

CS240E: Data Structures and Data Management (Enriched)

University of Waterloo

Instructor: Armin Jamshidpey

Winter 2023

Andrew Wang

Table of Contents

Asymptotic Analysis	6
Algorithm Efficiency	6
Experimental Studies	6
Random Access Machine (RAM) Model	7
Computational Model: Word RAM	7
Order Notation	8
Algebra of Order Notations	9
Techniques for Order Notation	9
Growth Rate	10
Relationships Between Order Notations	11
Algorithm Analysis	11
Complexity of Algorithms	12
Recursive Algorithm Analysis	12
Mergesort	12
Analysis of MergeSort	13
The Master Method	14
Useful Recurrence Relations	15
Other Useful Formulas	16
Priority Queues	17
Abstract Data Type (ADT)	17
Stack ADT	17
Queue ADT	17
Priority Queue ADT	17
Binary Heaps	18
Storing Heaps in Arrays	19
insert Operation	20
deleteMax Operation	20
Priority Queue Realization Using Heaps	21
Heapify	21
Heapsort	23
Mergeable Heaps	25
PQ Merge Operation	25
Meldable Heaps	25
Binomial Heaps	28
Sorting, Average-case, and Randomization	31
Average-case Analysis	31
Sorting Permutations	31
Example: Average-case Run-time of avgCaseDemo	32
Randomized Algorithms	33
Expected Running Time	33
Example: Expected Running Time of expectedDemo	34
QuickSelect	34
The Selection Problem	34
Subroutines	35
QuickSelect Algorithm	36
Randomized QuickSelect	37

Expected Run-time vs Average-case Run-time	39
QuickSort	40
QuickSort Analysis	40
QuickSort Improvements	41
Comparison-based Sorting Lower Bound	42
The Comparison Model	42
Decision Trees	42
Lower Bound for Sorting in the Comparison Model	43
Non-comparison-based Sorting	44
Bucket Sort	44
MSD Radix Sort	46
LSD Radix Sort	47
Dictionaries	47
Binary Search Tree (BST)	48
AVL Tree	49
AVL Tree Height	50
AVL Rebalance	52
AVL Insertion	54
AVL Deletion	55
AVL Tree Operations Runtime	56
Amortized Analysis	56
Scapegoat Trees	58
Scapegoat Tree Analysis	60
Skip Lists	62
Search in Skip Lists	63
Insert in Skip Lists	64
Delete in Skip Lists	66
Analysis of Skip Lists	67
Re-ordering Items	68
Static Ordering	68
Dynamic Ordering	68
More on Dictionaries	69
Expected height of BST	69
Treap	70
Treap Operations	70
Optimal Static BST	72
MTF-Heuristic for a BST	73
Splay tree	74
Dictionaries for Special Keys	77
Search	77
Lower Bound for Search	77
Binary Search	78
Interpolation Search	78
Trie	79
Trie Operations	79
Trie Variations	81
Compressed Trie Operations	82

Multiway Tries	84
Dictionaries Via Hashing	85
Direct Addressing	85
Hash Collisions	86
Chaining	87
Complexity of Chaining	88
Open Addressing	89
Linear Probing	90
Independent Hash Function	92
Double Hashing	92
Cuckoo Hashing	93
Summary of Open Addressing Strategies	95
Choosing a Good Hash Function	95
Carter-Wegman's Universal Hashing	96
Multi-Dimensional Data	96
Hashing vs Balanced Search Trees	97
Range Searches	97
Multi-Dimensional Data	98
Quadtrees	98
Quadtree Dictionary Operations	100
Quadtree Range Search	101
Quadtree Analysis	102
Quadtrees in Other Dimensions	102
Quadtree Summary	103
kd-Trees	103
Constructing a kd-Tree	103
kd-Tree Dictionary Operations	106
kd-Tree Range Search	106
kd-Tree in Higher Dimensions	107
Range Trees	107
Range Tree Dictionary Operations	109
BST Range Search	109
Range Tree Range Search	110
Range Trees in Higher Dimensions	113
Summary of Range Search Data Structures	113
String Matching	114
Brute-Force Algorithm	115
Improving Brute Force	115
Karp-Rabin Algorithm	116
Boyer-Moore	117
Reverse Order Searching	117
Bad Character Heuristic	117
Good Suffix Heuristic	119
Boyer-Moore Summary	119
Knuth-Morris-Pratt (KMP) Algorithm	120
String Matching with Finite Automata	120
KMP Algorithm	120

KMP Failure Array	121
KMP Runtime	122
Suffix Trees	122
Suffix Array	123
String Matching Summary	124

Asymptotic Analysis

In earlier CS courses the emphasis was on program *correctness*, however, in this course we will shift the focus to *efficiency* (typically in terms of processor time).

We will study methods of managing large collections of data (*adding, deleting, searching, and sorting*).

This course will mainly use pseudo-code and place emphasis on mathematical analysis.

- **Problem:** given a problem instance, carry out a computational task
- **Problem Instance:** input for the specified problem
- **Problem Solution:** (correct) output for the specified problem instance
- **Size of problem instance:** $Size(I)$ a positive integer which which measures size of the instance I
- **Algorithm:** a *step-by-step process* on arbitrary problem instance I (described in pseudo-code)
- **Program:** implementation of an algorithm using a programming language

Algorithm \mathcal{A} **solves** a problem Π if, for every instance I of Π , \mathcal{A} finds a valid solution in finite time.

For a problem Π there can be many possible algorithms.

For an algorithm \mathcal{A} solving Π there can be many possible programs (implementations).

In practice to solve a problem Π :

1. Design an algorithm \mathcal{A} that solves Π (**Algorithm Design**)
2. Assess *correctness* and *efficiency* of \mathcal{A} (**Algorithm Analysis**)
3. If acceptable (correct and efficient), implement \mathcal{A}

Algorithm Efficiency

- **Running Time:** *amount of time* for program to complete (primary concern for this course)
- **Auxiliary Space:** *amount of additional memory* the program requires

Both of these depend on the $Size(I)$ (size of given problem instance I)

How do we actually measure and compare algorithm/program efficiency for a given problem?

Experimental Studies

One option is to implement the algorithm then time the runtime of the program with inputs of varying size and composition. However there are many shortcomings to this approach:

- Implementation and testing can be complicated/costly
- Timings are affected by many factors:
 - *Hardware:* speed of processor/memory, special CPU instructions, CPU architecture, etc
 - *Software Environment:* OS, compiler, programming language, etc
 - *Human Factors:* quality of programmer

- Impossible to test all inputs, so need to find representative *sample inputs*?

With so many factors to consider it is not easy to compare two algorithms/programs using this method.

Random Access Machine (RAM) Model

To generalize we use *pseudo-code*, count *primitive operations* instead of time, and measure efficiency by *growth rate* of these operations relative to input size.

The Random Access Machine (RAM) model is an *idealized computer model*

- Set of memory cells, each storing one item of data
 - Implicit assumption: memory cells are big enough to hold items that we store
- Any access to memory location takes constant time
- Any primitive operation takes constant time
- Running time is proportional to number of memory accesses plus number of primitive operations

Note that these assumptions may not hold for a real computer.

Computational Model: Word RAM

- Memory locations contain interger words of b bits each
- Assume $b \geq \log(n)$ for input size n (always assume \log_2 for \log)
- Random Access Memory: can access any memory location at unit cost
- Basic operations on *single* words have unit costs

Example 1:

```

1 Sum(A[1..n])
2   s <- 0
3   for i = 1, ..., n
4     s <- s + A[i]
```

Performing $s + A[i]$ the length of the result is at most $n + 1$, so at the end s will be at most of size $2n$. Thus we can say that runtime will be linear since word size scales linearly.

Example 2:

```

1 Product(A[1..n])
2   s <- 0
3   for i = 1, ..., n
4     s <- s × A[i]
```

When performing $s \times A[i]$ the length of the result is at most $2n$, so at the end s will be at most of size n^2 . Thus we can say that runtime will be non-linear since word size scales quadratically.

In this course we will treat multiplication as a constant time operation, but note that it does not actually behave that way.

Order Notation

Note: absolute values signs are irrelevant for run-time or space analysis in this course, but can be useful in other applications of these definitions. (i.e. they can be ignored for this course)

- O -notation: $f(n) \in O(g(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that

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

- Ω -notation: $f(n) \in \Omega(g(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that

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

- Θ -notation: $f(n) \in \Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that

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

- o -notation: $f(n) \in o(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ such that

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

- ω -notation: $f(n) \in \omega(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 \geq 0$ such that

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

It directly follows that:

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

Examples: proofs from first principles

- $\frac{1}{2}n^2 - 5n \in \Omega(n^2)$. Begin by splitting up the $\frac{1}{2}n^2$ term:

$$\frac{1}{2}n^2 - 5n = \frac{1}{4}n^2 + \underbrace{\left(\frac{1}{4}n^2 - 5n\right)}_{\text{show } \geq 0} \rightarrow \frac{1}{4}n^2 - 5n \geq 0 \rightarrow n \geq 20$$

When $n \geq 20$ the second term is greater than 0 so we get $n_0 = 20$ and $c = \frac{1}{4}$

$$\frac{1}{2}n^2 - 5n \geq \frac{1}{4}n^2 \quad \text{for all } n \geq 20$$

- $\log_b(n) \in \Theta(\log n)$ for all $b > 1$ can be proved by using the log change of base rule

$$\log_b(n) = \frac{\log n}{\log b} \rightarrow \frac{\log n}{\log b} \leq \log_b(n) \leq \frac{\log n}{\log b} \rightarrow c_1 = c_2 = \frac{1}{\log b}$$

to be more formal we could show $\log_b n \in O(\log n)$ and $\log_b n \in \Omega(\log n)$

- $2000n^2 \in o(n^n)$. Begin by letting $c > 0$, find $n_0 \geq 0$ (which depends on c) such that if $n \geq n_0$

$$2000n^2 \leq cn^n \quad \rightarrow \quad 2000 \leq cn^{n-2}$$

Notice that when $n \geq 3$ then $n \leq n^{n-2}$. If we also take $n \geq \frac{2000}{c}$ then

$$\frac{2000}{c} \leq n \leq n^{n-2} \quad \rightarrow \quad 2000 \leq cn^{n-2}$$

To ensure that both $n \geq 3$ and $n \geq \frac{2000}{c}$ we have

$$n_0 = \max \left\{ 3, \frac{2000}{c} \right\}$$

Algebra of Order Notations

- **Identity rule:** $f(n) \in \Theta(f(n))$

- **Transitivity:**

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

$$f(n) \in \Theta(g(n)) \text{ and } g(n) \in \Theta(h(n)) \quad \implies \quad f(n) \in \Theta(h(n))$$

- **Maximum rules:** Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$, then:

$$f(n) + g(n) \in O(\max\{f(n), g(n)\})$$

$$f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$$

Proof: $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 \max\{f(n), g(n)\}$

Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

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

If the limit exists then

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

Note: this result is a *sufficient* (but not necessary) condition for the stated conclusions to hold.

This technique will often be computed using l'Hôpital's rule:

$$\lim_{x \rightarrow c} f(x) = \lim_{x \rightarrow c} g(x) = 0 \text{ or } \pm \infty \quad \text{and} \quad \lim_{x \rightarrow c} \frac{f'(x)}{g'(x)} \text{ exists} \quad \implies \quad \lim_{x \rightarrow c} \frac{f(x)}{g(x)} = \lim_{x \rightarrow c} \frac{f'(x)}{g'(x)}$$

To emphasize this is only valid if *both* denominator and numerator tend to 0 or $\pm\infty$

Example: prove $n(2 + \sin n\pi/2) \in \Theta(n)$. Note that $\lim_{n \rightarrow \infty} (2 + \sin n\pi/2)$ does not exist.

$$\begin{aligned} -1 \leq \sin(n\pi/2) \leq 1 &\implies 1 \leq 2 + \sin(n\pi/2) \leq 3 \\ &\implies n \leq n(2 + \sin(n\pi/2)) \leq 3n \end{aligned}$$

Example: (using l'Hôpital's rule) compare the growth rates of $\log n$ and n

$$\lim_{n \rightarrow \infty} \frac{\log n}{n} = \lim_{n \rightarrow \infty} \frac{1}{n} = 0 \quad \rightarrow \quad \log n \in o(n)$$

Now compare the growth rates of $\log^c n$ and n^d for arbitrary $c > 0$ and $d > 0$

$$\lim_{n \rightarrow \infty} \frac{\log^c n}{n^d} = \lim_{n \rightarrow \infty} \left(\frac{\log n}{n^{\frac{d}{c}}} \right)^c = \left(\lim_{n \rightarrow \infty} \frac{\log n}{n^{\frac{d}{c}}} \right)^c = \left(\lim_{n \rightarrow \infty} \frac{1}{\frac{d}{c} n^{\frac{d}{c}-1}} \right)^c = 0 \quad \rightarrow \quad \log^c n \in o(n^c)$$

Growth Rate

Typically $f(n)$ may be *complicated* while $g(n)$ is chosen to be simple

- If $f(n) \in \Theta(g(n))$ then growth rates of $f(n)$ and $g(n)$ are the *same*
- If $f(n) \in o(g(n))$ then $f(n)$ growth rate is *less than* $g(n)$ growth rate
- If $f(n) \in \omega(g(n))$ then $f(n)$ growth rate is *greater than* $g(n)$ growth rate

The common growth rates (in increasing order of growth rate):

- $\Theta(1)$ constant complexity
- $\Theta(\log n)$ logarithmic complexity
- $\Theta(n)$ linear complexity
- $\Theta(n \log n)$ linearithmic
- $\Theta(n \log^k n)$ quasi-linear (k is some constant)
- $\Theta(n^2)$ quadratic complexity
- $\Theta(n^3)$ cubic complexity
- $\Theta(2^n)$ exponential complexity

Example: how is running time $T(n)$ effected when the size of the problem instance doubles (i.e. $n \rightarrow 2n$)

- constant complexity:

$$T(n) = c \quad \rightarrow \quad T(2n) = c$$

- logarithmic complexity:

$$T(n) = c \log n \quad \rightarrow \quad T(2n) = T(n) + c$$

- linear complexity:

$$T(n) = cn \quad \rightarrow \quad T(2n) = 2T(n)$$

- linearithmic:

$$T(n) = cn \log n \quad \rightarrow \quad T(2n) = 2T(n) + 2cn$$

- quadratic complexity:

$$T(n) = cn^2 \quad \rightarrow \quad T(2n) = 4T(n)$$

- cubic complexity:

$$T(n) = cn^3 \quad \rightarrow \quad T(2n) = 8T(n)$$

- exponential complexity:

$$T(n) = c2^n \quad \rightarrow \quad T(2n) = (T(n))^2/c$$

Relationships Between Order Notations

$$f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$$

$$f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$$

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

$$f(n) \in o(g(n)) \implies f(n) \in O(g(n)) \text{ and } f(n) \notin \Omega(g(n))$$

$$f(n) \in \omega(g(n)) \implies f(n) \in \Omega(g(n)) \text{ and } f(n) \notin O(g(n))$$

Algorithm Analysis

Using asymptotic notation we can simplify run-time analysis by finding how the run-time of an algorithm depends on the input size n :

- Identify *primitive operations* that require $\Theta(1)$ time
- Complexity of a loop is expressed as the *sum* of the complexities of each iteration of the loop
- For nested loops we start by analyzing the innermost and proceeding outward (*nested summation*)

We have two general strategies:

- Strategy I: Use Θ -bounds *throughout the analysis* and obtain an Θ -bound for the algorithm
- Strategy II: Prove O -bound and a matching Ω -bound separately
 - This is useful because upper/lower bounds are usually easier to sum

Example:

```

1 Test(A, n)
2   max ← 0
3   for i ← 1 to n do
4     for j ← i to n do
5       sum ← 0
6       for k ← i to j do
7         sum ← A[k]
8   return max

```

Let $T(n)$ be the runtime of `Test`. Then $T(n) \in \Theta(S(n))$ where $S(n)$ is the number of the times we execute `sum ← A[k]`

$$S(n) \leq \sum_{i=1}^n \sum_{y=1}^n \sum_{k=1}^n 1 \in O(n^3)$$

$$\begin{aligned} S(n) &\geq \sum_{i=1}^{\frac{n}{3}} \sum_{j=i}^n \sum_{k=i}^j 1 \geq \sum_{i=1}^{\frac{n}{3}} \sum_{j=\frac{2n}{3}}^n \sum_{k=i}^j 1 \geq \sum_{i=1}^{\frac{n}{3}} \sum_{j=\frac{2n}{3}}^n \sum_{k=\frac{n}{3}}^{\frac{2n}{3}} 1 \\ &\geq \left(\frac{n}{3}\right)^3 \implies S(n) \in \Omega(n^3) \end{aligned}$$

Complexity of Algorithms

An algorithm can also have different running times on two problem instances of the same size:

```

1 // A: array of size n
2 Test(A, n)
3   for i ← 1 to n - 1 do
4     j ← i
5     while j > 0 and A[j] < A[j - 1] do
6       swap A[j] and A[j - 1]
7       j ← j - 1

```

Let $T_{\mathcal{A}}(I)$ denote running time of the algorithm \mathcal{A} on instance I :

- Worst-case complexity of an algorithm: function $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$ mapping n to *longest* running time of \mathcal{A} for all input instances of size n :

$$T_{\mathcal{A}}(n) = \max\{T_{\mathcal{A}} : \text{Size}(I) = n\}$$

- Average-case complexity of an algorithm: function $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$ mapping n to *average* running time of \mathcal{A} over all instances of size n :

$$T_{\mathcal{A}}^{\text{avg}}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I : \text{Size}(I)=n\}} T_{\mathcal{A}}(I)$$

Note: it is important to avoid making *comparisons* between algorithms using O -notation (compare using Θ -notation instead)

- Say we have algorithm \mathcal{A}_1 with worst-case run-time $O(n^3)$ and \mathcal{A}_2 with worst-case run-time $O(n^2)$
- We cannot say \mathcal{A}_2 is more efficient than \mathcal{A}_1 because:
 1. Worst-case run-time could be extremely rare
 2. O -notation is an *upper bound*. \mathcal{A}_1 could have a worst-case run-time of $O(n)$.

Recursive Algorithm Analysis

Mergesort

Input: Array A of n integers

1. Split A into two subarrays:
 - A_L consists of first $\lceil \frac{n}{2} \rceil$ elements in A

- A_R consists of last $\lfloor \frac{n}{2} \rfloor$ elements of A

2. Recursively run MergeSort on A_L and A_R

3. After A_L and A_R are sorted, merge into into a single sorted array using Merge

```

1 // A: array of size n, 0 ≤ ℓ ≤ r ≤ n - 1
2 MergeSort(A, n, ℓ ← 0, r ← n - 1, S ← NIL)
3   if S is NIL initialize it as array S[0..n - 1]
4   if (r ≤ ℓ) then
5     return
6   else
7     m = (r + ℓ) / 2
8     MergeSort(A, n, ℓ, m, S)
9     MergeSort(A, n, m + 1, r, S)
10    Merge(A, ℓ, m, r, S)

```

This uses two tricks to reduce run-time and auxiliary space:

- Recursion uses parameters that indicate range of array that needs to be sorted
- Array used for copying is passed along as a parameter

```

1 // A[ℓ..r] is an array, A[ℓ..m] is sorted, A[m + 1..r] is sorted
2 // S[0..n - 1] is an array
3 Merge(A, ℓ, m, r, S)
4   copy A[ℓ..r] into S[ℓ..r]
5   int iL ← ℓ; int iR ← m + 1;
6   for (k ← ℓ; k ≤ r; k++) do
7     if (iL > m) A[k] ← S[iR++]
8     else if (iR > r) A[k] ← S[iL++]
9     else if (S[iL] ≤ S[iR]) A[k] ← S[iL++]
10    else A[k] ← S[iR++]

```

Merge takes $\Theta(r - \ell + 1) \rightarrow \Theta(n)$ time to merge n elements

Analysis of MergeSort

Let $T(n)$ denote the time to run MergeSort on an array of length n

- Step 1 takes time $\Theta(n)$
- Step 2 takes time $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$
- Step 3 takes time $\Theta(n)$

The recurrence relation for $T(n)$ is:

$$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

We then convert this to an *exact recurrence* by replacing Θ 's with constant factor c :

$$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

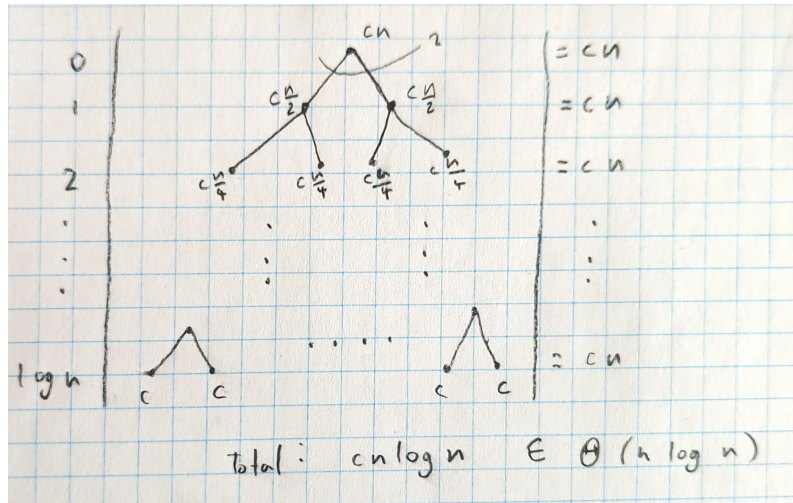
This can be turned to a *sloppy recurrence* if we remove the floors and ceilings:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

By default in this course we can always convert to a sloppy recurrence (except when explicitly told not to)

- When n is a power of 2 the exact and sloppy recurrences are identical
- While it is possible to show that $T(n) \in \Theta(n \log n)$ by analyzing the exact recurrence it is easier to use the sloppy recurrence

Example: analysis of MergeSort using the sloppy recurrence



The Master Method

The *Master Theorem* provides a formula to find the solution of many common recurrence relations found when analyzing divide-and-conquer algorithms. We will only look at a simplified version.

Theorem (Master Theorem): suppose that $a \geq 1$ and $b > 1$ then for the recurrence

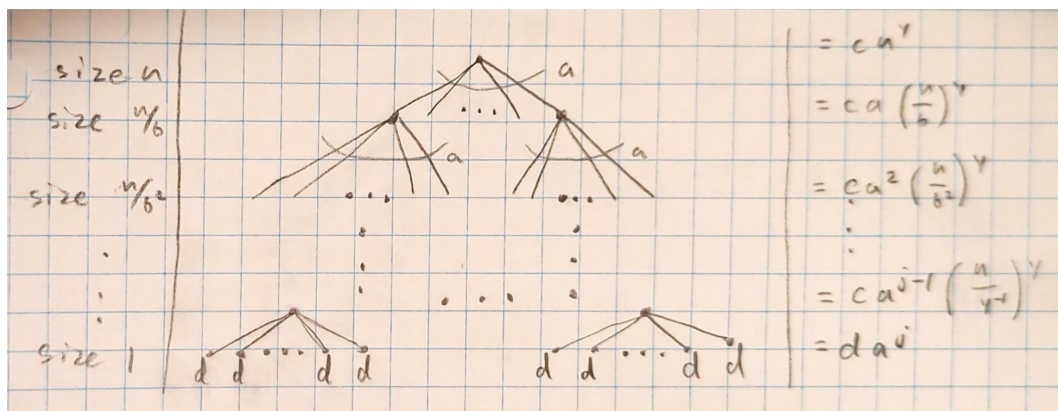
$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y)$$

in sloppy or exact form. Denote $x = \log_b a$ then

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^y) & \text{if } y > x \end{cases}$$

Since we must show recursion steps we won't actually be using this much in this course.

Proof: (ISSUES IN THIS PROOF)



Then letting $r = \frac{a}{b^y}$ we have a total

$$\begin{aligned} da^j + cn^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i &= da^{\log_b n} + cn^y \sum_{i=0}^{j-1} r^i \\ &= dn^{\log_b a} + cn^y \sum_{i=0}^{j-1} r^i \\ &= dn^x + cn^y \sum_{i=0}^{j-1} r^i \end{aligned}$$

Then we have 3 cases:

- $r = 1$

$$b^y = a \quad \rightarrow \quad \log_b b^y = \log_b a = x \quad \rightarrow \quad y = x$$

$$\sum_{i=0}^{j-1} r^i = j = \log_b h \quad \implies \quad \text{Total: } dn^x + cn^y \log_b n \in \Theta(n^x \log n)$$

- $r < 1$

$$b^y > a \quad \rightarrow \quad y > x$$

$$\sum_{i=0}^{j-1} r^i \in \Theta(1) \quad \implies \quad \text{Total: } dn^x + cn^y \in \Theta(n^y)$$

- $r > 1$

$$b^y < a \quad \rightarrow \quad y < x$$

$$\begin{aligned} \sum_{i=0}^{j-1} r^i \in \Theta(r^j) \quad \rightarrow \quad r^j &= r^{\log_b n} = n^{\log_b r} = n^{\log_b b^{x-y}} = n^{x-y} \\ \implies \quad \text{Total: } dn^x + cn^y(n^{x-y}) &= dn^x + cn^x \in \Theta(n^x) \end{aligned}$$

Useful Recurrence Relations

Once you know the result proving using induction is easy. Note that $0 < c < 1$.

Recursion	Resolves to	Example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify
$T(n) = T(cn) + \Theta(n)$	$T(n) \in \Theta(n)$	Selection
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range Search
$T(n) = T(\sqrt{n}) + \Theta(\sqrt{n})$	$T(n) \in \Theta(\sqrt{n})$	Interpol. Search
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpol. Search

Other Useful Formulas

- Arithmetic Sequence:

$$\sum_{i=0}^{n-1} i \rightarrow \sum_{i=0}^{n-1} (a + di) = an + \frac{dn(n-1)}{2} \in \Theta(n^2) \quad \text{if } d \neq 0$$

- Geometric Sequence:

$$\sum_{i=0}^{n-1} 2^i \rightarrow \sum_{i=0}^{n-1} ar^i = \begin{cases} a \frac{r^n - 1}{r - 1} \in \Theta(r^{n-1}) & \text{if } r > 1 \\ an \in \Theta(n) & \text{if } r = 1 \\ a \frac{1 - r^n}{1 - r} \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$$

- Harmonic Sequence:

$$\sum_{i=1}^n \frac{1}{i} \rightarrow \sum_{i=1}^n \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$$

- Other Sequences:

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

$$\sum_{i=1}^n i^k \rightarrow \sum_{i=1}^n i^k \in \Theta(n^{k+1}) \quad \text{for } k \geq 0$$

Logarithms:

- In this course $\log a$ always means $\log_2 a$
- Basic rules:

$$c = \log_b a \iff b^c = a \quad \log ac = \log a + \log c \quad \log a^c = c \log a$$

- Change of base:

$$\log_b a = \frac{\log_c a}{\log_c b} = \frac{1}{\log_a b} \quad a^{\log_b c} = c^{\log_b a}$$

- Concavity:

$$\alpha \log x + (1 - \alpha) \log y \leq \log(\alpha x + (1 - \alpha)y) \quad \text{for } 0 \leq \alpha \leq 1$$

Factorial:

$$n! := n(n-1)(n-2) \cdots 2 \cdot 1 = \# \text{ of ways to permute } n \text{ elements}$$

$$\log(n!) = \log n + \log(n-1) + \cdots + \log 2 + \log 1 \in \Theta(n \log n)$$

Linearity of expectation:

$$E[aX] = aE[X] \quad E[X + Y] = E[X] + E[Y]$$

Priority Queues

Abstract Data Type (ADT)

Definition: an *abstract data type* (ADT) is a description of *information* and a collection of *operations* on that information (information is *only* accessed through the given operations)

There can be various *realizations* of a ADT, which specify:

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

Note: an integer is an ADT

Stack ADT

Definition: a *stack* is an ADT consisting of a collection of items with operations:

- *push*: insert an item
- *pop*: remove (and typically returning) most recently inserted item

Items are removed in LIFO (*last-in first-out*) order (items enter at the top and are removed from the top)

Extra operations: *size*, *isEmpty*, *top*

The Stack ADT can be realized using an array or a linked list.

Queue ADT

Definition: a *queue* is an ADT consisting of a collection of items with operations:

- *enqueue*: insert an item
- *dequeue*: remove (and typically returning) the least recently inserted item

Items are removed in FIFO (*first-in first-out*) order (items enter at the rear and are removed from front)

Extra operations: *size*, *isEmpty*, and *front*

The Queue ADT can be realized using a (circular) array or a linked list.

Priority Queue ADT

Definition: a *priority queue* consists of a collection of items (each having a *priority*) with operations:

- *insert*: insert an item tagged with a priority
- *deleteMax*: remove and return the item of *highest* priority (also called *extractMax* or *getmax*)

The priority is also call *key*. The above definition is for a *max-oriented* priority queue, a *min-oriented* priority queue will use a *deleteMin* operation instead.

In both stacks and queues there is a implicit priority for each item that enters (stack gives highest priority to last element, queue gives lowest priority to last element).

We can use a priority queue to sort a list:

```

1 PQ-Sort(A[0..n - 1])
2   initialize PQ to an empty priority queue
3   for i ← 0 to n - 1 do
4     PQ.insert(A[i])
5   for i ← n - 1 down to 0 do
6     A[i] ← PQ.deleteMax()

```

Run-time of this depends on how we implement the priority queue $O(\text{init} + n \cdot \text{insert} + n \cdot \text{deleteMax})$.

- Realization 1: unsorted array

- *insert*: $O(1)$

- *deleteMax*: $O(n)$

PQ-sort with this realization yields *selection sort*

- Realization 2: sorted array

- *insert* $O(n)$

- *deleteMax*: $O(1)$

PQ-sort with this realization yields *insertion sort*

Note: we assume *dynamic arrays*, i.e. expand by doubling as needed (amortized to $O(1)$ extra time)

Using these naive implementations of priority queue both require $O(n^2)$ to sort an array.

Realization 3 using heaps will allow us to do this much faster (see next section).

Binary Heaps

Definition: A *binary tree* is either:

- empty
- a node with two binary trees (left subtree and right subtree)

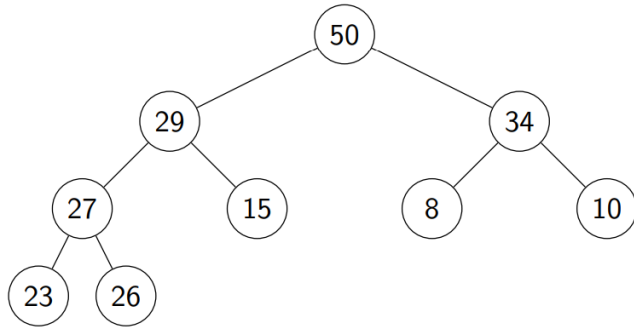
Any binary tree with n nodes has height at least $\log(n + 1) - 1 \in \Omega(\log n)$ (height of empty tree is -1)

Definition: a (*binary*) *heap* is a binary tree that obeys the following two properties:

- *Structural Property:* all levels of a heap are completely filled, except (possibly) for the last level which is filled from left to right
- *Heap-order Property:* for any node i , the key of the parent of i is larger than or equal to key of i

full name for this is a *max-oriented binary heap*

Example: here is an example of a heap, notice how it obeys the structural and heap-order properties



Lemma: The height of a heap with n nodes is $\Theta(\log n)$

Proof: for a heap of height h we have

- at least $1 + 2 + 4 + \dots + 2^{h-1} + 1 = 2^h$
- at most $1 + 2 + 4 + \dots + 2^{h-1} + 2^h = 2^{h+1} - 1$

Let n be the number of nodes, then

$$\begin{aligned}
 2^h \leq n \leq 2^{h+1} - 1 &\rightarrow 2^h \leq n \leq 2^{h+1} \\
 &\rightarrow h \leq \log n \leq h + 1
 \end{aligned}$$

Using this we get

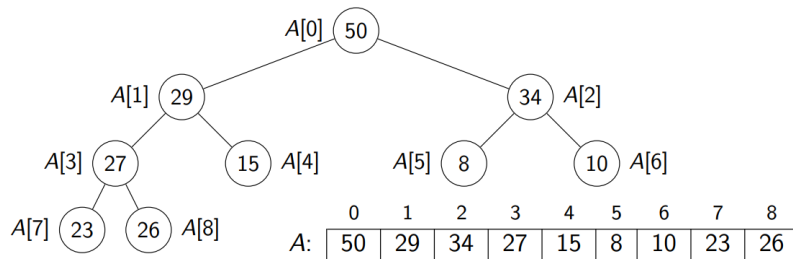
$$\log n - 1 \leq h \text{ and } h \leq \log n \rightarrow \log n - 1 \leq h \leq \log n$$

Storing Heaps in Arrays

Heaps should *not* be stored in a binary tree!

Let H be a heap of n items and let A be an array of size n . Store the root in $A[0]$ and continue with elements *level-by-level* from top to bottom, in each level left-to-right.

Example: the earlier example of a heap stored in an array



To navigate this array representation of a heap (node and index are used interchangeably):

- *root* node: index 0
- *last* node: index $n - 1$
- *left child* of node i : node $2i + 1$

- *right child* of node i : node $2i + 2$
- *parent* of node i : node $\lfloor \frac{i-1}{2} \rfloor$
- a node exists if its index falls in the range $\{0, \dots, n - 1\}$

We usually hide this implementation details using helper functions

- `root()`, `last()`, `left(i)`, `right(i)`, `parent(i)`, etc.

Some of these helper-function need to know n but we omit it for simplicity.

insert Operation

Place the new key at the first free leaf then call `fix-up` to fix heap-order property.

```

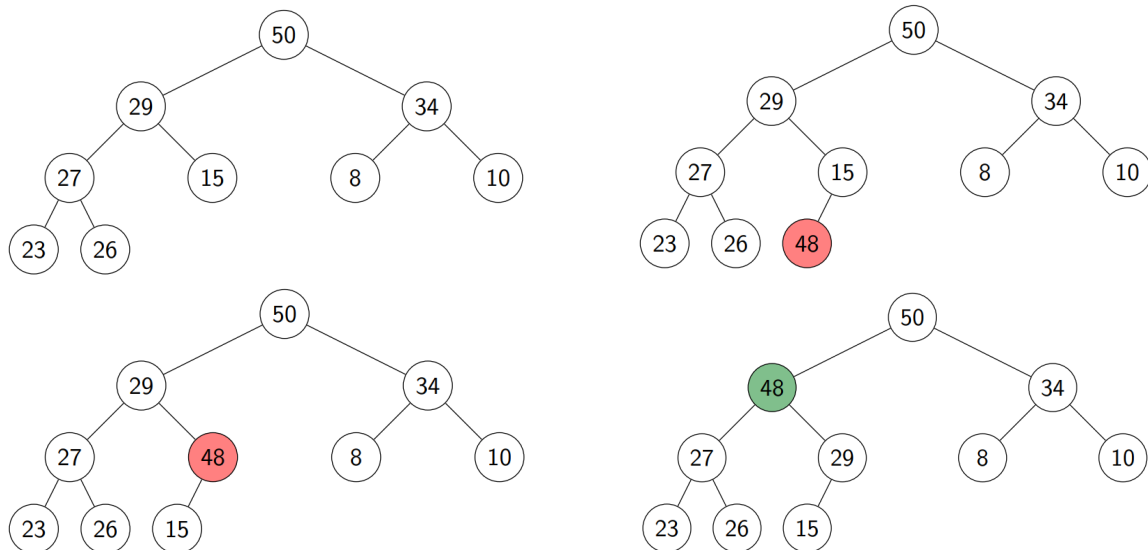
1 // i: an index corresponding to a node of the heap
2 fix-up(A, i)
3   while parent(i) exists and A[parent(i)].key < A[i].key do
4     swap A[i] and A[parent(i)]
5     i ← parent(i)

```

The new item *bubbles up* until it reaches its correct place in the heap.

Time: $O(\text{height of heap}) = O(\log n)$

Example:



deleteMax Operation

The root contains the maximum item of a heap:

1. Swap root node and last leaf
2. Remove last leaf and perform `fix-down` on the root node to fix heap-order property

```

1 // A: an array that stores a heap of size n
2 // i: an index corresponding to a node of the heap
3 fix-down(A, i, n ← A.size)
4   while i is not a leaf do
5     j ← left child of i // Find the child with the larger key

```



```

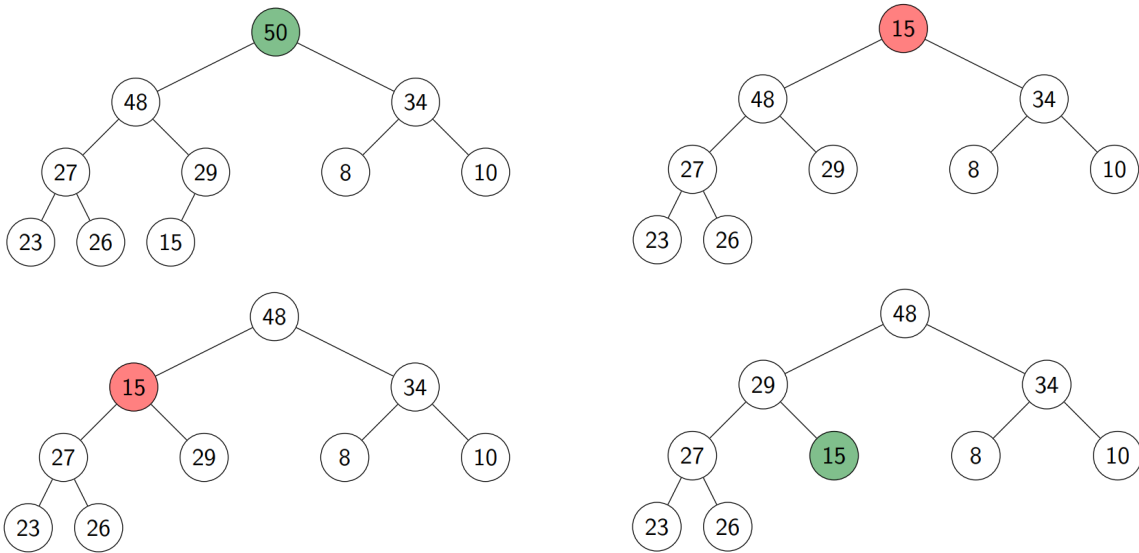
6   if (i has right child and A[right child of i].key > A[j].key)
7       j ← right child of i
8   if A[i].key ≥ A[j].key break
9   swap A[j] and A[i]
10  i ← j

```

The root node *bubbles down* until it reaches a correct place.

