Algorithms and Data Structures

Optimal Search Trees; Tries

Ulf Leser

Static Key Sets

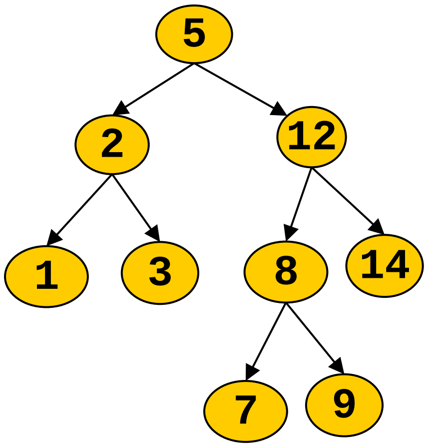
- Sometimes, the set of keys is “fixed”
  - Names in a city, cities in a country, elements of a prog. language ...
- “Fixed”: Searches are much more frequent than changes
  - We may spent more effort for reorganizing the tree after updates
- Not unusual – example large-scale search engines
  - Recall: 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
  - Changes are buffered and bulk-inserted periodically

Scenario

- Assume a set K of keys and a bag R of requests
  - Every search searches one k∈K; k's appear multiple times in R
  - In contrast to SOL, we here don't care about the order of requests to make life simpler (like SOL with fixed access probabilities)
- Naïve approach
  - Build an AVL tree over K
  - Every r∈R costs O(log(|K|)), i.e., we need O(|R|*log(|K|))
  - This is optimal, if every k∈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

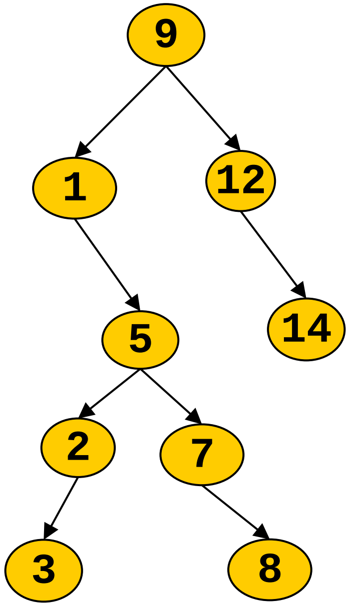
- K={1,2,3,5,7,8,9,12,14}
- We build an AVL tree



- 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



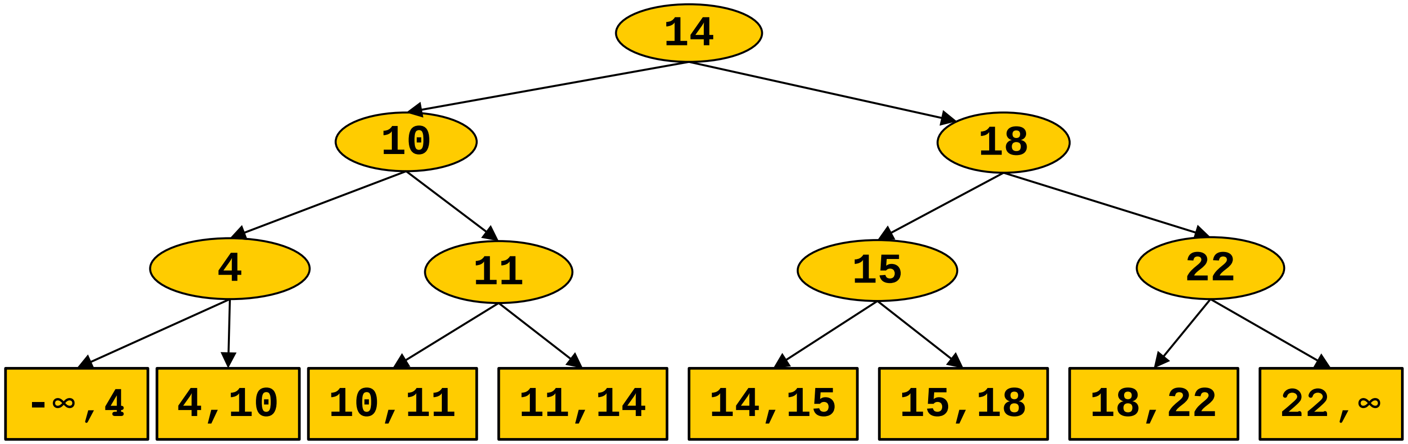
- $R_2=\{9,9,1,9,2,9,5,3,9,1\}$
  $=\{9,9,9,9,9,1,1,2,5,3\}$
  - 9 and 1 should be high in the tree
  - $1+1+1+1+1+2+2+4+3+5=21$
    - Versus 32
- Not good for R_1
  - $R_1=\{2,5,8,7,3,12,1,8,8\}$
  - $4+3+5+4+5+2+2+5+5=35$
    - Versus 31

