

Algorithms and Data Structures

Optimal Search Trees; Tries

Ulf Leser

Content of this Lecture

- Optimal Search Trees
 - Definition
 - Construction
 - Analysis
- Searching Strings: Tries

Static Key Sets, Varying Access Frequencies

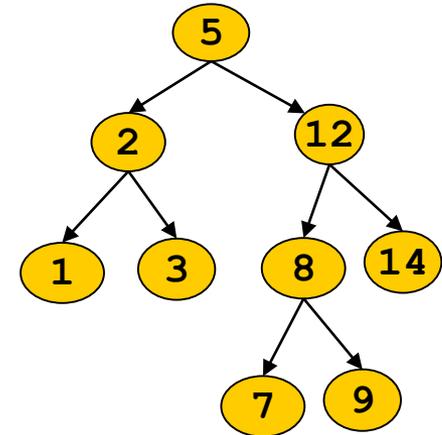
- Sometimes, the **set of keys is “fixed”**
 - Streets of a city, cities in a country, keywords of a prog. lang., ...
- Often, searches are much more frequent than updates
 - We may **spent more effort for reorganizing** the tree after updates
- Example: Large-scale web search engines
 - Recall: A search engine creates a dictionary; every word has a link to the set of documents containing it
 - The dictionary must be accessed very fast, changes are rare
 - Often, engines build complex structures to optimally support searching over the current set of documents **considered as static**
 - Defer updates: **Changes are buffered** and bulk-inserted periodically
 - Search either searches two data structures, or misses are accepted

Scenario

- Assume a set K of keys and a **bag R of requests (workload)**
 - Every request searches a $k \in K$; k 's may appear multiple times in R
 - In contrast to SOL, we now don't care about the order of requests
 - Like SOL with fixed access frequencies – but now we consider trees
- Naïve approach
 - Build an AVL tree over K
 - Every $r \in R$ costs $O(\log(|K|))$, i.e., we need **$O(|R| * \log(|K|))$**
 - This is optimal, if every $k \in K$ appears with the same frequency in R
- What if R is **highly skewed**?
 - Skewed: k 's are not equally distributed in R
 - Rather the norm than the exception in real life (Zipf, ...)
 - In contrast to SOL, finding an **optimal search tree for R** is not trivial

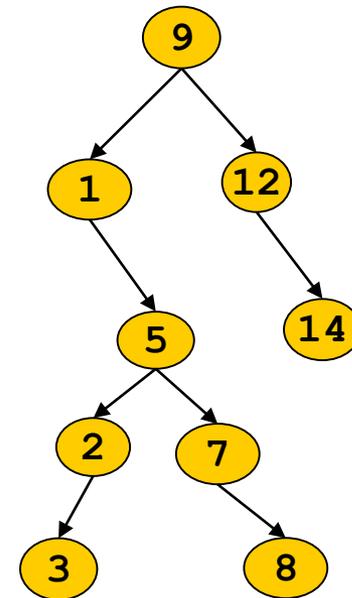
Example

- $K = \{1, 2, 3, 5, 7, 8, 9, 12, 14\}$
- We build an AVL tree
- $R_1 = \{2, 5, 8, 7, 3, 12, 1, 8, 8\}$
 - $2 + 1 + 3 + 4 + 3 + 2 + 3 + 3 + 3 = 31$ comparisons
- $R_2 = \{9, 9, 1, 9, 2, 9, 5, 3, 9, 1\}$
 - $4 + 4 + 3 + 4 + 2 + 4 + 1 + 3 + 4 + 3 = 32$ comparisons