Time: $O(\text{height of heap}) = O(\log n)$

Example:



Priority Queue Realization Using Heaps

Store item in array A and globally keep track of *size* (size denotes the location of the last element):

```

1 insert(x)
2   increase size
3   l ← last()
4   A[l] ← x
5   fix-up(A, l)

```

```

1 deleteMax()
2   l ← last()
3   swap A[root()] and A[l]
4   decrease size
5   fix-down(A, root(), size)
6   return A[l]

```

Both *insert* and *deleteMax* take $O(\log n)$ time

Heapify

Convert a given array $A[0, \dots, n - 1]$ with n items into a heap:

- Solution 1: start with an empty heap and insert items one at a time:

```

1 // A: an array
2 simpleHeapBuilding(A)
3   initialize H as an empty heap

```

```

4   for i ← 0 to A.size() - 1 do
5   H.insert(A[i])

```

This corresponds to doing *fix-ups* and has a worst-case of $\Theta(n \log n)$

- Solution 2: use *fix-downs* instead:

```

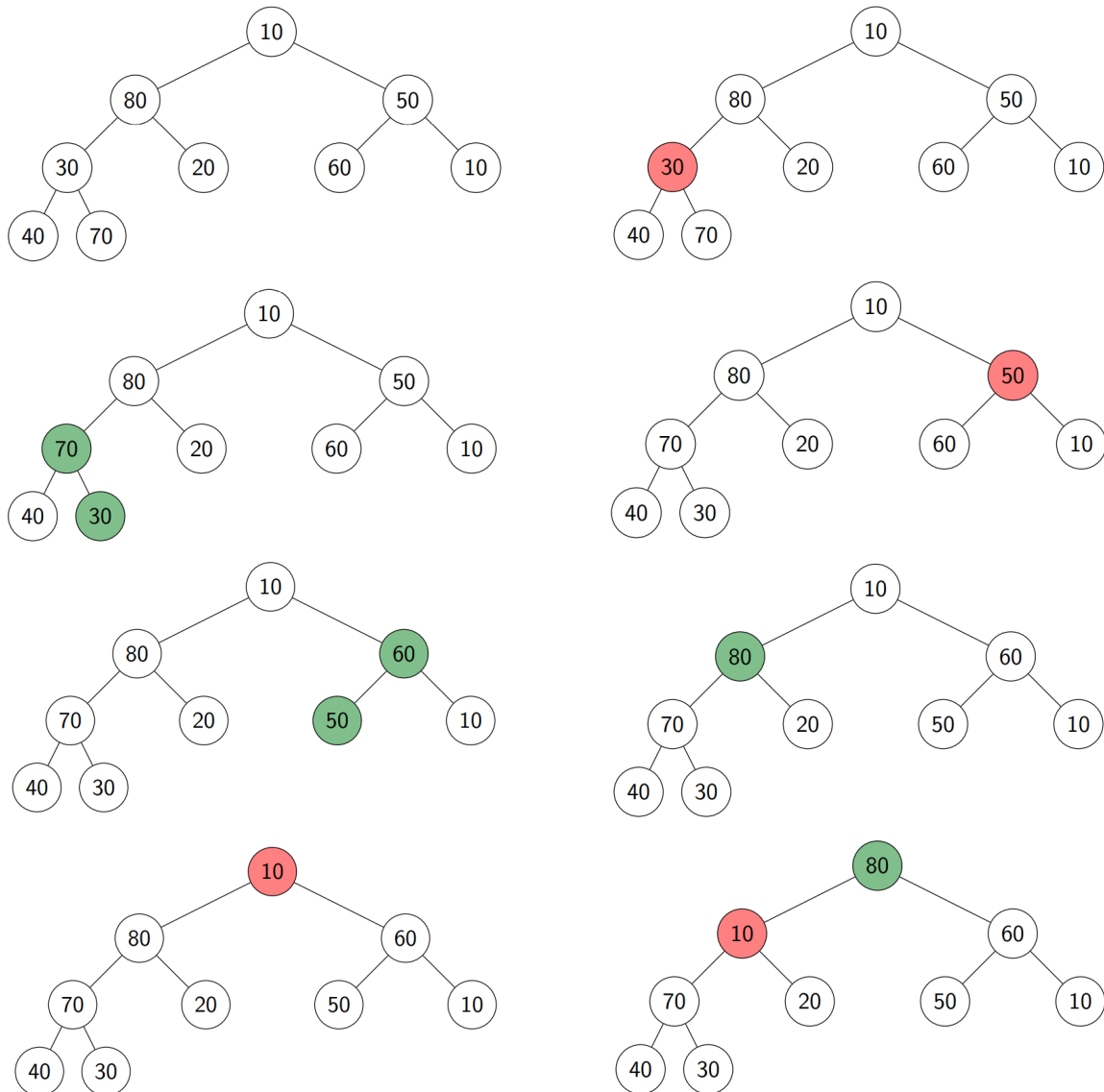
1 // A: an array
2 heapify(A)
3   n ← A.size()
4   for i ← parent(last()) downto root() do
5     fix-down(A, i, n)

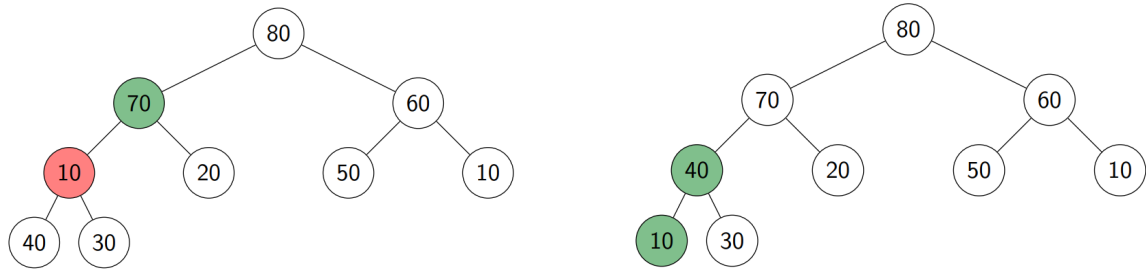
```

This has a worst-case complexity of $\Theta(n)$ which means we can build a heap in linear time.

This is all done *in-place* it does not require auxiliary space.

Example: notice that the fix-downs start at the bottom so later fix-downs will only require up to 1 swap





Heapsort

Notice that any priority queue can be sorted in time

$$O(\text{initialization} + n \cdot \text{insert} + n \cdot \text{deleteMax})$$

The naive implementation of sorting with heaps is:

```

1 PQsortWithHeaps(A)
2   initialize H to an empty heap
3   for i ← 0 to n - 1 do
4     H.insert(A[i])
5   for i ← n - 1 down to 0 do
6     A[i] ← H.deleteMax()

```

Using the *heapify* operation and a modification to *deleteMax* we arrive at *Heapsort*:

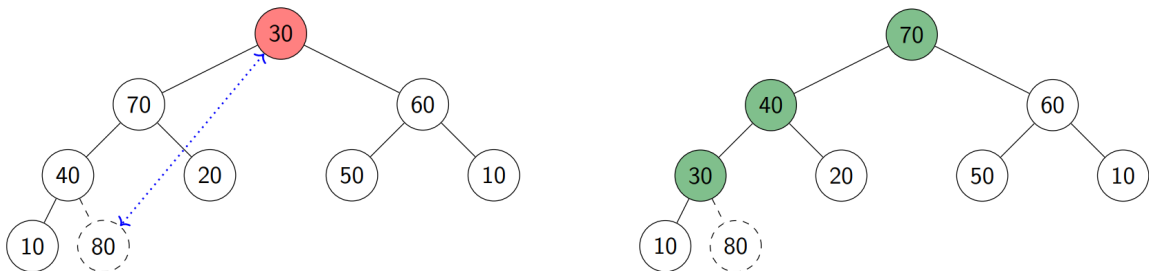
```

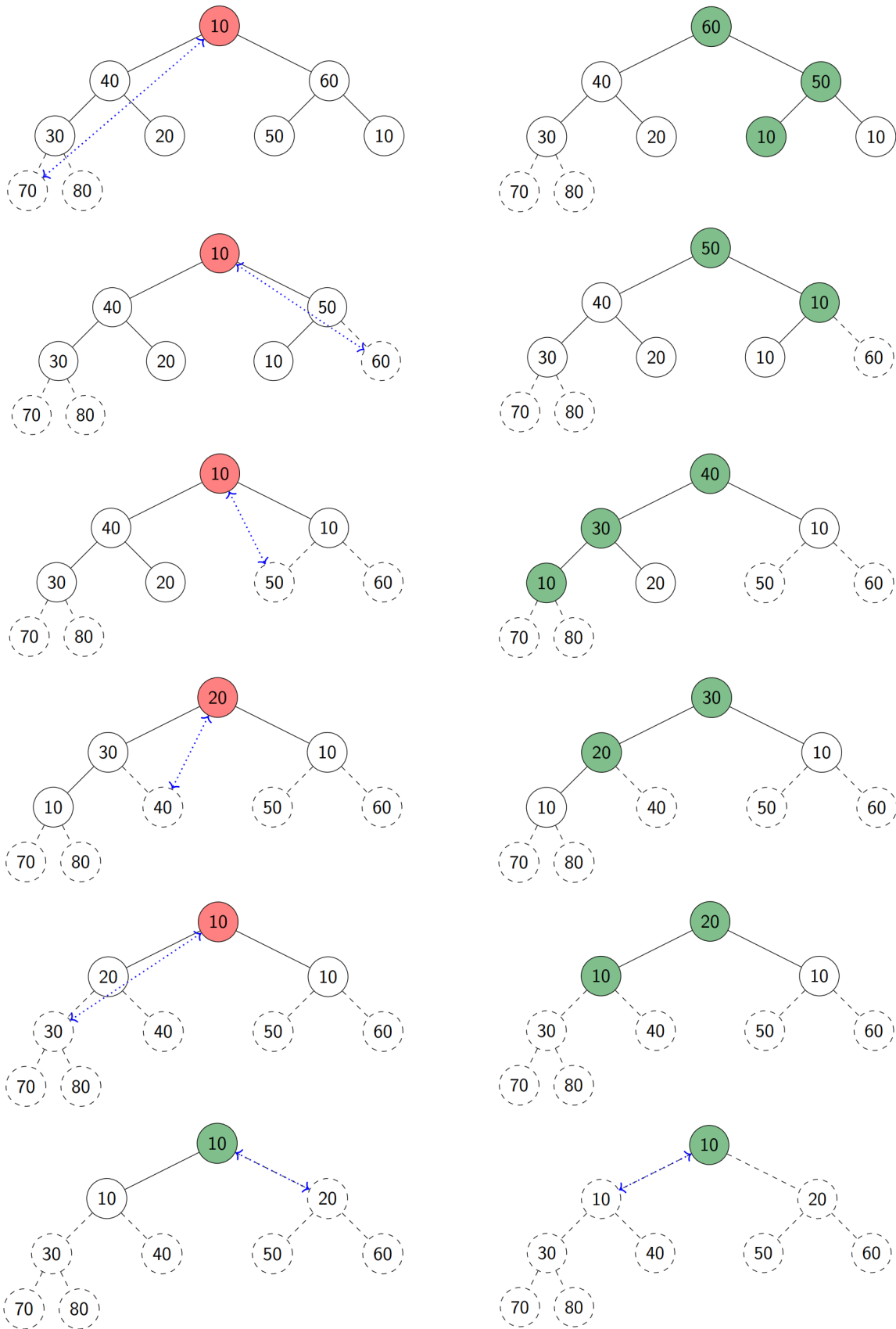
1 HeapSort(A, n)
2   // heapify
3   n ← A.size()
4   for i ← parent(last()) downto 0 do
5     fix-down(A, i, n)
6
7   // repeatedly find maximum
8   while n > 1
9     // 'delete' maximum by moving to end and decreasing n
10    swap items at A[root()] and A[last()]
11    decrease n
12    fix-down(A, root(), n)
13

```

Both methods have runtime of $O(n \log n)$ however *Heapsort* can be done with $O(1)$ auxiliary space.

Example: continuing from the *heapify* example





The array underlying the heap is now in sorted order.

Mergeable Heaps

PQ Merge Operation

$merge(P_1, P_2)$

- Input: two priority queues P_1, P_2 of size n_1, n_2
- Output: one priority queue P that contains all their items

This will take $\Omega(\min\{n_1, n_2\})$ if PQ is stored as array (since we need to copy at least one of them)

This can be done faster if PQ is stored as a tree, here are three approaches:

- Merge binary heaps (stored in a tree instead of array)
 - $O(\log^3 n)$ worst-case time (no details)
- Merge *meldable heaps* that have heap-property but not structural property
 - $O(\log n)$ expected run-time
- Merge *binomial heaps* that have heap-property but different structural property
 - $O(\log n)$ worst-case run-time

Notice that both *insert* and *deleteMax* can be done by *reduction to merge*:

- $P.insert(k, v)$
 - Create a 1-node heap P' that stores (k, v) then merge P' with P
- $P.deleteMax()$
 - Stash item at root and let P_ℓ, P_r be the left and right children of root
 - Update $P \leftarrow merge(P_\ell, P_r)$ then return the stashed item

Both operations have a run-time of $O(merge)$

Meldable Heaps

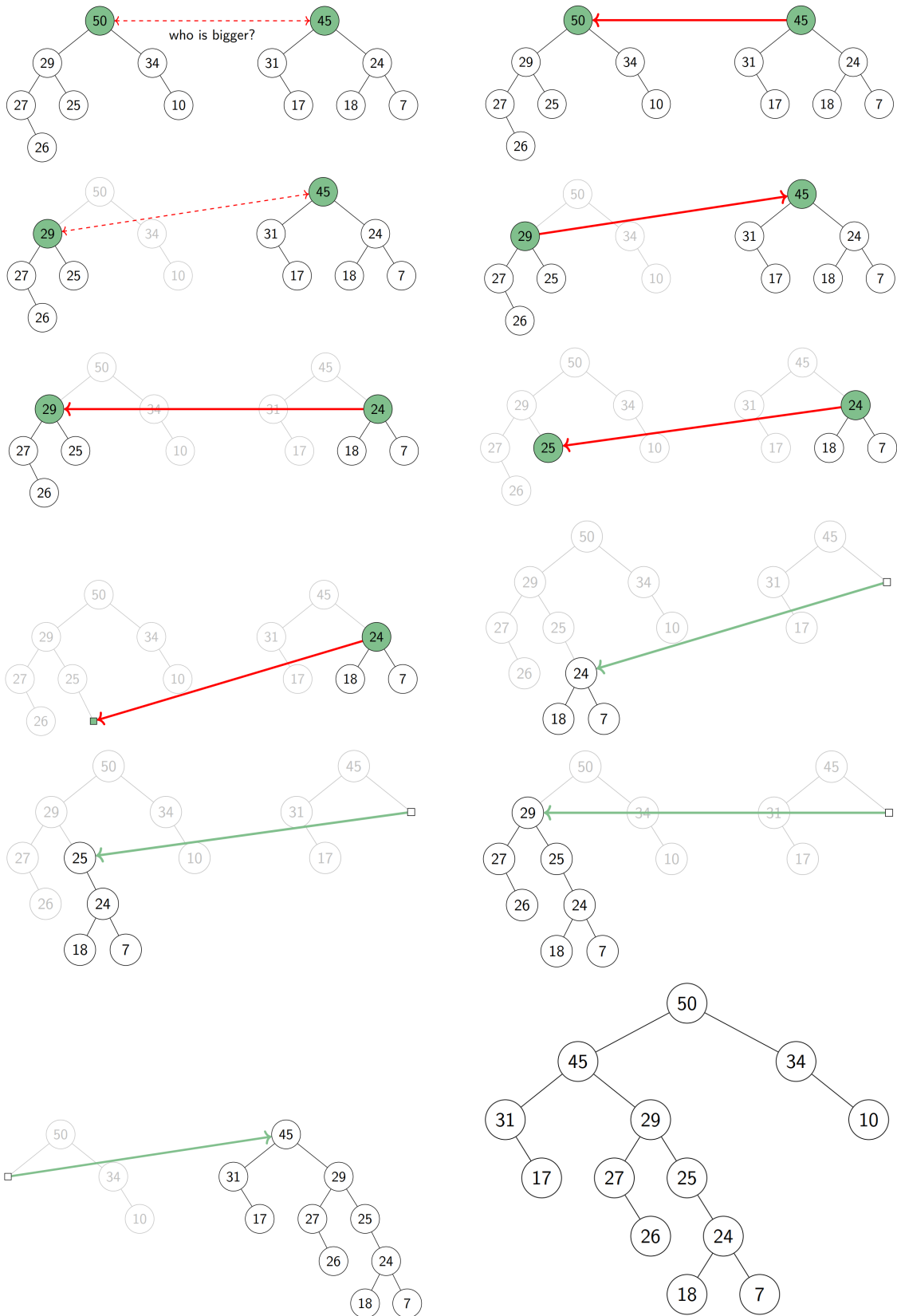
A meldable heap is a *tree-based* priority queue (nodes have references to left/right)

- *Heap order property*: parent not smaller than child
- *No structural property*: any binary tree is allowed

Idea: merge heap with smaller root into the other one by *randomly* choosing which sub-heap to merge.

```
1 // r1, r2: roots of two heaps (possibly NIL)
2 // returns root of merged heap
3 meldableHeap::merge(r1, r2)
4     if r1 is NIL return r2
5     if r2 is NIL return r1
6     if r1.key < r2.key swap(r1, r2)
7     // now r1 has max-key and becomes the root.
8     randomly pick one child c of r1
9     replace subheap at c by heapMerge(c, r2)
10    return r1
```

Example:



Theorem: the expected runtime to *merge* two meldable heaps is $T(n) \in O(\log n)$

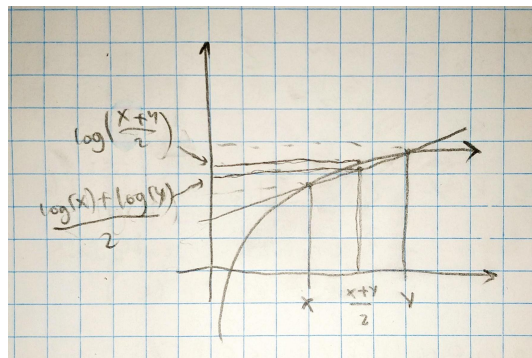
Proof: $T(n)$ = expected length of a random downward walk

- *Claim:* $T(n) \leq \log(n + 1)$
- *Base case:* $n = 1$ gives $T(1) \leq \log 2 = 1$
- *Inductive step:* for steps $n \geq 1$ let $R = \langle c, R' \rangle$ where
 - c is a coin flip at root
 - R' is the remaining outcomes
 - $P(R) = P(c)P(R')$

$$\begin{aligned}
 T^{\text{exp}}(n) &= \max_{I \in \mathcal{I}_n} \sum_R P(R)T(I, R) \\
 &= \sum_c \sum_{R'} P(c)P(R')T(I, \langle c, R' \rangle) \\
 &= \frac{1}{2} \sum_{R'} P(R')(1 + T(I_{\text{left}}, R')) + \frac{1}{2} \sum_{R'} P(R')(1 + T(I_{\text{right}}, R')) \\
 &= \frac{1}{2} \sum_{R'} P(R')(1 + T^{\text{exp}}(I_{\text{left}}, R')) + \frac{1}{2} \sum_{R'} P(R')(1 + T^{\text{exp}}(I_{\text{right}}, R')) \quad (\text{unexplained}) \\
 &= \max_I \left\{ 1 + \frac{1}{2}T^{\text{exp}}(n_L) + \frac{1}{2}T^{\text{exp}}(n_R) \right\} \\
 &= \max_{n_L+n_R=n-1} \left\{ 1 + \frac{1}{2}T^{\text{exp}}(n_L) + \frac{1}{2}T^{\text{exp}}(n_R) \right\} \\
 &\leq \max_{n_L+n_R=n-1} \left\{ 1 + \frac{1}{2} \log(n_L + 1) + \frac{1}{2} \log(n_R + 1) \right\} \quad (\text{induc hypo}) \\
 &\leq 1 + \log\left(\frac{n+1}{2}\right) = \log(n + 1)
 \end{aligned}$$

This last step is because log is concave which allows us to say that

$$\frac{\log(x) + \log(y)}{2} \leq \log\left(\frac{x+y}{2}\right) \quad \rightarrow \quad \frac{\log(n_L + 1) + \log(n_R + 1)}{2} \leq \log\left(\frac{n_L + 1 + n_R + 1}{2}\right)$$



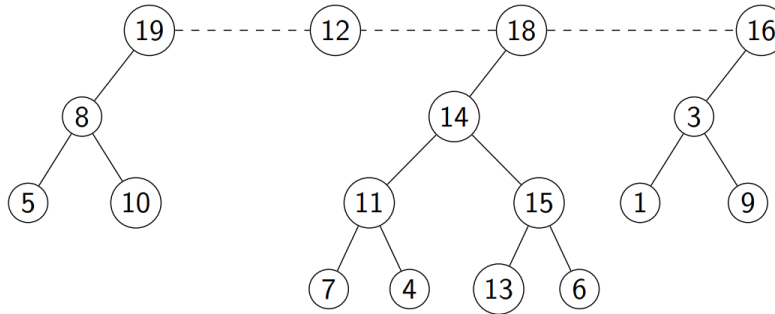
Since the runtime is no more than two *random downward walks* in a binary tree (we are walking two two trees) we get $T(n) \in O(\log n)$

I don't really understand this proof, for better check out video on learn.

So merge (and also *insert* and *deleteMax*) takes $O(\log n)$ expected time.

Binomial Heaps

This has a very different structure from binary heaps and melable heaps:



- Uses a list of L binary trees
 - Each binary tree is a *flagged tree*: a complete binary tree T as the left subtree of root r
 - * Flagged tree of height h has exactly 2^h nodes
 - * So $h \leq \log n$ for all flagged trees
- Order-property: nodes in *left* subtree have no larger keys. No restrictions on nodes in right subtree.
 - Root node will always be the largest (since the subtree is placed on the left)

Binomial Heap Operations

- *insert*: reduce to *merge* as before
- *findMax*: search every root for maximum and take that, so $L \rightarrow O(|L|)$ time
 - To minimize $|L|$ we want the least number of flagged trees

Proper Binomial Heap: no two flagged trees have the same height

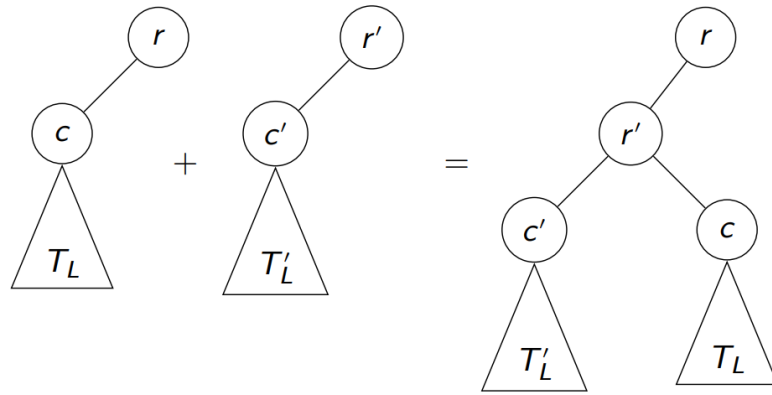
- Flagged tree of largest height h has $h \leq \log n$
- Only one flagged tree of each height in $\{0, \dots, h\}$
- Every number can be represented in binary so our binomial representation is:

$$n = c_k 2^k + c_{k-1} 2^{k-1} + \dots + c_0 2^0$$

- This ensures that $|L| \leq \log n + 1$

Making Binomial Heaps proper

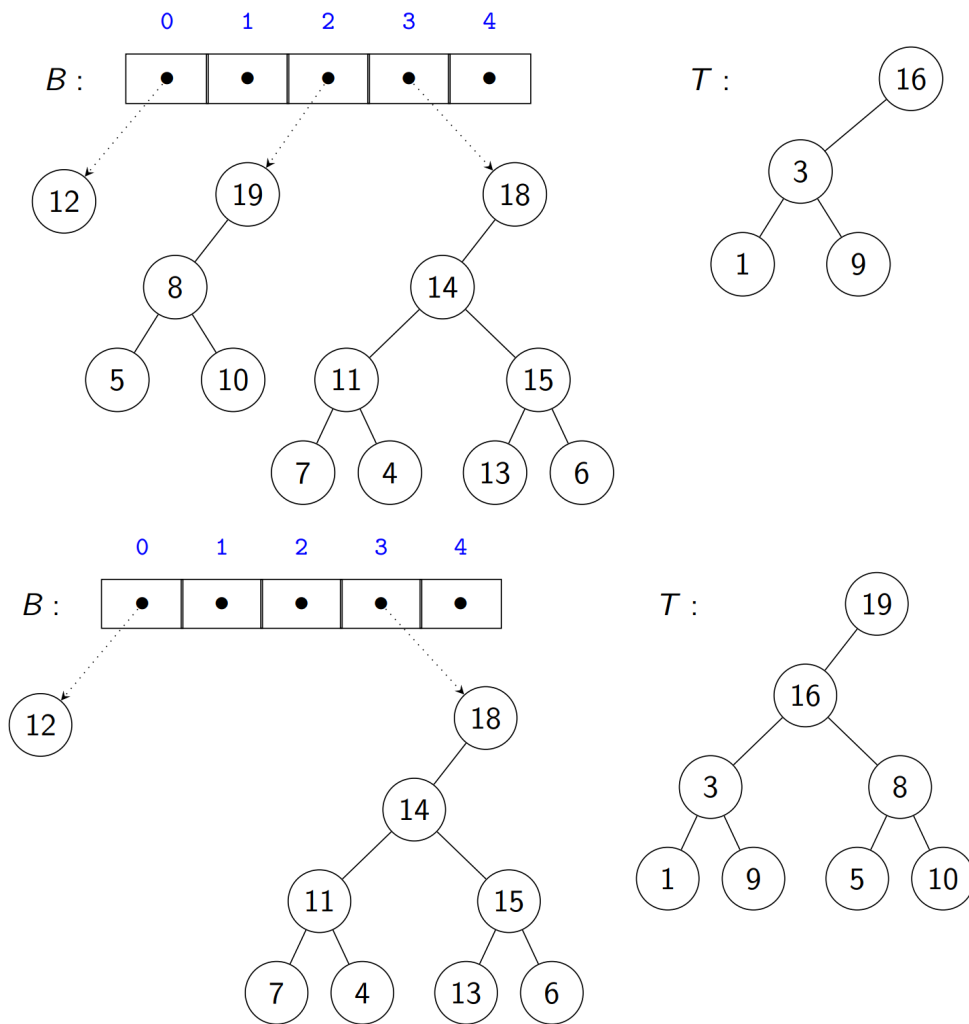
- Combining two flagged trees of the same height can be done in constant time. If $r.\text{key} \geq r'.\text{key}$:

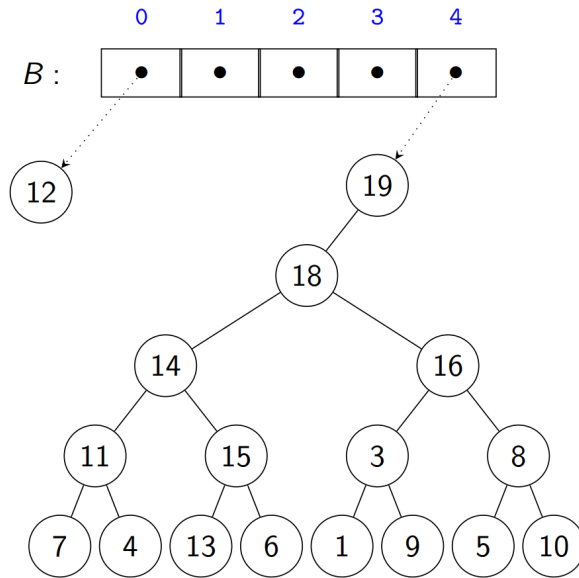


- The idea is to just do this whenever two flagged trees have the same height
- Runtime to make proper takes $O(|L| + \log n)$ if implemented suitably

The convention for L is that each spot i holds a flag tree of height i (with 2^i nodes)

Example: merging binomial heaps B and T then making the result proper





Computing $\log n$ takes $\log n$ time (done by divided by 2 repeatedly)

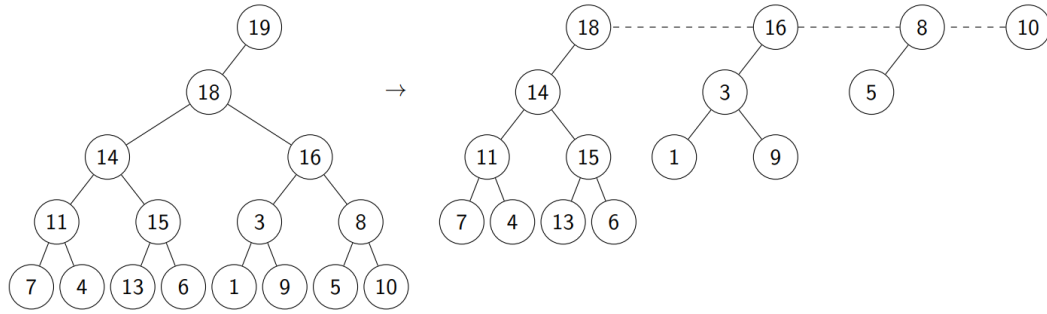
```

1 binomialHeap::makeProper()
2   n ← size of the binomial heap
3   compute  $\ell \leftarrow \lfloor \log n \rfloor$ 
4   B ← array of size  $\ell + 1$ , initialized all-NIL
5   L ← list of flagged trees
6   while L is non-empty do
7     T ← L.pop(), h ← T.height
8     while T' ← B[h] is not NIL do
9       if T.root.key < T'.root.key do swap T and T'
10      // combine T with T'
11      T'.right ← T.left, T.left ← T', T.height ← h+1
12      B[h] ← NIL, h++
13      B[h] ← T
14  // copy B back to list
15  for (h = 0; h ≤ ℓ; h++) do
16    if B[h] != NIL do L.append(B[h])

```

Proper binomial heap operations:

- Make binomial heap property after every operation
 - L always as length $O(\log n)$
 - *makeProper* takes $O(\log n)$ time
- *findMax*: $O(\log n)$ worst-case time
- *merge*: $O(\log n)$ worst-case time (concatenate the two lists then *makeProper*)
- *insert*: $O(\log n)$ worst-case time via *merge*
- *deleteMax*: $O(\log n)$ worst-case
 - Let T be the maximum among roots
 - After finding T split $T \setminus \{\text{root}\}$ into flagged trees T_1, \dots, T_k



- Finally merge $L \setminus T$ with $\{T_1, \dots, T_k\}$
- This has a $k \leq \log n \rightarrow O(\log n)$ worst-case time

In summary all operations have $O(\log n)$ worst-case run-time

Sorting, Average-case, and Randomization

Average-case Analysis

Definition: the *average-case run-time* of an algorithm is:

$$T^{\text{avg}}(n) = \frac{\sum_{I:\text{size}(I)=n} T(I)}{\# \text{ instances of size } n} = \frac{\sum_{I \in \mathcal{I}_n} T(I)}{|\mathcal{I}_n|}$$

Note: we need \mathcal{I}_n to be finite.

Sorting Permutations

There is a infinite number of sets with n numbers so in order to take the average running time over all inputs we need some method to characterize the set so we can look at it more abstractly.

For *comparison-based* algorithms notice we don't care about the absolute size of each element.

$$A = [14, 3, 2, 6, 1, 11, 7] \quad A' = [14, 4, 2, 6, 1, 12, 8]$$

Both sets will have the same run-time. The actual numbers don't matter, only their *relative order*

S_n (symmetric group of n th order) contains $n!$ different classes to characterize our lists of n elements into. Think of it as the set of possible permutations of n elements. (this course we will use Π_n instead)

To characterize relative order via *sorting permutation* we have a permutation $\pi \in \Pi_n$ for which:

$$A[\pi(0)] \leq A[\pi(1)] \leq \dots \leq A[\pi(n-1)]$$

Example: notice that π^{-1} has the same sorting permutation as the list A

index		0	1	2	3	4	5	6
A	=	14	3	2	6	1	11	7
π	=	4	2	1	3	6	5	0
π^{-1}	=	6	2	1	3	0	5	4

If we assume that all $n!$ sorting permutations are *equally likely* so we have $T^{\text{avg}}(n) = \frac{1}{n!} \sum_{\pi \in \Pi} T(\pi)$ where

$$\begin{aligned} T(\pi) &= \text{run-time on any instance with sorting-permutation } \pi \\ &= \text{run-time on } \pi^{-1} \end{aligned}$$

Then π^{-1} can be used to represent all arrays in the class that π sorts (same runtime for every array in the class)

Example: Average-case Run-time of avgCaseDemo

```

1 // A: array of size n with distinct items
2 avgCaseDemo(A, n)
3   if n ≤ 2 return
4   if A[n-2] < A[n-1]
5     avgCaseDemo(A[0..n/2-1], n/2) // Good case
6   else avgCaseDemo(A[0..n-3], n-2) // Bad case

```

Let $T(n)$ be the number of *recursions* (this will asymptotically be the same as the run-time)

- Worst-case analysis: recursive call could always have size $n - 2$:

$$T(n) = 1 + T(n - 2) = 1 + 1 + \dots + T(2) = n/2 - 1 \in \Theta(n)$$

- Best-case analysis: recursive call always have size $n/2$:

$$T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = \dots = \log n - 1 \in \Theta(\log n)$$

- Average-case analysis:

– For one instance of π we get the recursive formula:

$$T(\pi) = \begin{cases} 1 + T(\text{first } n/2 \text{ items}) & \text{if } \pi \text{ is good} \\ 1 + T(\text{first } n - 2 \text{ items}) & \text{if } \pi \text{ is bad} \end{cases}$$

May be tempted to write $1 + T^{\text{avg}}(n/2)$ and $1 + T^{\text{avg}}(n - 2)$ but this is incorrect because we are finding the runtime on a certain problem instance.

– Recursive formula for all instances π together (not at all trivial):

$$\sum_{\pi \in \Pi_n} T(\pi) = \sum_{\pi \in \Pi_n: \pi \text{ good}} (1 + T^{\text{avg}}(n/2)) + \sum_{\pi \in \Pi_n: \pi \text{ bad}} (1 + T^{\text{avg}}(n - 2))$$

– Now we find the recursive formula for $T^{\text{avg}}(n)$

$$\begin{aligned} T^{\text{avg}}(n) &= \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi) = \frac{1}{|\Pi_n|} \left(\sum_{\pi \in \Pi_n: \pi \text{ good}} T(\pi) + \sum_{\pi \in \Pi_n: \pi \text{ bad}} T(\pi) \right) \\ &= \frac{1}{|\Pi_n|} \left(\sum_{\pi \in \Pi_n: \pi \text{ good}} (1 + T^{\text{avg}}(n/2)) + \sum_{\pi \in \Pi_n: \pi \text{ bad}} (1 + T^{\text{avg}}(n - 2)) \right) \\ &= 1 + \frac{1}{|\Pi_n|} \left(|\{\pi \in \Pi_n : \pi \text{ good}\}| \cdot T^{\text{avg}}(n/2) + |\{\pi \in \Pi_n : \pi \text{ bad}\}| \cdot T^{\text{avg}}(n - 2) \right) \\ &= 1 + \frac{1}{2} T^{\text{avg}}(n/2) + \frac{1}{2} T^{\text{avg}}(n - 2) \quad (\text{since exactly half the permutations are good}) \end{aligned}$$

- Claim: $T^{\text{avg}}(n) \leq 2 \log n$
- Base case: statement holds for $n \leq 2$
- Inductive step: assume that the statement holds for any $m < n$ where $n \geq 3$ then

$$\begin{aligned} T^{\text{avg}}(n) &= 1 + \frac{1}{2}T^{\text{avg}}(n/2) + \frac{1}{2}T^{\text{avg}}(n-2) \\ &\leq 1 + \frac{1}{2}(2 \log(n/2)) + \frac{1}{2}(2 \log(n-2)) \\ &\leq 1 + \log n - \log 2 + \log n = 2 \log n \end{aligned}$$

Thus the claim holds and the average-case run-time is $O(\log n)$

Randomized Algorithms

Definition: a *randomized algorithm* is one that relies on some random numbers in addition to the input

Remark: software cannot generate randomness instead a *pseudo-random number generators* (PRNG) is given a *seed* and generates a sequence for seemingly random numbers

- The quality of the randomized algorithms depends on the quality of the PRNG
- There does exist hardware for true randomness (e.g. detecting radioactive decay or cosmic rays)

The goal such an algorithm is to shift run-time dependency on the input to the random numbers. Removing bad instances and only leaving unlucky numbers makes the run-time of the algorithm more stable.

e.g. if you are given a bad case maybe you can shuffle the elements to get a better case.

Expected Running Time

Definition: $T(I, R)$ is the running time of the randomized algorithm \mathcal{A} with:

- Problem instance I
- Sequence of random numbers R

Definition: $T^{\text{exp}}(I)$ is the *expected running time* on I .

- The expected value for runtime over all possible random inputs.

$$T^{\text{exp}}(I) = E[T(I, R)] = \sum_R T(I, R)P(R)$$

Definition: $T^{\text{exp}}(n)$ is the *expected running time* of \mathcal{A} on problems of size n .

- The max expected runtime over all instances of size n .

$$T^{\text{exp}}(n) := \max_{I \in \mathcal{I}_n} T^{\text{exp}}(I)$$

Occasionally we discuss the very worst that could happen: $\max_I \max_R T(I, R)$.

