length — Encoding

Code 11.2: Encoding

Assume we have some character encoding $E : \Sigma_S \rightarrow \Sigma_C^*$, so E is a dictionary with keys in Σ_S :

```

1  singleChar :: Encoding(E, S, C)
2  E: encoding dictionary, S: input stream with characters in  $\Sigma_S$ 
3  C: output stream
4  while S is non-empty
5      w ← E.search(S.pop())
6      append each bit of w to C

```

11.2.4 Variable-Length — Decoding

Comment 11.4

The code must be uniquely decodable.

Example 11.4

Morse code uses “end of character” pause to avoid ambiguity, this is equivalent to adding ‘\$’ at the end of each word.

Result 11.1

Encoding is prefix-free if ‘\$’ added, meaning no codeword is a prefix of another codeword.

Comment 11.5

However, adding ‘\$’, would mean coded alphabet is not binary, which is not desirable. So we will require encoding to be prefix free.

Theorem 11.1

From now on only consider prefix-free codes E , and we store codes in a trie with characters of Σ_S at the leaves.

Code 11.3

```
1  prefixFree :: decoding(T, C, S)
2  T : trie of a prefix-free code, C : input-stream with characters in  $\Sigma_C$ 
3  S : output-stream
4
5  while C is non-empty // iterate over all codewords
6      z ← T.root
7      while z is not a leaf // read next codeword
8          if C is empty or z has no child labelled C.top()
9              return "invalid encoding"
10         z ← child of z that is labelled with C.pop()
11         S.append(character stored at z)
```

Run-time: $O(|C|)$. It also detects if the encoding is invalid.

Discovery 11.4

It is easy to see how to encode from a table, which we simply map each character with its coded message, but table wastes space, and codewords can be quite long. So the better idea is:

store codewords via links to the trie leaves

Code 11.4: Modified Version

```
1  prefixFree::encoding(T, S, C)
2  T: prefix-free code trie, S: input-stream with characters in  $\Sigma_S$ 
3
4  E  $\leftarrow$  array of nodes in T indexed by  $\Sigma_S$ 
5  \textbf{for} all leaves l in T
6      E[character at l]  $\leftarrow$  l
7
8  \textbf{while} S is non-empty
9      w  $\leftarrow$  empty bitstring; v  $\leftarrow$  E[S.pop()]
10     \textbf{while} v is not the root
11         w.prepend(character from v to its parent)
12         v  $\leftarrow$  parent(v)
13     // now w is the encoding of S
14     append each bit w of to C
```

Run-time: $O(|T| + |C|)$ as first for loop is $O(|T|)$ and the while loop is $O(|C|)$. We have to visit all trie nodes, and insert leaves into *E*. We also assume *T* has no node with only one leaf, so it is essentially $O(|\Sigma_S| + |C|)$ because we can show

$$|\Sigma_S| \cdot 2 - 1 \geq \# \text{internal nodes} + \# \text{leaves} = |T|$$

11.3 Huffman Codes — Build the best trie

Question 11.3.

How to determine the best trie for a given source text *S*? i.e., how do we get the shortest $|C|$?

Solution: Infrequent characters should be far down in the trie.



11.3.1 Huffman Tree Construction

Algorithm 11.1

1. determine frequency of each character $c \in \Sigma$ in *S*, and for each $c \in \Sigma$, create trie of height 0 holding only *c*;
2. assign weight to each trie based on the frequency;
3. find and merge two tries with the minimum weight;
 - new interior node added
 - the new weight is the sum of merged tries weights
4. repeat Steps 4 until there is only 1 trie left.

Comment 11.6

Use min-heap for efficient implementation: step 3 is two `delete-min` one `insert`.

Discovery 11.5

This is a greedy algorithm as always pair up least frequent characters.

