Please initial:

Order Notation Summary

O-notation: $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $|f(n)| \leq c\,|g(n)|$ for all $n \geq n_0$.

Ω-notation: $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $c\,|g(n)| \leq |f(n)|$ for all $n \geq n_0$.

Θ-notation: $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that $c_1\,|g(n)| \leq |f(n)| \leq c_2\,|g(n)|$ 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 $|f(n)| \leq c\,|g(n)|$ 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 $c\,|g(n)| \leq |f(n)|$ for all $n \geq n_0$.

Some Recurrence Relations

Recursion	resolves to	example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify (*)
$T(n) = T(cn) + \Theta(n)$ for some $0 < c < 1$	$T(n) \in \Theta(n)$	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 (*)

- Once you know the result, it is (usually) easy to prove by induction.
- Many more recursions, and some methods to find the result, in CS341.

(*) These will be studied later in the course.

Useful Sums

Arithmetic sequence:

$_{i=0}^{n-1} i = ???$ $\quad$ $_{i=0}^{n-1}(a + di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2)$ if $d \neq 0$.

Geometric sequence:

$_{i=0}^{n-1} 2^i = ???$ $\quad$ $_{i=0}^{n-1} a\,r^i = \begin{cases} a\dfrac{r^n - 1}{r - 1} & \in \Theta(r^{n-1}) \quad \text{if } r > 1 \\ na & \in \Theta(n) \quad \text{if } r = 1 \\ a\dfrac{1 - r^n}{1 - r} & \in \Theta(1) \quad \text{if } 0 < r < 1. \end{cases}$

Harmonic sequence:

$_{i=1}^{n} \frac{1}{i} = ???$ $\quad$ $H_n := \; _{i=1}^{n} \frac{1}{i} = \ln n + \gamma + o(1) \in \Theta(\log n)$

A few more:

$_{i=1}^{n} \frac{1}{i^2} = ???$ $\quad$ $_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^2}{6} \in \Theta(1)$

$_{i=1}^{n} i^k = ???$ $\quad$ $_{i=1}^{n} i^k \in \Theta(n^{k+1})$ for $k \geq 0$

Lower bounds for sorting

We have seen many sorting algorithms:

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

Question: Can one do better than $\Theta(n \log n)$ running time?
Answer: Yes and no! *It depends on what we allow*.

- No: Comparison-based sorting lower bound is $\Omega(n \log n)$.
- Yes: Non-comparison-based sorting can achieve $O(n)$ (under restrictions!). $\quad \to$ see below

Please initial:

Analysis of Skip Lists

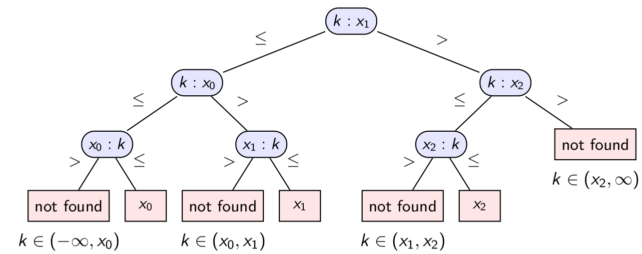
- Expected **space** usage: $O(n)$
- Expected **height**: $O(\log n)$
- Crucial for all operations:
  - How often do we *drop down* (execute $p \leftarrow p.below$)?
  - How often do we *step forward* (execute $p \leftarrow p.after$)?
- *skipList::search*: $O(\log n)$ expected time
  - # drop-downs = 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

Lower bound for search

The fastest realizations of *ADT Dictionary* require $\Theta(\log n)$ time to search among n items. Is this the best possible?

Theorem: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-n dictionary.

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



But can we beat the lower bound for special keys?

Interpolation Search

- Code very similar to binary search, but compare at interpolated index
- Need a few extra tests to avoid crash during computation of m.

```
interpolation-search(A, n, k)
A: Sorted array of size n, k: key
    ℓ ← 0, r ← n − 1
    while (ℓ ≤ r)
        if (k < A[ℓ] or k > A[r]) return "not found"
        if (k == A[r]) then return "found at A[r]"
        m ← ℓ + ⌊(k − A[ℓ]) / (A[r] − A[ℓ]) · (r − ℓ)⌋
        if (k == A[m]) then return "found at A[m]"
        else if (A[m] < k) then ℓ ← m + 1
        else r ← m − 1
    // We always return from somewhere within while-loop
```


Multiway Tries: Summary

- Operations *search*(x), *insert*(x) and *delete*(x) are exactly as for tries for bitstrings.
- Run-time $O(|x| \cdot (\text{time to find the appropriate child}))$

Each node now has up to $|\Sigma| + 1$ children. How should they be stored?

Solution 1: Array of size $|\Sigma| + 1$ for each node.
Complexity: $O(1)$ time to find child, $O(|\Sigma|)$ space per node.

Solution 2: List of children for each node.
Complexity: $O(|\Sigma|)$ time to find child, $O(\#\text{children})$ space per node.

Solution 3: Dictionary (AVL-tree?) of children for each node.
Complexity: $O(\log(\#\text{children}))$ time, $O(\#\text{children})$ space per node.
Best in theory, but not worth it in practice unless $|\Sigma|$ is huge.

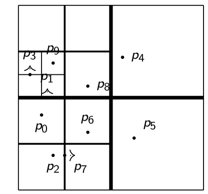
In practice, use *hashing* (keys are in (typically small) range Σ).

Please initial:

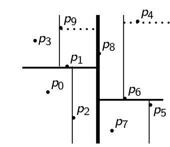
Complexity of chaining

- To analyze what happens 'on average', switch to *randomized* hashing.
- How can we randomize?
  Assume that the *hash-function* is chosen randomly.
- **Uniform Hashing Assumption**: U is finite and any possible hash-function is equally likely to be chosen as hash-function.
  (This is not at all realistic, but the assumption makes analysis possible.)
- Can show:
  - $P\Big(h(k) = i\Big) = \frac{1}{M}$ for any key k and slot i,
  - Hash-values of any two keys are independent of each other.

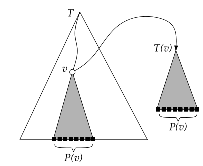
Range search data structures summary



- Quadtrees
  - simple (also for dynamic set of points)
  - work well only if points evenly distributed
  - wastes space for higher dimensions



- kd-trees
  - linear space
  - range search time $O(\sqrt{n} + s)$
  - inserts/deletes destroy balance and range search time (no simple fix)



- range-trees
  - range search time $O(\log^2 n + s)$
  - wastes some space
  - inserts/deletes destroy balance (can fix this with occasional rebuilt)

Convention: Points on split lines belong to right/top side.

Boyer-Moore Algorithm

```
Boyer-Moore::patternMatching(T,P)
1.   L ← lastOccurrenceArray(P)
2.   S ← good suffix array computed from P
3.   i ← m − 1,     j ← m − 1
4.   while i < n and j ≥ 0 do
          // current guess begins at index i − j
5.        if T[i] = P[j]
6.             i ← i − 1
7.             j ← j − 1
8.        else
9.             i ← i + m−1 − min{L[T[i]], j−1}
10.            j ← m − 1
11.  if j = −1 return "found at T[i+1..i+m]"
12.  else return FAIL
```

If good suffix heuristic is used, then line 9 should be

$$i \leftarrow i + m{-}1 - \min\{L[T[i]], S[j]\}$$

where S will be explained below.