Example: Expected Running Time of `expectedDemo`

```
1 // A: array of size n with distinct items
2 expectedDemo(A, n)
3   if n ≤ 2 return
4   if random(2) swap A[n-1] and A[n-2]
5   if A[n-2] ≤ A[n-1]
6     expectedDemo(A[0..n/2-1], n/2) // Good case
7   else expectedDemo(A[0..n-3], n-2) // Bad case
```

- Assume that $\text{random}(n)$ returns an integer uniformly from $\{0, 1, 2, \dots, n-1\}$.
- Observe that $P(\text{good case}) = \frac{1}{2} = P(\text{bad case})$

Run-time on array A if random outcomes are $R = \langle x, R' \rangle$

$$T(A, R) = \begin{cases} 1 + T(A[0 \dots \frac{n}{2} - 1], R') & \text{if } x = \text{good} \\ 1 + T(A[0 \dots n - 3], R') & \text{if } x = \text{bad} \end{cases}$$

Summing up over all sequences of random outcomes:

$$\begin{aligned} \sum_R P(R)T(A, R) &= P(X \text{ good}) \sum_{R'} P(R') \left(1 + T(A[0 \dots \frac{n}{2} - 1], R') \right) \\ &\quad + P(X \text{ bad}) \sum_{R'} P(R') \left(1 + T(A[0 \dots n - 3], R') \right) \\ &= 1 + \frac{1}{2} \sum_{R'} P(R')T(A[0 \dots \frac{n}{2} - 1], R') + \frac{1}{2} \sum_{R'} P(R')T(A[0 \dots n - 3], R') \\ &\leq 1 + \frac{1}{2} \underbrace{\max_{A' \in \mathcal{I}_{n/2}} \sum_{R'} P(R')T(A', R')}_{T^{\text{exp}}(\lfloor n/2 \rfloor)} + \frac{1}{2} \underbrace{\max_{A' \in \mathcal{I}_{n-2}} \sum_{R'} P(R')T(A', R')}_{T^{\text{exp}}(n-2)} \\ &\leq 1 + \frac{1}{2}T^{\text{exp}}(n/2) + \frac{1}{2}T^{\text{exp}}(n-2) \end{aligned}$$

Then since this holds for *all* A we have:

$$T^{\text{exp}}(n) = \max_{A \in \mathcal{I}_n} \sum_R P(R)T(A, R) \leq 1 + \frac{1}{2}T^{\text{exp}}(n/2) + \frac{1}{2}T^{\text{exp}}(n-2)$$

- This the same recursion as for $T_{\text{avgCaseDemo}}^{\text{avg}}(n)$
- So we can perform the same analysis to arrive at $T_{\text{expectedDemo}}^{\text{exp}}(n) \in O(\log n)$

In general, we cannot say that the expected time of a randomized version of an algorithm is the same as the average-case time of the deterministic version. However, later we will see a case when they are equal.

QuickSelect

The Selection Problem

Definition: the *selection problem* is given an array A of n numbers with $0 \leq k < n$ find the element that would be at position k of the sorted array

- Special case: *median finding* = selection with $k = \lfloor \frac{n}{2} \rfloor$
- Using heapify then performing k deleteMax, selection can be done with heaps in $\Theta(n + k \log n)$
 - Median-finding with this takes $\Theta(n \log n)$ (same cost as our best sorting algorithms)

We will see that using *QuickSelect* we can do this in linear time.

Subroutines

QuickSelect and the related algorithm *QuickSort* rely on two subroutines:

- *choose-pivot*(A): return an index p in A . We will use the *pivot-value* $v \leftarrow A[p]$ to rearrange the array
 - We want this to be close to median value for best performance
 - For now just assume it always selects rightmost element in array (`return A.size - 1`)
- *partition*(A, p): rearrange A and return pivot index i such that
 - pivot-value v is in $A[i]$
 - all items in $A[0, \dots, i - 1]$ are $\leq v$
 - all items in $A[i + 1, \dots, n - 1]$ are $\geq v$

Simple linear-time *partition* algorithm

```
1 // A: array of size n
2 // p: integer s.t. 0 ≤ p < n
3 partition(A, p)
4   Create empty lists smaller, equal and larger.
5   v ← A[p]
6   for each element x in A
7     if x < v then smaller.append(x)
8     else if x > v then larger.append(x)
9     else equal.append(x).
10  i ← smaller.size
11  j ← equal.size
12  Overwrite A[0..i-1] by elements in smaller
13  Overwrite A[i..i+j-1] by elements in equal
14  Overwrite A[i+j..n-1] by elements in larger
15  return i
```

It is possible to do this in both linear-time and with $O(1)$ auxiliary space.

Efficient in place *partition* algorithm (Hoare)

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

The idea is to keep swapping the outer-most wrongly-positioned pairs.

Loop invariant: A

	$\leq v$?	$\geq v$	v
	i		j	$n-1$

Start from left and start from right, then scan toward the center doing swaps.
 (complete pain to implement)

Example:

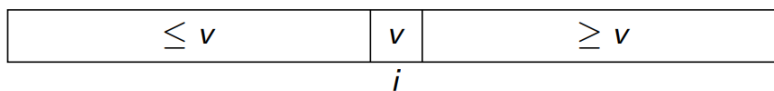
$i=-1$	0	1	2	3	4	5	6	7	8	$j=9$
	30	60	10	0	50	80	90	20	40	$v=70$
	0	1	2	3	4	$i=5$	6	7	$j=8$	9
	30	60	10	0	50	80	90	20	40	$v=70$
	0	1	2	3	4	$i=5$	6	7	$j=8$	9
	30	60	10	0	50	40	90	20	80	$v=70$
	0	1	2	3	4	5	$i=6$	$j=7$	8	9
	30	60	10	0	50	40	90	20	80	$v=70$
	0	1	2	3	4	5	$i=6$	$j=7$	8	9
	30	60	10	0	50	40	20	90	80	$v=70$
	0	1	2	3	4	5	$j=6$	$i=7$	8	9
	30	60	10	0	50	40	20	90	80	$v=70$
	0	1	2	3	4	5	$j=6$	$i=7$	8	9
	30	60	10	0	50	40	20	70	80	90

QuickSelect Algorithm

```

1 // A: array of size n
2 // k: integer s.t.  $0 \leq k < n$ 
3 QuickSelect(A, k)
4   p ← choose-pivot(A)
5   i ← partition(A, p)
6   if i = k then
7     return A[i]
8   else if i > k then
9     return QuickSelect(A[0, 1, ..., i-1], k)
10  else if i < k then
11    return QuickSelect(A[i+1, i+2, ..., n-1], k - (i+1))
    
```

After each partition we have:



- If $k = i$ then return v (value has been found and we are done)
- If $k < i$ then perform QuickSelect on $\leq v$ part of the array
- If $k > i$ then perform QuickSelect on $\geq v$ part of the array

Let $T(n, k)$ be the number of *key-comparisons* in a size n array with parameter k (asymptotically the same as run-time)

Note that *partition* uses n key-comparisons

- Worst-case analysis: Pivot-index is last, $k = 0$

$$T(n, 0) \geq n + (n - 1) + (n - 2) + \dots + 1 \in \Omega(n^2)$$

- Best-case analysis: first chosen pivot could be k th element (no recursive calls)

$$T(n, k) = n \in \Theta(n)$$

- Average-case analysis: using sorting permutations and ignoring the parameter k we have

$$T^{\text{avg}}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi)$$

Assume that sorting permutation π gives pivot-index i then if array after partition is A' then

$$T(\pi) \leq n + \max \left\{ \underbrace{T(A'[0 \dots i - 1])}_{\text{size } i}, \underbrace{T(A'[i + 1 \dots n - 1])}_{\text{size } n - i - 1} \right\}$$

- Option 1: perform a *very* complicated proof to show that

$$\sum_{\pi \in \Pi_n: \text{pivot-idx } i} T(\pi) \leq \sum_{\pi \in \Pi_n: \text{pivot-idx } i} \left(n + \max \{ T^{\text{avg}}(i), T^{\text{avg}}(n - i - 1) \} \right)$$

- Option 2: prove the avg-case run-time via randomized version of algorithm

- * Convert *QuickSelect* to *RandomizedQuickSelect*
- * Determine the expected run-time of *RandomizedQuickSelect*
- * Find how the expected run-time implies the avg-case run-time of *QuickSelect*

It is too difficult to use option 1 so we will use option 2 (note that option 2 is not always valid)

Randomized QuickSelect

To create a randomized version of *QuickSelect*:

- First idea: randomly permute the input first using *shuffle*

```

1 // A: array of size n
2 shuffle(A)
3   for i ← 1 to n-1 do
4     swap(A[i], A[random(i+1)])

```

This works well, but we can do it directly within the routine

- Second idea: change the pivot selection

```

1 RandomizedQuickSelect(A, k)
2   ...
3   p ← random(A.size)
4   i ← partition(A, p)
5   ...

```

We observe that $P(\text{pivot has index } i) = \frac{1}{n}$

We use the second idea to create *RandomizedQuickSelect*

Assume we know that the first *random* gave pivot-index i :

- We either recurse into an array of size i , $n - i - 1$, or not at all
- If new array after *partition* is A' and $R = \langle i, R' \rangle$ then

$$T(\pi, k, \langle i, R' \rangle) \leq n + \begin{cases} T(A'[0 \dots i - 1], k, R') & \text{if } i > k \\ T(A'[i + 1 \dots n - 1], k - i - 1, R') & \text{if } i < k \\ 0 & \text{otherwise} \end{cases}$$

$$T(\pi, k, \langle i, R' \rangle) \leq n + T(\text{some array of size } i \text{ or } n - i - 1, \text{ some } k', R')$$

Claim: over all choices of i and R'

$$\sum_R T(\pi, k, R) \leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T^{\text{exp}}(i), T^{\text{exp}}(n - i - 1)\}$$

- Proof is similar to *expectedDemo* (note $T^{\text{exp}}(\cdot)$ uses maximum over all instances)
- This bound holds for all π, k

$$T^{\text{exp}}(n) = \max_{\pi} \max_k \sum_R T(\pi, k, R) \leq n + \frac{1}{n} \sum_{i=0}^{n-1} \max\{T^{\text{exp}}(i), T^{\text{exp}}(n - i - 1)\}$$

Claim: this recursion resolves to $O(n)$

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

Proof: show that $T^{\text{exp}}(n) \leq 4n$

- Base case: $n = 1$ holds
- Inductive step: assume $T^{\text{exp}}(m) \leq 4m$ for all $m < n$

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

Derivation for (*):

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

Thus we can say *RandomizedQuickSelect* has expected run-time of $O(n)$

- This is generally the *fastest QuickSelect* implementation
- CS341 has variation with worst-case $O(n)$ running time, but double recursion and is slower in practice

Expected Run-time vs Average-case Run-time

In general the expected and average run-time is not the same, however here is a special case.

- Assume we have an algorithm \mathcal{A} that solves selection or sorting
- Create a randomized algorithm \mathcal{B} as follows:
 - Let I be the given instance (an array)
 - Randomly (and uniformly) permute I to get I'
 - * Done with *shuffle* but for *QuickSelect* random pivot has same effect
 - Call algorithm \mathcal{A} on input I'

Claim: $T_{\mathcal{B}}^{\text{exp}}(n) = T_{\mathcal{A}}^{\text{avg}}(n)$

Proof:

- I is a instance with sorting permutation π
- σ is the sorting permutation for shuffling I so we have $\sigma(I) = I'$
- Then the sorting permutation of I' is $\pi \circ \sigma^{-1}$

$$\sigma(I) = I' \quad \rightarrow \quad \sigma^{-1}(I') = I \quad \rightarrow \quad \pi \circ \sigma^{-1}(I') = \underbrace{\pi(I)}_{\text{sorted}}$$

- For a group G and a subgroup H of G , if we let $g \in G$ then

$$gH = \{gh : h \in H\}$$

- Since G is a subgroup of G then $gG = G$ since $gG \subseteq G$ and $gG \supseteq G$

$$h \in G \quad \rightarrow \quad h \in gG \quad \rightarrow \quad h = g(g^{-1}h)$$

$$\begin{aligned}
T_{\mathcal{B}}^{\text{exp}}(n) &= \max_{I \in \mathcal{I}_n} \sum_R T_{\mathcal{B}}(I, R) Pr(R) \\
&= \max_{I \in \mathcal{I}_n} T_{\mathcal{A}}(I') Pr(R) \\
&= \max_{\pi \in \Pi_n} \frac{1}{n!} \sum_{\sigma \in \Pi_n} T_{\mathcal{A}}(\pi \circ \sigma^{-1}) \\
&= \max_{\pi \in \Pi_n} \frac{1}{n!} \sum_{\tau \in \Pi_n} T_{\mathcal{A}}(\tau) \\
&= T_{\mathcal{A}}^{\text{avg}}(n)
\end{aligned}$$

Using this we conclude that since *RandomizedQuickSelect* has expected run-time $O(n)$, then *QuickSelect* has average-case run-time $O(n)$

QuickSort

Hoare developed *partition* and *QuickSelect* in 1960, he then applied the ideas to create *QuickSort*:

```

1 // A: array of size n
2 QuickSort(A)
3   if n ≤ 1 then return
4   p ← choose-pivot(A)
5   i ← partition(A, p)
6   QuickSort(A[0, 1, ..., i-1])
7   QuickSort(A[i+1, ..., n-1])

```

QuickSort Analysis

Set $T(n) := \#$ of key-comparison for *QuickSort* in a size- n array

- Worst-case analysis: recursive call could always have size $n - 1$

$$T(n) \geq n + T(n - 1) \in \Omega(n^2)$$

this is tight since the recursion depth is at most n

- Best-case analysis: if pivot-index is always in the middle we recurse in two sub-arrays of size $\leq n/2$

$$T(n) \leq n + 2T(n/2) \in O(n \log n)$$

this can be shown to be tight

- Average-case analysis: will be proved via randomization.

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

– Assume that $n = 4q$

– Claim: $T^{\text{exp}}(n) \leq 2n \log_{4/3} n$

– Base case: $n = 1$ we know that $T^{\text{exp}}(1) = 0$ directly which is ≤ 0

– Inductive step: we call an index i small if $i \leq \frac{3}{4}n$ and large otherwise

* For small i :

$$T^{\text{exp}}(i) \leq 2i \log_{4/3} i \leq 2i \log_{4/3} \left(\frac{3}{4}n \right) = 2i \log_{4/3} n - 2i$$

* For large i :

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

Easier method to find this using expected recursion-depth in module 3 slides 31/45 to 32/45

QuickSort Improvements

- Auxiliary space is $\Omega(\text{recursion depth})$
 - $\Theta(n)$ in the worst-case, $\Theta(\log n)$ in average-case
 - Worst case can be reduced to $\Theta(\log n)$ by recursing in smaller sub-array first and replacing the other recursion by a while-loop
- Stop recursing when $n \leq 10$ then run *InsertionSort* to sort in $O(n)$ since all items are within 10 units of their required position
- Array with many duplicates can be sorted by changing *partition* to produce three subsets

$\leq v$	$= v$	$\geq v$
----------	-------	----------

- Two programming tricks that apply in many situations
 - Instead of passing full arrays, pass only the range of indices
 - Avoid recursion altogether by keeping an explicit stack

```

1 QuickSortImproved(A, n)
2   Initialize a stack S of index-pairs with { (0, n-1) }
3   while S is not empty
4     (l,r) ← S.pop()
5     while (r-l+1 > 10) do
6       p ← choose-pivot-improved(A, l,r)
7       i ← partition-improved(A, l,r, p)
8       if (i-l > r-i) do
9         S.push( (l, i-1) )
10        l ← i+1
11      else
12        S.push( (i+1,r) )
13        r ← i-1
14   InsertionSort(A)

```

Usually in practice this is most efficient sorting algorithm (even though worst-case is still $\Theta(n^2)$)

Comparison-based Sorting Lower Bound

We have seen a good variety of sorting algorithms:

Algorithm	Running time	Analysis
Selection Sort	$\Theta(n^2)$	worst-case
Insertion Sort	$\Theta(n^2)$	worst-case
Merge Sort	$\Theta(n \log n)$	worst-case
Heap Sort	$\Theta(n \log n)$	worst-case
QuickSort	$\Theta(n \log n)$	average-case
RandomizedQuickSort	$\Theta(n \log n)$	expected

Is it possible to do better than $\Theta(n \log n)$ running time?

- No: for comparison-based sorting the lower bound is $\Omega(n \log n)$
- Yes: non-comparison-based sorting methods can achieve $O(n)$ with restrictions on input

The Comparison Model

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

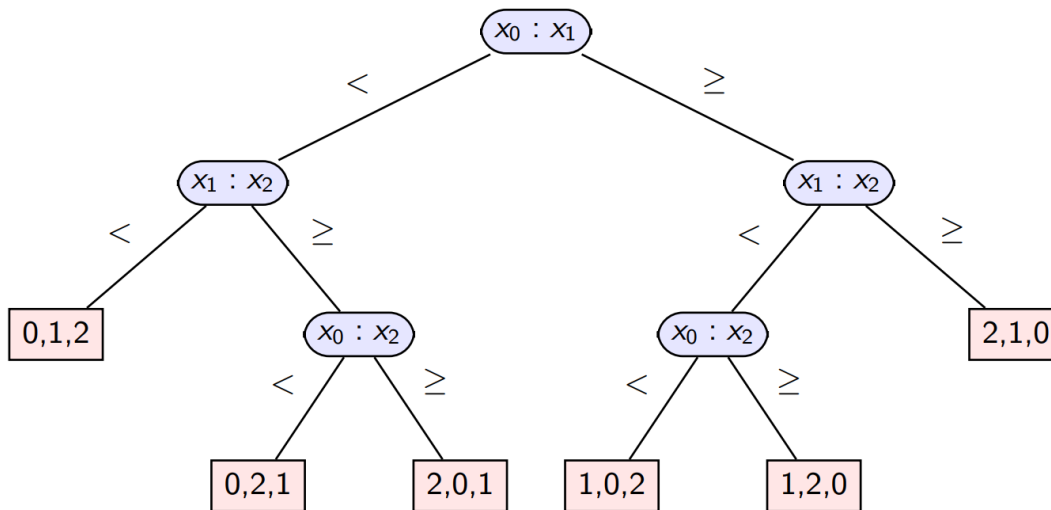
- Comparing two elements
- Moving elements around (e.g. copying, swapping)

All the sorting algorithms we have seen so far use the comparison model

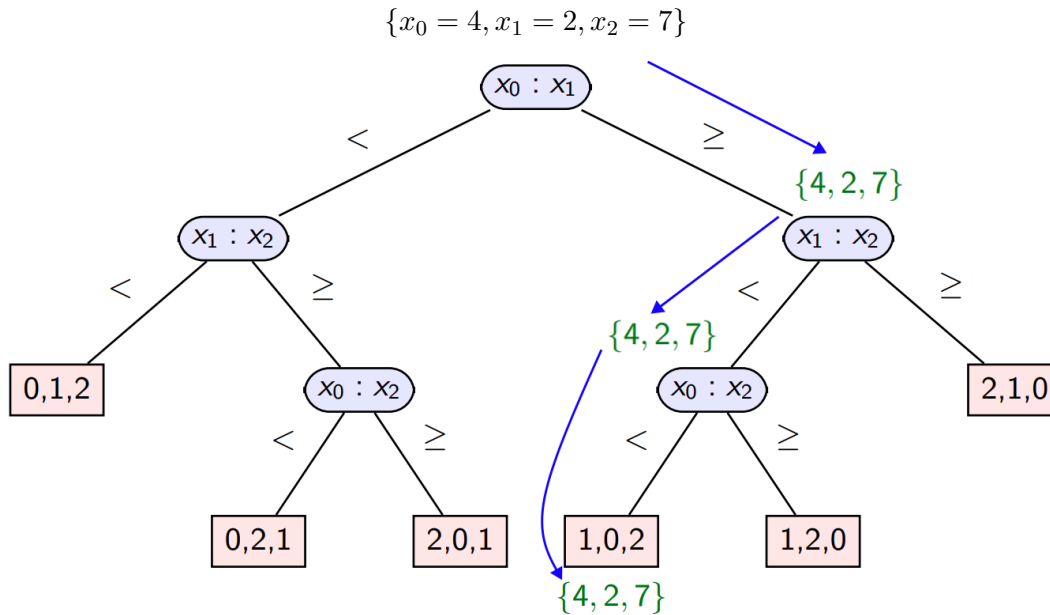
This is useful because it makes very few assumptions on the kind of things we are sorting.

Decision Trees

We can express comparison-based algorithms as a *decision tree*. To sort $\{x_0, x_1, x_2\}$ we have:



Example:



As a result we say the input $\{4, 2, 7\}$ has sorting permutation $\langle 1, 0, 2 \rangle$

Lower Bound for Sorting in the Comparison Model

Theorem: any correct *comparison*-based sorting algorithm requires at least $\Omega(n \log n)$ comparison operations to sort n distinct items

Proof:

- For n distinct elements there are $n!$ permutations, so there are at least $n!$ leaves in the decision tree
 - A sorting algorithm can be thought of as producing a permutation that sorts a list
 - In general a decision tree will have at least as many leaves as outputs of the algorithm
 - Since there are $n!$ ways to reverse list permutations the decision tree will have at least $n!$ leaves
- Binary tree with height h has at most 2^h leaves
- Since $n! \leq \# \text{ leaves}$ and $2^h \geq \# \text{ leaves}$ we get

$$2^h \geq \# \text{ leaves} \geq n! \quad \rightarrow \quad h \geq \log(\# \text{ leaves}) \geq \log n! \quad \rightarrow \quad h \geq \log n!$$

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

Thus the tree has depth $h \in \Omega(n \log n)$ thus we need at least $\Omega(n \log n)$ comparisons to sort a list.

Non-comparison-based Sorting

- Assume the keys are numbers in base R (also called *radix* R)
 - Common choices for radix are $R = 2, 10, 128, 256$
- Assume that all keys have m digits (by padding with leading 0s)

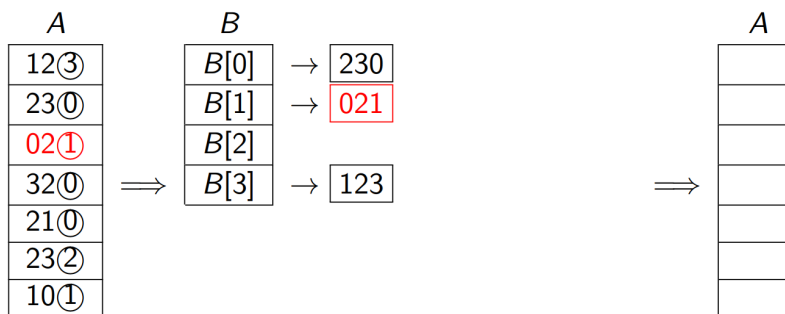
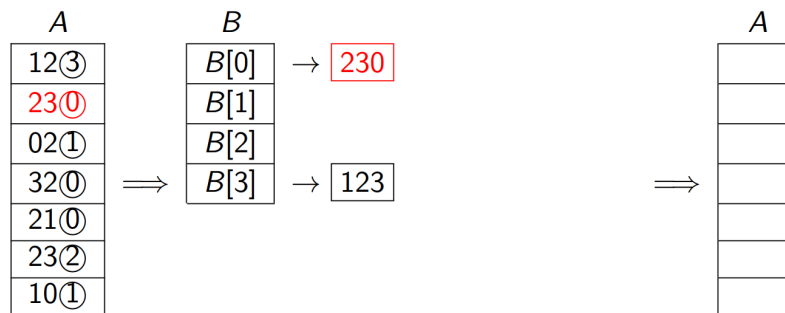
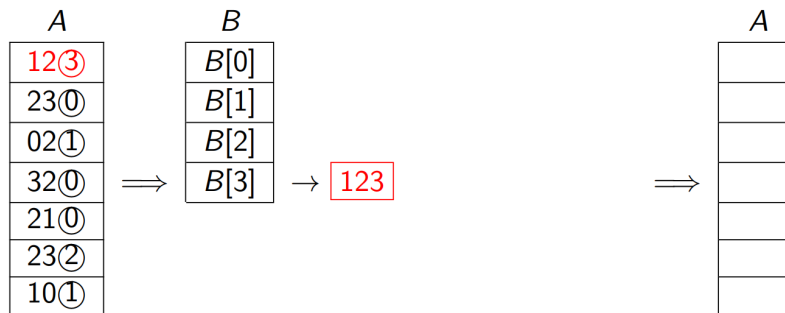
Example: $R = 4$ and $m = 3$

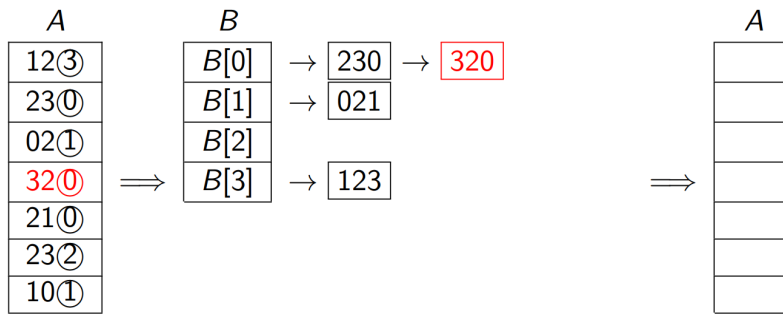
123	230	021	320	210	232	101
-----	-----	-----	-----	-----	-----	-----

Bucket Sort

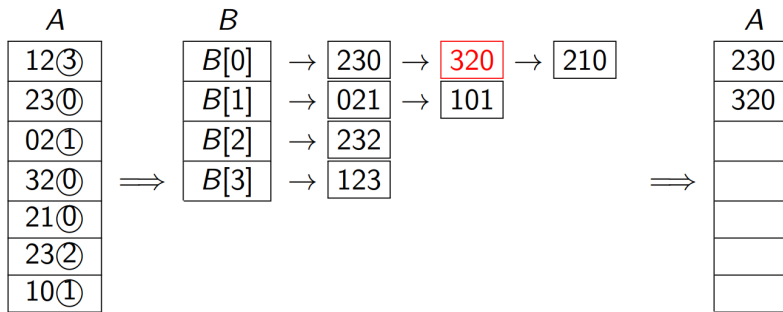
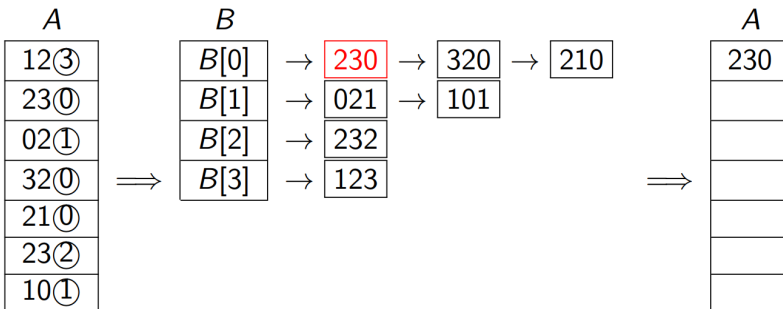
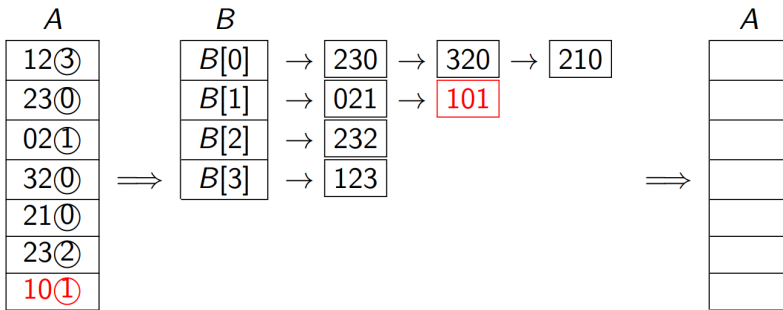
Create an array as large as the radix then choose an digit and sort the list by that digit.

Example: sorting the array A by its last digit

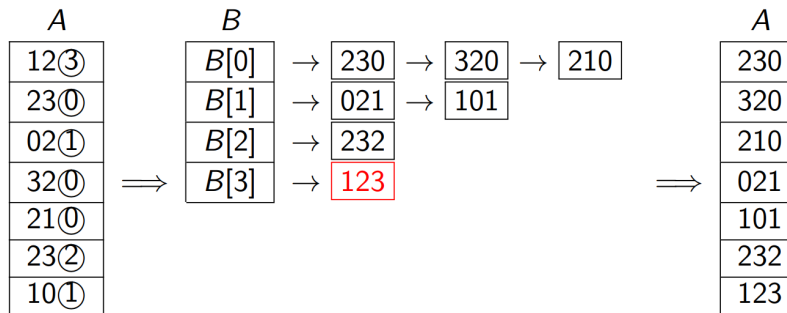




...



...



Important: always append to the *end* of the linked list and read from the *beginning*

Bucket sort on its own only works when $m = 1$ (numbers are one digit long)

For later it is *extremely* important that this sorting is *stable*
 (elements that are considered *equal* stay in the same relative order)

```

1 // A: array of size n, contains numbers with digits in {0,...,R-1}
2 // d: index of digit by which we wish to sort
3 Bucket-sort(A, d)
4   Initialize an array B[0..R-1] of empty lists (buckets)
5   for i ← 0 to n-1 do
6     Append A[i] at end of B[digit d of A[i]]
7   i ← 0
8   for j ← 0 to R-1 do
9     while B[j] is non-empty do
10      move first element of B[j] to A[i++]

```

Run-time $\Theta(n + R)$ and auxiliary space $\Theta(n + R)$

MSD Radix Sort

Sorts an array of m -digit radix- R numbers recursively, starting from leading digit.

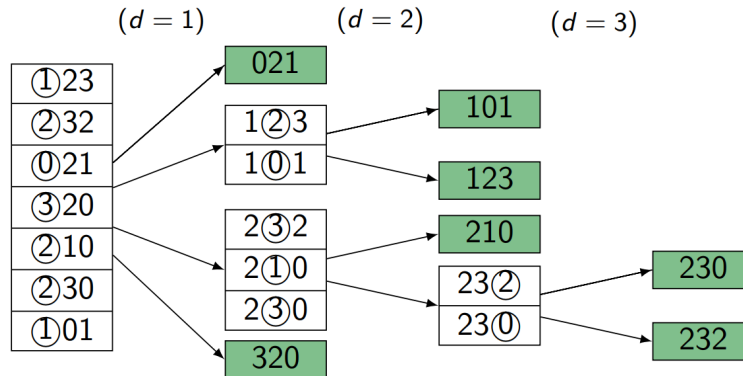
```

1 // ℓ,r: range of what we sort, 0 ≤ ℓ, r ≤ n-1
2 MSD-Radix-sort(A, ℓ ← 0, r ← n-1, d ← index of leading digit)
3   if ℓ < r
4     bucket-sort(A[ℓ..r], d)
5     if there are digits left // recurse in sub-arrays
6       ℓ' ← ℓ
7       while (ℓ' < r) do
8         Let r' ≥ ℓ' be maximal s.t. A[ℓ'..r'] all have same dth digit
9         MSD-Radix-sort(A, ℓ', r', d+1)
10      ℓ' ← r' + 1

```

- Auxiliary space: $\Theta(n + R + m)$ for *bucket-sort* and the recursion stack
- Run-time: $\Theta(mnR)$ since we may have $\Theta(mn)$ subproblems

Example:



The drawback of *MSD-Radix-Sort* is that there are many recursions

LSD Radix Sort

Sort an array of m -digit radix- R numbers starting from the last digit.

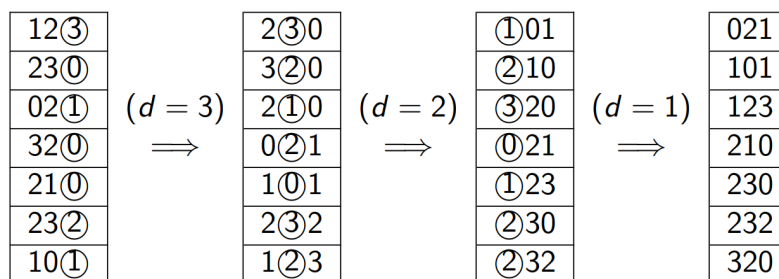
```

1 // A: array of size n, contains m-digit radix-R numbers
2 LSD-radix-sort(A)
3   for d ← least significant to most significant digit do
4     Bucket-sort(A, d)

```

- Loop-invariant: A is sorted w.r.t. digits d, \dots, m of each entry
 - This is due to the *stability* of bucket sort
- Time cost: $\Theta(m(n + R))$
- Auxiliary space: $\Theta(n + R)$

Example:



- When sorting *physical* items MSD radix sort works better
 - Works better than LSD radix sort if we expect lots of variation in length m
 - e.g. sorting books in alpha order

Dictionaries

Definition: a *dictionary* ADT is a collection of elements each of which is a *key-value pair* (KVP)

- Keys can be compared and are typically unique

Operations:

- *search*(k)
- *insert*(k, v)
- *delete*(k)

Optional: *isEmpty*, *size*, *join*, *closestKeyBefore*, etc.

Common assumptions:

- Dictionary has n KVPs
- Keys can be compared in constant time
- Each KVP uses constant space (if not, then *value* could be a pointer)

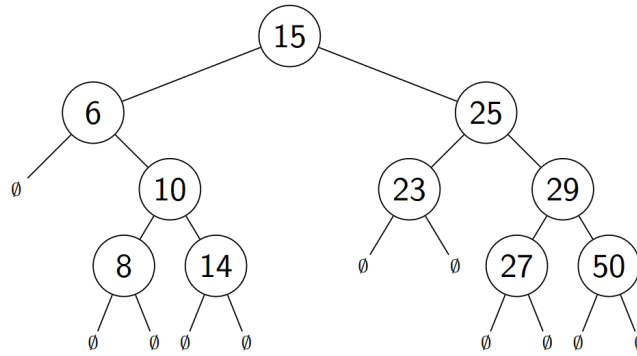
Elementary implementations:

- Unordered array or linked list
 - *search*: $\Theta(n)$
 - *insert*: $\Theta(1)$ (array may occasionally need resize)
 - *delete*: $\Theta(n)$ (need to search)
- Ordered array
 - *search*: $\Theta(\log n)$ (via binary search)
 - *insert*: $\Theta(n)$
 - *delete* $\Theta(n)$

Binary Search Tree (BST)

- Structure:
 - All nodes have two (possibly empty) subtrees and store a KVP
- Ordering:
 - Every key in $T.\text{left}$ is less than $T.\text{key}$
 - Every key in $T.\text{right}$ is more than $T.\text{key}$

Example:



BST can be used to realize an ADT dictionary.

- $BST::search(k)$
 - Start at root, compare k to current node's key
 - * If found stop
 - * If k is greater than current node's key then go right
 - * If k is less than current node's key then go left
- $BST::insert(k, v)$
 - Search for k until arriving at a null child, then insert (k, v) as a new node
- $BST::delete(k)$
 - Search for the node x that contains the key
 - * If x is a leaf then just delete it
 - * If x has one non-empty subtree, move that child up
 - * Else swap key at x with key at *successor* or *predecessor* node and then delete that node

$BST::search$, $BST::insert$, $BST::delete$ all have cost $\Theta(h)$ where $h =$ height of tree. Size of h :

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

AVL Tree

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

$$\text{balance}(v) := \text{height}(R) - \text{height}(L) \text{ must be in } \{-1, 0, 1\}$$

The heights of the left and right subtree differ by at most 1 (height of empty tree is defined to be -1)

If node v has left subtree L and right subtree R then

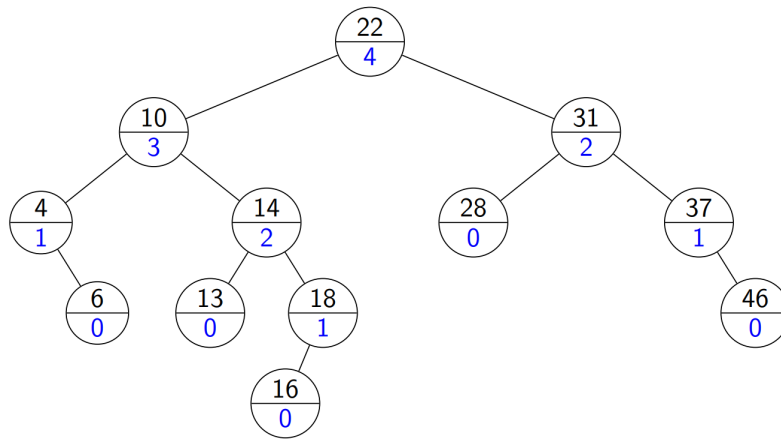
$$\text{balance}(v) := \text{height}(R) - \text{height}(L) \text{ must be in } \{-1, 0, 1\}$$

- $\text{balance}(v) = -1$ means v is *left-heavy*
- $\text{balance}(v) = +1$ means v is *right-heavy*

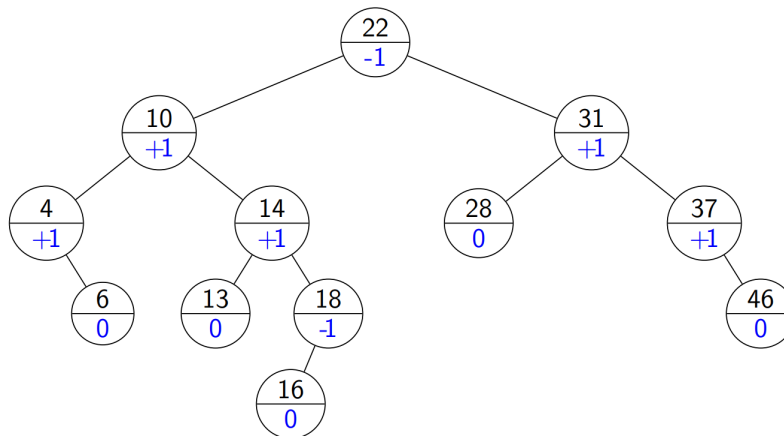
To keep things simple we assume each node v stores the max height of subtrees rooted at it

- It is possible to only store $\text{balance}(v)$ instead but the code gets more complex

Example: the numbers in blue indicate the height of the subtree



Example: storing balance instead of height at each node



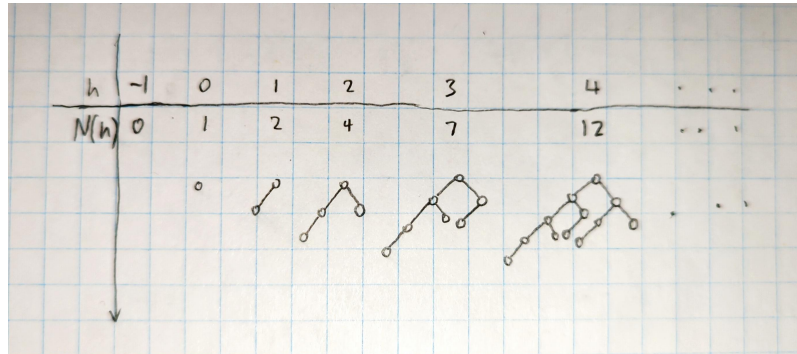
AVL Tree Height

Theorem: an AVL tree on n nodes has $\Theta(\log n)$ height

This directly implies that *search*, *insert*, *delete* all cost $\Theta(\log n)$ in *worst case*

Proof:

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



- Notice that the recurrence relation for $N(h)$ is similar to the Fibonacci sequence

n	0	1	2	3	4	5	6	7	8	...
Fib(n)	0	1	1	2	3	5	8	13	21	...
Fib(n) - 1	-1	0	0	1	2	4	7	12	20	...