Code 11.5

```

1  \textit{Huffman::encoding}(S, C)
2  S: input-stream (length n) with characters in  $\Sigma_S$ , C: output-stream, initially empty

3  f  $\leftarrow$  array indexed by  $\Sigma_S$ , initialized to 0
4
5  while S is non-empty do increase f[S.pop()] by 1      get frequencies
6
7  Q  $\leftarrow$  min-oriented priority queue to store tries
8
9  for all c  $\in \Sigma_S$  with f[c] > 0
10     Q.insert(single-node trie for c, f[c])
11
12  while Q.size() > 1
13     (T1, f1)  $\leftarrow$  Q.deleteMin()
14     (T2, f2)  $\leftarrow$  Q.deleteMin()
15     Q.insert(trie with T1, T2 as subtrees, f1 + f2)
16
17  T  $\leftarrow$  Q.deleteMin() // trie for decoding
18  reset input-stream S // read all of S, need to read again for encoding
19
20  prefixFree::encoding(T, S, C) // perform actual encoding

```

Total time is $O(|\Sigma_S| \log |\Sigma_S| + |C|)$. First while loop is $O(n)$, for loop is $O(|\Sigma_S| \log |\Sigma_S|)$, second while loop is $O(|\Sigma_S| \log |\Sigma_S|)$, and actual encoding is $O(|\Sigma_S| + |C|)$.

Theorem 11.2

The constructed trie is optimal in the sense that the coded text C is shortest among all prefix-free character encodings with $\Sigma_C = \{0, 1\}$.

Proof. See course notes. □

Result 11.2

Decoding run-time: $O(|C|)$.

11.4 Lempel-Ziv-Welch

Comment 11.7

Huffman takes advantage of frequent or repeated single characters.

Discovery 11.6

Certain substrings are much more frequent than others.

Theorem 11.3

Ingredient 1 for Lempel-Ziv-Welch compression: encode characters and frequent substrings. Algorithm discovers and encodes frequent substring as we process text, so there is no need to know frequent substrings beforehand.

Definition 11.7: Single Character Encoding

Single character encoding: each source-text character receives one codeword.

Example 11.5

$$S = \underbrace{b}_{01} \underbrace{a}_1 \underbrace{n}_{11} \underbrace{a}_1 \underbrace{n}_{11} \underbrace{a}_1$$

Definition 11.8: Multi-Character Encoding

Multi-character encoding: multiple source-text characters can receive one codeword

Example 11.6

$$S = \underbrace{b}_{01} \underbrace{ana}_{11} \underbrace{na}_{101}$$

Algorithm 11.2

Lempel-Ziv-Welch is a multi-character encoding.

11.4.1 Encoding

Theorem 11.4

Ingredient 2 for LZW: adaptive dictionary, which means that dictionary constructed during encoding/decoding.

Comment 11.8

Dictionary is constructed during encoding/ decoding. Start with some initial fixed dictionary D_0 (usually ASCII). At iteration $i \geq 0$, D_i is used to determine the i^{th} output, and after i^{th} output (iteration i), we update D_i to D_{i+1}

$$D_{i+1} \leftarrow D_i.\text{insert}(\text{new character combination})$$

The decoder knows (i.e. be able to reconstruct from the coded text) how encoder changed the dictionary.

Example 11.7: LZW Encoding: Main Idea

At iteration i of encoding, our current $D_i = \{a : 65, b : 66, c : 67, ab : 128, bb : 129\}$.

$$S = \text{ a b bb aad} \quad C = 65 \ 66 \ 129$$

What we have done is finding longest substring that starts at current pointer and is in the dictionary. Then we perform

$$D_{i+1} = D_i.\text{insert}('bba', \text{nextAvailableCodenum} = 130)$$

Hence after iteration i ,

$$D_{i+1} = \{a : 65, b : 66, c : 67, ab : 128, bb : 129, bba : 130\}$$

Algorithm 11.3

We also need to store (string, codenumber) pairs in a trie, with string being the key.

Comment 11.9

Trie stores codenumbers at all nodes (external and internal) except the root, we do not store the string key explicitly, store only the codenumber.

Theorem 11.5

We also use fixed length (12 bits) per codenumber, so in total of $2^{12} = 4096$ codesnumbers available during encoding.

Code 11.6

```

1  LZW::encoding(S, C)
2  // S : input stream of characters, C : output stream
3
4  initialize dictionary D with ASCII in a trie
5  idx ← 128
```

```

6
7   while S is not empty do
8       v ← root of trie D
9       // following while loop is trie search
10      while S is non-empty and v has a child c labelled S.top()
11          v ← c
12          S.pop()
13      C.append(codenumber stored at v)
14      // new dictionary entry
15      if S is non-empty
16          create child of v labelled S.top() with code idx
17          idx++

```

Running time is $O(|S|)$, assuming can look up child labeled with c in $O(1)$ time.