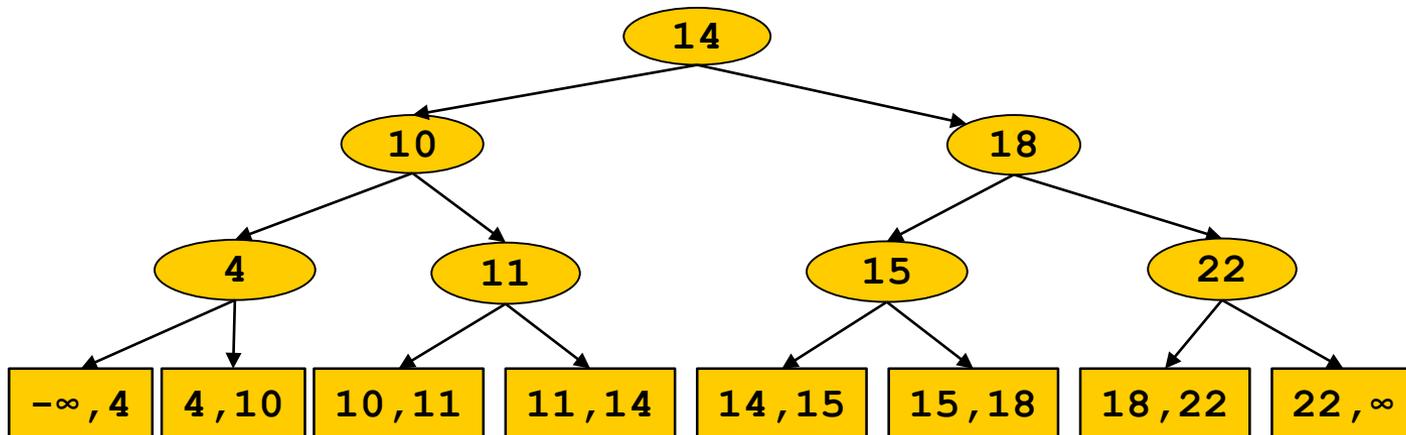
Example

- Let's **optimize the tree** for R_2
 - Not a AVL tree any more
- $R_2 = \{9, 9, 1, 9, 2, 9, 5, 3, 9, 1\}$
 $= \{9, 9, 9, 9, 9, 1, 1, 2, 5, 3\}$
 - 9 and 1 should be high in the tree
 - $1+1+1+1+1+2+2+4+3+5=21$
 - Versus 32
- Not good for R_1
 - $R_1 = \{2, 5, 8, 7, 3, 12, 1, 8, 8\}$
 - $4+3+5+4+5+2+2+5+5=35$
 - Versus 31
- Is this truly the **optimal search tree** for R_2 ?



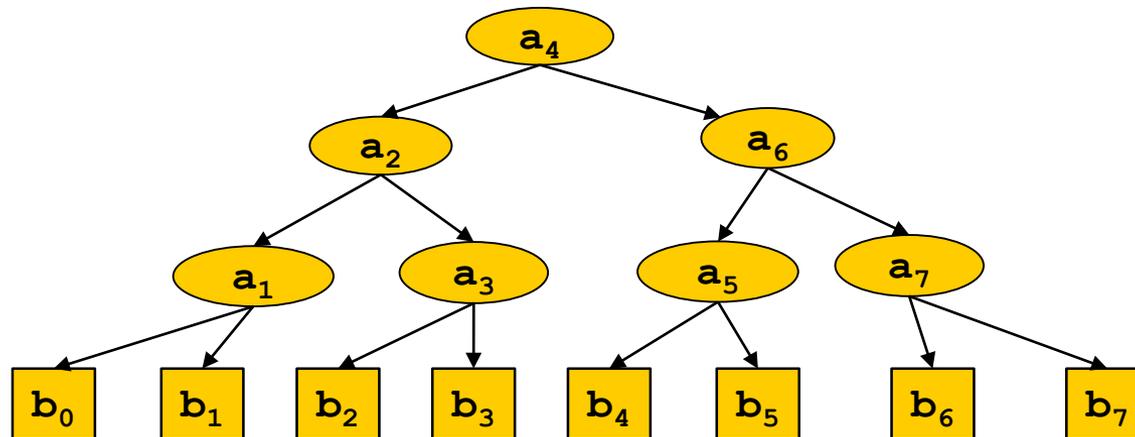
Request Model

- Assume an (ordered) set K of keys, $K = \{k_1, k_2, \dots, k_n\}$
- Every k is searched with frequency a_1, a_2, \dots, a_n
- **No-key intervals** $]-\infty, k_1[$, $]k_1, k_2[$, \dots , $]k_{n-1}, k_n[$, $]k_n, +\infty[$ are searched with frequencies b_0, b_1, \dots, b_n
 - We need to consider costs of searches that fail
- Together: $R = \{a_1, a_2, \dots, a_n, b_0, b_1, \dots, b_n\}$



Request Model

- Assume an (ordered) set K of keys, $K = \{k_1, k_2, \dots, k_n\}$
- Every k is searched with frequency a_1, a_2, \dots, a_n
- **No-key intervals** $]-\infty, k_1[$, $]k_1, k_2[$, \dots , $]k_{n-1}, k_n[$, $]k_n, +\infty[$ are searched with frequencies b_0, b_1, \dots, b_n
 - We need to consider costs of searches that fail
- Together: $R = \{a_1, a_2, \dots, a_n, b_0, b_1, \dots, b_n\}$



Optimal Search Trees

- Definition

*Let T be a search tree for K and R a workload. The **cost** $P(T)$ of T for R is defined as*

$$P(T) = \sum_{i=1}^n (\text{depth}(k_i) + 1) * a_i + \sum_{j=0}^n (\text{depth}(]k_j, k_{j+1}[) + 1) * b_j$$

- Definition

*Let K be a set of keys and R a workload. A search tree T over K is **optimal for R** iff*

$$P(T) = \min\{P(T') \mid T' \text{ is search tree for } K\}$$

One More Definition

- Definition

Let T be a search tree over K and R a workload. The weight $W(T)$ of T for R is:

$$W(T) = \sum_{i=1}^n a_i + \sum_{j=0}^n b_j$$

- Thus, the weight of T is simply $|R|$
- We will need this [definition for subtrees](#)

Content of this Lecture

- Optimal Search Trees
 - Definition
 - Construction
 - Analysis
- Searching Strings: Tries