- Notice that we build an AVL tree of height h by using AVL trees of heights $h - 1$ and $h - 2$

$$\text{Fib}(\ell) - 1 = \underbrace{\text{Fib}(\ell - 1) - 1}_{\text{left children}} + \underbrace{1}_{\text{root}} + \underbrace{\text{Fib}(\ell - 2) - 1}_{\text{right children}}$$

- Claim: $\sqrt{2}^h - 1 \leq N(h)$

– Base case: $h = 0$ then $\sqrt{2}^0 - 1 = 0 \leq N(0) = 0$

– Inductive step: assume claim holds for $m < h$ where $h \geq 1$ then

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

Thus the claim holds by induction

- Since $n \geq N(h)$ we have

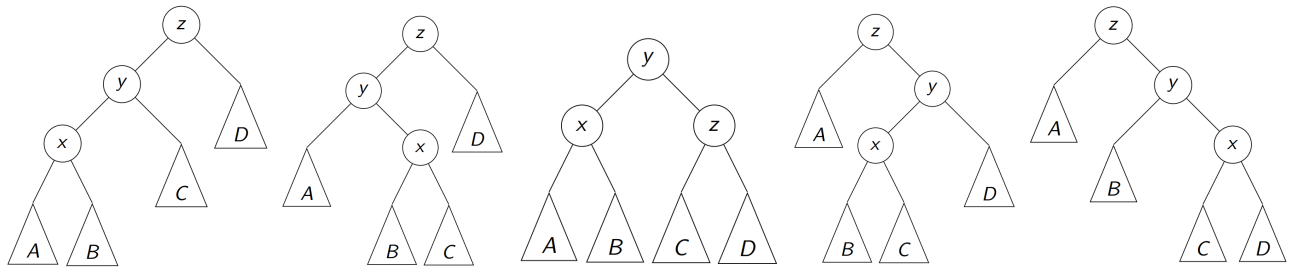
$$\begin{aligned} n \geq N(h) \geq \sqrt{2}^h - 1 &\implies n \geq \sqrt{2}^h - 1 \\ &\implies n + 1 \geq \sqrt{2}^h \\ &\implies \log_{\sqrt{2}}(n + 1) \geq h \\ &\implies h \in O(\log n) \end{aligned}$$

- Not shown in class: I think $N(h+1) \geq n$ allows us to get $h \in \Omega(\log n)$ and conclude that $h \in \Theta(\log n)$

AVL Rebalance

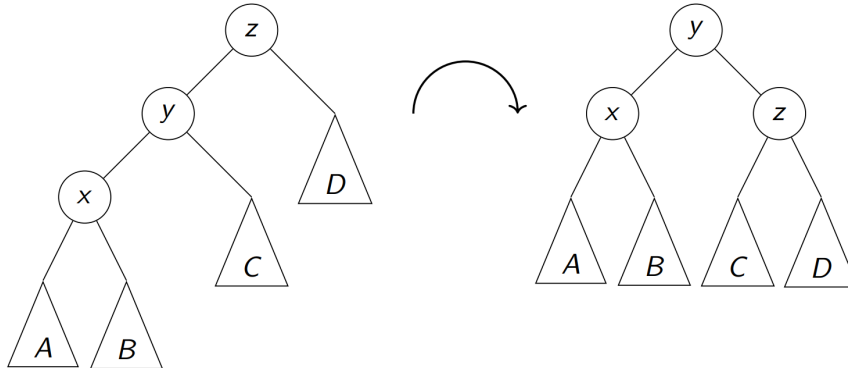
There are many different BSTs with the same keys.

If take A, B, C, D as same height then there are 5 cases for every subtree after BST *insert* or *delete*:



After a BST *insert* or *delete* to fix the *height-balance* property we find the node closest to the modified one that breaks the property and fix that subtree to turn the entire BST back into a AVL tree.

- **Right Rotation:** *right rotation* on node z

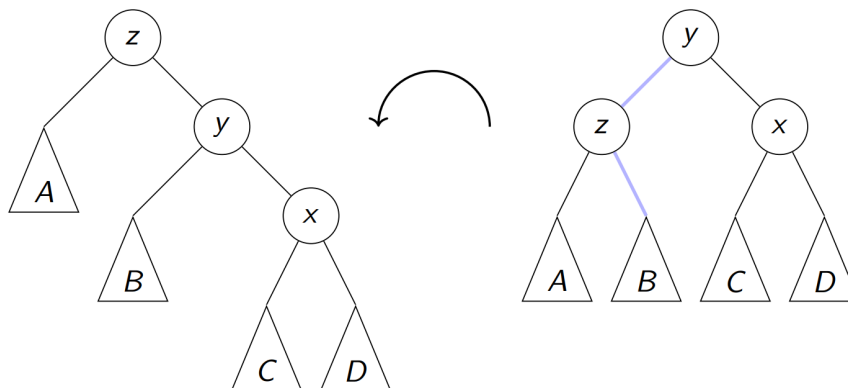


```

1 rotate-right(z)
2   y ← z.left, z.left ← y.right, y.right ← z
3   setHeightFromSubtrees(z), setHeightFromSubtrees(y)
4   return y // returns new root of subtree

```

- **Left Rotation:** *left rotation* on node z

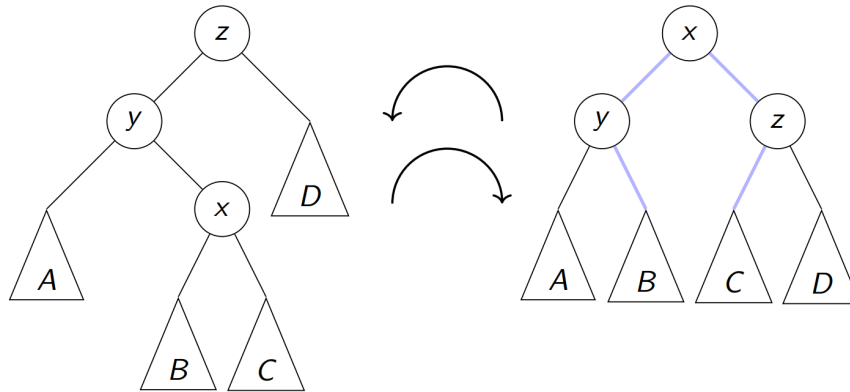


```

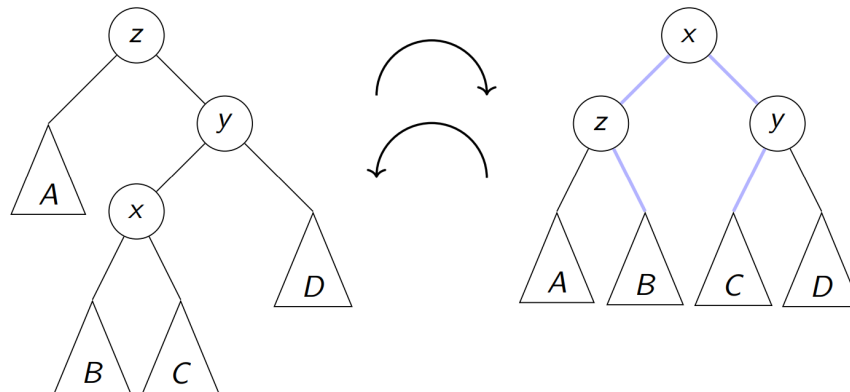
1 rotate-left(z)
2   y ← z.right, z.right ← y.left, y.left ← z
3   setHeightFromSubtrees(z), setHeightFromSubtrees(y)
4   return y // returns new root of subtree

```


- **Double Right Rotation:** *double right rotation* on node z :
 - First, a left rotation at y
 - Second, a right rotation at z



- **Double Left Rotation:** *double left rotation* on node z :
 - First, a right rotation at y
 - Second, a left rotation at z



In summary to fix a slightly-unbalanced AVL tree we have

```

restructure(x, y, z)
node x has parent y and grandparent z
1. case
  z
  y
  x : // Right rotation
      return rotate-right(z)
  z
  y
  x : // Double-right rotation
      z.left ← rotate-left(y)
      return rotate-right(z)
  z
  y
  x : // Double-left rotation
      z.right ← rotate-right(y)
      return rotate-left(z)
  z
  y
  x : // Left rotation
      return rotate-left(z)

```

AVL Insertion

- Insert (k, v) with the usual BST insertion (returns leaf z where the key is stored)
- Then move up the tree from z , updating heights
 - `BST::Insert` could return full path to z or we need parent pointers
- When height difference becomes ± 2 at node z then z is *unbalanced* and we need to rebalance

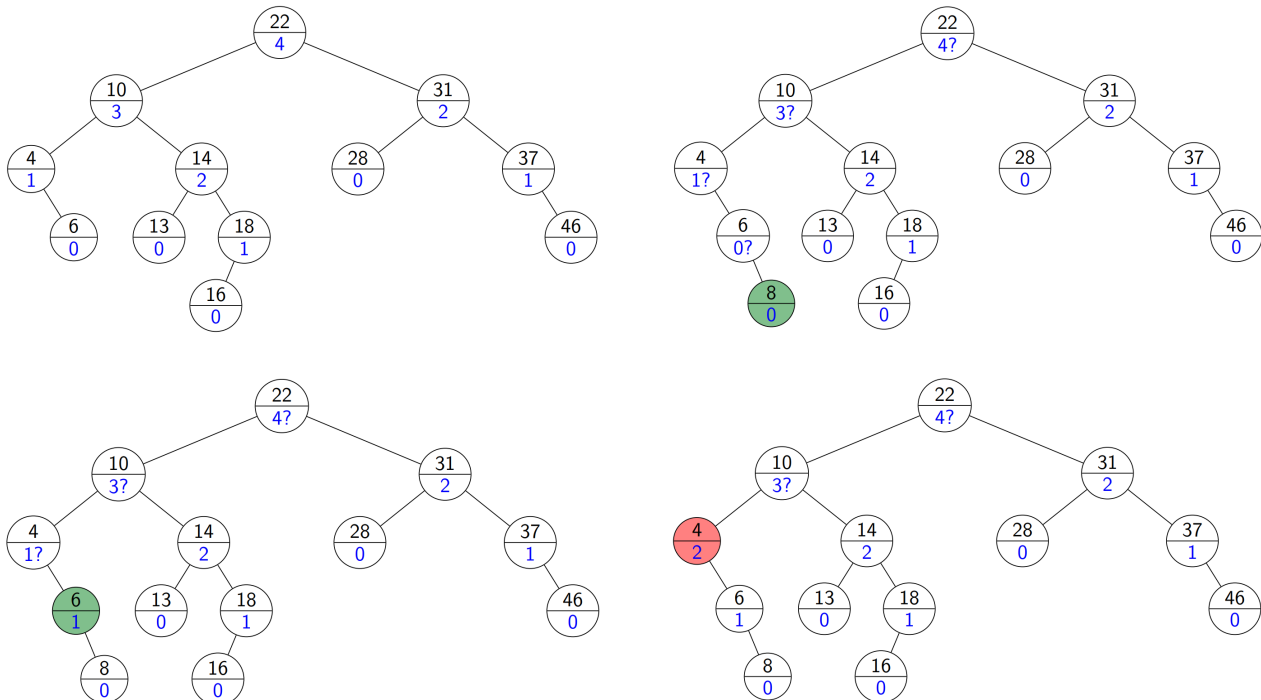
```

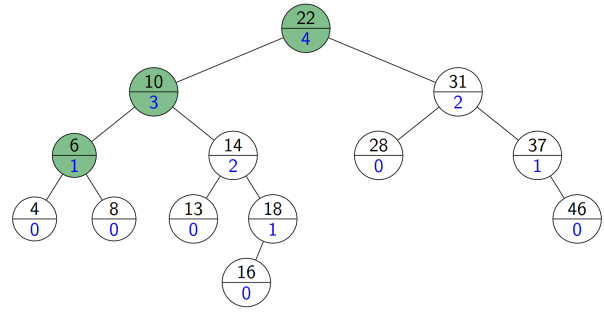
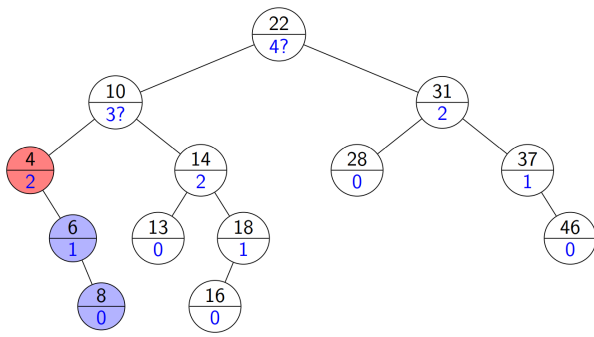
1 setHeightFromSubtrees(u)
2   1. u.height ← 1 + max{u.left.height, u.right.height}

1 AVL::insert(k, v)
2   z ← BST::insert(k, v) // leaf where k is now stored
3   while (z is not NIL)
4     if (|z.left.height - z.right.height| > 1) then
5       Let y be taller child of z
6       Let x be taller child of y
7       z ← restructure(x, y, z)
8       break // can argue that we are done
9     setHeightFromSubtrees(z)
10  z ← z.parent
  
```

After a single restructure the height of the overall tree becomes the same as before the added node, since we had a AVL tree previously.

Example: `AVL::insert(8)`





AVL Deletion

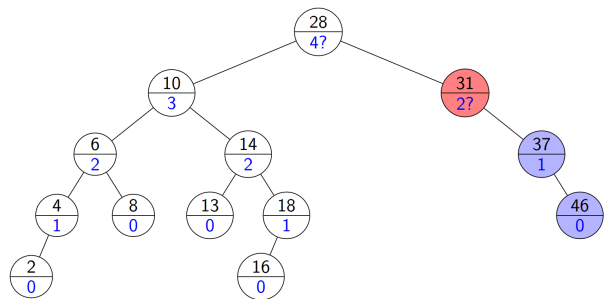
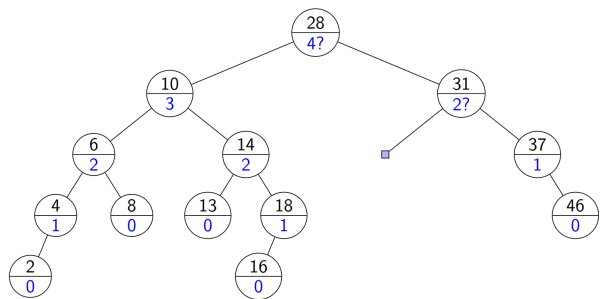
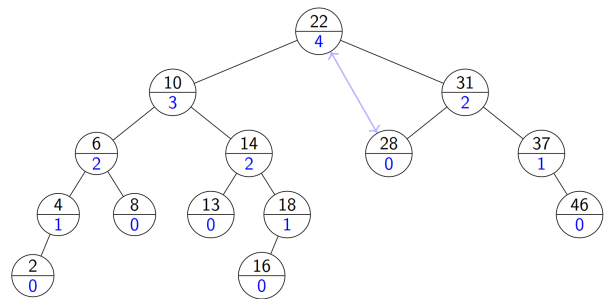
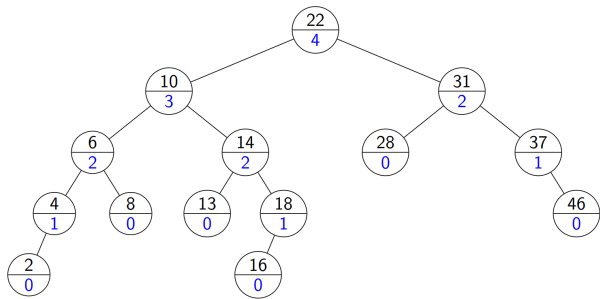
- Remove the key k with $BST::delete$ (returns node z which is parent of deleted node)
- Find the node where *structural* change happened (not necessarily near node that had k)
- Go back up to root updating heights and rotate if needed

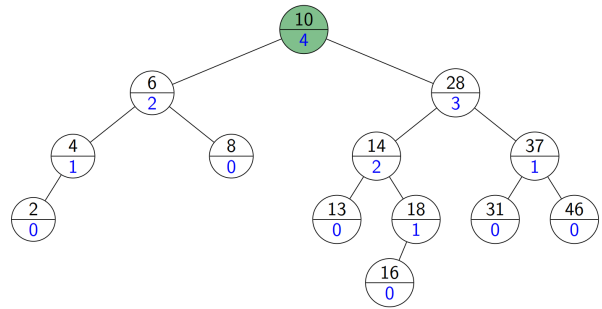
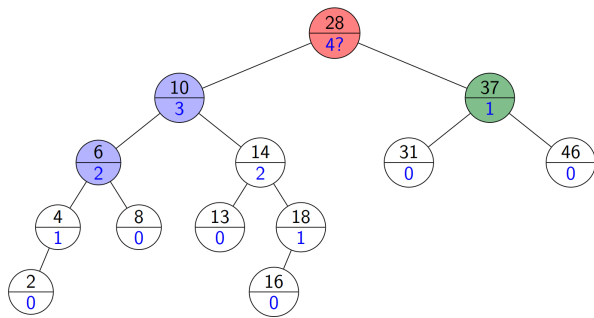
```

1 AVL::delete(k)
2   z ← BST::delete(k)
3   // Assume z is the parent of the BST node that was removed
4   while (z is not NIL)
5     if (|z.left.height - z.right.height| > 1) then
6       Let y be taller child of z
7       Let x be taller child of y (break ties to prefer single rotation)
8       z ← restructure(x, y, z)
9     // Always continue up the path and fix if needed.
10    setHeightFromSubtrees(z)
11    z ← z.parent

```

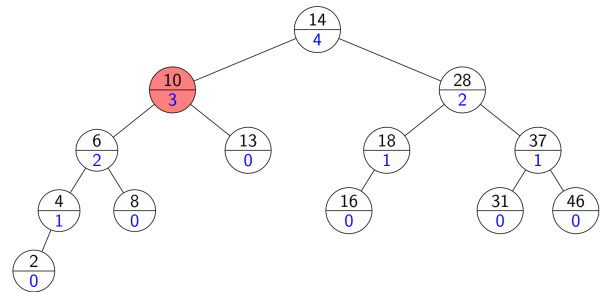
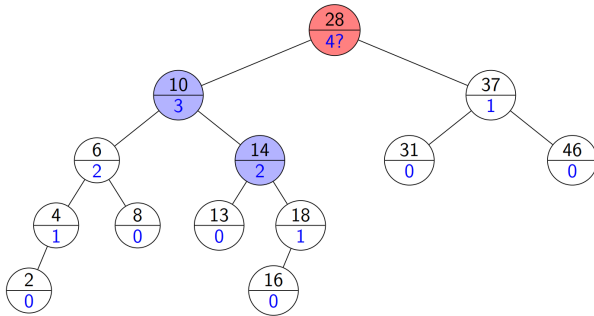
Example: $AVL::delete(22)$





Important: ties *must* be broken with a single rotation

- In the example above the key 10 has two children of the same height
- If we instead chose its left child and performed double-rotation we would get:



The resulting tree is *not* an AVL tree.

AVL Tree Operations Runtime

- *search*: $\Theta(\text{height})$
 - Same operation as BST
- *insert*: $\Theta(\text{height})$
 - Perform *BST::insert* then check along path to new leaf
 - *restructure* restores height of subtree to before inserting new leaf (call *at most once*)
- *delete*: $\Theta(\text{height})$
 - Perform *BST::insert* then check along path to delete node
 - *restructure* may be called $\Theta(\text{height})$ times

Worst-case for all operations is $\Theta(\text{height}) = \Theta(\log n)$ but in practice the constant is quite large due to many rotations.

Amortized Analysis

A common pattern is for some operations is:

- usually fast
- occasionally slow

The worst-case run-time does not reflect how over a long periods such an operation can work quite well.

Definition: an *amortized run-time bound* is a bound that holds if we add the bounds over all operations

$$\sum_{i=1}^k T^{\text{actual}}(\mathcal{O}_i) \leq \sum_{i=1}^k T^{\text{amort}}(\mathcal{O}_i)$$

Where $\mathcal{O}_1, \dots, \mathcal{O}_k$ is any feasible sequence of operations and:

- $T^{\text{actual}}(\cdot)$ is the actual run-time
- $T^{\text{amort}}(\cdot)$ is the amortized run-time (or an upper bound for it)

We have three methods to perform amortized run-time analysis:

$$c_i = T^{\text{actual}}(\mathcal{O}_i) \quad \hat{c}_i = T^{\text{amort}}(\mathcal{O}_i)$$

- *Aggregate Analysis:*

- For n operations we let $\hat{c}_i := T(n)/n$
- We know that $\sum_{i=0}^n c_i \leq T(n)$ and we have

$$\sum_{i=0}^n c_i \leq \sum_{i=1}^n \hat{c}_i = n \frac{T(n)}{n} = T(n)$$

- *Accounting Method:*

- Instead of having the same amortized time for all operations
- Overcharge on the many fast operations to pay for the occasionally slower operation

- *Potential Method* (method we will use):

- Similar to the accounting method but we consider credit as potential energy
- Potential is given by function $\Phi(\cdot)$ which depends on the current state of the data structure
- * We *require* that $\Phi : D \rightarrow \mathbb{R}$ satisfies $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0 \forall i$ then we can define

$$T^{\text{amort}}(\mathcal{O}_i) := T^{\text{actual}}(\mathcal{O}_i) + \Phi(D_i) - \Phi(D_{i-1})$$

- * This is often written as

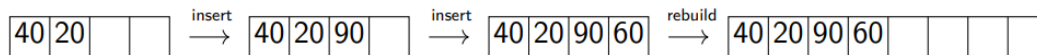
$$T^{\text{amort}}(\mathcal{O}) := T^{\text{actual}}(\mathcal{O}) + \Phi^{\text{after}} - \Phi^{\text{before}}$$

- *Lemma:* this satisfies $\sum_{i=0}^n T^{\text{actual}}(\mathcal{O}_i) \leq \sum_{i=0}^n T^{\text{amort}}(\mathcal{O}_i)$

- *Proof:*

$$\begin{aligned} \sum_{i=0}^n \hat{c}_i &= \sum_{i=0}^n (c_i + \overbrace{\Phi(D_i) - \Phi(D_{i-1})}^{\text{telescoping summation}}) \\ &= \left(\sum_{i=0}^n c_i \right) + \overbrace{\Phi(D_n) - \Phi(D_0)}^{\geq 0 \text{ for upper bound}} \leq \sum_{i=0}^n c_i \end{aligned}$$

Example: potential method analysis of dynamic arrays



- Define potential function $\Phi(i) = \max\{0, 2 \cdot \text{size} - \text{capacity}\}$
- To make analysis easier set time units such that

$$T^{\text{actual}}(\text{insert}) \leq 1 \quad \text{and} \quad T^{\text{actual}}(\text{resize}) \leq n$$

- *insert* increases *size* while not changing *capacity*

$$\Delta\Phi = \Phi^{\text{after}} - \Phi^{\text{before}} \leq 2 - 0 = 2$$

$$T^{\text{amort}}(\text{insert}) = T^{\text{actual}}(\text{insert}) + \Delta\Phi \leq 1 + 2 = 3 \in O(1)$$

- *rebuild* happens only when $\text{size} = \text{capacity} = n$ and sets capacity to $2n$

$$\Phi^{\text{before}} = 2n - n = n \quad \text{and} \quad \Phi^{\text{after}} = 2n - 2n = 0$$

$$T^{\text{amort}}(\text{rebuild}) = T^{\text{actual}}(\text{rebuild}) + \Delta\Phi \leq n + (0 - n) = 0 \in O(1)$$

- As a result the amortized run-time of dynamic arrays is $O(1)$

There is no general recipe to find potential functions but we have some guidelines:

- Study the expensive operation to find what gets smaller when it occurs
 - Dynamic arrays: *rebuild* increases *capacity* so $-\text{capacity}$ gets smaller
- Study the conditions $\Phi(\cdot) \geq 0$ and $\Phi(0) = 0$
 - Dynamic arrays: have $\text{capacity} \leq 2 \cdot \text{size}$ so

$$2 \cdot \text{size} - \text{capacity} \geq 0$$

- We can add a $\max\{0, \dots\}$ so that $\Phi(0) = 0$

- Then compute the amortized time and see if you get good bounds

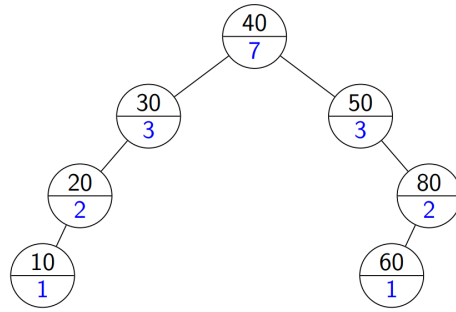
Scapegoat Trees

AVL-trees in practice are quite expensive due to large overhead from rotations, a scapegoat tree is a balanced binary search tree that is maintained *without* rotations.

Idea: instead of performing local changes (rotations) we wait a while before performing a global rebuild on a subtree. Using this we can get $O(\log n)$ height with $O(\log n)$ amortized time for all operations.

Definition: fix α with $\frac{1}{2} < \alpha < 1$ then a scapegoat tree is a BST where every node v with a parent has

$$v.\text{size} \leq \alpha \cdot v.\text{parent.size}$$



Example of a scapegoat tree with $\alpha = 2/3$

- $v.size$ is needed during updates so it must be stored in each node
- Since each subtree is a constant fraction smaller for height h we have

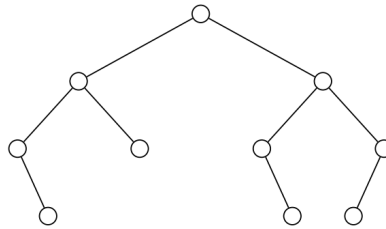
$$1 \leq \alpha^h n \rightarrow \left(\frac{1}{\alpha}\right)^h \leq n \rightarrow h \leq c \log n \rightarrow h \in O(\log n)$$

- Since a leaf is of size 1 and has at most h parents we have $1 \leq \alpha^h n$

Scapegoat tree operations

- *search*: same as a BST, has $O(\text{height}) = O(\log n)$
- *insert* and *delete*: requires occasionally restructuring a subtree into a *perfectly balanced tree*
 - Perfectly balanced trees have for all nodes z

$$|\text{size}(z.\text{left}) - \text{size}(z.\text{right})| \leq 1$$

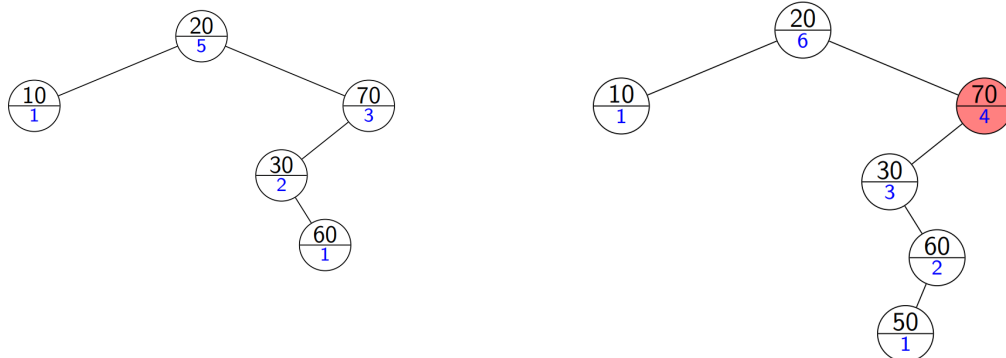


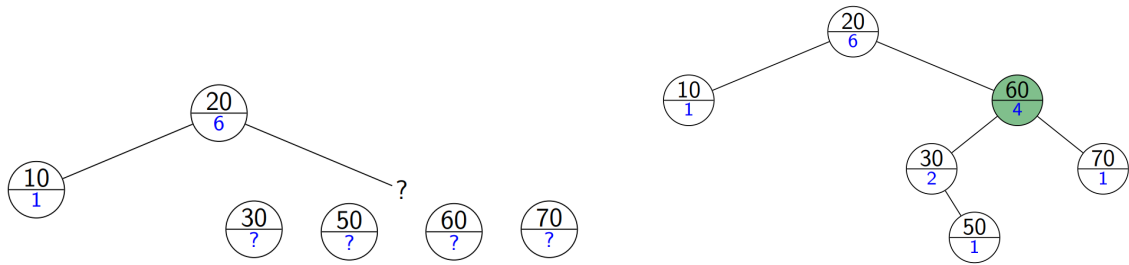
Example of a perfectly balanced tree

- This restructuring is done at the *highest* node that violates the scapegoat tree size-condition

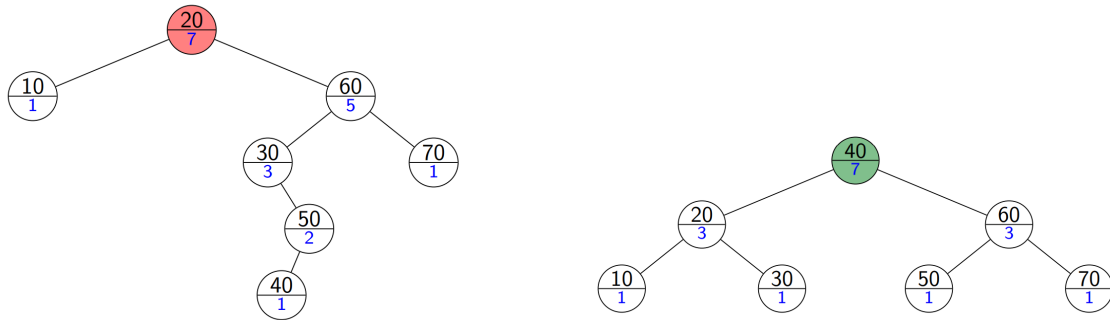
Example:

- `Scapegoat::insert(50)`





- Scapegoat::insert(40)



After we rebalance the subtree the whole tree will respect the size condition because we restructured the highest violating node.

Scapegoat tree insertion algorithm:

```

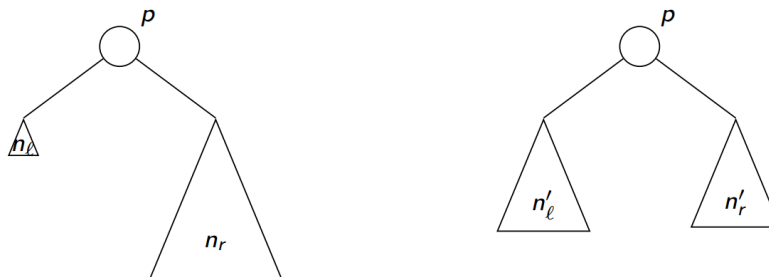
1 scapegoatTree::insert(k, v)
2   z ← BST::insert(k, v)
3   S ← stack initialized with z
4   while (p ← z.parent ≠ NIL)    // update sizes, get path
5     increase p.size
6     S.push(p)
7     z ← p
8   while (S.size ≥ 2)           // size-condition violated?
9     p ← P.pop()
10    if (p.size < α e max{p.left.size, p.right.size})
11      rebuild subtree at p into perfectly balanced tree
12    return

```

Rebuilding at p (line 11) can be done in $O(p.size)$ time by converting the BST to a sorted list then converting that list into a perfectly balanced BST.

Scapegoat Tree Analysis

Relative to the rest of the operations, rebuilding at P is quite expensive:



Claim: if we rebuild at p , then

$$|n_L - n_R| \geq (2\alpha - 1)n + 1$$

Proof: assume wlog $n_R \geq n_L$ then we know that $n = n_L + n_R + 1$ and $n_R > \alpha n$ so

$$\begin{aligned} 2n_R &> 2\alpha n \\ n_R + (n - n_L - 1) &> 2\alpha n \\ n_R - n_L &> (2\alpha - 1)n + 1 \\ |n_L - n_R| &\geq (2\alpha - 1)n + 1 \end{aligned}$$

The potential function should involve $\sum_v |\text{size}(v.\text{left}) - \text{size}(v.\text{right})|$

The goal for the potential function is to amortize the most expensive operation to 0 so we want

$$T^{\text{amort}}(\text{rebuild}) = 0 \leq n + \Delta\Phi$$

So we need $\Delta\Phi = -n$ which can be found by looking at what goes down during the rebuild

- This will not work because there are some nodes without children that get children after rebalance

$$\Phi_1(i) = \sum_v |\text{size}(v.\text{left}) - \text{size}(v.\text{right})|$$

- This will cause possible negative values for the potential function

$$\Phi_2(i) = \sum_v (|\text{size}(v.\text{left}) - \text{size}(v.\text{right})| - 1)$$

- This one fulfills all the requirements

$$\Phi_3(i) = \sum_v \max\{0, |\text{size}(v.\text{left}) - \text{size}(v.\text{right})| - 1\}$$

- Our final potential function will include a $c > 0$ for later adjustments

$$\Phi(i) = c \sum_v \max\{0, |\text{size}(v.\text{left}) - \text{size}(v.\text{right})| - 1\}$$

We have $\Phi(i) \geq 0$ and $\Phi(0) = 0$ so Φ is a potential function

$$T^{\text{amort}}(\mathcal{O}) = T^{\text{actual}}(\mathcal{O}) + \Phi^{\text{after}} - \Phi^{\text{before}}$$

- *search* does not change the data structure so $\Delta\Phi = 0$

$$T^{\text{actual}}(\text{search}) \leq \log n \quad \rightarrow \quad T^{\text{amort}}(\text{search}) \leq \log n$$

- *insert* and *delete* increases contribution at ancestors by at most 1 and does not increase others

$$\Delta\Phi = c \sum_v \Delta\Phi_v \leq c \cdot \underbrace{\#\text{ ancestors of } z}_{O(\log n)} \quad \rightarrow \quad T^{\text{amort}}(\text{insert}) \leq \log n + c \cdot c' \log n \in O(\log n)$$

- *rebuild* decreases contribution at p by $(2\alpha - 1)n + 1$ and does not increase other contributions

$$\Phi = c \sum_v \Phi_v \quad \rightarrow \quad \Delta\Phi = \Phi^{\text{after}} - \Phi^{\text{before}} = c \sum_v (\Phi_v^{\text{after}} - \Phi_v^{\text{before}})$$

There are three cases for v

- Case 1: v is not in the subtree of p

$$\Phi_v^{\text{after}} = \Phi_v^{\text{before}} \quad \rightarrow \quad \Delta\Phi_v = 0$$

- Case 2: v is in the subtree of p , then after rebuilding v perfectly balanced

$$|v.\text{left.size} - v.\text{right.size}| \leq 1 \quad \rightarrow \quad \Phi_v^{\text{after}} = 0 \quad \Phi_v^{\text{before}} \geq 0 \quad \rightarrow \quad \Delta\Phi_v \leq 0$$

- Case 3: v is p

$$\Phi_p^{\text{after}} = 0$$

$$\begin{aligned} \Phi_p^{\text{before}} &= \max\{0, |p.\text{left.size} - p.\text{right.size}| - 1\} \\ &\geq \max\{0, (2\alpha - 1)n_p + 1 - 1\} \\ &\geq (2\alpha - 1)n \end{aligned}$$

$$\Delta\Phi_p \leq -(2\alpha - 1)n$$

Now we have

$$\begin{aligned} T^{\text{amort}}(\text{rebuild}) &= T^{\text{actual}}(\text{rebuild}) + \Delta\Phi \\ &\leq n + c \sum_v \Delta\Phi_v \\ &\leq n + c(-(2\alpha - 1)n) \\ &\leq (1 - c(2\alpha - 1))n \end{aligned}$$

Finally we have

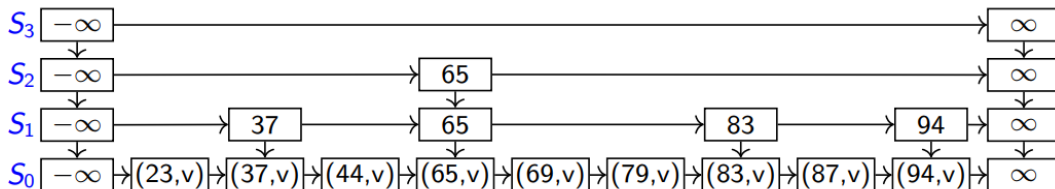
$$c = \frac{1}{2\alpha - 1} \quad \rightarrow \quad T^{\text{amort}}(\text{rebuild}) \leq 0$$

Thus the amortized cost of rebuild is free

We have an amortized run-time for search, insert, and delete being in $O(\log n)$

Skip Lists

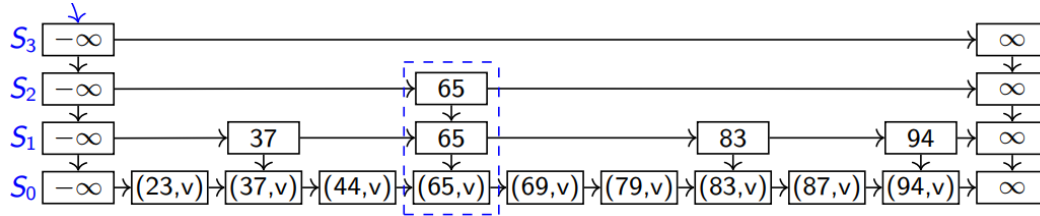
Definition: skip lists are a hierarchy S of ordered linked lists *levels* S_0, \dots, S_h



Example of a skip list

- Each list S_i contains special keys $-\infty$ and $+\infty$ (sentinels)

- Only S_0 contains KVPs and they are ordered in non-decreasing order
 - Other lists stores only keys, or links to nodes in S_0
- Each list is a subsequence of the previous one, i.e. $S_0 \subseteq \dots \subseteq S_h$
- List S_h contains only sentinels, the left of which is the *root* (top left blue arrow)



- Each KVP belongs to a *tower* of nodes (blue dashed box)
- Each node p has references to $p.after$ and $p.below$

Search in Skip Lists

search, *insert*, *delete* all use the `getPredecessors(k)` method gets a list of nodes leading up the node that is right *before* where k would be

```

1 getPredecessors (k)
2   p ← root
3   P ← stack of nodes, initially containing p
4   while p.below != NIL do
5     p ← p.below
6     while p.after.key < k do p ← p.after
7     P.push(p)
8   return P

```

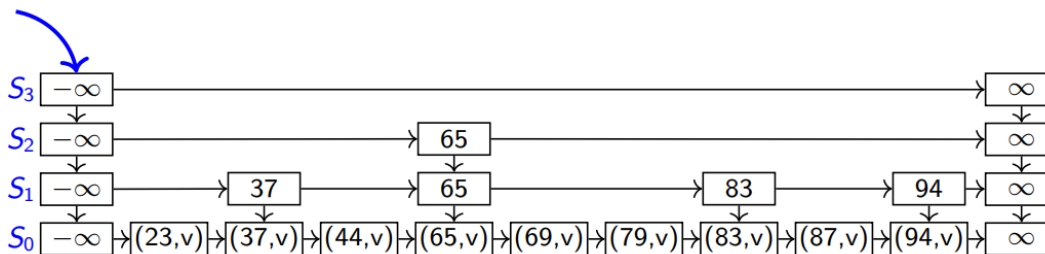
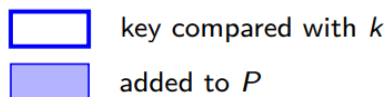
To search for a specific item in the skip list we find the predecessor node then check the node after.