11.4.2 Decoding

Code 11.7

```

1   LZW::decoding(C, S)
2   // C : input-stream of integers
3   // S : output-stream
4
5   D ← dictionary that maps {0,...,127} to ASCII
6   idx ← 128 // next available code
7
8   code ← C.pop()
9   s ← D.search(code)
10  S.append(s)
11
12  while there are more codes in C do
13      s_prev ← s
14      code ← C.pop()
15
16      if code < idx then
17          s ← D.search(code) // code in D, look up string s
18      elif code == idx then // code not in D yet, reconstruct string
19          s ← s_prev + s_prev[0]
20      else
21          Fail // invalid encoding
22
23      append each character of s to S
24      D.insert(idx, s_prev + s[0])
25      idx++

```

Running time is $O(|S|)$.

Comment 11.10

In practice, compression rate is around 45% on English text.

11.5 Combining Compression Schemes: bzip2

Theorem 11.6

This method combines multiple compression schemes and text transforms.

Example 11.8

Burrows-Wheeler
transform

$T_0 = \text{alfeatsalfalfa}$

if T_0 has repeated longer substrings, then T_1 has long runs of characters

Move-to-front
transform

$T_1 = \text{affs$eflllaata}$

if T_1 has long runs of characters, then T_2 has long runs of 0 and *skewed frequencies*

0-runs encoding

text $T_2 = 13053435006006$

if T_2 has long runs of zeroes, then T_3 is *shorter*; *skewed frequencies remain*

Huffman encoding

text T_3

compresses well since frequencies are skewed

text T_4

Definition 11.9: Text Transform

Text transform refers to changing input text into a different text.

Comment 11.11

Output is not shorter, but likely to compress better.

11.5.1 Move-to-Front transform

Algorithm 11.4

Source alphabet is Σ_S with size $|\Sigma_S| = m$, we first put alphabet in array L , initially in sorted order, but allow L to get unsorted in the process: **After each encoding, update L with Move-To-Front heuristic.**

Discovery 11.7

Dictionary D is changed dynamically just like LZW, but unlike LZW, no new items added to dictionary and codeword for one or more letters can change at each iteration.

11.5.2 Move-to-Front Transform: Properties

1. If a character in S repeats k times, then C has a run of $k - 1$ zeros;
2. C contains a lot of small numbers and a few big ones (skewed frequencies);
3. C has the same length as S , but better properties for encoding.

11.5.3 0-runs Encoding

Input consists of ‘characters’ in $\{0, \dots, 127\}$ with long runs of zeros. We replace k consecutive zeros by $(k)_2$ (takes approximately $\log k$ bits) bits using two new characters A, B .

Example 11.9

65, 0, 0, 0, 0, 67, 0, 0, 72 becomes 65, AB , 67, $B72$.

11.6 Burrows-Wheeler Transform

Definition 11.10: Burrows-Wheeler Transform

Burrows-Wheeler Transform is a transformation algorithm, it is not a compression algorithm. It transforms source text to coded text with same letters but in different order.

Theorem 11.7

It is required that the source text S ends with end-of-word character $\$, \$$ occurs nowhere else in S and is counted towards length of S .

Comment 11.12

Burrows-Wheeler Transform is based on cyclic shifts for a string.

Definition 11.11: Cyclic Shift

A **cyclic shift** of string X of length n is the concatenation of $X[i + 1, \dots, n - 1]$ and $X[0, \dots, i]$, for

$$0 \leq i < n.$$

11.6.1 BWT Algorithm and Example

Consider

$$S = \text{alfeatsalfalfa\$}$$

1. We first write all consecutive cyclic shifts, forming an array of shifts (see the middle column).
2. We then sort (lexographically) cyclic shifts (see the rightmost column), note that we have strict sorting order due to \$.

alfeatsalfalfa\$	\$alfeatsalfalfa a
lfeatsalfalfa\$a	a\$alfeatsalfalf f
featsalfalfa\$al	alfa\$alfeatsalf f
eatsalfalfa\$alf	alfalfa\$alfeats s
atsalfalfa\$alfe	alfeatsalfalfa\$ \$
tsalfalfa\$alfea	atsalfalfa\$alfe e
salfalfa\$alfeat	eatsalfalfa\$alf f
alfalfa\$alfeats	fa\$alfeatsalfal l
lfalfa\$alfeatsa	falfa\$alfeatsal l
falfa\$alfeatsal	featsalfalfa\$al l
alfa\$alfeatsalf	lfa\$alfeatsalfa a
lfa\$alfeatsalfa	lfalfa\$alfeatsa a
fa\$alfeatsalfal	lfeatsalfalfa\$a a
a\$alfeatsalfalf	salfalfa\$alfeat t
\$alfeatsalfalfa	tsalfalfa\$alfea a

Result 11.3

Last column can be decoded.