Finding the Optimal Search Tree

- Bad news: There are **exponentially many search trees**
 - We cannot enumerate all search trees, compute their cost, and then choose the cheapest
 - Proof omitted
- Good news: We don't need to look at all possible search trees
 - We can use a divide & conquer approach
 - **Dynamic programming**: Build large solutions from smaller ones
 - Recall max_subarray etc.
 - Here: Build larger optimal search trees from smaller optimal STs

General Idea

- Observation: We can **define $P(T)$ recursively**
 - Let k_r be root of T and $T_{lr} = \text{leftChild}(k_r)$, $T_{rr} = \text{rightChild}(k_r)$
 - “lr: Left-of-r”; “rr: Right-of-r”
 - Clearly:
$$P(T) = P(T_{lr}) + P(T_{rr}) + a_r + W(T_l) + W(T_{rr})$$
$$= P(T_{lr}) + P(T_{rr}) + W(T)$$
 - Since $W(T)$ is the same for every possible search tree, the cost of a tree only depends on the cost of its subtrees
- Problem: We do not know k_r , but we **need to find it**
 - k_r divides T into a left part (T_{lr}) and a right part (T_{rr})
 - Both T_{lr} and T_{rr} are smaller than T
 - Assume we knew $P(T_{lr})$ and $P(T_{rr})$ for every possible k_r
 - Both are smaller, so we can compute T_l/T_r values bottom-up
 - We can test all n different k_r 's and find the one maximizing the term $P(T_{lr}) + P(T_{rr}) + W(T)$

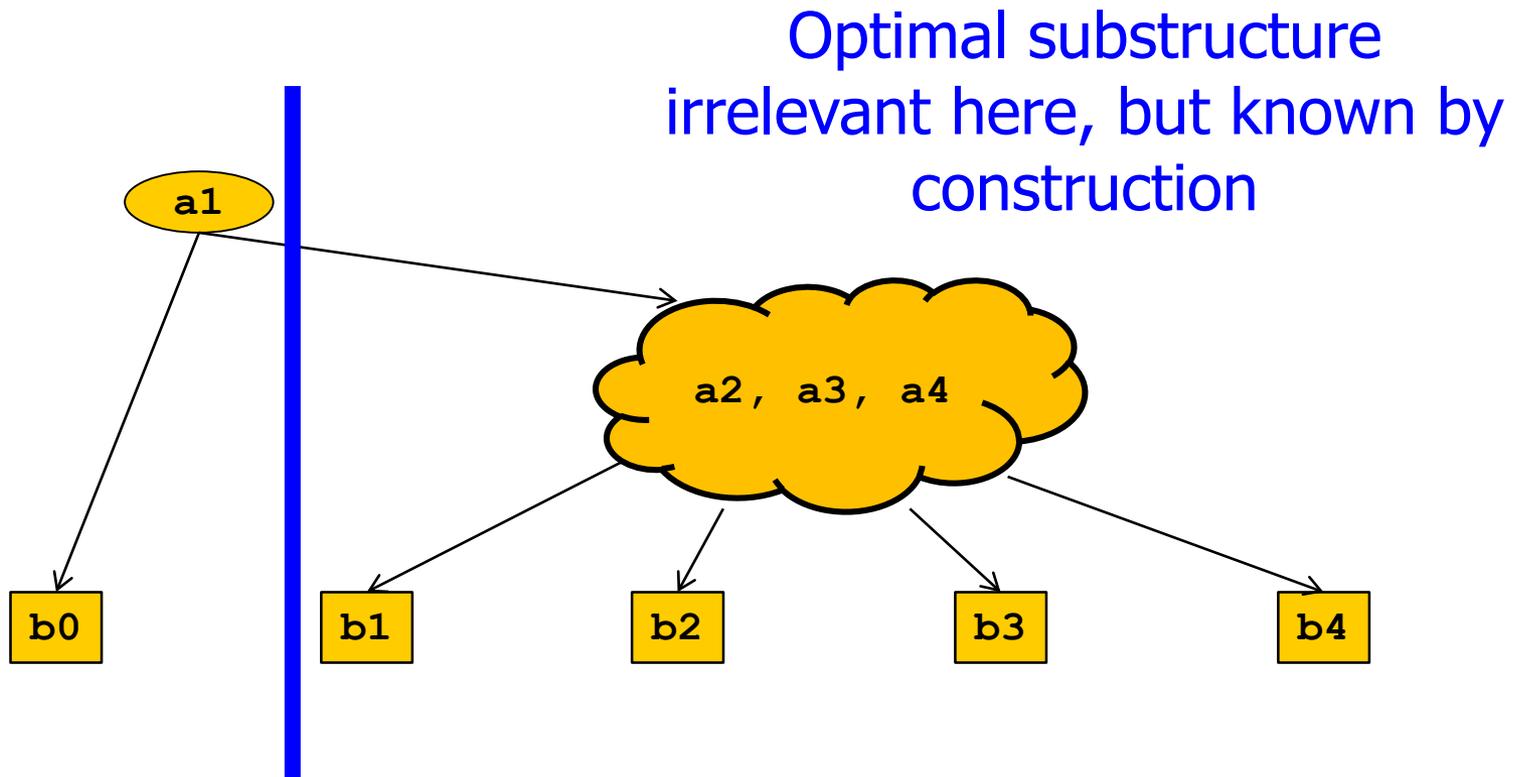
Example

- We want to compute the optimal search tree T for the keys a_1 - a_4 and no-key ranges b_0 - b_5
- One of the keys a_1, a_2, a_3, a_4 , must be the root