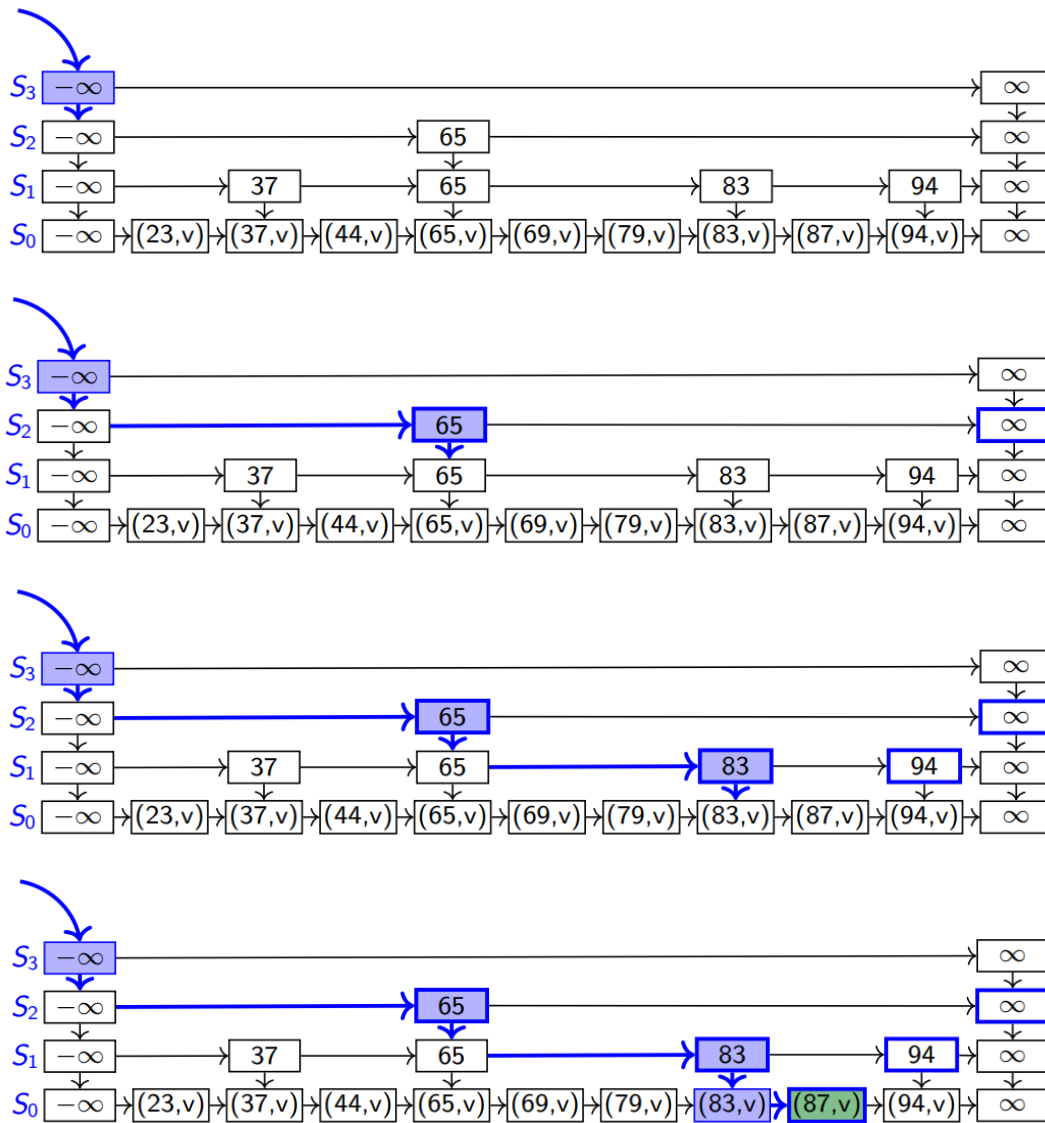
```

1 skipList::search(k)
2   P ← getPredecessors(k)
3   p0 ← P.top() // predecessor of k in S0
4   if p0.after.key = k return p0.after
5   else return "not found, but would be after p0"

```

Example: `search(87)`





Insert in Skip Lists

`skipList::insert(k,v)`

- Randomly repeatedly toss a coin until you get a tails
 - Negative binomial distribution on number of heads
- Let i be the number of heads
 - Then the key k should be in the lists S_0, \dots, S_i
 - This gives k a tower with height i the probability

$$P(\text{tower of key } k \text{ has height } \geq i) = \left(\frac{1}{2}\right)^i$$

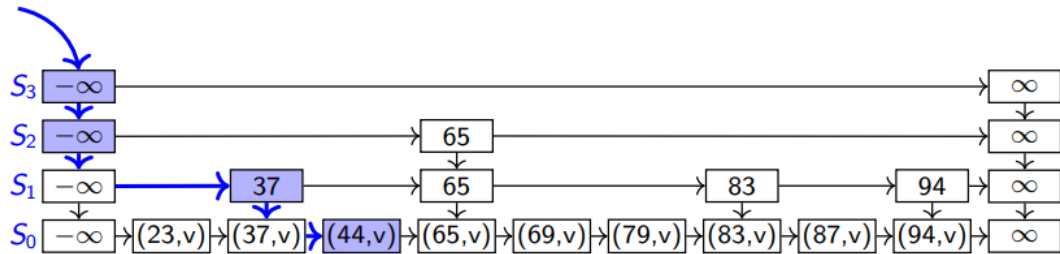
- As needed we also increase the height h of the skip list, such that $h > i$
- Use `getPredecessors(k)` to get stack P
 - The top i items of P are the predecessors p_0, \dots, p_i of where k should be in each list S_0, \dots, S_i

- Finally insert (k, v) after p_0 in S_0 , and k after p_j in S_j for $1 \leq j \leq i$

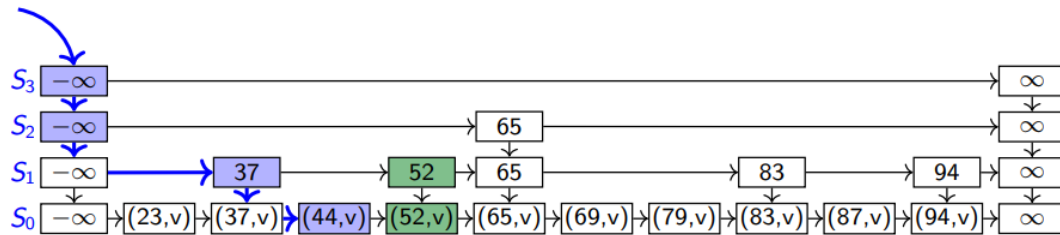
Example:

- `skiplist::insert(52, v)`

– Coin tosses: $H, T \implies i = 1$ then we call `getPredecessors(52)`

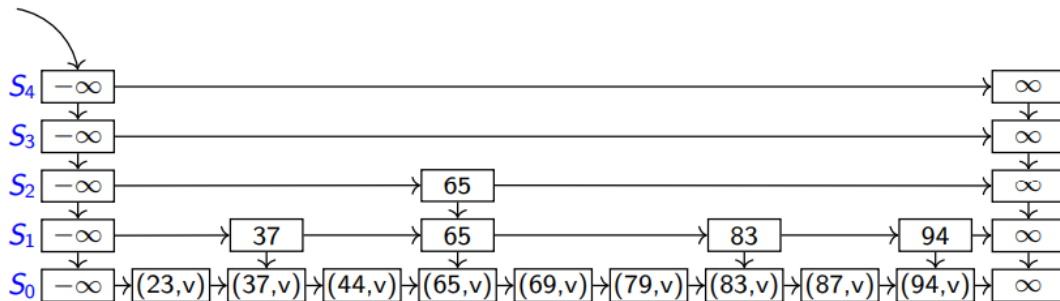


– Now create a tower of height 1 for the key 52

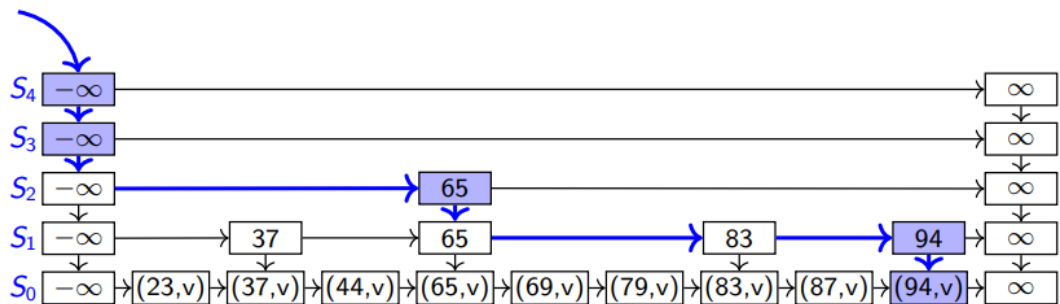


- `skiplist::insert(100, v)`

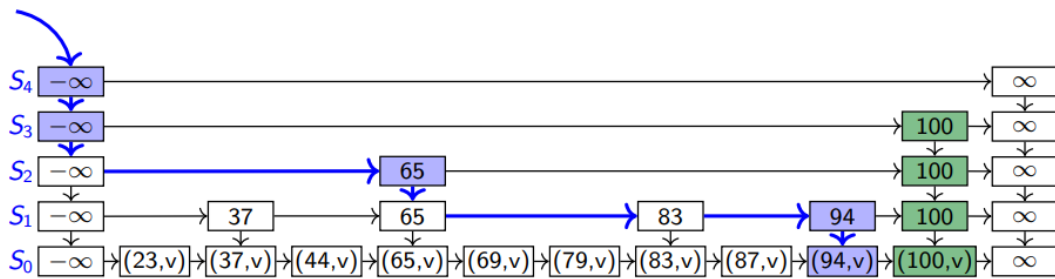
– Coin tosses: $H, H, H, T \implies i = 3$ then we perform a height increase



– Make a call to `getPredecessors(100)`



– Create a tower of height 3 for the key 100



```

1 skipList::insert(k, v)
2   P ← getPredecessors(k)
3   for (i ← 0; random(2) = 1; i ← i+1) {} // random tower height
4   while i ≥ P.size() // increase skip-list height?
5     root ← new sentinel-only list, linked in appropriately
6     add left sentinel of root at bottom of stack P
7   p ← P.pop() // insert (k, v) in S_0
8   z_below ← new node with (k, v), inserted after p
9   while i > 0 // insert k in S_1, . . . , S_i
10    p ← P.pop()
11    z ← new node with k added after p
12    z.below ← z_below; z_below ← z
13    i ← i - 1

```

Delete in Skip Lists

It is easy to remove a key since we have `getPredecessors`. Then eliminate extra layers if there are multiple ones with only sentinels.

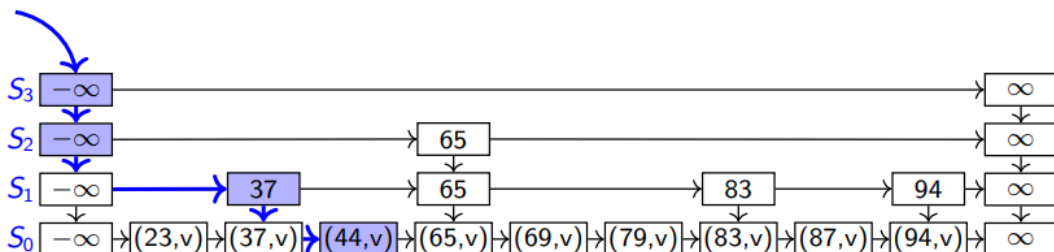
```

1 skipList::delete(k)
2   P ← getPredecessors(k)
3   while P is non-empty
4     p ← P.pop() // predecessor of k in some layer
5     if p.after.key = k
6       p.after ← p.after.after
7     else break // no more copies of k
8
9   p ← left sentinel of the root-list
10  while p.below.after is the ∞-sentinel
11    // the two top lists are both only sentinels, remove one
12    p.below ← p.below.below
13    p.after.below ← p.after.below.below

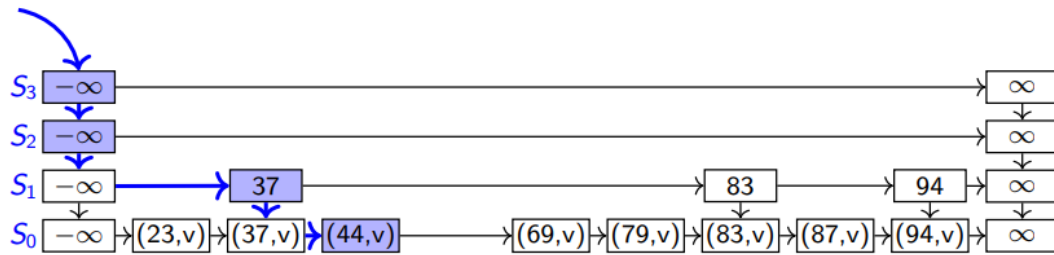
```

Example: `skipList::delete(65)`

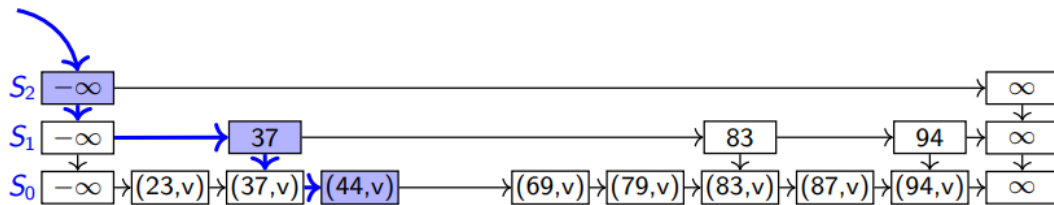
- Make a call to `getPredecessors(65)`



- Delete the key and its tower



- Perform a height decrease



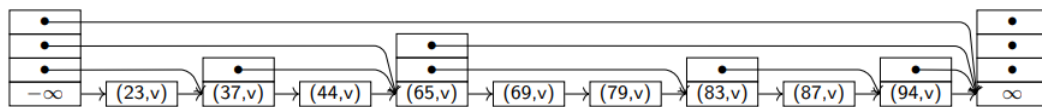
Analysis of Skip Lists

- Expected *space*: $O(n)$
- Expected *height*: $O(\log n)$
- For all operations:
 - How often do we *drop down* (execute $p \leftarrow p.\text{below}$)
 - How often do we *step forward* (execute $p \leftarrow p.\text{after}$)
- `skiplist::search`: $O(\log n)$ expected time
 - # drop-downs = height
 - expected # forward-steps is ≤ 1 in each level
 - expected total # forward-steps is in $O(\log n)$
- `skiplist::insert`: $O(\log n)$ expected time
- `skiplist::delete`: $O(\log n)$ expected time

We have $O(n)$ expected space and all operations take $O(\log n)$ expected time.

Remarks:

- We can show that a biased coin-flip to determine tower-height gives smaller expected run-times
- Can save links (hence space) by implementing towers as an array



- Then skip lists are simple to implement. They are also fast with good cache locality but can still suffer from cache misses.

Re-ordering Items

Recall using an unordered list to implement ADT dictionary. While lists are simple and popular we cannot make them any more effective in practice, *unless*, we have some knowledge of the item access probabilities.

For short lists with extremely unbalanced distributions this can be much faster than AVL trees or Skip Lists and much easier to implement.

Static Ordering

For *static orderings* sorting items by non-increasing access-probabilities minimizes expected access cost. (proof idea: for any other ordering, exchanging two out of order items makes total cost decrease)

Example: *static ordering* with full knowledge of access probabilities

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access probability	2/26	8/26	1/26	10/26	5/26

For accessing the key in the i th position we count the cost as $i + 1$

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

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

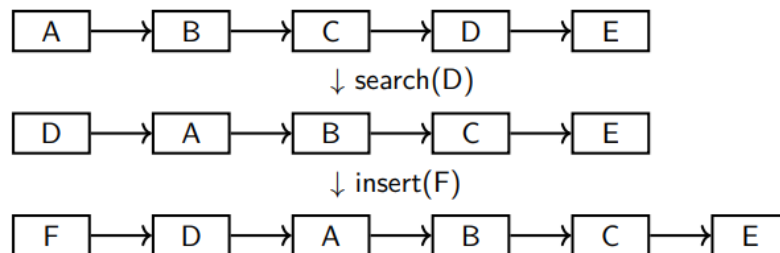
- Order D, B, E, A, C has expected access cost of

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

Dynamic Ordering

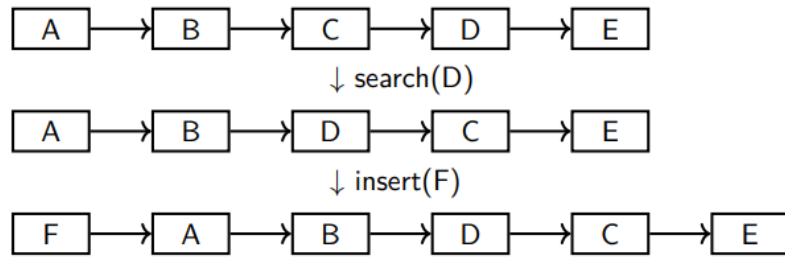
If we do not know the access probabilities ahead of time we still have the *temporal locality* heuristic which says that the most recently accessed items is likely to be used soon again.

- **Move-To-Front** (MTF) heuristic: upon a successful search, move the accessed item to front



- MTF can be done with an array, but, insert and search should start from the *back* so we have room to grow

- **Transpose** heuristic: upon successful search, swap the accessed item with the item immediately preceding it



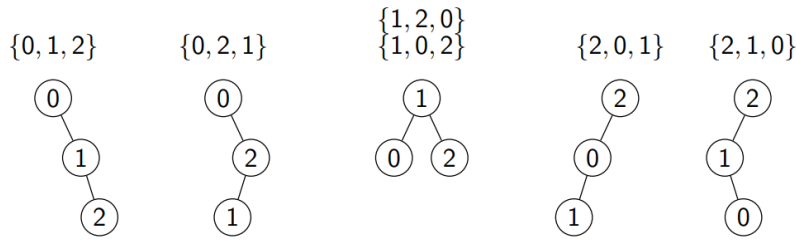
– Transpose does not adapt quickly to changing access patterns

In practice MTF works pretty well and it can be shown that it is *2-competitive* (not more than twice as bad as the optimal static ordering)

More on Dictionaries

Expected height of BST

Assume we *randomly* choose a permutation of $\{0, \dots, n-1\}$ and build a BST in this order:



Theorem: expected height of a BST is $O(\log n)$

Proof: (NEEDS WORK) Let π be the root node and $H(\pi)$ the height of the tree

$$H(\pi) = 1 + \max\{H(\pi_L), H(\pi_R)\}$$

We also let

$$y(\pi) = 2^{H(\pi)} \quad \rightarrow \quad y(n) = E[y(\pi)]$$

Then since log is concave we have

$$H(n) = E[H(\pi)] = E[\log(y(\pi))] \leq \log[E(y(\pi))]$$

To complete this we just show that $E[y(\pi)] \leq (n+1)^3$ which can be done by proof by induction

This does *not* imply that the average height of a BST is $O(\log n)$

- The average height is $\Theta(\sqrt{n})$ (no details)
- Average height (over all BST) \neq expected height (over all randomly built BST)
- This difference is most obvious for $n = 3$
 - Expected height over 6 possible permutations is

$$\frac{1}{6}(2 + 2 + 1 + 1 + 2 + 2) \approx 1.66$$

– Average height over 5 possible BST is

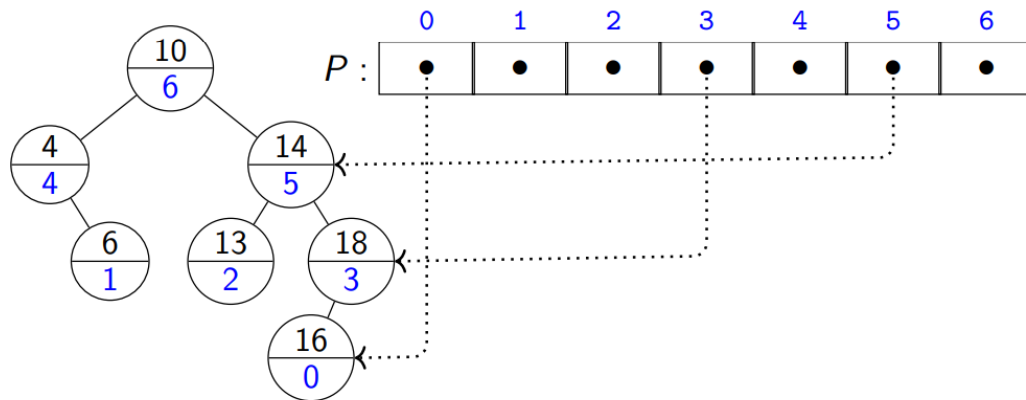
$$\frac{1}{5}(2 + 2 + 1 + 2 + 2) \approx 1.8$$

Randomization does *not* automatically imply an average-case bound, it depends on what we average over and how we randomize.

Treap

To build a binary search tree that acts if it had been built in random insertion order we use a BST but also store a priority with each node.

- Priorities are a permutation of $\{0, \dots, n - 1\}$
- The permutation should be picked *randomly* and all permutations should be *equally likely*
- Apply heap order property to priorities (*decreasing* downwards)
- The treap (= tree + heap) can be stored in array P where $P[i]$ stores the node with priority i



A treap is a BST with respect to the KVP and a max heap with respect to the priorities.

The treap is the BST that results from inserting KVP $p = n - 1$, then KVP $p = n - 2$, etc while maintaining BST order on the KVP (this gives us heap property on priorities due to order of insert).

Theorem: the expected height of a treap is $O(\log n)$

Proof: root-item has priority $n - 1$, the rest of the priorities were picked randomly so proof of expected height of the BST applies

Treap Operations

Inserting a KVP into the treap we need to consider what priority to give it.

- We *have* a random permutation of $\{0, \dots, n - 2\}$ and *want* a random permutation of $\{0, \dots, n - 1\}$
 - recall *shuffle* (inside-out-shuffle)

```

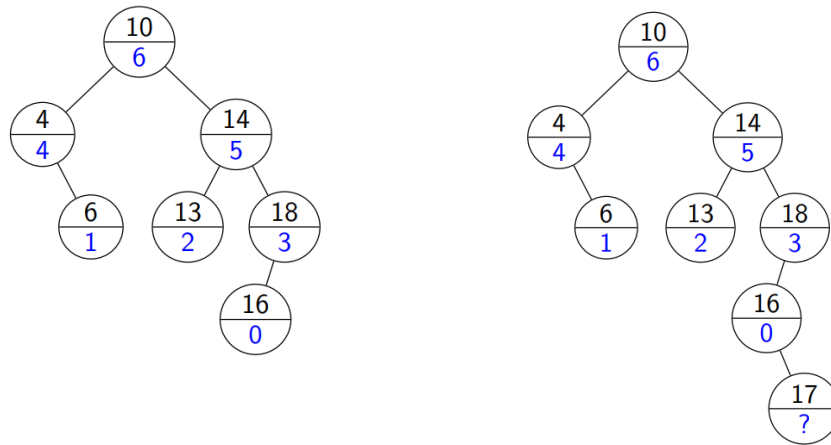
1 // A: array of size n stores (0, ..., n-1)
2 shuffle(A)
3   for i ← 1 to n-1 do
4     swap(A[i], A[random(i + 1)])

```

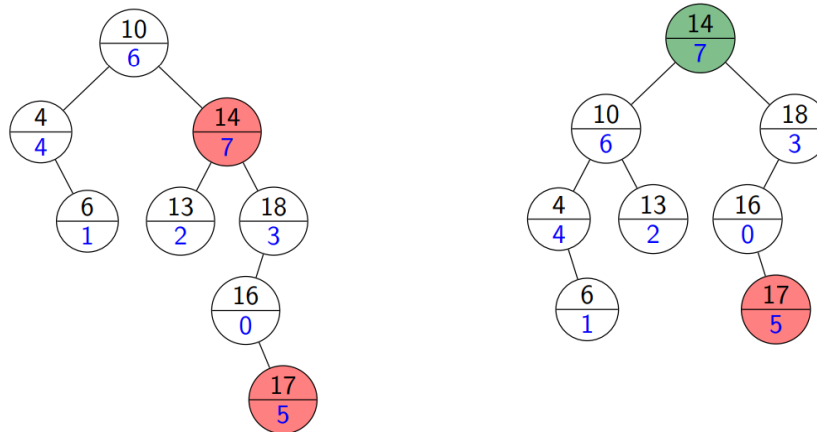
* After iteration $n - 2$ we have random permutation of $\{0, \dots, n - 2\}$

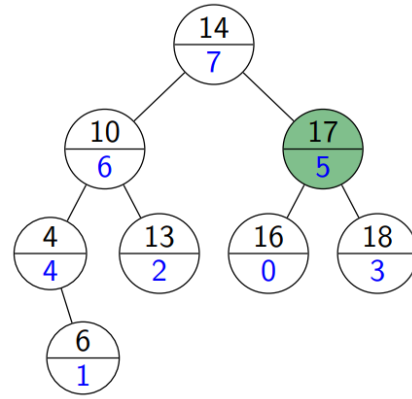
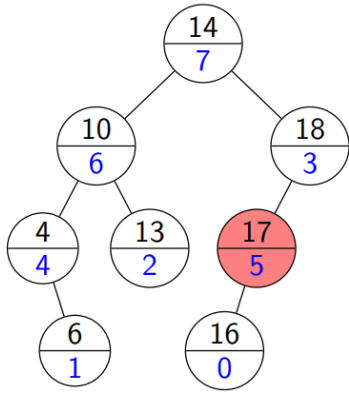
- * After iteration $n - 1$ we have random permutation of $\{0, \dots, n - 1\}$
- * Idea: swap new item with a randomly item to make a random permutation one larger
- Begin by inserting the new item as if the treap was a BST
- Pick a *random* priority p (where $0 \leq p \leq n - 1$) for the new item
 - The item that had priority p now has priority $n - 1$
 - Perform fix-up with rotations to bubble the priorities upwards
 - * We fix the higher node first so lower node can assume proper treap structure above

Example: `treap::insert(17)`



Randomly pick priority $5 \in \{0, \dots, 7\}$





```

1 treap::insert(k, v)
2   n ← P.size // current size
3   z ← BST::insert(k, v); n++
4   p ← random(n)
5   if p < n-1 do
6     z' ← P[p], z'.priority ← n-1, P[n-1] ← z'
7     fixUpWithRotations(z')
8   z.priority ← p; P[p] ← z
9   fixUpWithRotations(z)

```

```

1 treap::fixUpWithRotations(z)
2   while (y ← z.parent is not NIL and z.priority > y.priority) do
3     if z is the left child of y do rotate-right(y)
4     else rotate-left(y)

```

- Since the result is a randomized BST the expected height is $O(\log n)$
- This implies $O(\log n)$ expected time for *search* and *insert*
- *delete* can be handled in a similar manner we have seen but needs more exchanges
- Large space overhead due to parent-pointers, priorities, P itself
 - Extra space overhead is pretty worth if we can get much faster operations
 - There are methods to avoid some of the space overhead

This is not particularly efficient in practice for the things we are currently considering (later we will give priorities have meaning → cartesian trees for range searches)

Optimal Static BST

- Static: the access frequencies are known (offline algorithm)
- Dynamic: access frequencies are inferred from usage (online algorithm)

When finding the optimal static order it is natural to use a greedy algorithm.

- Put item with highest access probability at the root
- Then insert insert tree in order of access probabilities (from highest to lowest)
 - Keys need to respect the BST order-property

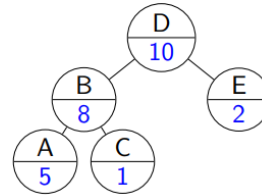
This greedy-algorithm is pretty good but does *not* actually give the optimum!

- To actually find the optimum we need to effectively try *all* possible BST
- Native method would take exponential time so we speed up using *dynamic programming*
- Idea: store and re-use solutions of subproblems to achieve polynomial run-time

More details in CS 341 (for dynamic programming).

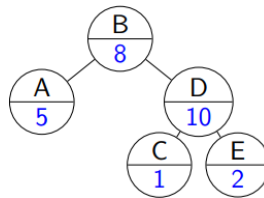
Example:

k_i	A	B	C	D	E
$P(k_i)$	5/26	8/26	1/26	10/26	2/26



$$\text{Access-cost} = \sum_k P(k) \cdot (1 + \text{depth of } k) = 1 \cdot \frac{10}{26} + 2 \cdot \frac{8}{26} + 2 \cdot \frac{2}{26} + 3 \cdot \frac{5}{26} + 3 \cdot \frac{1}{26} = \frac{48}{26}$$

The optimal solution for this is:



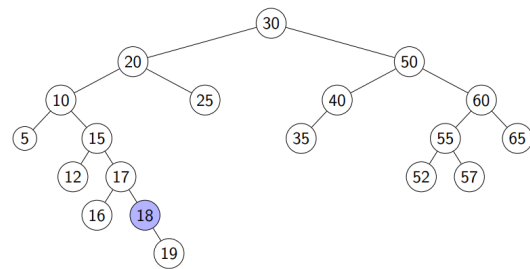
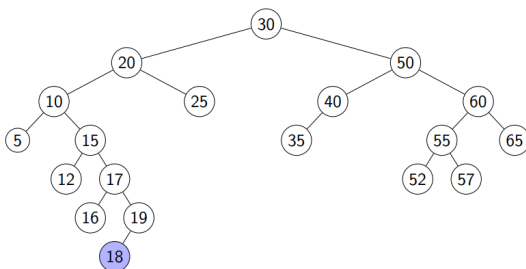
$$\text{Access-cost} = \sum_k P(k) \cdot (1 + \text{depth of } k) = 1 \cdot \frac{8}{26} + 2 \cdot \frac{5}{26} + 2 \cdot \frac{10}{26} + 3 \cdot \frac{1}{26} + 3 \cdot \frac{2}{26} = \frac{47}{26}$$

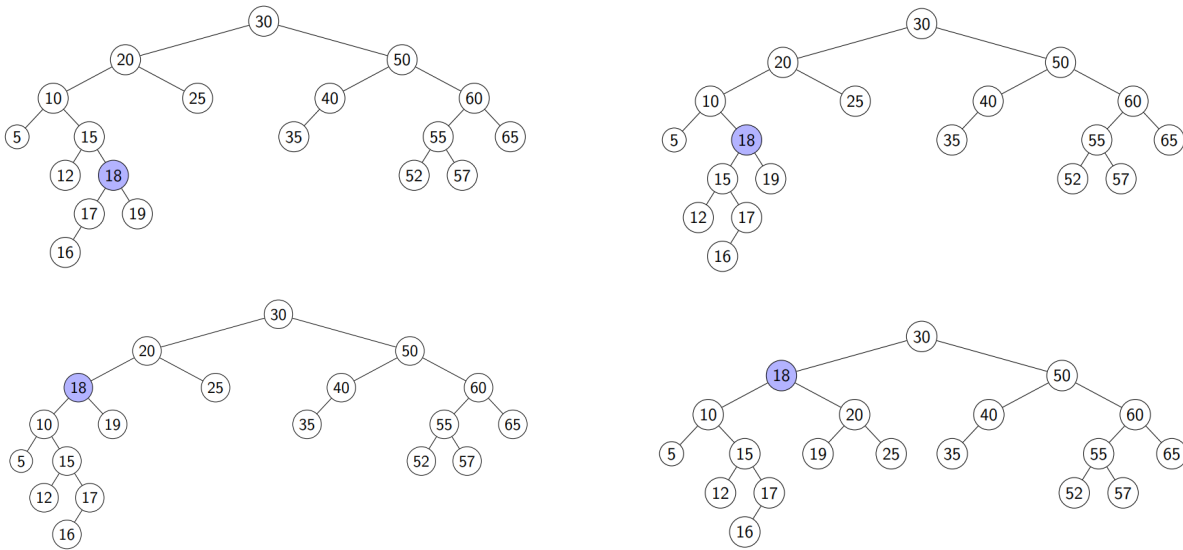
MTF-Heuristic for a BST

In the move-to-front heuristic the front is the place that is easiest to access which in a BST is the root.

So applying MTF-heuristic to a BST, we use rotations to bring the accessed node to the top.

Example: BST-MTF::search(18)





This should work well, but we can do better by moving it up two levels at a time.

This is actually really bad because there are cases when we end up needing to rotate the entire tree.

Example: BST of 70, 60, 50, 40 (inserted in that order) then insert 40

- MTF would require us to rotate tree until 40 is root (rotates over the entire tree)
- The resulting height will be one more than previous

The issue is that our rotations have too much *symmetry* so our height stays the same after each rotate.

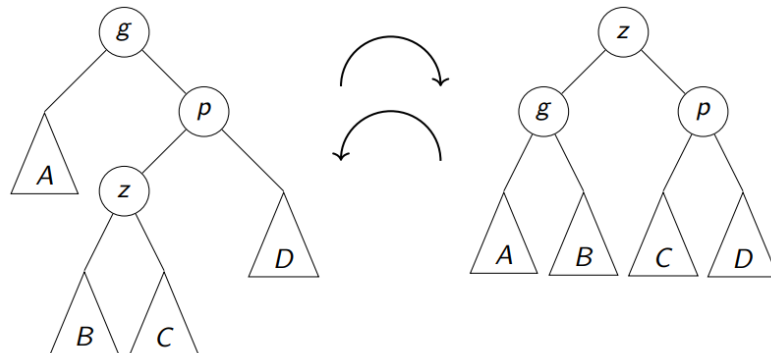
Splay tree

A BST with no extra information (e.g. height, balance, size) at nodes with amortized run-time $O(\log n)$

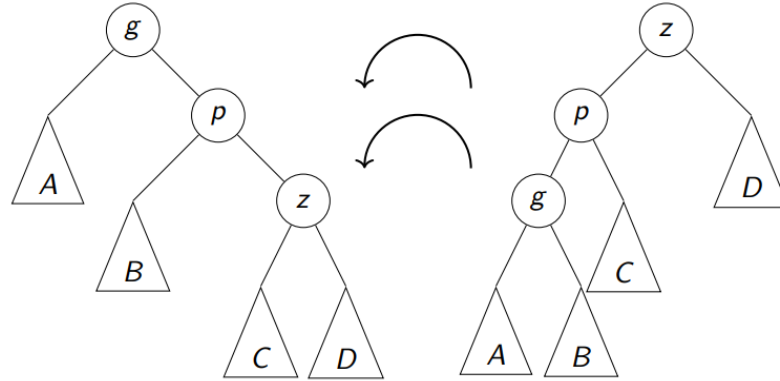
- After search/insert, bring accessed node to the root with rotations
- Move node up two layers at a time (except when near root)
 - Use zig-zig-rotation or zig-zag-rotation to move up two levels

Let z be the node that we want to move up and p and g be parent and grandparent.