Hence in our example,

$$C = \text{affs$eflll1aaata}$$

11.6.2 BWT Fast Encoding: Efficient Sorting

Theorem 11.8

We can read BWT encoding from suffix array in $O(n)$ time:

$$C[i] = S[A^s[i] - 1]$$

Proof. $A^s[i]$ gives the starting index of the i^{th} lex smallest suffix, hence taking the one before it yields us the character at the end of the row. \square

11.6.3 BWT Decoding

Discovery 11.8

In unsorted shifts array, first column is S , so decoding is equivalent to determining the first letter of each row in unsorted shifts array.

Observation 1: First column has exactly the same characters as the last column, and they must be sorted. Therefore, after sorting the the characters and align them with C , the character matches with \$ is the start of row 0.

Observation 2: if row i starts with character c , then row $i + 1$ has to end with character c . Moreover, if row i is the k^{th} among the rows start with character c , then row $i + 1$ is the k^{th} among the rows end with character c .

Code 11.8

```
1  BWT::decoding(C, S)
2  // C: string of characters over alphabet  $\Sigma_C$ , one of which is $
3  // S: output stream
4  initialize array A // leftmost column
5  for all indices i of C
6      A[i]  $\leftarrow$  (C[i], i) // store character and index
7  stably sort A by character (the first aspect)
8  for all indices j of C // find $
9      if C[j] = $ break
10 do
11     S.append(character stored in A[j])
12     j  $\leftarrow$  index stored in A[j]
13 while appended character is not $
```

11.6.4 BWT Summary

Discovery 11.9

BWT needs all of the text (no streaming possible).

Result 11.4: BWT Summary

1. Encoding cost $O(n \log n)$ with special sorting algorithm – read encoding from the suffix array;
2. Decoding cost $O(n + |\Sigma_S|)$, which is faster than encoding.

Comment 11.13

Encoding and decoding both use $O(n)$ space.

11.6.5 bzip2 Summary

Result 11.5

1. encoding cost: $O(n[\log n + |\Sigma|])$ with a big multiplicative constant;
2. decoding cost: $O(n|\Sigma|)$ with a big multiplicative constant.

11.7 Compression Summary

Huffman	Lempel-Ziv-Welch	bzip2 (uses Burrows-Wheeler)
variable-length	fixed-width	multi-step
single-character	multi-character	multi-step
2-pass, must send dictionary	1-pass	not streamable
optimal 01-prefix-code	good on English text	better on English text
requires uneven frequencies	requires repeated substrings	requires repeated substrings
rarely used directly	frequently used	used but slow
part of pzip, JPEG, MP3	GIF, some variants of PDF, compress	bzip2 and variants

12 External Memory

Our current architecture is:

1. registers: super fast, very small
2. cache L1, L2: very fast, less small
3. main memory: fast, large
4. disk or cloud: slow, very large

Question 12.1.

How to adapt algorithms to take memory hierarchy into consideration?

Comment 12.1

desirable to minimize transfer between slow/ fast memory.

12.1 Stream based algorithms

We studied algorithms that handle input/ output with streams, recall:

1. take item off top of the input;
2. process item output;
3. put the result of processing at the tail of output.

Discovery 12.1

Data in external memory has to be placed in internal memory before it can be processed.

Solution: perform the same algorithm as before, but in “block-wise” manner:

- have one block for input, one block for output in internal memory
- transfer a block (size B) to internal memory, process it as before, store result in output block
- when output stream is of size B (full block), transfer it to external memory
- when current block in internal memory is fully processed, transfer next unprocessed block from external memory



Result 12.1

Methods below use stream input/output model, therefore need $\Theta\left(\frac{n}{B}\right)$ transfers, if output size is $O(n)$.