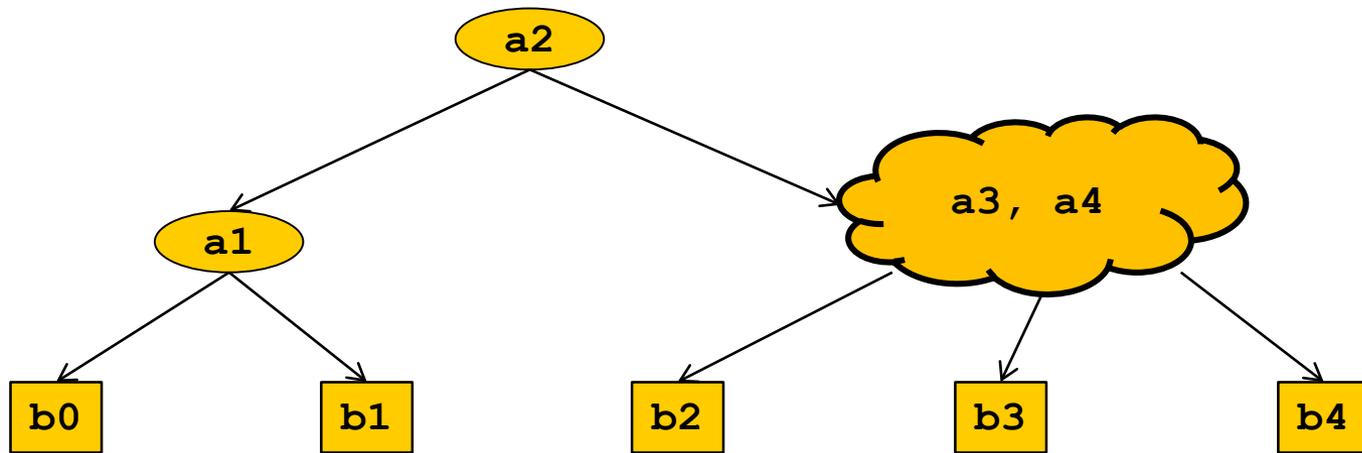
Example Continued

- If a_1 would be the “optimal root”, the cost of $P(T)$ would be $P(b_0) + P(b_1 \dots b_4) + W(T)$



Example Continued

- If a_2 would be the “optimal root”, the cost of $P(T)$ would be $P(b_0..b_1) + P(b_2..b_4) + W(T)$