- *Zig-zag rotation:* if they are in zig-zag formation, apply a double-rotation



- *Zig-zig rotation*: if they are in zig-zig formation, apply left rotation at g then left rotation at p



```

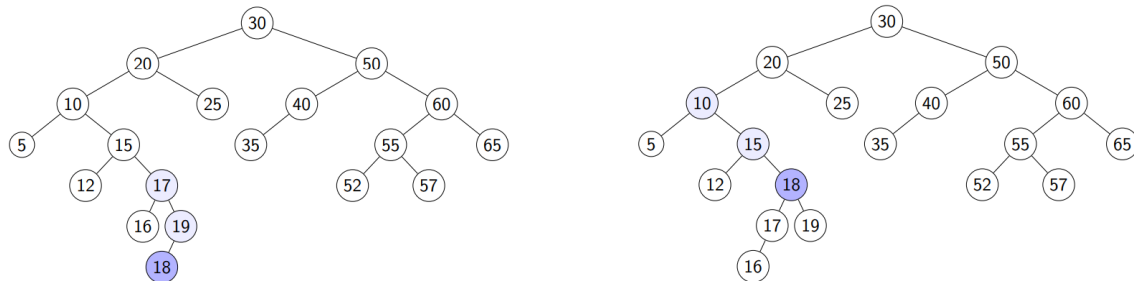
SplayTree::insert(k, v)
1.  z ← BST::insert(k, v)
2.  while (z is not the root)
3.    p ← z.parent
4.    if (z is the left child of p)
5.      if (p is the root) rotate-right(p)
6.      else g ← p.parent
7.      case  : // Zig-zig rotation
           rotate-right(g)
           rotate-right(p)
8.      case  : // Zig-zag rotation
           rotate-right(p)
           rotate-left(g)
9.    else ... // symmetric case, z is right child

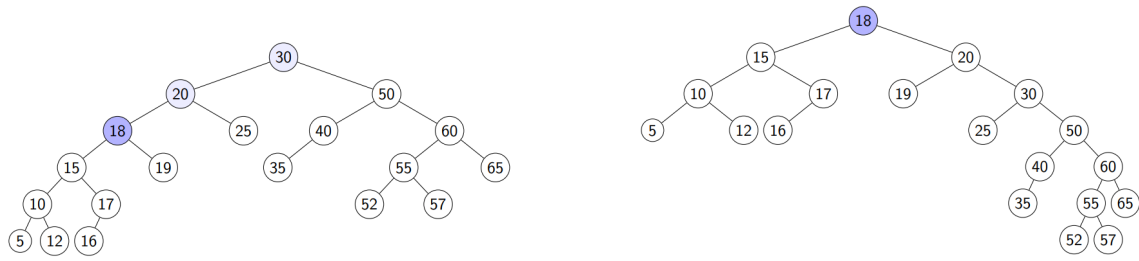
```

search and *delete* use the *BST*-method then rotate the lowest visited node up.

- This code assumed parent-references which can be avoided by storing search-path

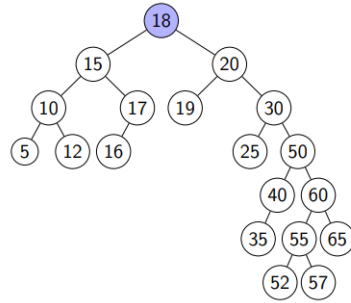
Example: *SplayTree::search*(18)



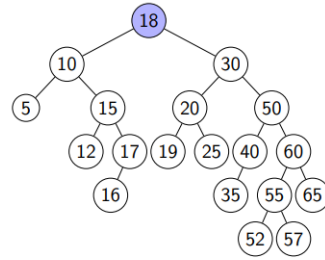


You may notice that this result is actually worse than when using single rotations:

With zig-zig rotations:

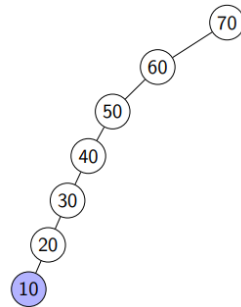


With single rotations:

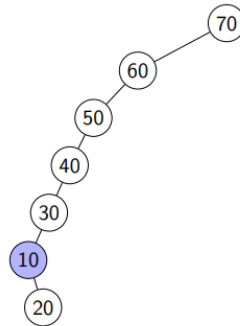
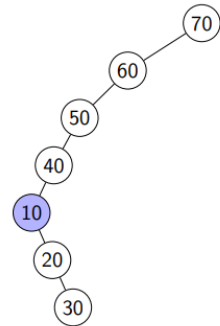
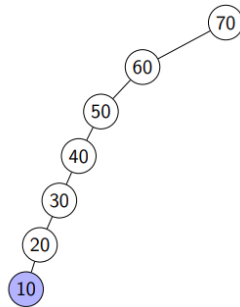


Example: however we will see where it really works when when use a different initial tree

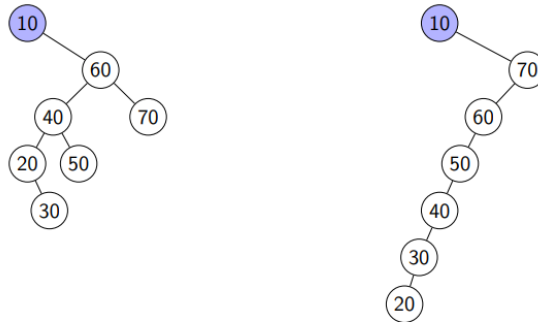
With zig-zig rotations:



With single rotations:



...



Splay tree intuition:

- For any node on the search-path, the depth (roughly) halves
- For all nodes, depth increases by at most 2

Theorem: in a splay tree, all operations take $O(\log n)$ amortized time

Proof: (see textbook) (formal proof does not follow intuition and uses a potential function)

Dictionaries for Special Keys

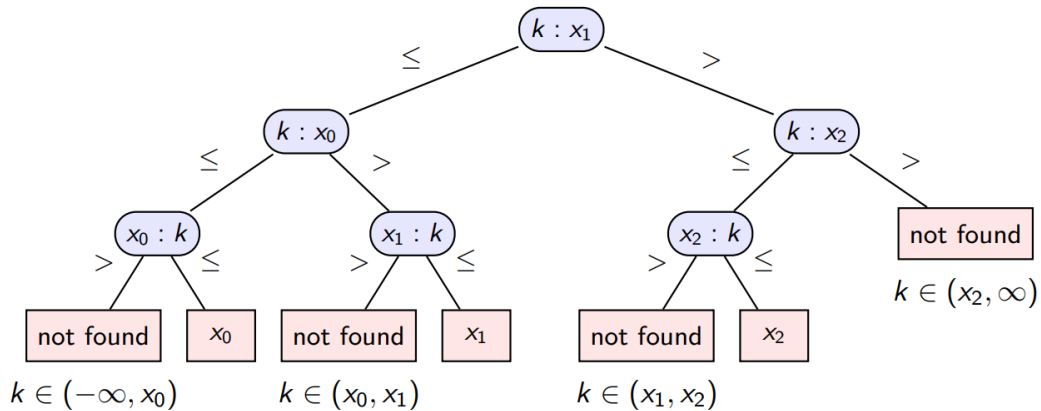
Search

Lower Bound for Search

The fastest realization of *ADT Dictionary* requires $\Theta(\log n)$ time to search among n items.

Theorem: Under the comparison model, $\Omega(\log n)$ comparisons are required to search a size n dictionary

Proof: via decision tree for items x_0, \dots, x_{n-1}



k can be found at the n items x_0, \dots, x_{n-1} or in the $n + 1$ cases when k is between items x_i and x_j . Thus all together the number of leaves is bounded from below by $2n + 1$ so:

$$n \leq 2n + 1 \leq \# \text{ of leaves} \leq 2^h \rightarrow \log n \leq h \rightarrow h \in \Omega(\log n)$$

However recall from sorting that we may be able to beat this lower bound if we had certain properties.

Binary Search

Recall that the binary search algorithm and its run-times and its runtime of $\Theta(\log n)$ on a sorted array

```
1 // A: Sorted array of size n, k: key
2 binary-search(A, n, k)
3    $\ell \leftarrow 0, r \leftarrow n-1$ 
4   while ( $\ell \leq r$ )
5      $m \leftarrow \lfloor (\ell+r)/2 \rfloor$ 
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   return "not found, but would be between A[ $\ell-1$ ] and A[ $\ell$ ]"
```

Interpolation Search

Binary search does not assume anything about the keys are distributed between the max and min keys, however, if we can say that they can uniformly distributed then *interpolation search* can perform better.

Interpolation search performs a linear interpolation using the lower and upper key values then finds a index would give k if there is a linear relationship between index and key.

left index: ℓ right index: r search key: k key at i : $A[i]$

binary search: $\ell + \lfloor \frac{1}{2}(r - \ell) \rfloor$ interpolation search: $\ell + \lfloor \frac{k-A[\ell]}{A[r]-A[\ell]}(r - \ell) \rfloor$

- Binary search just takes the middle index value
- Interpolation search finds how far k is between $A[\ell]$ and $A[r]$ and converts that to an index value

```
1 // A: Sorted array of size n, k: key being search for
2 interpolation-search(A, n, k)
3    $\ell \leftarrow 0, r \leftarrow n - 1$ 
4   while ( $\ell \leq r$ )
5     if ( $k < A[\ell]$  or  $k > A[r]$ ) return "not found"
6     if ( $k == A[r]$ ) then return "found at A[r]"
7      $m \leftarrow \ell + \lfloor (r - \ell)((k-A[\ell])/(A[r]-A[\ell])) \rfloor$ 
8     if ( $A[m] == k$ ) then return "found at A[m]"
9     else if ( $A[m] < k$ ) then  $\ell \leftarrow m + 1$ 
10    else  $r \leftarrow m - 1$ 
11 // We always return from somewhere within while-loop
```

Example: interpolation-search($A[0..10], 449$)

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

- Initially begin with $\ell = 0$ and $r = n - 1 = 10$

$$m = \ell + \left\lfloor \frac{449-0}{1500-0}(10-0) \right\rfloor = \ell + 2 = 2$$

- Since $A[2] = 2 < k = 449$ we update $\ell = m + 1 = 3$ and keep $r = 10$

$$m = \ell + \left\lfloor \frac{449-3}{1500-3}(10-3) \right\rfloor = \ell + 2 = 5$$

- Since $A[4] = 449 = k$ we found k at index 4

This works well if the keys are *uniformly* distributed:

- Recurrence relation is $T^{(\text{avg})}(n) = T^{(\text{avg})}(\sqrt{n}) + \Theta(1)$
- Recurrence relation resolves to $T^{(\text{avg})}(n) \in \Theta(\log \log n)$

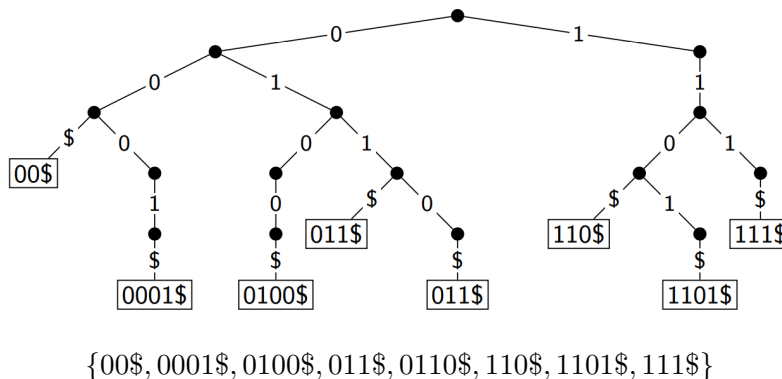
However the worst case performance is $\Theta(n)$ (occurs when keys increase, e.g. $A[i] = 10^i$)

Trie

Definition: *Trie* (also known as a *radix tree*) is a dictionary for bitstrings

- Name comes from **re**trieval, but pronounced as “try”
- Each edge of the tree is labelled with a bit and comparisons are done bitwise.
- Requires: dictionary is *prefix-free* (no string is a prefix of another)
 - Alternative: satisfied if all strings have the same length
 - Alternative: satisfied if all strings end with some *end-of-word* character (we will use \$)
- The items (keys) are stored *only* in the leaf nodes

Example:



Trie Operations

Performing *search* of x on a trie:

- start from the root and the most significant (first) bit of x
- follow the link that corresponds to the current bit in x
 - return failure if link is missing
 - return success if we reach a leaf (it must store x)
 - else recurse on the new node and go to the next bit of x

```

1 // v: node of trie; d: level of v, x: word stored as array of chars
2 Trie::search(v ← root, d ← 0, x)
3   if v is a leaf
4     return v
5   else
6     let v' be child of v labelled with x[d]

```

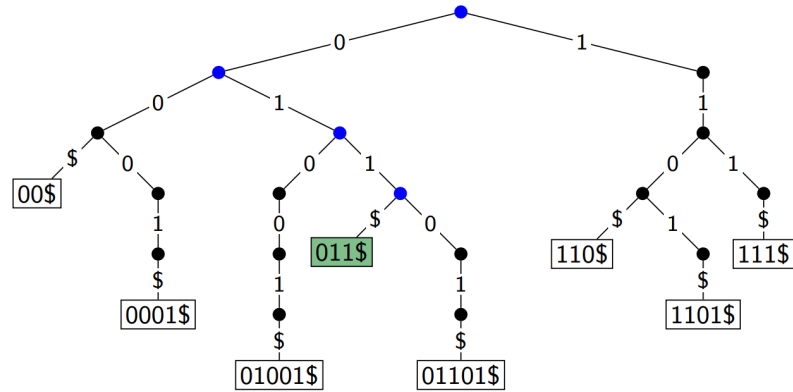
```

7   if there is no such child
8       return "not found"
9   else Trie::search(v', d + 1, x)

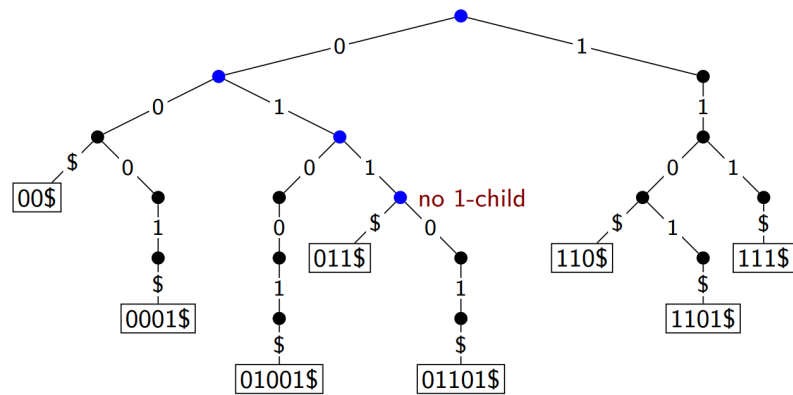
```

Examples:

- *Trie::search(011\$)* success



- *Trie::search(0111\$)* unsuccessful



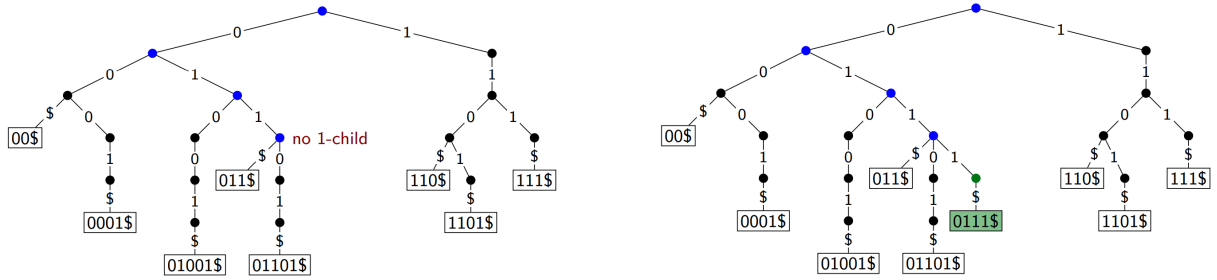
insert and *delete* are simple extensions of *search*:

- *Trie::insert(x)*
 - search for x (should be unsuccessful)
 - We finish the search at a node v that does not have a child for the current bit of x
 - Expand the trie from node v by adding necessary nodes that correspond to extra bits of x
- *Trie::delete(x)*
 - Search for x (should be successful)
 - Let v be the leaf where x is found
 - Delete v and ancestors of v until reaching an ancestor with two or more children

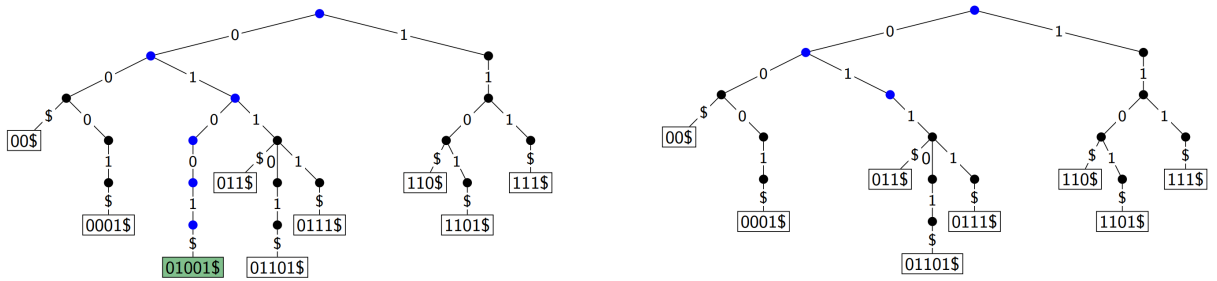
For all operations the *time complexity* is $\Theta(|x|)$ (where $|x|$ is the length of binary string x)

Examples:

- *Trie::insert(0111\$)*

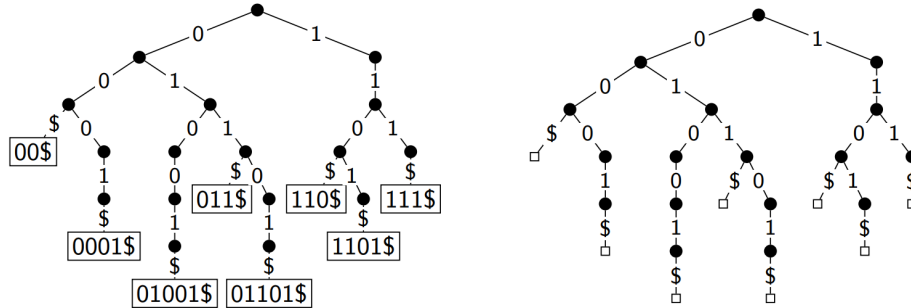


- *Trie::delete(01001\$)*

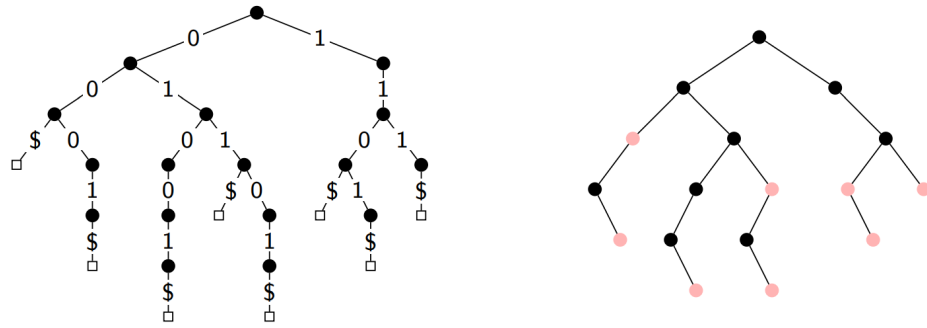


Trie Variations

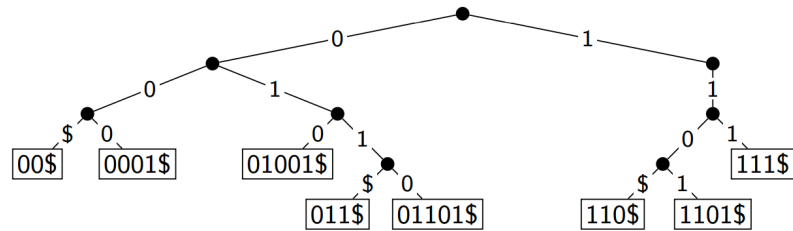
- Variation 1: no leaf labels
 - Key is stored implicitly through the characters along the path to the leaf
 - Halves the amount of the space needed



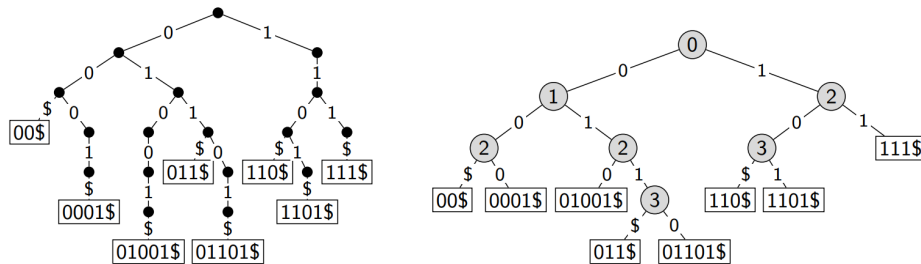
- Variation 2: allow proper prefixes
 - Allow internal nodes to represent keys (use *flag* to indicate such nodes)
 - No need for end-of-word character \$
 - 0-child and 1-child can be expressed as left and right child of a binary tree



- Variation 3: *pruned trie*
 - Stop adding nodes as soon as the key is unique
 - Note that each leaf must now store the full key
 - Saves space if only a few bitstrings are long and can store infinite bitstrings (e.g. real numbers)
 - More efficient but operations get a bit more complicated



- Variation 4: *compressed trie*
 - Compress paths of nodes with only a single child
 - Each node stores an *index*, corresponding to the depth in the uncompressed trie
 - * Use this index to know which bit of x to test against
 - A compressed trie with n keys has at most $n - 1$ internal nodes
 - Also known as Patricia-tries:
 - * Practical Algorithm To Retrieve Information Coded In Alphanumeric



Compressed Trie Operations

Just like with the basic trie search is used in every compressed trie operation

```

1 // v: node of trie; x: word
2 CompressedTrie::search(v ← root, x)
3   if v is a leaf
4     return strcmp(x, v.key)

```

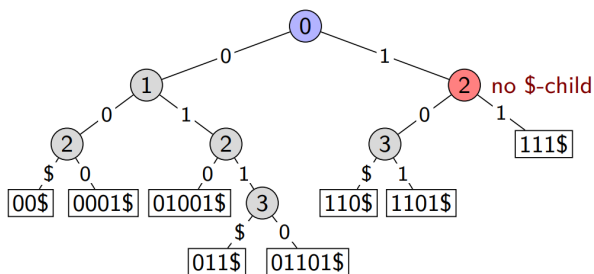
```

5  d ← index stored at v
6  if x has at most d bits
7      return "not found"
8  v' ← child of v labelled with x[d]
9  if there is no such child
10     return "not found"
11  CompressedTrie::search(v', x)

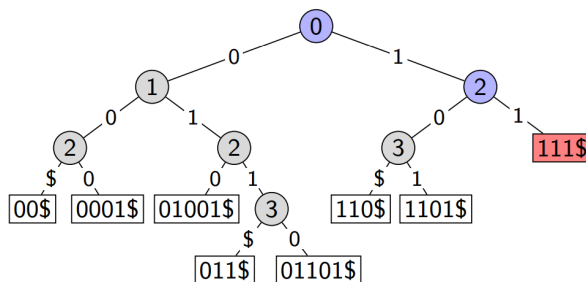
```

Examples:

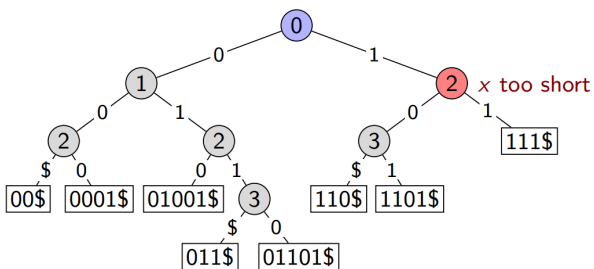
- *CompressedTrie::search(10\$)*



- *CompressedTrie::search(101\$)*



- *CompressedTrie::search(1\$)*



- *CompressedTrie::search(x)*
 - Start from the root and the bit of x indicated at that node
 - Follow the link that corresponds to the current bit in x
 - * Return failure if the link is missing
 - If we reach a leaf, explicitly check whether word stored at leaf is x
 - Else recurse on the new node and the bit of x indicate for that node
- *CompressedTrie::delete(x)*

- Perform $search(x)$
- Remove the node v that stored x
- Compress along path to v whenever possible
- $CompressedTrie::insert(x)$
 - Perform $search(x)$
 - Let v be the node where the search ended
 - Conceptually simplest approach:
 - * Uncompress path from root to v
 - * Insert x as in an uncompressed trie
 - * Compress paths from root to v and from root to x
 - This can also be done by only adding those nodes as they are needed
 - * Requires *leaf-links* where each node stores a link to a leaf that is a descendant

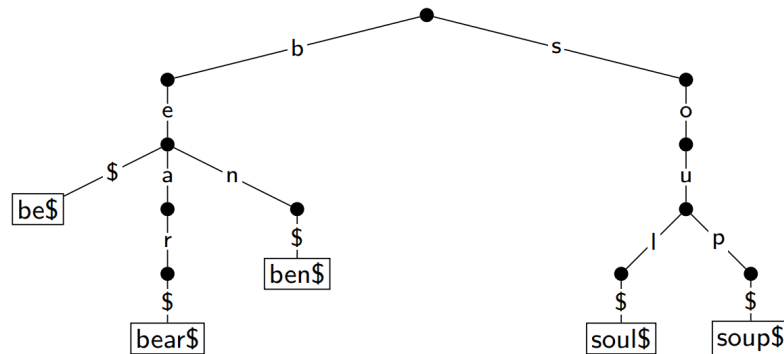
All these operations take $O(|x|)$ time.

This is much more complicated, but the space-saving can be worth it if the words are unevenly distributed.

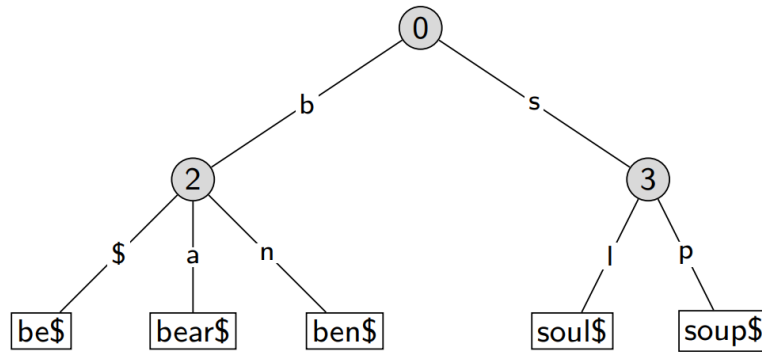
Multiway Tries

In order to represent *strings* over any *fixed alphabet* Σ we use a multiway trie:

- Allow nodes to have at most $|\Sigma| + 1$ children (one child for end-of-word character \$)
- **Example:** a multiway trie holding strings {bear\$, ben\$, be\$, soul\$, soup\$}



- A *compressed multiway trie* is compressed in the same way as a compressed trie
- **Example:** a compressed multiway trie holding strings {bear\$, ben\$, be\$, soul\$, soup\$}



The operations $search(x)$, $insert(x)$, $delete(x)$ are exactly the same as before, but, the runtime is now:

$$O(|x| \cdot (\text{time to find appropriate child}))$$

Each node has up to $|\Sigma| + 1$ children and there are multiple ways to store them:

- Solution 1: array of size $|\Sigma| + 1$ for each node
 - $O(1)$ time to find child
 - $O(|\Sigma|)$ space per node
- Solution 2: linked list of children for each node
 - $O(|\Sigma|)$ time to find child
 - $O(\# \text{ children})$ space per node
- Solution 3: dictionary (e.g. AVL-tree) of children for each node
 - $O(\log(\# \text{ children}))$ time to find child
 - $O(\log(\# \text{ children}))$ space per node
 - Best in theory, but not worth it in practice unless $|\Sigma|$ is huge

In practice, use *hashing* (since keys are in a typically small range Σ)

Dictionaries Via Hashing

Direct Addressing

For a known $M \in \mathbb{N}$, let every key k be an integer with $0 \leq k < M$.

Then we can implement a dictionary using an array T of size M that stores (k, v) via $T[k] \leftarrow v$

- $search(k)$: check if $T[k]$ is NIL
- $insert(k, v)$: $T[k] \leftarrow v$
- $delete(k)$: $T[k] \leftarrow \text{NIL}$

Each operation is $\Theta(1)$ and total space is $\Theta(M)$

This is called *direct addressing* (using key as the array's index) and it has two issues:

- Cannot be used if the keys are not integers
- Wastes space if M is unknown or $n \ll M$

Idea: use a *hash* to map arbitrary keys into integers in range $\{0, \dots, M - 1\}$ then use direct addressing:

- We assume all keys come from some *universe* U (typically $U = \mathbb{Z}_{\geq 0}$, sometimes may be finite)
- The *hashing function* is defined as $h : U \rightarrow \{0, \dots, M - 1\}$ (commonly use $h(k) = k \bmod M$)
- We call the array T of size M the *hash table*
- An item with key k should ideally be stored in *slot* $h(k)$, i.e. $T[h(k)]$

Example: $U = \mathbb{N}$, $M = 11$, $h(k) = k \bmod 11$

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

hash table stores: 7, 13, 43, 45, 49, 92

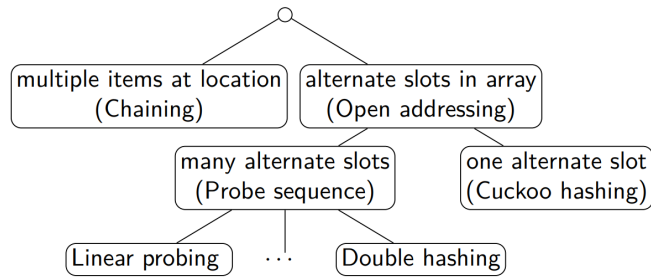
Hash Collisions

Definition: A hash collision occurs when inputs hash to the same output

Generally the hash function h is not injective (many keys may map to the same integer)

- e.g. $h(46) = 2 = h(13)$ if $h(k) = k \bmod 11$
- Then we have a case where we want to insert (k, v) but $T[h(k)]$ is already occupied

There are many solutions to this problem:



Chaining

The most straightforward solution to collisions is to have each slot of the T store a *bucket*.

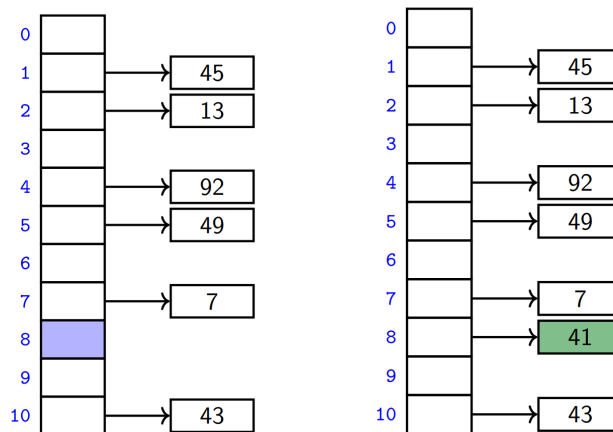
- Each bucket contains 0 or more KVPs
- Usually use a unsorted linked list (but any dictionary realization, even another hash table works)
- This method of collision resolution (via linked list) is called *chaining*

Operations:

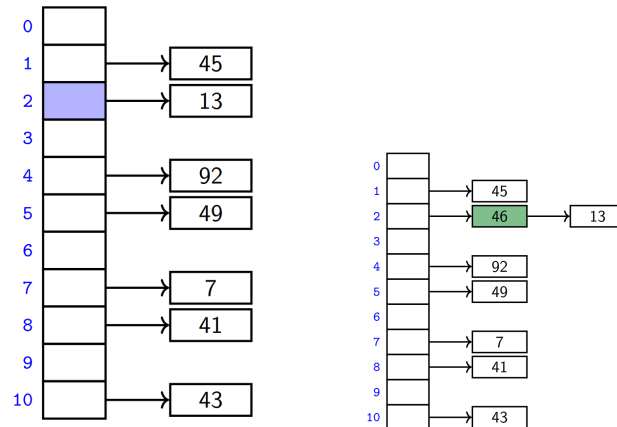
- $search(k)$: look for key k in the list at $T[h(k)]$ (apply MTF-heuristic)
- $insert(k, v)$: add (k, v) to the front of the list at $T[h(k)]$
- $delete(k)$ perform a search, then delete from the linked list

Example: $M = 11$, $h(k) = k \bmod 11$

- $insert(41)$ ($h(41) = 8$)



- $insert(46)$ ($h(46) = 2$)



Complexity of Chaining

- $insert$ takes time $\Theta(1)$
- $search$ and $delete$ have run-time $\Theta(1 + \text{size of bucket } T[h(k)])$

We define $\alpha := n/M$ as the *load factor* of the hash map.

Although the *average* bucket-size is α we cannot say the average case cost of $search$ and $delete$ is $\Theta(1 + \alpha)$

- If all keys hash to the same slot then operations will take $\Theta(n)$
- To analyze what happens *on average* we switch to *randomized hashing*
- To randomize we assume that the hash-function is chosen randomly

Definition (*Uniform Hashing Assumption*): any possible hash function is equally likely to be chosen

- This is not realistic, but it is close enough and makes analysis possible
- We can show that for any key k and slot i

$$P(h(k) = i) = \frac{1}{M}$$

and that the hash-values of any two keys are independent of each other

- Using this assumption, each key of the dictionary is expected to collide with $(n - 1)/M$ other keys

Lemma: the expected cost of $search$ and $delete$ is $\Theta(1 + \alpha)$

Proof: under uniform hashing we have keys k_1, \dots, k_n with two properties:

- $\forall j \in \{0, \dots, M - 1\}$ and $\forall i \in \{0, \dots, n\}$

$$Pr(h(k_i) = j) = \frac{1}{M}$$

- $\forall j, j' \in \{0, \dots, M - 1\}$ and $\forall i, i' \in \{1, \dots, n\}$ with $i \neq i'$

$$Pr(h(k_i) = j \text{ and } h(k_{i'}) = j') = Pr(h(k_i) = j) \cdot Pr(h(k_{i'}) = j') = \frac{1}{M^2} \quad (\text{iff independent})$$

The above are actually consequences of uniform hashing but many people will actually use these results as the definition of uniform hashing.

- Then $\forall i, i' \in \{1, \dots, n\}$ with $i \neq i'$

$$Pr(h(k_i) = h(k_{i'})) = \sum_{j=0}^{M-1} \underbrace{Pr(h(k_i) = h(k_{i'}) = j)}_{1/M^2} = M \frac{1}{M^2} = \frac{1}{M}$$

- Let

$$\chi_{ii'} = \begin{cases} 0 & \text{if } h(k_i) \neq h(k_{i'}) \\ 1 & \text{if } h(k_i) = h(k_{i'}) \end{cases}$$

- Then

$$E[\# \text{ of collision solving } k_i] = \sum_{i' \neq i} E[\chi_{ii'}] = \sum_{i' \neq i} \frac{1}{M} = \frac{n-1}{M}$$

In all collision resolution strategies run-time is in terms of the *load factor* $\alpha := n/M$

- The larger the load factor the larger the bucket sizes and the slower the search/delete
- Load factor can be kept small by *rehashing* which is done by creating a hash table twice the size with new hash-function(s) and re-inserting all the items into the new table
- Rehashing costs $\Theta(M + n)$ but since it happens rarely we can amortize it away
- Also rehash when α gets too small so that $M \in \Theta(n)$ in order for space to always be in $\Theta(n)$
- If we maintain $\alpha \in \Theta(1)$, then (under the uniform hashing assumption) the expected cost for hashing with chaining is $O(1)$ with space in $\Theta(n)$

Open Addressing

Idea: avoid the linking required for chaining by allowing key k to be in possible multiple slots

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

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

until an empty spot or k is found.

delete is a little problematic:

- We cannot leave an empty spot behind (may cause next search to terminate too early)
- Idea 1: move later items in probe sequence forward
 - still problematic because may move stuff from other hashes
- Idea 2: *lazy deletion*: mark the spot as *deleted* instead of NIL
 - if searching continue searching past deleted spot
 - if inserting replace the delete spot with the new value

Linear Probing

The simplest method for open addressing is *linear probing* which is

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

for some hash function h

Example: $M = 11$, $h(k) = k \bmod 11$, $h(k, i) = (h(k) + i) \bmod 11$

- $insert(41)$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

$$h(41, 0) = 8, \quad h(41, 1) = 9, \quad h(41, 2) = 9$$

- $insert(20)$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

$$h(20, 0) = 9, \quad h(20, 1) = 10, \quad h(20, 2) = 0$$

- *delete*(43)

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	deleted

$$h(43, 0) = 10$$

- *search*(63) not found

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	deleted

$$h(63, 0) = 8, \quad h(63, 1) = 9, \quad h(63, 2) = 10, \quad h(63, 3) = 0, \quad h(63, 4) = 1, \quad h(63, 5) = 2, \quad h(63, 6) = 3$$

```

1 probe-sequence::insert(T, (k, v))
2   for (j = 0; j < M; j++)
3     if T[h(k, j)] is NIL or "deleted"
4       T[h(k, j)] = (k, v)
5       return "success"
6   return "failure to insert" // need to re-hash

```

```

1 probe-sequence-search(T, k)
2   for (j = 0; j < M; j++)
3     if T[h(k, j)] is NIL
4       return "item not found"
5     else if T[h(k, j)] has key k
6       return T[h(k, j)]
7     // ignore "deleted" and keep searching
8   return "item not found"

```

Independent Hash Function

Some hashing methods require *two* hash functions h_0 and h_1

- These hash functions should be *independent* in the sense that

$$P(h_0(k) = i) \text{ and } P(h_1(k) = j)$$

are independent

- This means we cannot pick two modular hash-functions

For the second hash function we can use the *multiplication method*:

$$h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$$

- A is some floating-point number with $0 < A < 1$
- $kA - \lfloor kA \rfloor$ computes to just the fractional part of kA which is in $[0, 1)$
- Then we multiply by M to get a floating-point number in $[0, M)$
- Finally we round down to get an integer in $\{0, \dots, M - 1\}$

The quality of the key scrambling is dependent on how good the chosen A is:

- Notice that if we use $A = 1/2$ we only have two keys and $A = 1/3$ leads to only three keys
- Good scrambling can be achieved with $A = \varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749 \dots$
- We should use at least $\log |U| + \log |M|$ bits of A

Double Hashing

Definition: *double hashing* is performing open addressing with probe sequence

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

- Assume h_0 and h_1 are two independent hash functions
- Assume that $h_1(k) \neq 0$ and that $h_1(k)$ is a relative prime with table-size M for all keys k
 - This can be satisfied by choosing M to be prime
 - Modify standard hash-functions to ensure that $h_1(k) \neq 0$
 - * e.g. modified multiplication method: $h(k) = 1 + \lfloor (M - 1)(kA - \lfloor kA \rfloor) \rfloor$

search, insert, delete work just like for linear probing but with this different probe sequence.