1. Pattern matching: Karp-Rabin, Knuth-Morris-Pratt, Boyer-Moore

Comment 12.2

assuming pattern P fits into internal memory

2. Text compression: Huffman, run-length encoding, Lempel-Ziv-Welch
3. Sorting: merge-sort can be implemented with $O\left(\frac{n}{B} \log n\right)$ block transfers
4. Bzip2 cannot be streamed as we described

Comment 12.3

can compress in ‘blocks’, though not as good as the whole text compression, but better than nothing.

12.2 External dictionaries

AVL tree based dictionary implementations have poor *memory locality*, meaning ‘nearby’ tree nodes are unlikely to be in the same block.

Discovery 12.2

In an AVL tree $\Theta(\log n)$ blocks are loaded in the worst case.

Proof. Each AVL tree node could be stored in a different disk block, so you might have to load up to $O(\log n)$ different blocks just to follow one path. \square

Theorem 12.1

Idea: allow trees that have many children per node.

Example 12.1

suppose store $n = 2^{50}$ items total, and $B = 2^{15}$ children per node, tree height is

$$\log_B n = \frac{\log_2 n}{\log_2 B} = \frac{50}{15}$$

which means that it is 15 times less block transfers.

12.3 2-4 Trees

Definition 12.1: 2-4 Tree

1. **Structural properties:** Every node is either
 - 1-node: one KVP and two subtrees (possibly empty), or
 - 2-node: two KVPs and three subtrees (possibly empty), or
 - 3-node: three KVPs and four subtrees (possibly empty)

Comment 12.4

Allowing 3 types of nodes simplifies insertion/ deletion.

- All empty subtrees are at the same level. (necessary for ensuring height is logarithmic in the number of KVP stored).

2. Order Property: keys at any node are between the keys in the subtrees.

Theorem 12.2

The height of a 2-4 tree is $O(\log n)$.

Proof. Proof is given in the section (a,b)-tree, since 2-4 tree is a special case of (a,b)-tree. \square

Comment 12.5

Empty subtrees are not part of height computation (Often times, we do not even show empty subtrees).

12.3.1 2-4 Tree, Search

Code 12.1

`search(k)` compares key k to k_1, k_2, k_3 , and either finds k among k_1, k_2, k_3 or figures out which subtree to recurse into. If key is not in tree, search returns parent of empty tree where search stops.

Code 12.2

```

1  24Tree::search( $k, v \leftarrow \text{root}, p \leftarrow \text{empty subtree}$ )
2   $k$ : key to search,  $v$ : node where we search;  $p$ : parent of  $v$ 
3
4  if  $v$  represents empty subtree
5      return "not found, would be in  $p$ "
6  let  $\langle T_0, k_1, \dots, k_d, T_d \rangle$  be key-subtrees list at  $v$ 
7  if  $k \geq k_1$ 
8       $i \leftarrow$  maximal index such that  $k_i \leq k$ 
9      if  $k_i = k$ 
10         return "at  $i$ th key in  $v$ "
11     else '24Tree::search'( $k, T_i, v$ )
12 else '24Tree::search'( $k, T_0, v$ )

```

12.3.2 2-4 Tree, Insert

Code 12.3

```

1  24Tree::insert( $k$ )
2       $v \leftarrow$  24Tree::search( $k$ ) // leaf where  $k$  should be
3      add  $k$  and an empty subtree in key-subtree-list of  $v$ 
4

```

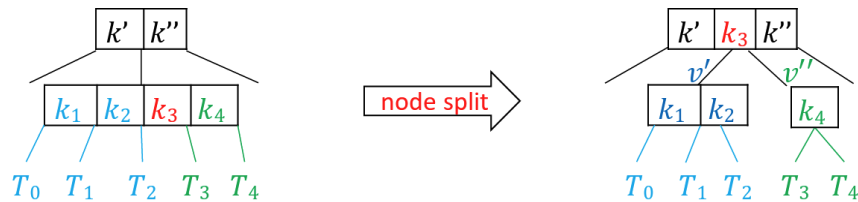
```

5   while  $v$  has 4 keys (overflow  $\rightarrow$  node split!)
6       let  $\langle T_0, k_1, \dots, k_4, T_4 \rangle$  be key-subtrees list at  $v$ 
7       if  $v$  has no parent
8           create an empty parent of  $v$ 
9        $p \leftarrow$  parent of  $v$ 
10       $v' \leftarrow$  new node with keys  $k_1, k_2$  and subtrees  $T_0, T_1, T_2$ 
11       $v'' \leftarrow$  new node with key  $k_4$  and subtrees  $T_3, T_4$ 
12      replace  $\langle v \rangle$  by  $\langle v', k_3, v'' \rangle$  in key-subtree-list of  $p$ 
13       $v \leftarrow p$  @// continue checking for overflow upwards@