Formal: A Divide & Conquer Approach

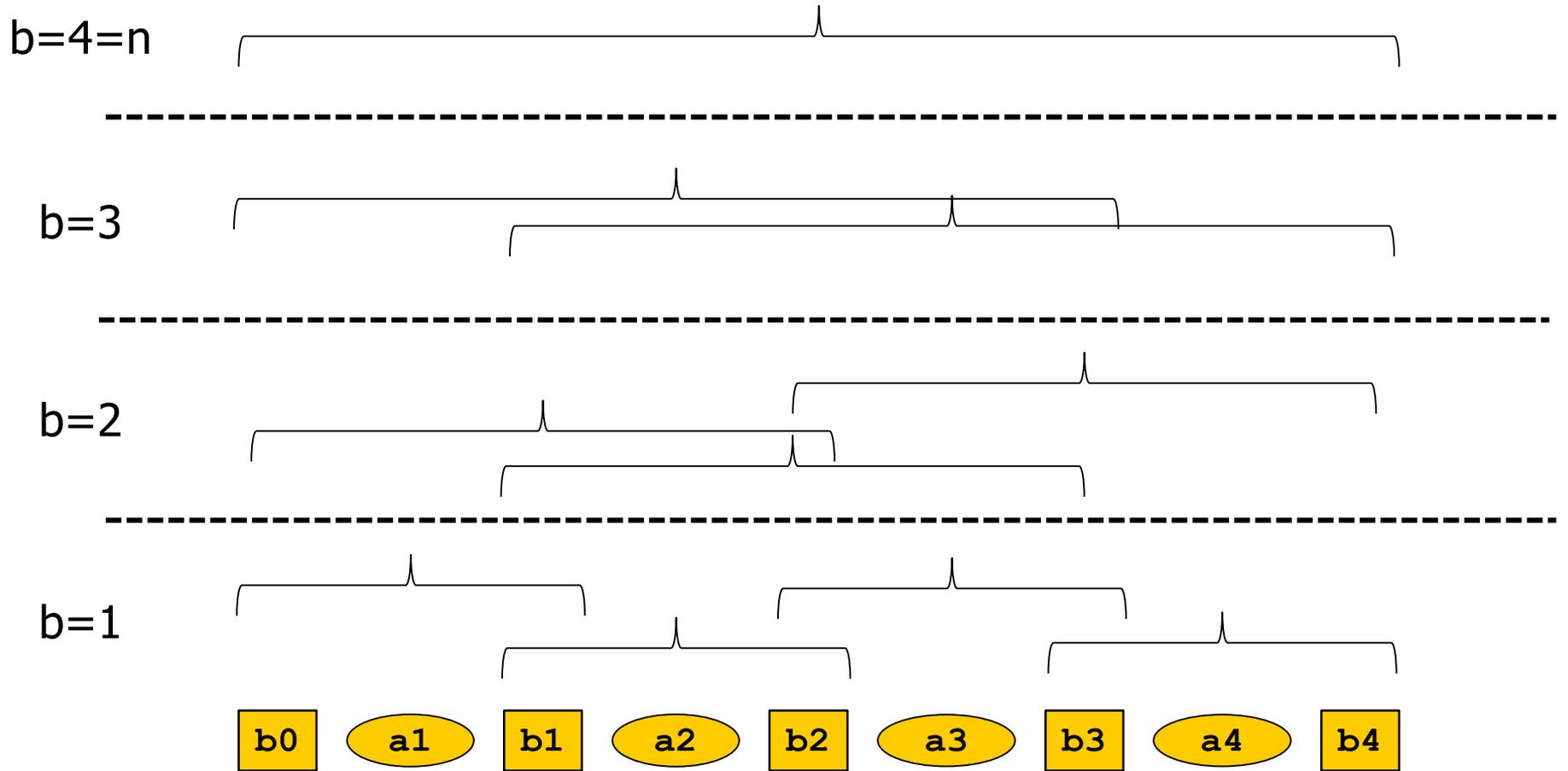
- Consider a **range $R(i,j)$ of keys and intervals**
 - $R(i,j) = \{]k_i, k_{i+1}[, k_{i+1},]k_{i+1}, k_{i+2}[, k_{i+2}, \dots, k_j,]k_j, k_{j+1}[\}$
- Assume that $R(i,j)$ is represented as subtree $T(i,j)$ of $T(1,n)$
 - That's not the case in all topologies for T ; the "left" part of R could lie in a different subtree than the "right" part
- One of the **$k_r \in R(i,j)$ must be the root** of this subtree
- Thus, k_r divides $R(i,j)$ in two halves $R(i,r-1)$, $R(r,j)$
- Assume we know the optimal trees for **all sub-ranges** $R(i,i+1)$, $R(i,i+2)$, ..., $R(i,j-1)$, $R(i+1,j)$, ..., $R(j-1, j)$
- Then, **we find the r creating the** optimal tree $T(i,j)$ using

$$P(T(i, j)) = W(T(i, j)) + \min_{r=i+1..j} (P(T(i, r-1)) + P(T(r, j)))$$

Bottom-Up Computation

- We **systematically enumerate** smaller $R(i,j)$ and puzzle them together to larger ones
- Let $P(i,j)$ be the cost of the optimal search tree for $R(i,j)$
- To compute $P(i,j)$, we (1) need the P and W -values of all possible enclosed subtrees and we (2) need to find the optimal value of r
- We perform **induction over the breadth b** of intervals: All intervals of breadth $0, 2 \dots n$ (and we are done)
 - Breadth of an interval: Number of keys contained