Example: $M = 11$, $h_0(k) = k \bmod 11$, $h_1(k) = \lfloor 10\varphi k - \lfloor \varphi k \rfloor \rfloor + 1$, $insert(194)$

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

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

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

$$h_0(194) = 7 \quad h_1(194) = 9$$

$$h(194, 0) = 7, \quad h(194, 1) = 5, \quad h(194, 2) = 3$$

Cuckoo Hashing

This requires two independent hash functions h_0, h_1 and two tables T_0, T_1

Idea: an item with key k can *only* be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$

- *search* and *delete* take constant time
- *insert* initially puts a new item into $T_0[h_0(k)]$
 - If $T_0[h_0(k)]$ is occupied then we *kick out* current occupant and attempt to reinsert at $T_1[h_1(k)]$
 - * Each time we kick out an element we attempt to reinsert it in the *other* array
 - This may lead to a loop of *kicking out* so we abort after too many attempts
 - In the case of failure we rehash with a larger M and new hash functions

insert may be slow, but is expected to be constant time if the load factor is small enough.

```

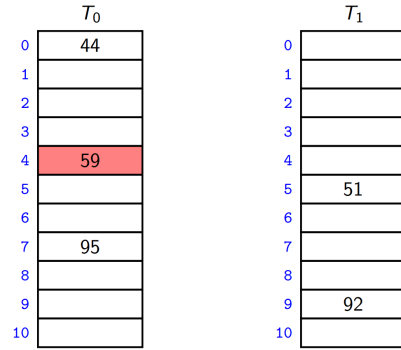
1 cuckoo::insert(k, v)
2   i ← 0
3   do at most 2n times:
4     if Ti[hi(k)] is NIL
5       Ti[hi(k)] ← (k, v)
6       return "success"
7     swap((k, v), Ti[hi(k)])
8     i ← 1 - i
9   return "failure to insert" // need to re-hash

```

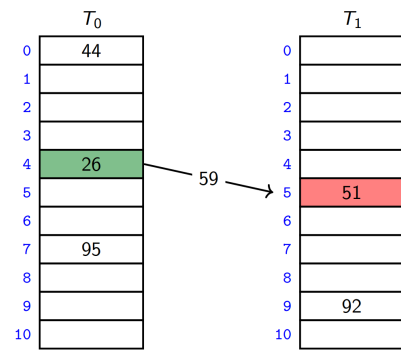
After $2n$ iterations there is definitely a loop in the *kicking out* sequence (in practice should stop earlier).

Example: $M = 11$, $h_0(k) = k \bmod 11$, $h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$, $insert(26)$

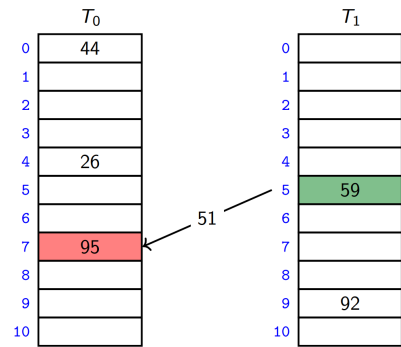
- $k = 26$, $h_0(k) = 4$, $h_1(k) = 0$



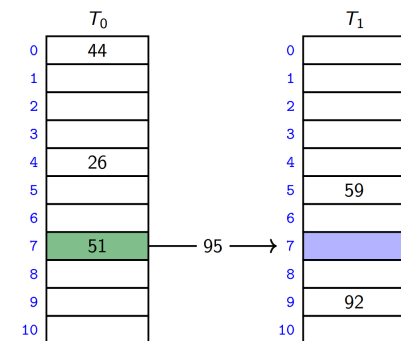
- $k = 59$, $h_0(k) = 4$, $h_1(k) = 5$

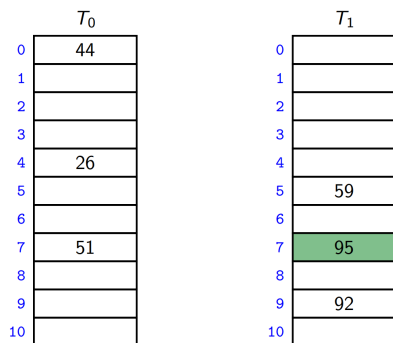


- $k = 51$, $h_0(k) = 7$, $h_1(k) = 5$



- $k = 95$, $h_0(k) = 4$, $h_1(k) = 7$





More on Cuckoo hashing:

- The two hash-tables do not need to be of the same size
- *Load factor* $\alpha = n/(\text{size of } T_0 + \text{size of } T_1)$
- If the load factor α is small enough then insertion has $O(1)$ expected run-time ($\alpha < 1/2$)

Variations on cuckoo hashing:

- The two hash-tables can be combined into one
- More flexible when inserting by considering both possible positions
- Use $k > 2$ allowed locations (i.e. k hash-functions)

Summary of Open Addressing Strategies

For any open addressing scheme we *must* have $\alpha < 1$ and Cuckoo hashing requires a stronger $\alpha < 1/2$

Table of expected # of probes over different strateiges:

	search (unsuccessful)	search (successful)	insert
Linear probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{1-\alpha}$ (avg. over keys)	$\frac{1}{(1-\alpha)^2}$
Double Hashing	$\frac{1}{1-\alpha} + o(1)$	$\frac{1}{1-\alpha} + o(1)$	$\frac{1}{1-\alpha} + o(1)$
Cuckoo Hashing	1 (worst-case)	1 (worst-case)	$\frac{\alpha}{(1-2\alpha)^2}$

All operations have $O(1)$ expected run-time if the hash-function is chosen uniformly and α is kept small.

Choosing a Good Hash Function

The uniform hashing assumption is impossible to satisfy since there too many possible hash functions.

Instead we need to compromise:

- Choose a hash-function that is easy to compute
- While aiming for $P(h(k) = i) = \frac{1}{M}$ w.r.t key-distribution

If all keys are used equally often then this is easy but in practice to get good performance we need:

- hash-function is unrelated to any possible patterns in the data, and
- depends on all parts of the key

The two basic methods for integer keys we saw were:

- *Modular method*:

$$h(k) = k \bmod M$$

where M should be chosen to be prime

- *Multiplicative method*:

$$h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$$

for some constant floating-point number A with $0 < A < 1$

Carter-Wegman's Universal Hashing

Idea: use randomization to create a family of easy to compute hash functions

- Requires: all keys are in $\{0, \dots, p-1\}$ for some (big) prime p
- Choose $M < p$ arbitrarily, power of 2 is ok
- Choose two *random* numbers $a, b \in \{0, \dots, p-1\}$ where $a \neq 0$
- Then we have the hash function

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

- Clearly $h(k)$ can be computed in $O(1)$ time (although p is large it is still a constant)
 - don't care about bit complexity in this course
- This method of choosing h does not satisfy uniform hashing but we get part of its implication
 - $P(h(k) = i) = 1/M$
 - Hash values of two keys are *not* independent of each other

Multi-Dimensional Data

What if the keys are multi-dimensional, such as strings in Σ^* ?

The standard approach is the *flatten* string w to integers $f(w) \in \mathbb{N}$

$$\begin{aligned} \text{APPLE} &\rightarrow (65, 80, 80, 76, 69) && \text{(ASCII)} \\ &\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R^1 + 69R^0 && \text{(Radix } R = 128) \\ &\rightarrow (((65R + 80)R + 80)R + 76)R + 69 \end{aligned}$$

If we use modular hash function $h(w) = f(w) \bmod M$ we can use Horner's rule to apply mod early:

$$h(\text{APPLE}) = \left(\left(\left(\left(\left((65R + 80) \bmod M \right) R + 80 \right) \bmod M \right) R + 76 \right) \bmod M \right) R + 69 \bmod M$$

Sometimes it is possible to compute the hash of multi-dimensional data in constant time

Hashing vs Balanced Search Trees

Balanced search tree advantages:

- $O(\log n)$ 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 entire structure
- Supports ordered dictionary operations (rank, select, etc)

Hash table advantages:

- $O(1)$ operation cost (if hash-function is random and load factor is small)
- Can choose space-time tradeoff via load factor
- Cuckoo hashing achieves $O(1)$ worst-case for search and delete

Range Searches

So far our $search(k)$ requests only looked for *one* specific item.

New operation *RangeSearch* looks for *all* items that fall within a given range

- Input: A *range*, i.e. an interval $I = (x, x')$ (may be open or closed at the ends)
- Output: report all KVPs in the dictionary whose key k satisfies $k \in I$
- e.g. $RangeSearch((18, 45])$ should return $\{19, 33, 45\}$

5	10	11	17	19	33	45	51	55	59
---	----	----	----	----	----	----	----	----	----

Measuring the the run-time will work a little different:

- Let s be the *output-size* (number of items in the range)
- We need $\Omega(s)$ time to simply report the items
- We don't know what s is so we keep it as a separate parameter when analyzing run-time
 - $O(s + n)$ doesn't really make sense because $0 \leq s \leq n$
 - $O(s + \log n)$ does make sense because s can dominate the expression

Using existing dictionary realizations:

- Unsorted list/array/hash table: range search requires $\Omega(n)$ time since we need to check for every item explicitly whether it is in the range
- Sorted array: range search can be done in $O(\log n + s)$ time
 - Using binary search, find i so that x_1 is at (or would be at) $A[i]$
 - Using binary search, find i' so that x_2 is at (or would be at) $A[i']$

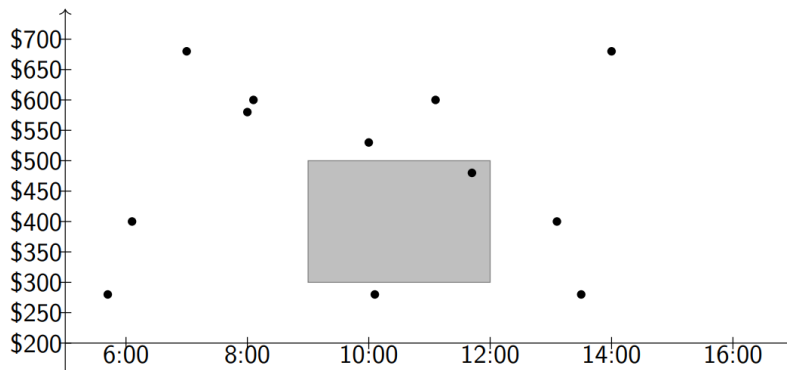
- Report all items $A[i + 1..i' - 1]$
- Report $A[i]$ and $A[i']$ if they are also in the range
- Binary search tree: range search can be done in $O(h + s)$ time (details later)

Multi-Dimensional Data

Range searches are of special interest for *multi-dimensional data*

- Each item has d aspects (coordinates): (x_0, \dots, x_{d-1})
- The values of the aspects x_i are numbers
- Each item corresponds to a point in d -dimensional space
- We will concentrate on $d = 2$, i.e. points in Euclidean plane

Example: we want to find flights that leave between 9am and noon, and cost between \$300 – \$500



d-dimensional range search: given a query rectangle A , find all points that lie within A :

- Assume that points are in *general position*: no two x -coordinates or y -coordinates are the same
 - data structures can be generalized to allow any points
- Could store a 1-dimensional dictionary (where key is some combination of the aspects)
 - Problem: range search on one aspect is not straightforward
- Could use one dictionary for each aspect
 - Problem: inefficient and wastes space

A better idea is to design new data structures specifically for points.

Quadtrees

Assume we have n points $S = \{(x_0, y_0), \dots, (x_{n-1}, y_{n-1})\}$ on the plane.

We begin with a *bounding box* R that contains all the points

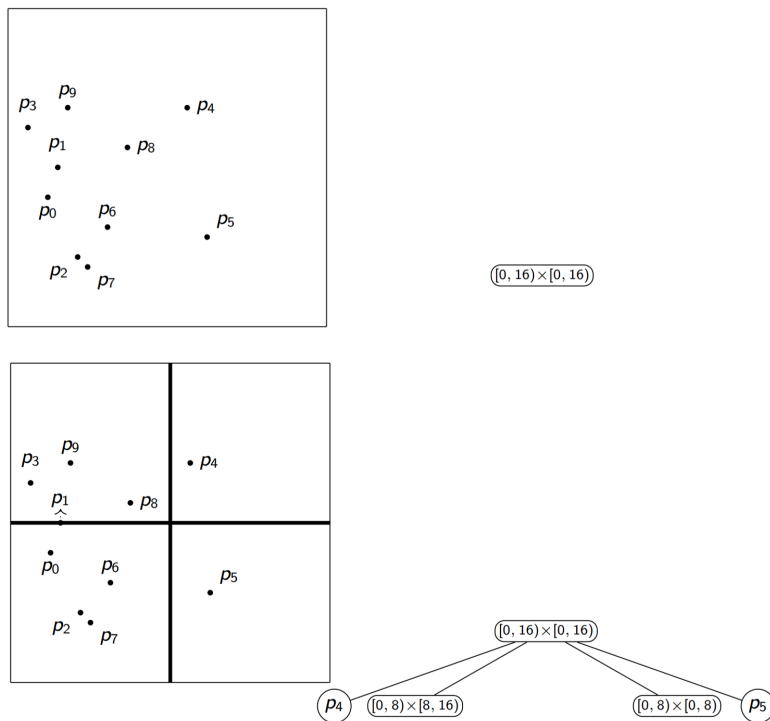
- This R can be found by computing max and min, x and y values in S
- Width and height of R should be a power of 2

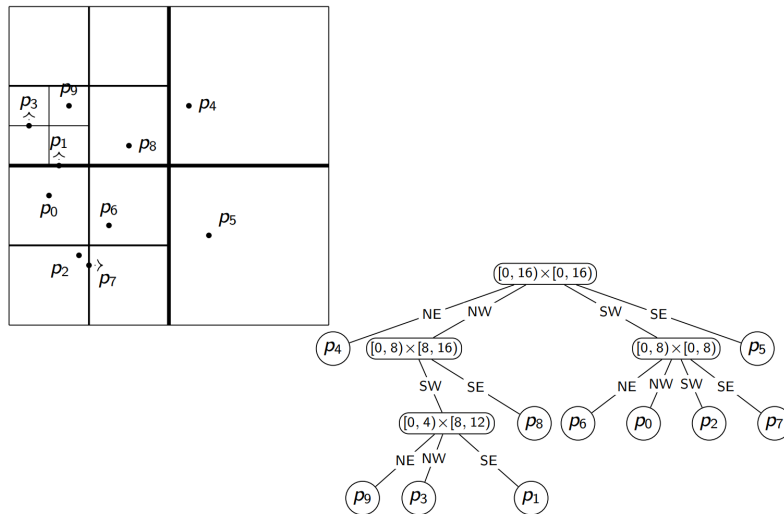
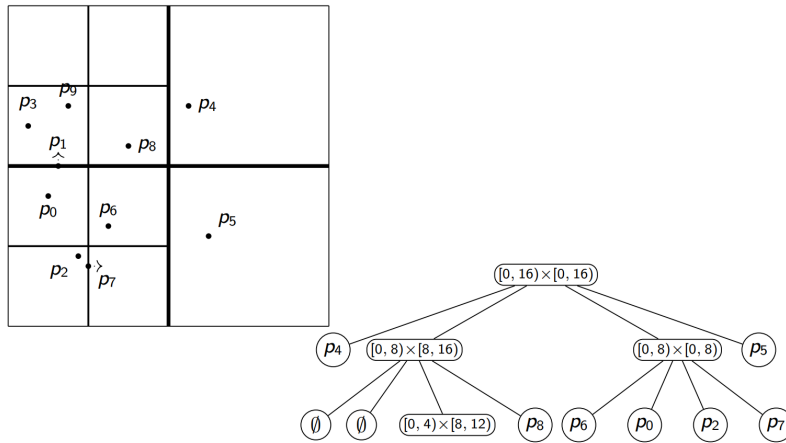
Then to build the quadtree that stores S we create the following structure:

- Root r of the quadtree is associated with the entire region R
 - If R contains 0 or 1 points, then root r is a leaf that stores the point and we are done
- If there are more than 1 point, then partition R into 4 equal subsquares (*quadrants*)
 - $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
- Distribute the points in S into the set where they belong: $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
 - *Convention*: points on the split lines belong to the right/top side (tie-breaker)
- Recursively build tree T_i for points S_i in region R_i and make them children of the root

The KVPs are only stored in a leaves and each leaf can only contain 0 or 1 KVP.

Example: steps to build a Quadtree

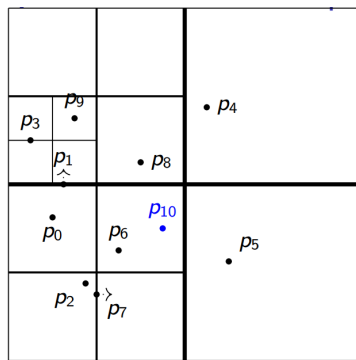




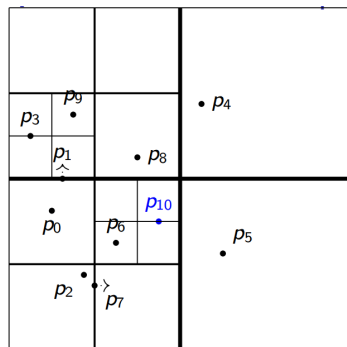
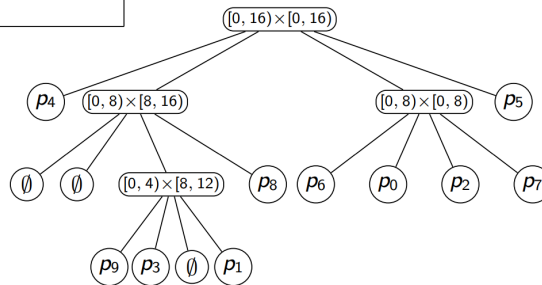
Quadtree Dictionary Operations

- *search*: same as BST and tries
- *insert*:
 - Search for the point then place at the region
 - Split the leaf if there are two points in a single region
- *delete*:
 - Search for the point then remove it
 - Recursively delete all ancestors that have only one point left

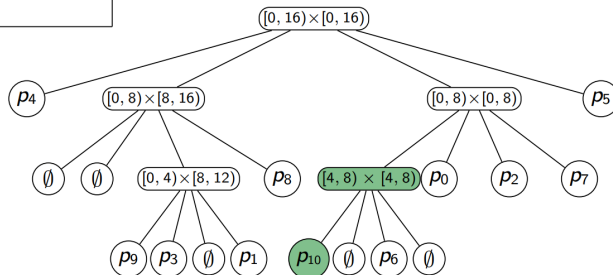
Example: Quadtree insert example



insert(p_{10})



insert(p_{10})



Quadtree Range Search

```

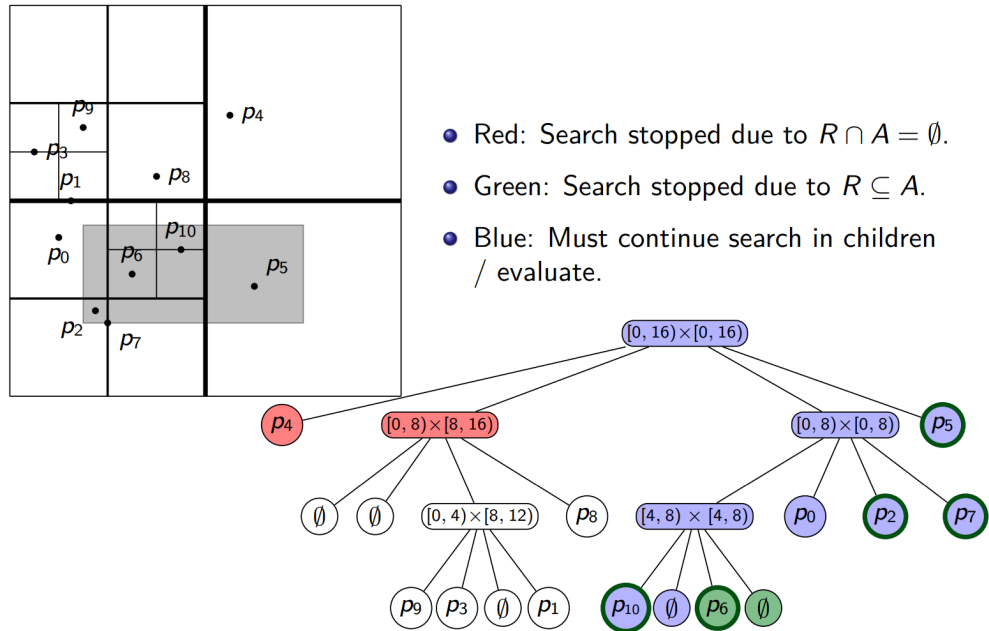
1 // r: The root of a quadtree, A: Query-rectangle
2 QTree::RangeSearch(r ← root, A)
3   R ← region associated with node r
4   if (R ⊆ A) then // inside node
5     report all points below r; return
6   if (R ∩ A is empty) then // outside node
7     return
8   // The node is a boundary node, recurse
9   if (r is a leaf) then
10    p ← point stored at r
11    if p is in A return p
12    else return
13   for each child v of r do
14     QTree::RangeSearch(v, A)

```

Note: we assume that each node of the quadtree stores its associated squared, but, they can also be

recomputed during the search (space-time tradeoff)

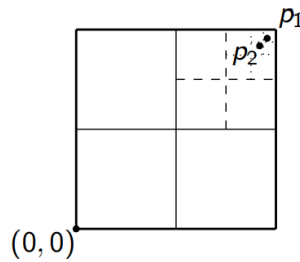
Example:



Quadtree Analysis

Notice that if we have some points really close to each other then we can end up with a really bad height.

Example: create a quadtree from 3 points: p_0 at $(0, 0)$ then p_1 and p_2 close to each other



We introduce *spread factor* of points P

$$\beta(S) = \frac{\text{sidelength of } R}{\text{minimum distance between points in } S}$$

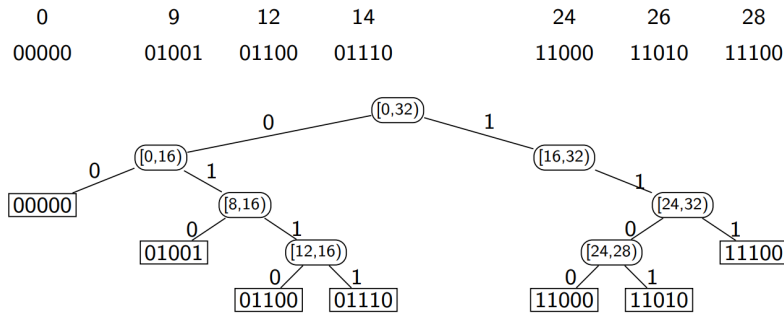
Can show that height h of the quadtree is $\Theta(\log \beta(S))$

- Complexity to build quadtree: $\theta(nh)$ worst-case
- Complexity of range search: $\theta(nh)$ worst-case (even if answer is \emptyset)

In practice these can be much faster

Quadtrees in Other Dimensions

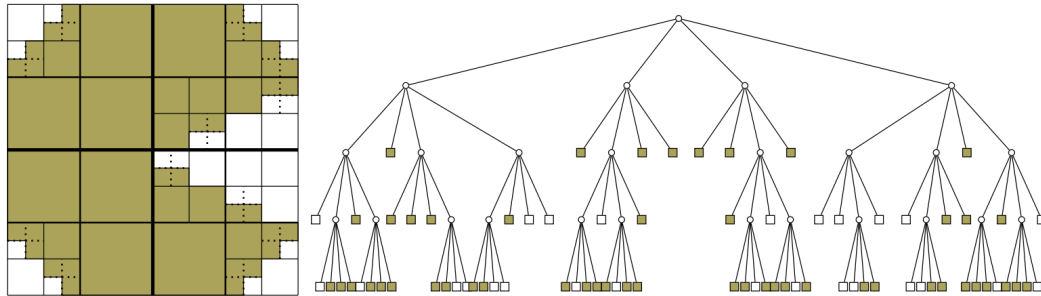
A quadtree of 1-dimensional points 1 or 0 acts like a pruned trie:



Can generalize quadtrees to higher dimensions (e.g. octrees) but will rarely use them beyond dimension 3.

Quadtree Summary

- Very easy to compute and handle
- No complicated arithmetic, only divisions by 2 (bit-shift) if height/width of R is power of 2
- Can potentially waste space if points are not well-distributed
- Variation: stop splitting earlier and allow up to S points in a leaf (for fixed S)
- Variation: use quadtree to store an image



kd-Trees

Suppose we have n points $S = \{(x_0, y_0), \dots, (x_{n-1}, y_{n-1})\}$.

Idea: instead of always splitting into quadrants (like quadtrees), split the region such that (roughly) half the points are each subtree

- Each node keeps track of a *splitting line* for one dimension
 - For the 2D case the line is either vertical or horizontal
- *Convention:* points on the split lines belong to right/top side
- Split, switching between making vertical and horizontal lines, until every point is in a separate region
 - *alternative:* split along the dimension that has more even length ratio for the resulting regions

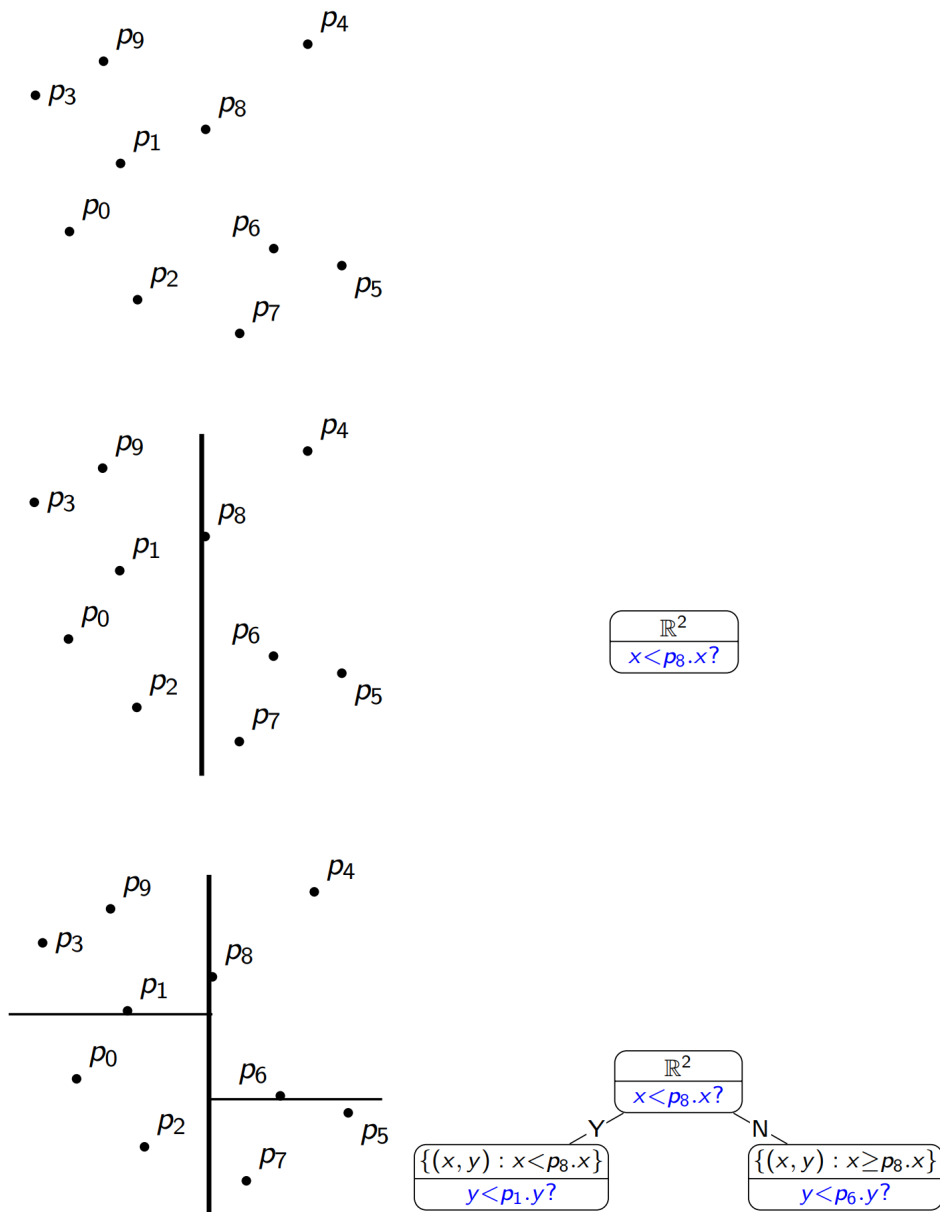
Constructing a kd-Tree

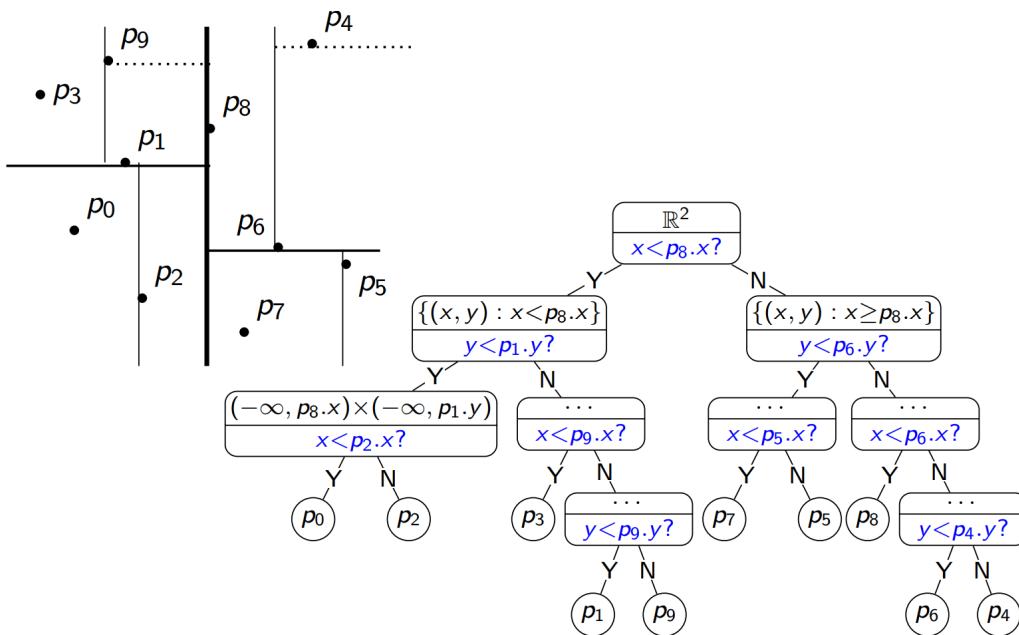
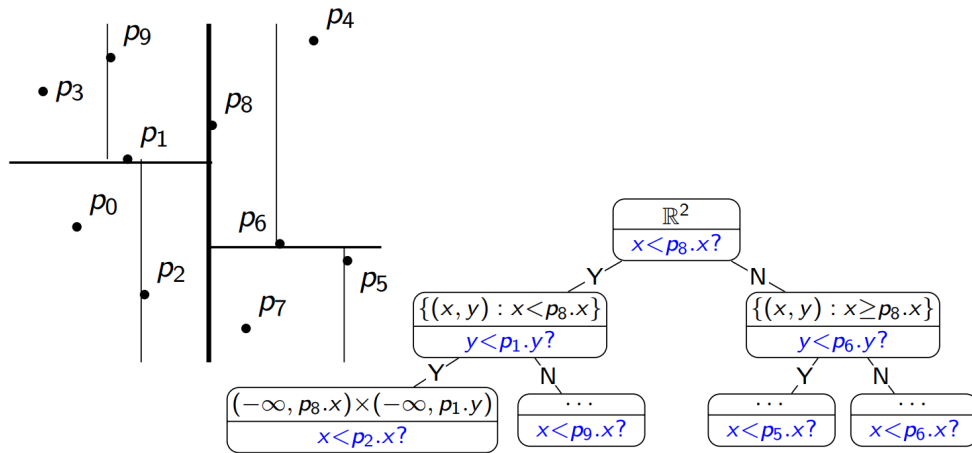
To build a kd-tree with with initial splitting x on points S :

- If $|S| \leq 1$ create a leaf and return
- Else use quick-select to select the median x -coordinate

- $X := \text{quick-select}(S, \lfloor n/2 \rfloor)$ (selected by x -coordinate)
- Partition S by x -coordinate into $S_{x < X}$ and $S_{x \geq X}$
 - This will lead to $\lfloor n/2 \rfloor$ points on one side and $\lceil n/2 \rceil$ points on the other
 - Recall that points are in general position
- Create left subtree recursively (splitting by y) on points $S_{x < X}$
- Create right subtree recursively (splitting by y) on points $S_{x \geq X}$

Example:





The time it takes to construct a kd-tree:

- Find X and partition S in $\Theta(n)$ expected time using *randomized-quick-select*
- Both subtrees have $\approx n/2$ points

$$T^{\text{exp}}(n) = 2T^{\text{exp}}(n/2) + O(n) \quad (\text{sloppy recurrence})$$

This resolves to $\Theta(n \log n)$ expected time

- This can be reduced $\Theta(n \log n)$ *worst-case* time by pre-sorting (no details)
 - heap on x -coordinates and another heap on y -coordinates

Height: $h(1) = 0, h(n) \leq h(\lceil n/2 \rceil) + 1$

- Resolves to $O(\lceil \log n \rceil)$

kd-Tree Dictionary Operations

- *search*: same as in a BST (via the indicated coordinate)
- *insert*: search then insert as new leaf
- *delete*: search then remove leaf

After insert or delete, the split might no longer be at the exact median so height may not longer be guaranteed to be $\lceil \log n \rceil$.

kd-trees overall do not handle insertion/deletion well so we maintain $O(\log n)$ height by occasionally rebuilding entire subtrees.

kd-Tree Range Search

Range search is *exactly* the same as for quadtrees, except that there are only two children

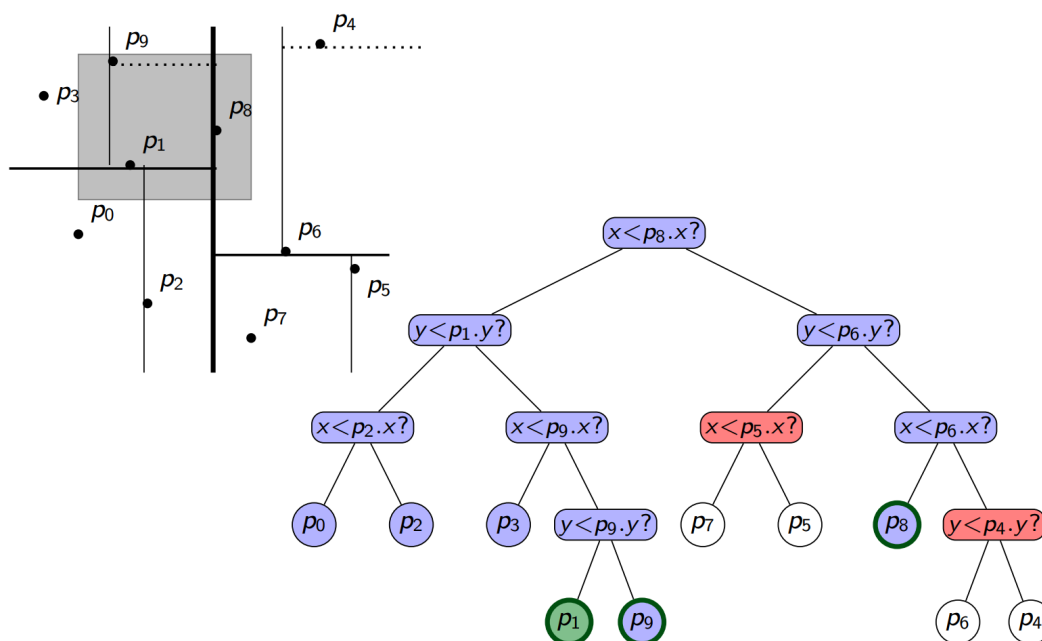
```

1 // r: The root of a kd-tree, A: Query-rectangle
2 kdTree::RangeSearch(r ← root, A)
3   R ← region associated with node r
4   if (R ⊆ A) then report all points below r; return
5   if (R ∩ A is empty) then return
6   if (r is a leaf) then
7     p ← point stored at r
8     if p is in A return p
9     else return
10  for each child v of r do
11    kdTree::RangeSearch(v, A)

```

- We again assume that each node stores its associated region
- To save space, we could instead pass the region as a parameter and compute region for each child using the splitting line

Example:



- Red: search stopped due to $R \cap A = \emptyset$ (no overlap)
- Green: search stopped due to $R \subseteq A$ (entirely contained in search area)

The complexity is $O(s + Q(n))$ where

- s is the output-size
- $Q(n)$ is the number of *boundary* nodes (blue)

Can show that $Q(n)$ satisfies the following recurrence relation (no details):

$$Q(n) \leq 2Q(n/4) + O(1)$$

This resolves to $Q \in O(\sqrt{n})$ therefore the complexity of range search is $O(s + \sqrt{n})$

kd-Tree in Higher Dimensions

kd-tree is short for k -dimensional tree

- At the root the point set is partitioned based on the first coordinate
- At the subtrees of the root the partition is based on the second coordinate
- At depth $k - 1$ the partition is based on the last coordinate
- At depth k we start all over again, partitioning on the first coordinate

For a k -dim tree assuming that k is constant we have:

- Storage: $O(n)$
- Height: $O(\log n)$
- Construction time: $O(n \log n)$
- Range search time: $O(s + n^{1-1/k})$

Notice how as k increases the range search run-time approaches linear time.

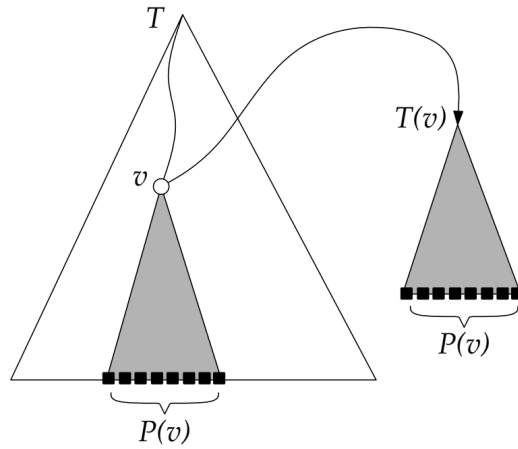
Range Trees

While quadtrees and kd-trees are intuitive and simple they are kind of slow for range searches.

Range Trees are built from a tree of trees (*multi-level* data structure) that although are more wasteful in space, will perform range searches much faster.