```

Example 12.2

Here is an example of node splitting:



Definition 12.2: Immediate Sibling

A node can have an immediate left sibling, immediate right sibling, or both.

Comment 12.6

Any node except the root must have an immediate sibling.

Definition 12.3: In Order Successor

Inorder successor of key k is the smallest key in the subtree immediately to the right of k .

Discovery 12.3

Inorder successor is guaranteed to be at a leaf node.

Proof. Otherwise would have something smaller in the leftmost subtree. \square

12.3.3 2-4 Tree, Delete

Result 12.2

1. If key not at a leaf node, swap with inorder successor (guaranteed at leaf node)
2. Delete key and one empty subtree from the leaf node involved in swap

3. If underflow

- If there is an immediate sibling with more than one key, transfer
 - no further underflows caused, do not forget to transfer a subtree as well
 - convention: if two siblings have more than one key, transfer with the right sibling
- If all immediate siblings have only one key, merge
 - there must be at least one sibling, unless root, if root, delete
 - convention: if two immediate siblings with one key, merge with the right one
 - merge may cause underflow at the parent node, continue to the parent and fix it, if necessary

Code 12.4

```
1  24Tree::delete(k)
2      v ← 24Tree::search(k) // node containing k
3      if v is not a leaf
4          swap k with its inorder successor k'
5          swap v with leaf that contained k'
6      delete k and one empty subtree in key-subtree-list of v
7      while v has 0 keys // underflow
8          if v is the root, delete v and break
9          if v has immediate sibling u with 2 or more KVPs // transfer, then done!
10             transfer the key of u that is nearest to v to p
11             transfer the key of p between u and v to v
12             transfer the subtree of u that is nearest to v to v
13             break
14         else // merge and repeat
15             u ← immediate sibling of v
16             transfer the key of p between u and v to u
17             transfer the subtree of v to u
18             delete node v
19             v ← p
```

12.4 Red Black Tree

Discovery 12.4

Problem with 2-4 trees:

Question 12.2.

How should we store keys and subtrees?

Comment 12.7

array of length 7 wastes space; linked list overhead for list-nodes, also wastes space. Theoretical bound not affected, but matters in practice.

Solution: Design a class of binary search trees that mirrors 2-4 tree.



12.4.1 2-4 Tree to Red Black Tree

There are three types of nodes in 2-4 tree:

1-node, 2-node, and 3-node

Algorithm 12.1

d -node becomes a black node with $d - 1$ red children (assembled in a way so that they form a BST of height at most 1).

Definition 12.4: Overhead

Red/ black 'colour' is stored with just 1 extra bit per node.

Result 12.3

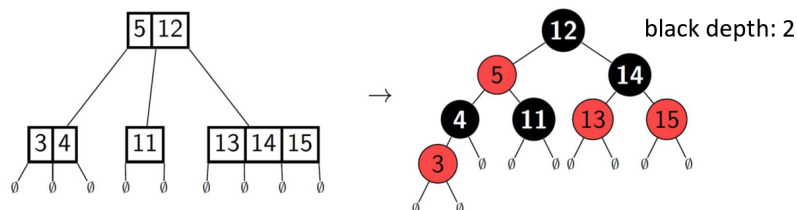
- Any red node has a black parent;
- Any empty subtree of T has the same black-depth.

Definition 12.5: Black-depth

Black-depth refers to the number of black nodes on the path from root to T .