Illustration

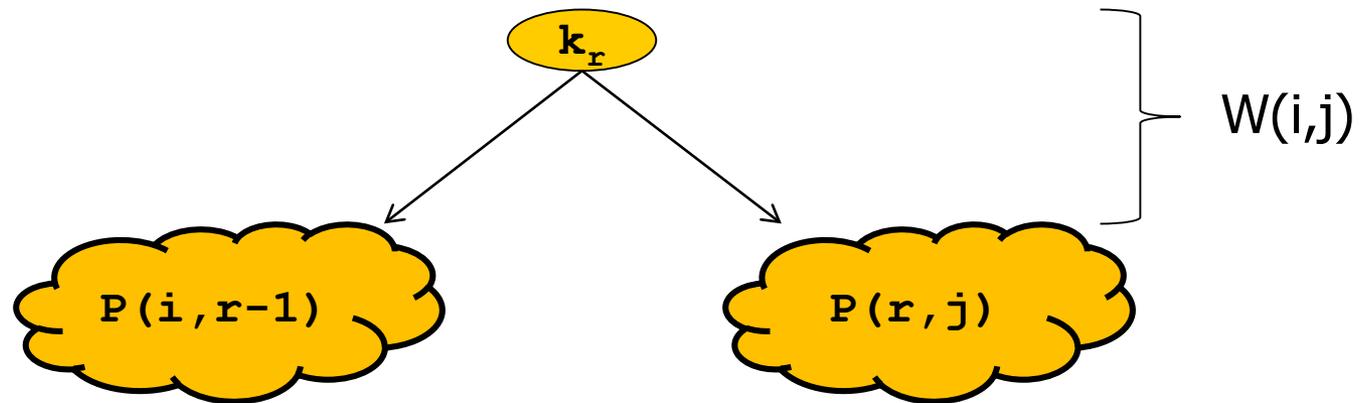


Induction Start

- $b=0$; all subintervals (i,i)
 - This is a leaf (an interval without keys), no root selection required
 - $\forall 0 \leq i < n+1$: $W(i,i) = b_i$
 $P(i,i) = W(i,i)$
- $b=1$; all subintervals $(i,i+1)$
 - The root is always k_{i+1}
 - The only key in this interval; $l=i+1$
 - $\forall 0 \leq i < n$: $W(i,i+1) = b_i + a_{i+1} + b_{i+1}$
 $P(i,i+1) = P(i,i) + W(i,i+1) + P(i+1,i+1)$

Induction

- General case: $b > 1$, subintervals (i, j) with $j - i = b > 1$
 - Induction hypothesis: We know W, P for all intervals of **breadth** $< b$
 - Find the **index r for the optimal root** of the subtrees
 - Then compute:
$$W(i, j) = W(i, r-1) + a_i + W(r, j)$$
$$P(i, j) = P(i, r-1) + W(i, j) + P(r, j)$$

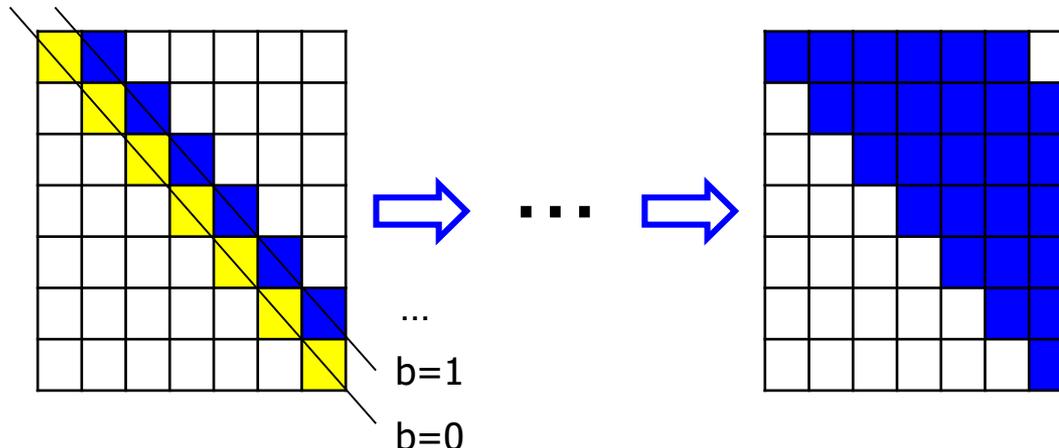


Content of this Lecture

- Optimal Search Trees
 - Definition
 - Construction
 - Analysis
- Searching Strings: Tries

Implementation

- There are only $(n+1) \cdot (n+1)$ different pairs i, j
- We essentially fill a **quadratic matrix** of size $(n+1) \cdot (n+1)$ for W and **one for P**
 - Since $j \geq i$, we actually only need half of each matrix
- Both matrixes are iteratively filled **from the main diagonal to the upper-right corner**



Analysis

- Space
 - We need 2 arrays of size $O(n*n)$
 - Space complexity: $O(n^2)$
- Time
 - Cases $b=0$ and $b=1$ are $O(n)$
 - We enumerate breadths from 2 to n
 - For each b , we consider all possible start positions: $O(n-b)$ many
 - In each range, we need to find the optimal l – this is $O(b)$
 - A range has max size $n-1$
 - Together: $O(n^3)$

```
1. initialize W(i,i);
2. initialize P(i,i);
3. initialize W(i,i+1);
4. initialize P(i,i+1);
5. for b = 2 to n do
6.   for i = 0 to (n-b) do
7.     j := i+b;
8.     find optimal l in [i,j];
9.     W(i,j) := ...
10.    P(i,j) := ...
11.   end for;
12. end for;
```

Constructing the tree

- We only showed how to compute the cost of the optimal tree, but **not how to build the tree itself**
- But this is simple since we never revise decisions
- We can “grow” the tree whenever we have computed a new optimal root l
- For instance, we can define a **$r(i,j):=l$ in every step**; the sequence of computed l -values fully determine the tree

Relevance

- Nice and instructive
- Runtime can actually be reduced to $O(n^2)$
- But: $O(n^2)$ is still **quite expensive** for large n
- Fortunately, one can compute „almost“ optimal search trees in linear time
 - Not this lecture