For a 2-dimensional range tree we have:

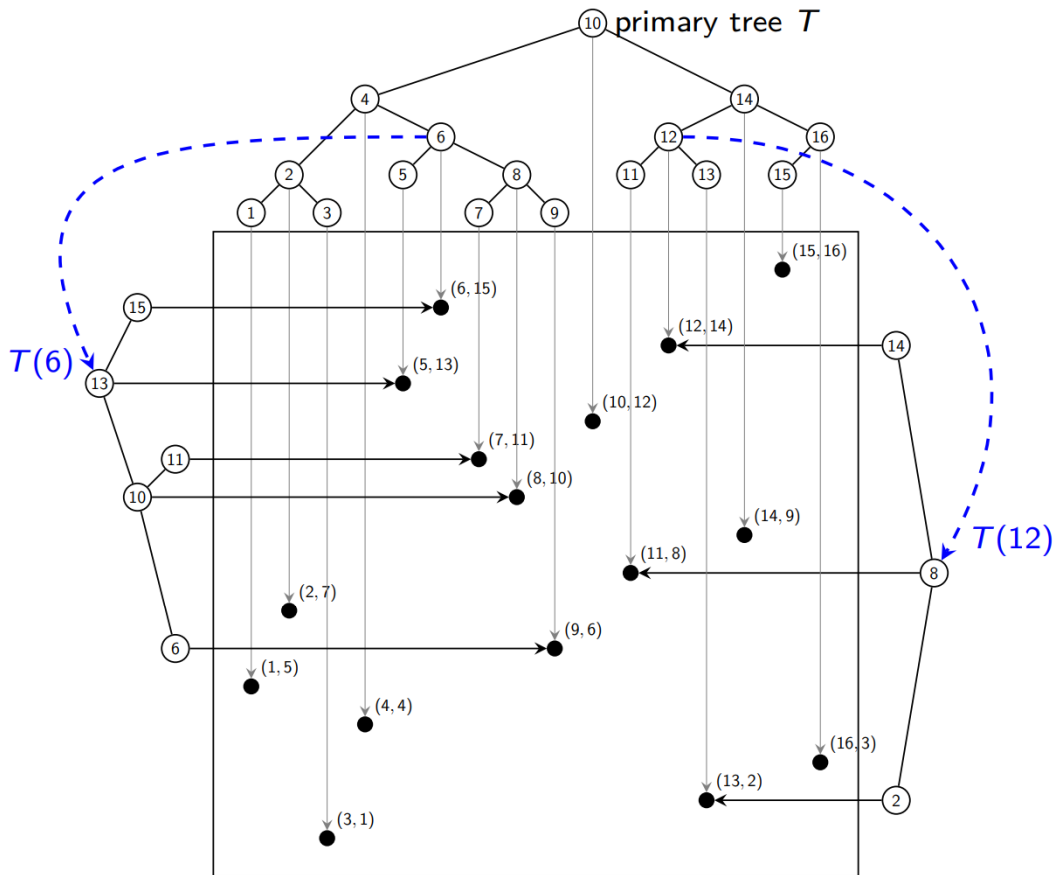
- *Primary structure*: balanced BST T that stores P and uses x -coordinates as keys
- Every node v of T stores an *associate structure* $T(v)$:
 - Let $P(v)$ be all the points in subtree of v in T (including v itself)
 - $T(v)$ stores $P(v)$ in a balanced BST using the y -coordinates as the key
 - v is not necessarily the root of $T(v)$



- The subtree v in T stores the points in a balanced BST using the x -coordinates
- The tree $T(v)$ stores the same points in a balanced BST using the y -coordinates

Note that we are assuming that points are in general position.

Example: partial range tree example (not all associate trees are shown)



Range tree space analysis:

- Primary tree uses $O(n)$ space

- Associate tree $T(v)$ uses $O(|P(v)|)$ space
 - Notice that $w \in P(v)$ implies that v is an ancestor of w in T
 - Every node w has $O(\log n)$ ancestors in T (we assume T to be balanced BST)
 - Every node w belongs to $O(\log n)$ sets $P(v)$
 - Then we have

$$\sum_v |P(v)| \leq \sum_w \#\{\text{ancestors of } w\} \in O(n \log n)$$

Thus a range-tree with n points uses $O(n \log n)$ space.

Range Tree Dictionary Operations

- *search*: search by x -coordinate in T
- *insert*: first insert point by x -coordinate into T , then
 - walk back up to root and insert the point by y -coordinate in *all* associate trees $T(v)$ of nodes v
- *delete*: analogous to insertion

One problem is that we want the BSTs to be balanced

- however if we use AVL-trees this makes *insert/delete* slow
 - rotation at v changes $P(v)$ and requires a rebuild of $T(v)$
- This is solved by completely rebuilding the the subtrees when they get highly unbalanced

BST Range Search

The main component to this is how to perform range search on a regular BST:

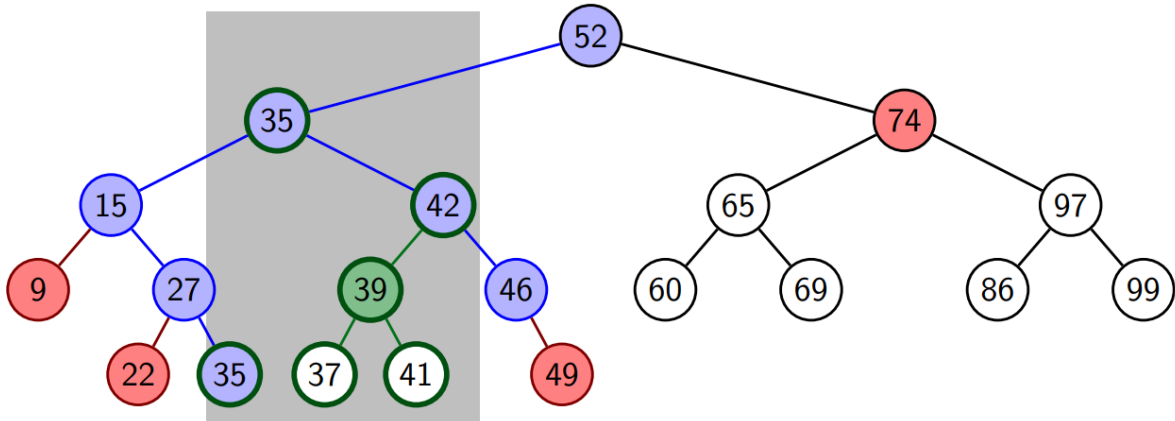
```

1 // r: root of a binary search tree, x1, x2: search keys
2 // Returns keys in subtree at r that are in range [x1, x2]
3 BST::RangeSearch-recursive(r ← root, x1, x2)
4   if r = NIL then return
5   if x1 ≤ r.key ≤ x2 then
6     L ← BST::RangeSearch-recursive(r.left, x1, x2)
7     R ← BST::RangeSearch-recursive(r.right, x1, x2)
8     return L ∪ r.{key} ∪ R
9   if r.key < x1 then
10    return BST::RangeSearch-recursive(r.right, x1, x2)
11  if r.key > x2 then
12    return BST::RangeSearch-recursive(r.left, x1, x2)

```

The keys are reported in-order, i.e. sorted order (useful but not required)

Example: $BST::RangeSearch-recursive(T, 28, 43)$



- Search for left boundary x_1 : gives path P_1
- Search for right boundary x_2 : gives path P_2
- This partitions T into three groups: outside, on, or between these paths
- Boundary nodes (blue): nodes in P_1 or P_2 (need to test if in range)
- Outside nodes (red): nodes that are left of P_1 or right of P_2 (will never be in range)
- Inside nodes (green): nodes that are right of P_1 and left of P_2 (all descendants are in range)

BST range search analysis

- Assume that the BST is balanced
- Searching for paths P_1 and P_2 take $O(\log n)$ each and produce $O(\log n)$ boundary nodes (blue)
- We spend $O(1)$ on each binary node
- We also spend $O(1)$ time per topmost inside node v (green)
 - Since they are children of boundary nodes, this takes $O(\log n)$ time
- For a 1D range search we also report the descendants of v

$$\sum_{v \text{ topmost inside}} \#\{\text{descendants of } v\} \leq s$$

since subtrees of topmost inside nodes are disjoint, so this takes $O(s)$ time overall

Thus the runtime for 1D range search is $O(\log n + s)$ (same as range search on sorted array)

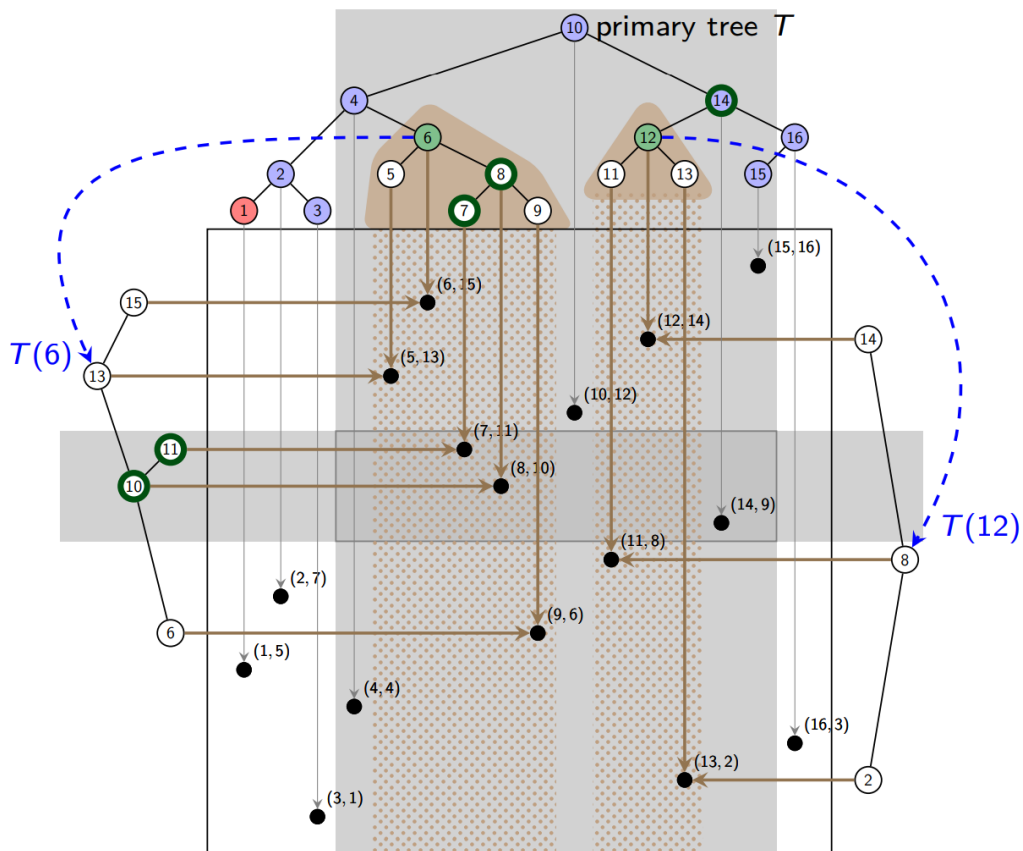
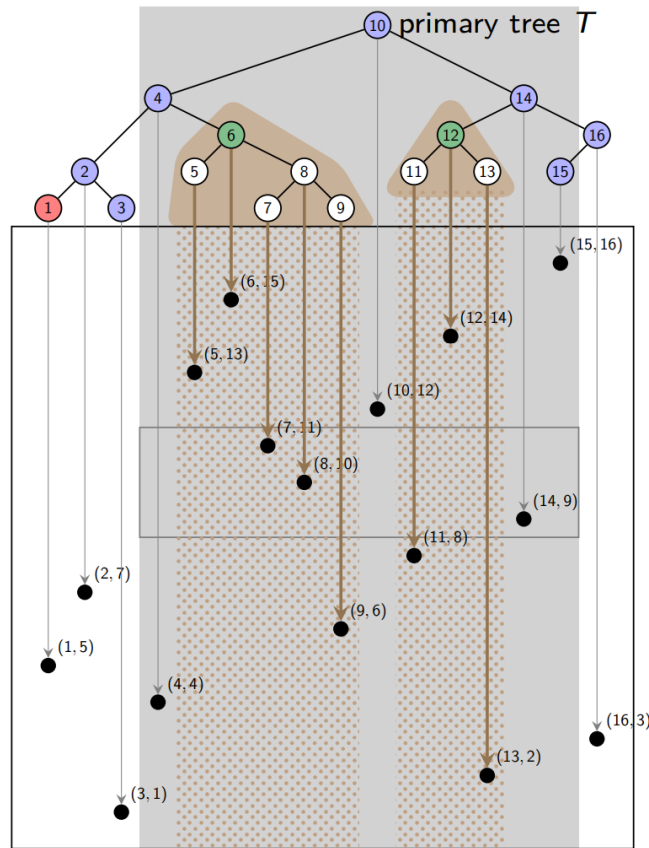
Range Tree Range Search

To perform a range search for $A = [x_1, x_2] \times [y_1, y_2]$ we use a two stage process:

- Perform a range search on x -coordinates for interval $[x_1, x_2]$ in primary tree T
 - $BST::RangeSearch(T, x_1, x_2)$
 - Note: for every *boundary node* (blue) test if corresponding point is actually in A
- For every *topmost inside node* (green) v :

- Let $P(v)$ be the points in the subtree of v in T
- We know that all x -coordinates of points in $P(v)$ are within $[x_1, x_2]$
- Find the subset of points in $P(v)$ where the y -coordinates are within the range as well
 - * this is done by performing a range search on $T(v)$
 - * $BST::RangeSearch(T(v), y_1, y_2)$

Example:



Range search run-time:

- $O(\log n)$ time to find boundary and topmost inside nodes in primary tree
 - there are $O(\log n)$ such des
- $O(\log n + s_v)$ time for each topmost inside node v
 - where s_v is the number of points in $T(v)$ that are reported
- Two topmost inside nodes have no common point in their trees
 - every point is reported in at most one associate structure
 - $\sum_{v \text{ topmost inside}} s_v \leq s$

So the overall time for range search on a range-tree is proportional to

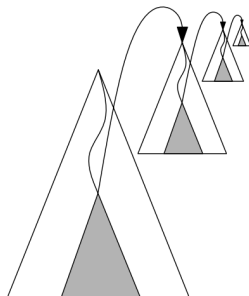
$$\sum_{v \text{ topmost inside}} (\log n + s_v) = \sum \log n + \sum s_v \in O(\log^2 n + s)$$

There are methods to make this faster but no details from this course.

Range Trees in Higher Dimensions

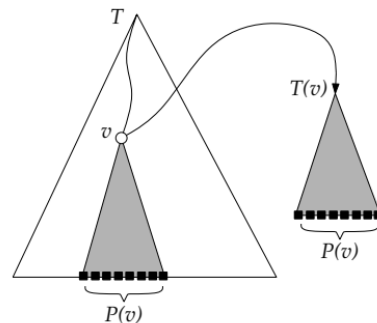
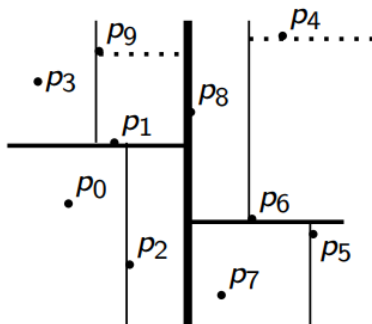
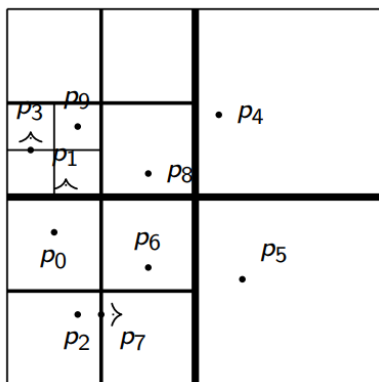
Range trees can be generalized in k -dimensional space, so assuming that k is constant:

- Space: $O(n(\log n)^{k-1})$
- Construction time: $O(n(\log n)^k)$
- Range search time: $O(s + (\log n)^d)$



Notice that kd-trees are actually better than range trees when going into higher dimensions.

Summary of Range Search Data Structures



- Quadtrees
 - simple (also for dynamic set of points)
 - works well only if points are evenly distributed
 - wastes space for higher dimensions
- kd-trees
 - linear space
 - range search time: $O(\sqrt{n} + s)$
 - insert/delete destroy balance (no simple fix)
- range-trees
 - range search time; $O(\log^2 n + s)$
 - wastes some space
 - insert/delete destroy balance (can fix with occasional rebuild)

Convention: points on split lines belong to right/top side

String Matching

The goal of string matching is to locate a string (pattern) in a large body of text:

- $T[0..n - 1]$ the text (or haystack) being searched within
- $P[0..m - 1]$ the pattern (or needle) being searched for
- All strings are all over an alphabet Σ
- We are finding the smallest i such that

$$P[j] = T[i + j] \quad \text{for } 0 \leq j \leq m - 1$$

- this is the first occurrence of P in T
- if P does not occur in T , return FAIL

Example: P_1 is found at index 1 of T while P_2 is not found in T

$$T = \text{"where is he?"} \quad P_1 = \text{"he"} \quad P_2 = \text{"who"}$$

- *Substring* of T : a string that consists of $T[i..j]$ for some $0 \leq i \leq j < n$
- *Prefix* of T : a string that consists of $T[0..i]$ for some $0 \leq i < n$
- *Suffix* of T : a string that consists of $T[i..n - 1]$ for some $0 \leq i < n$

In general pattern matching algorithms will consists of *guesses* and *checks*:

- A *guess* or *shift* is a position i such that P might start at $T[i]$
 - Valid guesses (initially) are $0 \leq i \leq n - m$

- A *check* of a guess is a single position j with $0 \leq j < m$ where we compare $T[i + j]$ to $P[j]$
 - Need to perform m checks to confirm a *correct* guess
 - If the guess is *incorrect* then may require much fewer checks

Brute-Force Algorithm

Idea: check every possible guess (note that *strcmp* takes $\Theta(m)$ time)

```

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

```

```

1 strcmp(T[i..i+m-1], P[0..m-1])
2   for j ← 0 to m - 1 do
3     if T[i + j] is before P[j] in  $\Sigma$  then return -1
4     if T[i + j] is after P[j] in  $\Sigma$  then return 1
5   return 0

```

Example: $T = abbbababbab$, $P = abba$

	a	b	b	b	a	b	a	b	b	a	b
a	a	b	b	a							
b		a									
b			a								
b				a							
a					a	b	b				
b						a					
a							a	b	b	a	

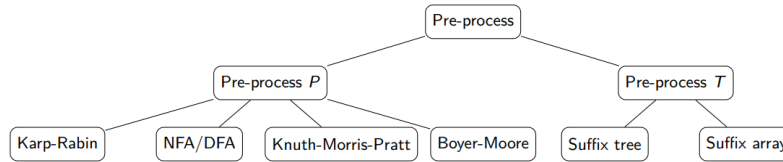
- A possible worst input is
$$P = a^{m-1}b \quad T = a^n$$
- Worst case performance $\Theta((n - m) \cdot m)$ which overall becomes $\Theta(mn)$

Improving Brute Force

We will follow two methods to improve this time:

- *preprocessing* on the pattern P
 - Karp-Rabin
 - Boyer-Moore
 - Deterministic finite automata (DFA), KMP
 - *eliminate guesses* based on completed matches and mismatches
- *preprocessing* on the text T
 - Suffix-trees
 - Suffix-arrays

- create a data structure to find matches easily



Karp-Rabin Algorithm

Idea: use hashing to eliminate guesses

- Compute hash of every guess and compare with hash of the pattern
- If the values are unequal then the guess cannot be an occurrence

$$a = b \implies h(a) = h(b) \quad \rightarrow \quad h(a) \neq h(b) \implies a \neq b$$

- if the hashes match then they *might* be the same

A crucial insight is that we can use the previous hash to compute the next hash in constant time:

- $O(1)$ time per hash, except the first one
- Precomputed: $10000 \bmod 97 = 9$ (based the value of the MSB)
- Current hash: $\underline{4}1592 \bmod 97 = 76$
- Next hash:

$$\begin{aligned}
 1592\underline{6} \bmod 76 &= (41592 - 4 \cdot 10000) \cdot 10 + 6 \\
 &= ((\underbrace{41592 \bmod 97}_{\text{current hash (76)}} - 4 \cdot \underbrace{10000 \bmod 97}_{\text{precomputed (9)}}) \cdot 10 + 6) \bmod 97 \\
 &= ((76 - 4 \cdot 9) \cdot 10 + 6) \bmod 97 \\
 &= 18
 \end{aligned}$$

```

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 ← 10^{m-1} mod M
6   for i ← 0 to n - m
7     if hT = hP
8       if strcmp(T[i..i+m-1], P) = 0
9         return "found at guess i"
10    if i < n - m // compute hash-value for next guess
11      hT ← ((hT - T[i] · s) · 10 + T[i+m]) mod M
12  return "FAIL"
  
```

- Choose table size M to be *random* prime in $\{2, \dots, mn^2\}$
- Expected time $O(m + n)$, worst-case time $O(mn)$ (very unlikely)
- Improvement: reset M if no match of $h_T = h_P$

Boyer-Moore

Boyer-Moore is the fastest pattern matching algorithm for English text and requires 3 components:

- *Reverse-order searching*: compare P with a guess starting from the end and moving *backwards*
- When a mismatch occurs then choose the better of the following two options:
 - *Bad character jumps*: eliminate guesses based on mismatched characters of T
 - *Good suffix jumps*: eliminate guesses based on matched suffix of P

Reverse Order Searching

For each guess we start checking from the end of the pattern and move towards the beginning

- If the letter seen in text does not appear in the pattern we can shift past it
- This works well with the bad character heuristic

Example: $P = \text{aldo}$, $T = \text{whereiswaldo}$

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

- Since r does not occur in P we shift past it
- Since w does not occur in P we shift past it

Bad Character Heuristic

Example:

$P: \text{p a p e r}$
 $T: \text{f e e d a l l p o o r p a r r o t s}$

				r												
			[a]			r										
				[p]		r										
										e	r					

1. Mismatched character in the text is a
2. Shift the guess until a in P aligns with a in T
3. Shift the guess until *last* p in P aligns with p in T
4. Shift completely past o since o is not in P
5. Finding r in this place does not help so just shift by one unit
 - Could get a better shift if use a good suffix jump

To implement this we build a *last-occurrence array* L mapping Σ to integers

- $L[c]$ is the largest index i such that $P[i] = c$
- If c is not in P then set $L[c] = -1$ (will be useful later)
- e.g.

Pattern:				
0	1	2	3	4
p	a	p	e	r

Last-Occurrence Array:					
char	p	a	e	r	all others
$L[\cdot]$	2	1	3	4	-1

This can be built in $O(m + |\Sigma|)$ time with a simple for loop:

```

1 BoyerMoore::lastOccurrenceArray(P[0..m-1])
2   initialize array L indexed by  $\Sigma$  with all -1
3   for j  $\leftarrow$  0 to m-1 do L[P[j]]  $\leftarrow$  j
4   return L

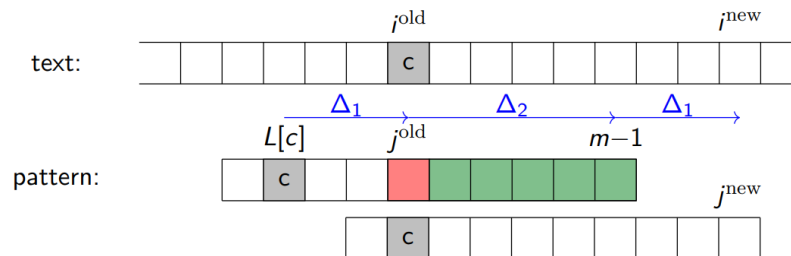
```

To update the location of the guess we have the formula:

$$i^{\text{new}} = i^{\text{old}} + (m - 1) - \min\{L[c], j^{\text{old}} - 1\}$$

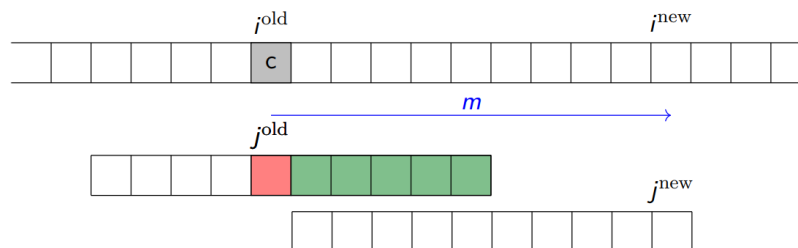
This formula captures 3 cases can occur in the case of a mismatch:

1. $L[c] < j$ so c is at the left of $P[j]$



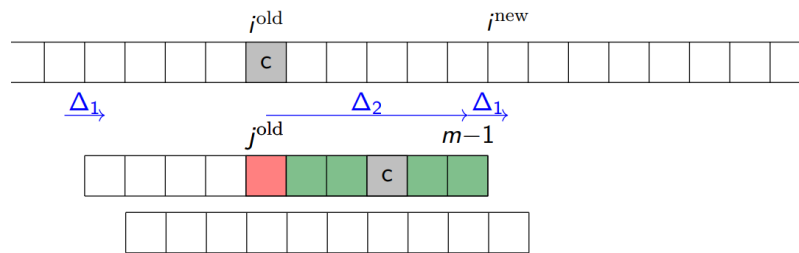
- In this case we only shift enough to line up c in pattern with c in text
- $i^{\text{new}} = i^{\text{old}} + \Delta_2 + \Delta_1 = i^{\text{old}} + (m - 1) - L[c]$
 - $\Delta_1 = j^{\text{old}} - L[c]$ (amount that we should shift)
 - $\Delta_2 = (m - 1) - j^{\text{old}}$ (how much we had compared)

2. c does not occur in P so we get $L[c] = -1$



- In this case we want to shift the pattern past i^{old}
- $i^{\text{new}} = i^{\text{old}} + m = i^{\text{old}} + (m - 1) - L[c]$

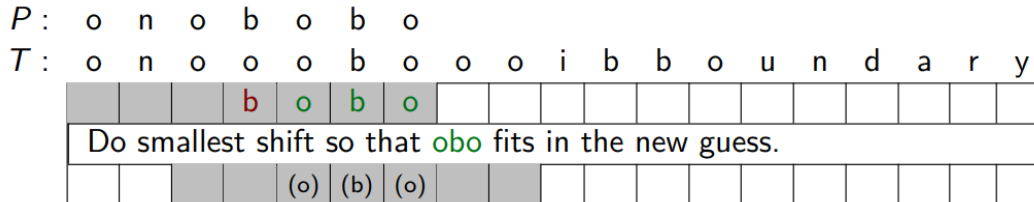
3. $L[c] > j$ so c is at the right of $P[j]$



- Bad character heuristic cannot make a shift more than 1 in this case
- $i^{new} = i^{old} + \Delta_2 + \Delta_1 = i^{old} + 1 + (m - 1) - j^{old} = i^{old} + (m - 1) - (j^{old} - 1)$
 - Notice that $\min\{L[c], j^{old} - 1\} = j^{old} - 1$ since $L[c] > j$ (since j means j^{old})
 - We also won't have $L[c] = j$ because that would imply a match

Good Suffix Heuristic

$S[j]$ contains that amount to shift when $P[j + 1..m - 1]$ are matched.



S can be found in $\Theta(m)$ time but the exact formula is complicated (no details)

Boyer-Moore Summary

```

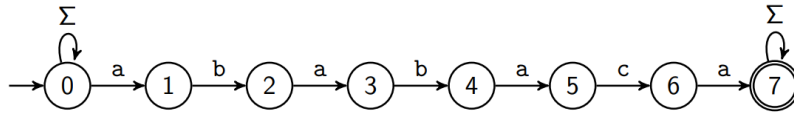
1 Boyer-Moore::patternMatching(T, P)
2   L ← lastOccurrenceArray(P)
3   S ← good suffix array computed from P
4   i ← m - 1, j ← m - 1
5   while i < n and j ≥ 0 do
6     // current guess begins at index i - j
7     if T[i] = P[j]
8       i ← i - 1
9       j ← j - 1
10    else
11      i ← i + m - 1 - min{L[T[i]], j - 1}
12      // if good suffix heuristic is used, then line above should be:
13      // i ← i + m - 1 - min{L[T[i]], S[j]}
14      j ← m - 1
15  if j = -1 return "found at T[i+1..i+m]"
16  else return FAIL
  
```

- Boyer-Moore performs well even without good suffix heuristic
- On typical english text Boyer-Moore only looks at $\approx 25\%$ of T
- Worst case run-time is $O(mn)$ but much faster in practice
 - There are methods to ensure a $O(n)$ run-time (no details)

Knuth-Morris-Pratt (KMP) Algorithm

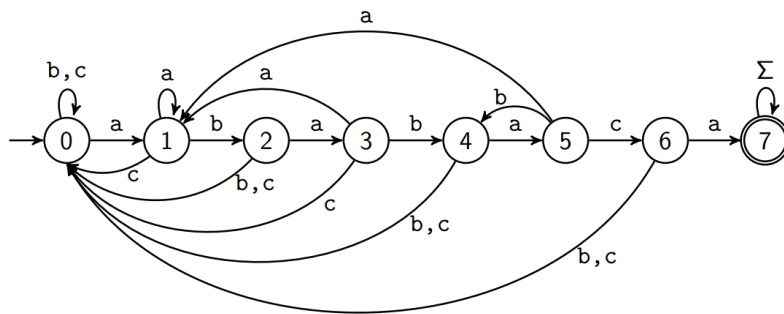
String Matching with Finite Automata

Example: NFA for matching the pattern $P = ababaca$



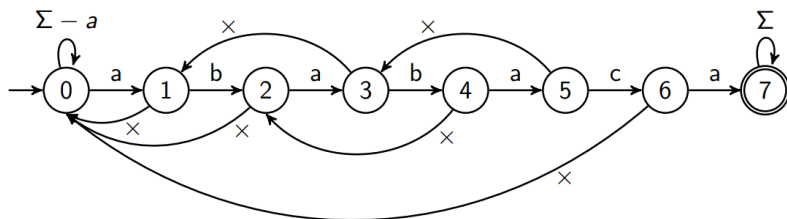
Each state q expresses that we have seen $P[0..q - 1]$ in the text

- NFA will accept T if and only if T contains P
- NFAs evaluation is quite slow so we convert to equivalent DFA ($\Sigma = \{a,b,c\}$)
 - It is always possible to convert a NFA to a DFA



We notice that finding this small DFA from a NFA can be difficult.

KMP Algorithm



- Introduce a new type of transition: \times (*failure*)
 - At most one per state, to be used only if no other transition fits
 - Does *not* consume a character
 - Computations of this automaton is deterministic (however formally this is not a valid DFA)
- The *failure-function* can be stored in an array $F[0..m - 1]$
 - Failure arc from state j leads to $F[j - 1]$

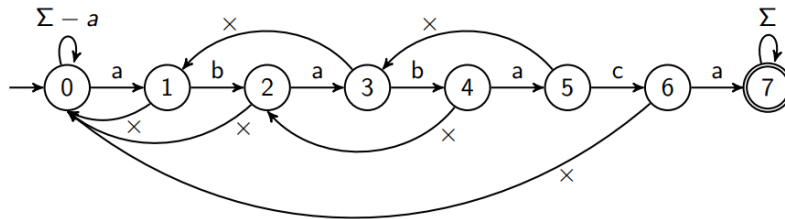
```
1 KMP::patternMatching(T, P)
2   F ← failureArray(P)
3   i ← 0 // current character of T to parse
4   j ← 0 // current state: we have seen P[0..j-1]
5   while i < n do
6     if P[j] = T[i]
7       if j = m - 1
8         return "found at guess i - m + 1"
```

```

9     else
10        i ← i + 1
11        j ← j + 1
12    else // i.e. P[j] != T[i]
13        if j > 0
14            j ← F[j - 1]
15        else
16            i ← i + 1
17    return FAIL

```

Example: $T = abababbcbabaca$, $P = ababaca$



T : a b a b a b b c a b a b a c a

a	b	a	b	a	x										
		(a)	(b)	(a)	b	x									
				(a)	(b)	x									
						x									
							x								
								x							
									x						
										x					
											x				
												x			
													x		
														x	
															x

state:

1	2	3	4	5	3,4	2,0	0	1	2	3	4	5	6	7
---	---	---	---	---	-----	-----	---	---	---	---	---	---	---	---

(after reading this character)

KMP Failure Array

$F[j]$ is the length of the longest prefix of P that is a suffix of $P[1..j]$

Example: Consider $P = ababaca$

j	$P[1..j]$	Prefixes of P	longest	$F[j]$
0	Λ	$\Lambda, a, ab, aba, abab, ababa, \dots$	Λ	0
1	b	$\Lambda, a, ab, aba, abab, ababa, \dots$	Λ	0
2	ba	$\Lambda, a, ab, aba, abab, ababa, \dots$	a	1
3	bab	$\Lambda, a, ab, aba, abab, ababa, \dots$	ab	2
4	baba	$\Lambda, a, ab, aba, abab, ababa, \dots$	aba	3
5	babac	$\Lambda, a, ab, aba, abab, ababa, \dots$	Λ	0
6	babaca	$\Lambda, a, ab, aba, abab, ababa, \dots$	a	1

```

1 // P: String of length m (pattern)
2 KMP::failureArray(P)
3     F[0] ← 0
4     j ← 1 // index within parsed text
5     l ← 0 // reached state
6     while j < m do
7         if P[j] = P[l]
8             l ← l + 1

```

```

9       F[j] ← ℓ
10      j ← j + 1
11      else if ℓ > 0
12          ℓ ← F[ℓ - 1]
13      else
14          F[j] ← 0
15          j ← j + 1

```

$F[j]$ is defined via pattern matching of P in $P[1..j]$ (built parts of F are used to expand it)

KMP Runtime

- *failureArray*
 - Consider how $2j - \ell$ changes in each iteration of the while loop:
 - * j and ℓ both increase by 1 ($2j - \ell$ increases) OR
 - * ℓ decreases ($2j - \ell$ increases) OR
 - * j increases ($2j - \ell$ increases)
 - Initially $2j - \ell \geq 0$, at the end $2j - \ell \leq 2m$
 - So no more than $2m$ iterations of the while loop gives runtime of $\Theta(m)$
- *KMP main function*
 - failureArray can be computed in $\Theta(m)$ time
 - Same analysis gives at most $2n$ iterations of the while loop since $2i - j \leq 2n$
 - The total runtime of KMP is $\Theta(n + m)$

Suffix Trees

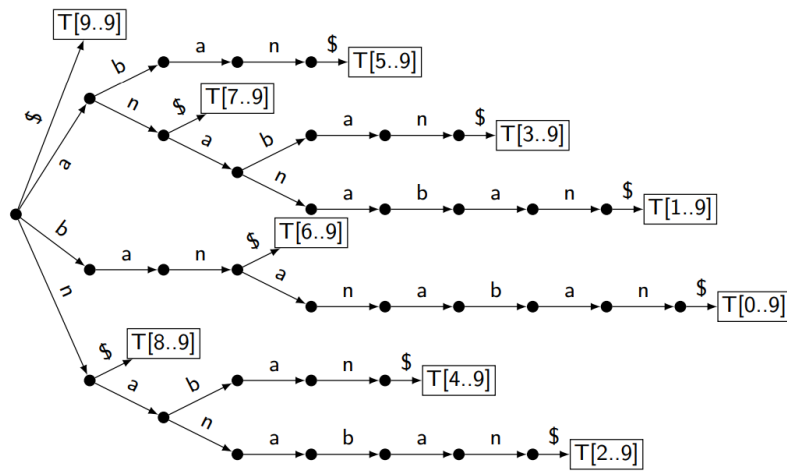
If we were looking to search for *many patterns* P within the same *fixed text* T it would make more sense to preprocess the text T rather than the pattern P .

Observation: P is a substring of T iff P is some prefix of some suffix of T

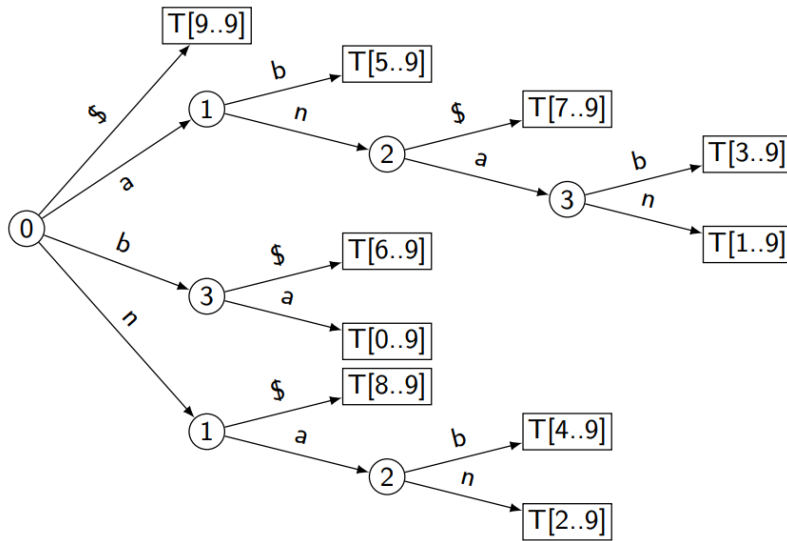
A *suffix tree* is a compressed trie of all suffixes of T

Example: $T = \text{bananaban}$ has suffixes $\{\text{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n, } \Lambda\}$

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



Suffix tree: compressed trie of suffixes



- Building:
 - Text T has n characters and $n + 1$ suffixes
 - Can build suffix tree by inserting each suffix of T into compressed trie: $\Theta(n^2|\Sigma|)$
 - There is a way to build a suffix tree in $\Theta(n|\Sigma|)$ time (complicated and beyond this course)
- Pattern Matching:
 - Essentially *search* for P in the compressed trie (runtime of $O(|\Sigma|m)$)
 - Some modification is needed as P may be some prefix of a stored word
 - * P is substring of T iff P is some prefix of some suffix of T

Although this is theoretically good the construction is slow, complicated, and lots of space-overhead.

Suffix Array

Relatively recent development (popularized in the 1990s)

TODO

String Matching Summary

TODO