Example 12.3

Here is an example converting 2-4 tree to red black tree:



12.4.2 Red Black Tree to 2-4 Tree

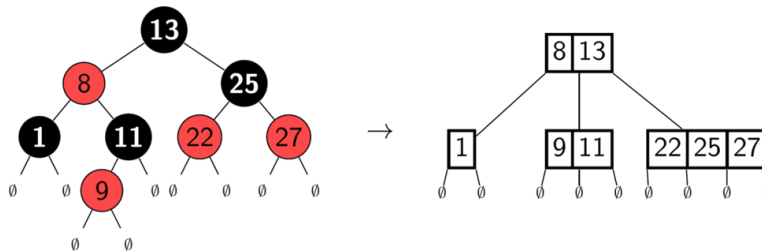
Theorem 12.3

Any red-black tree can be converted to a 2-4 tree.

Proof. black node with $0 \leq d \leq 2$ red children becomes a $(d + 1)$ -node, and this covers all nodes, as no red node has a red child. Empty subtrees on the same level due to the same blackdepth. \square

Example 12.4

Here is an example converting red black tree to a 2-4 tree:

**12.4.3 Red Black Trees Summary****Result 12.4**

- Red-black trees have height $O(\log n)$, each level of 2-4 tree creates at most 2 levels in red-black tree.
- Insert/delete can be done in $O(\log n)$ time
 - convert relevant part to 2-4 tree
 - do insert/delete as in 2-4 tree
 - convert relevant parts back to red-black tree
- Insert/delete can be done in $O(\log n)$ without conversion.

Comment 12.8

Red/ black trees are very popular balanced search trees (`std::map`).

12.5 (a,b)-tree

Definition 12.6: (a,b)-tree

(a, b)-tree satisfies

1. each node has at least a subtrees, unless it is the root (root must have at least 2 subtrees);
2. each node has at most b subtrees;
3. if node has d subtrees, then it stores $d - 1$ key-value pairs (KVPs);
4. all empty subtrees are at the same level;
5. keys in the node are between keys in the corresponding subtrees;
6. requirement: $b \geq 3$ and $2 \leq a \leq \lceil b/2 \rceil$.

Comment 12.9

Lower bound on a is needed to bound height, upper bound on a is needed during operations.

Question 12.3.

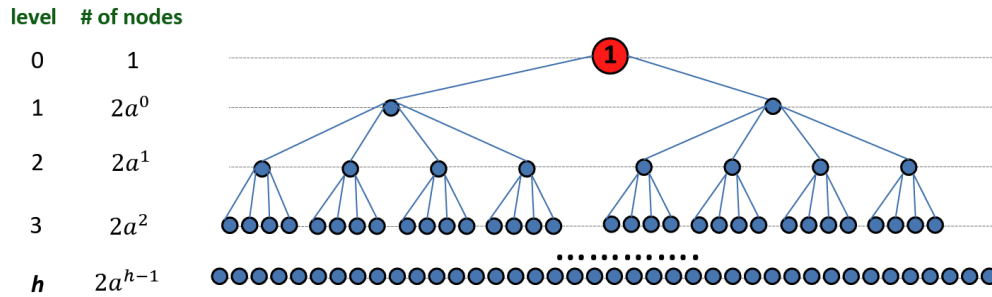
Why special condition for the root?

Solution: Otherwise we could not build the tree if for example, we force the root to have at least three children but we only have one KVP. 🎵

Theorem 12.4

Height of (a,b)-tree with n KVPs is $O(\log_a n)$.

Proof. Consider (a,b)-tree with the smallest number of KVP and of height h (red node (the root) has 1 KVP, blue nodes have $(a - 1)$ KVP).



Hence we have

$$\# \text{ of KVPs} = 1 + \sum_{i=0}^{h-1} 2a^i(a-1) = 1 + 2(a-1) \sum_{i=0}^{h-1} a^i = 2a^h - 1$$

Let n the number of KVP in any (a,b) -tree of height h , then

$$n \geq 2a^h - 1 \quad \rightarrow \quad \log_a \frac{n+1}{2} \geq h$$

which implies that the height of a tree with n KVPs is $O(\log_a n) = O(\log n / \log a)$. \square

Theorem 12.5

number of KVP = number of empty subtrees -1 in any (a,b) -tree.

Proof. Put one stone on each empty subtree and pass the stones up the tree. Each node keeps 1 stone per KVP, and passes the rest to its parent. Since for each node, $\#KVP = \# \text{ children} - 1$, each node will pass only 1 stone to its parent. This process stops at the root, and the root will pass 1 stone outside the tree. At the end, each KVP has 1 stone, and 1 stone is outside the tree. \square

12.6 B-tree

Comment 12.10

B-tree is a type of (a,b) -tree tailored to the external memory model. Each block in external memory stores one tree node.