Content of this Lecture

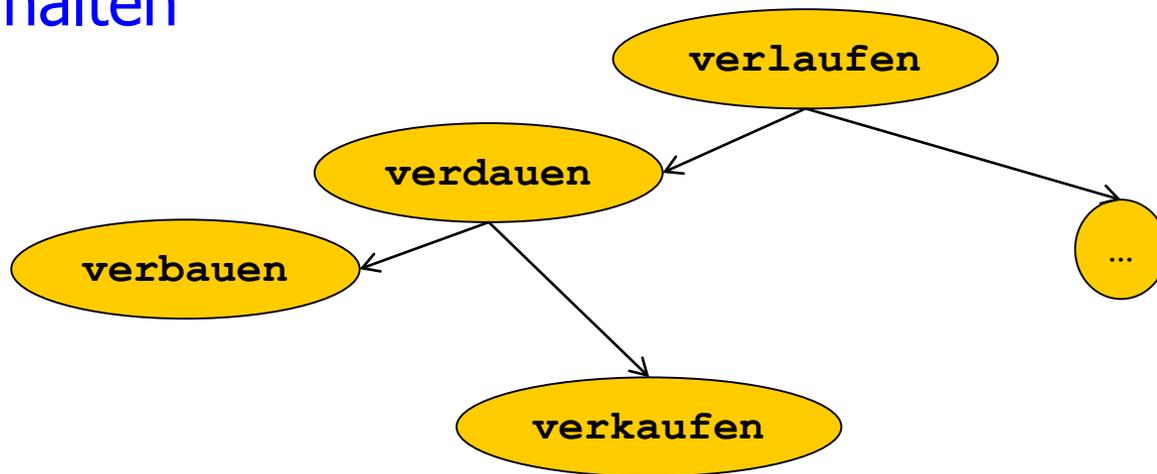
- Optimal Search Trees
- Searching Strings: Tries

Keys that are Strings

- Assume K is a **set of strings** of maximal length m
- We can build an AVL tree over K
- Searching requires $O(\log(n))$ key comparisons
- But: Each **string-comp** requires m **char-comps** in WC
 - Very pessimistic, but we do WC analysis
- Together: We need **$O(|k| * \log(n))$ character comparisons** for searching a key k
- Observation
 - “Similar” strings will be close neighbors in the tree
 - These will **share prefixes** (the longer, the more similar)
 - These prefixes are **compared again and again**

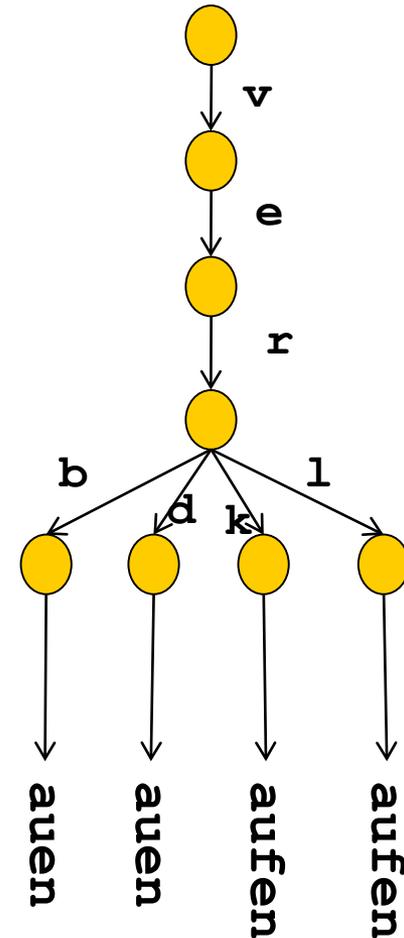
Example

k=„verhalten“



Tries

- Tries are **edge-labeled trees** of order $|\Sigma|$
 - Developed for Information Retrieval
- Edges are labeled with chars from Σ
- Idea: **Common prefixes** of keys are represented only once
- Problem: If “verl” is a key?
 - Trick: Add a “\$” (not in Σ) to every string
 - Then every and **only leaves** represent keys



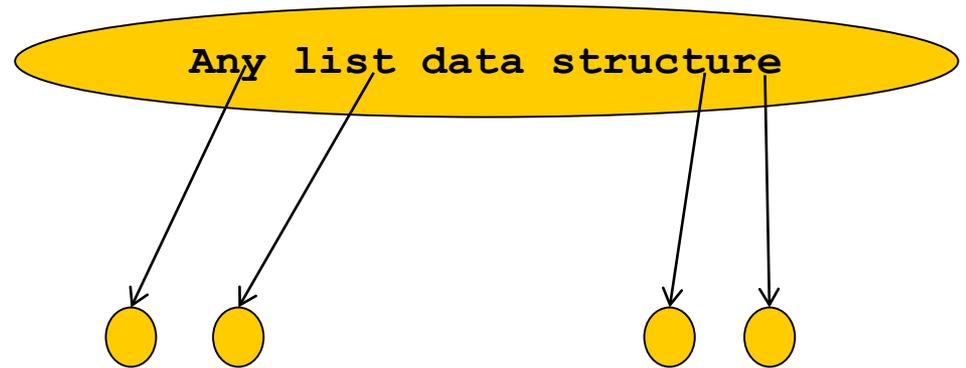
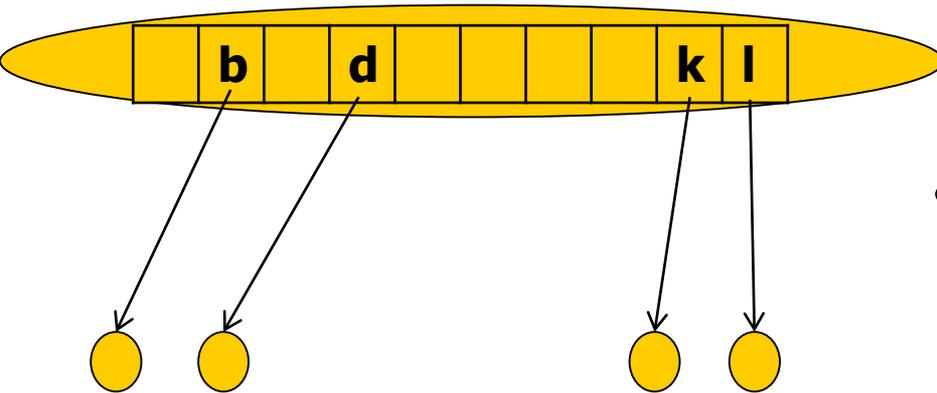
Analysis