Definition 12.7: B-tree

B-tree is (a,b) -tree such that $a = \lceil a/2 \rceil$.

Comment 12.11

For external memory, choose b s.t. the largest possible node (i.e. b subtrees) fits into a block.

13 Post-Midterm/ Final

Exercise 13.1

Hashing (with Linear Probing)

Suppose we have a hash table of size 7 and we use Linear Probing with the following hash function:

$$h(k, i) = ((k \bmod 6) + i) \bmod 6$$

Assuming each hash value is equally likely for a given key, what is the probability that slots 2, 3, and 4 are occupied after we insert 3 elements into the hash table?

Solution: Possible orders of hashing that result in slots 2, 3, and 4 being occupied are:

2	3	4	2	2	4	3	3	2
2	4	4	2	2	3	3	2	2
3	2	4	2	2	2	3	2	3
3	4	2	2	3	3	4	2	2
4	3	2	2	3	2			
4	2	3	2	4	2			

In total 16 possibilities, thus the probability is $\frac{16}{6^3}$.



Exercise 13.2

Cuckoo Hashing

T/ F: In cuckoo hashing, search and delete have guaranteed $O(1)$ run-time.

Solution: True.



Exercise 13.3

Hashing, Hashing Function Property

What is the desired property of a randomly chosen hash function $h : U \rightarrow \{0, 1, \dots, S - 1\}$?

1. It is a one-to-one function.
2. $S \geq U$
3. $\Pr(h(u) = i) = \frac{1}{S}$ for all keys u and slots i
4. For $u, v \in U$, $h(u) \neq h(v)$

Solution: 3 is the right answer.



Exercise 13.4

The height h of a 2-dimensional *kd*-tree is never larger than the height of a 2-dimensional quadtree where both trees store the same n 2-dimensional points in general position (i.e. distinct x and y coordinates) and $h > 0$.

Index

- Ω -notation, 11
- Θ -notation, 11
- ω -notation, 12
- o -notation, 12
- (a,b)-tree, 129
- 2-4 Tree, 123
- 2D Range Tree, 91

- ADT, 19
- Algorithm, 6
- Average Running Time, 17
- AVL Trees, 45

- B-tree, 130
- Balance, 45
- Best Case Running Time, 16
- BigO, 10
- Binary Tree, 20, 44
- Black-depth, 127
- Bucket, 70
- Burrows-Wheeler Transform, 118

- Chaining, 70
- Character Encoding, 109
- Check, 95
- Coded Text, 108
- Collision, 70
- Comparison Model, 39
- Compress Trie, 62
- Compression Ratio, 108
- Cyclic Shift, 118

- Decision Tree, 39
- Dictionary, 44
- Double Hashing, 75
- Double Right Rotation, 48

- Efficiency, 8
- Empty String, 94

- Fixed-Length Code, 109

- Guess, 95

- Hash Function, 69
- Hashing (idea), 69
- Heap, 20
- Height of Skip List, 52

- Immediate Sibling, 125
- In Order Successor, 125

- kd-Tree, 86

- Last Occurrence Array, 102
- Linear Probing, 73
- Load Factor (cuckoo Hashing), 77
- Load Factor (Probe Sequence), 72

- MTF Heuristic, 57
- Multi-Character Encoding, 114
- Multi-Dimensional Range Search, 81
- Multiway Trie, 64

- Overhead, 127

- Predecessor, 53
- Prefix, 94
- Priority Queue, 19
- Probe Sequence, 73
- Problem, 6
- Problem Instance, 6
- Program, 7
- Pruned Trie, 62
- Pseudocode, 7

- Queue, 19

- Random Access Machine (RAM) Idealized
 Computer Model, 8
- Randomized Algorithm, 27
- Range Search, 81
- Rank, 31

- Single Character Encoding, 114
- Size (of a word), 59
- Skip List, 51
- Solving, 6
- Sorting Permutation, 39
- Source Text, 108
- Spread Factor, 84
- Stack, 19

String Matching, 94
Substring, 94
Successor, 45
Suffix, 94
Suffix Tree, 104
Text Transform, 117

Tower, 52
Trie, 60

Variable-Length Code, 109

Words (Strings), 59
Worst Case Running Time, 16