- Construction of a trie over K ?
 - Let $\text{len}(K)$ be the sum of all key lengths in K
 - We start with an empty tree and **iteratively add** all $k \in K$
 - To add a key k , we **char-match k in the tree** as long as possible
 - As soon as no continuation is found, we build a new branch
 - This requires $O(|k|)$ operations (char-comps or node creations)
 - It follows: **Construction is in $O(\text{len}(K))$**
- Searching a key k (which maybe in K or not in K)
 - We match k from root down the tree
 - When k is exhausted and we are in a leaf: $k \in K$
 - If no continuation is found or we end in an inner node: $k \notin K$
 - It follows: **Searching is in $O(|k|)$**
 - But ...

Space Complexity

- We have at most $\text{len}(K)$ edges and $\text{len}(K)+1$ nodes
 - Shared prefixes make the actual number smaller
- But we also need **pointer to children**
- To achieve our search complexity, **choosing the right pointer** must be in $O(1)$
- This adds $O(\text{len}(K)*|\Sigma|)$ pointers
- Too much for any non-trivial alphabet
 - **Digital tries** are a popular data structure in coding theory
 - There, $|\Sigma|=2$, so the pointers don't matter much
 - But beware – the trees get very deep
- Furthermore, most of the pointers will be null
 - Depending on $|\Sigma|$, $|K|$, and lengths of shared prefixes

Alternatives

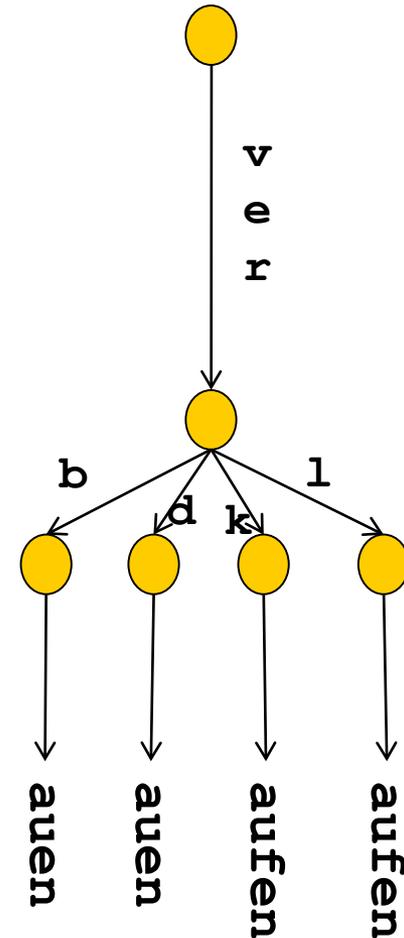


- Full array for children ptr
- Advantage: $O(|k|)$ search
- Disadvantage: Excessive space consumption

- Dense array for children ptr
- Advantage: $O(\text{len}(K))$ space
- Disadvantage: Search is $O(|k| * \log(|\Sigma|))$

Compressed Tries = Patricia Trees

- We can save further space
- A **patricia tree** (or radix tree) is a trie where edges are labeled with (sub-)strings, not with characters
- All sequences $S = \langle \text{node}, \text{edge} \rangle$ which do not branch are **compressed into a single edge** labeled with the concatenation of the labels in S
- More compact, less pointer
- Slightly more complicated implementation
 - E.g. insert requires splitting of labels



Exemplary Questions

- Recall the definition of a trie. Give an implementation (in pseudo code) for (a) searching a key k and (b) building a trie for a string set K . You may presuppose a data structure „list“ with operations $\text{add}(c, p)$ for adding a pair of character and pointer and $\text{retrieve}(c)$, which returns the pointer associated to c or nil .
- Build an optimal search tree for $K=\{5,12,15,20\}$ and $R=\{6,2,3,8,11,5,2,1,4\}$. Show the complete tables for W and P
- Prove that all tries for any permutation of a set of strings are identical