Content of this Lecture

- Optimal Search Trees
- Construction of Optimal Search Trees
- Searching Strings: Tries

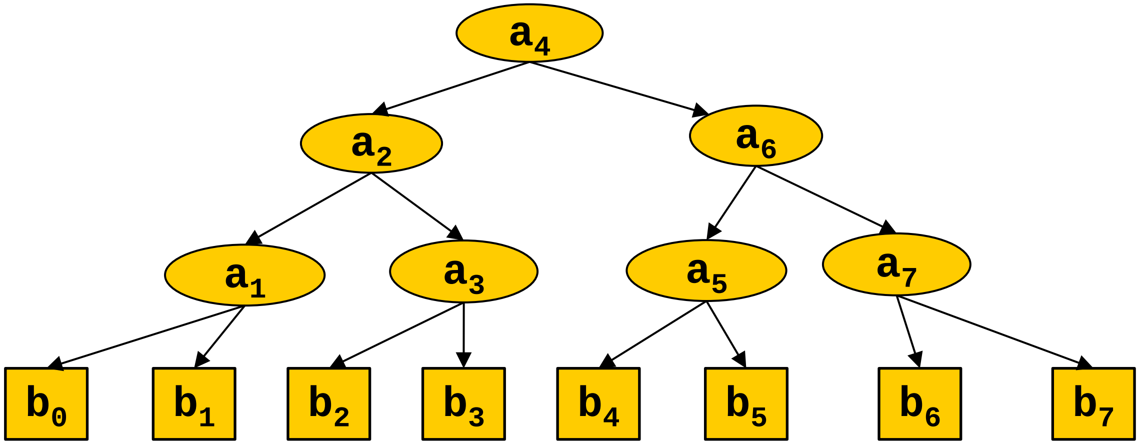
Setting

- Assume a (ordered) set K of keys, $K=\{k_1, k_2, \ldots, k_n\}$
- Every k is searched with frequency $a_1, a_2, \ldots, a_n$
- Intervals $]-\infty,k_1[$, $]k_1,k_2[, \ldots,]k_{n-1},k_n[,]k_n,+\infty[$ are searched with frequencies $b_0, b_1, \ldots, b_n$
  - Searches that fail
- We summarize these as $R=\{a_1, a_2, \ldots, a_n, b_0, b_1, \ldots, b_n\}$



Setting

- Assume a (ordered) set K of keys, $K=\{k_1, k_2, ..., k_n\}$
- Every k is searched with frequency $a_1, a_2, ..., a_n$
- Intervals $]-\infty,k_1[$, $]k_1,k_2[$, ..., $]k_{n-1},k_n[$, $]k_n,+\infty[$ are searched with frequencies $b_0, b_1, ..., b_n$
  - Searches that fail
- We summarize these as $R=\{a_1, a_2, ..., a_n, b_0, b_1, ..., b_n\}$



Optimal Search Trees

- Definition
  Let T be a search tree over K. The cost of T for R is:

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

- Definition
  Let K be a set of keys and R a set of requests. A search tree T over K is optimal for R iff

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

One More Definition

- Definition
  Let T be a search tree over K. The weight 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
- Construction of Optimal Search Trees
- Searching Strings: Tries

Finding the Optimal Search Tree

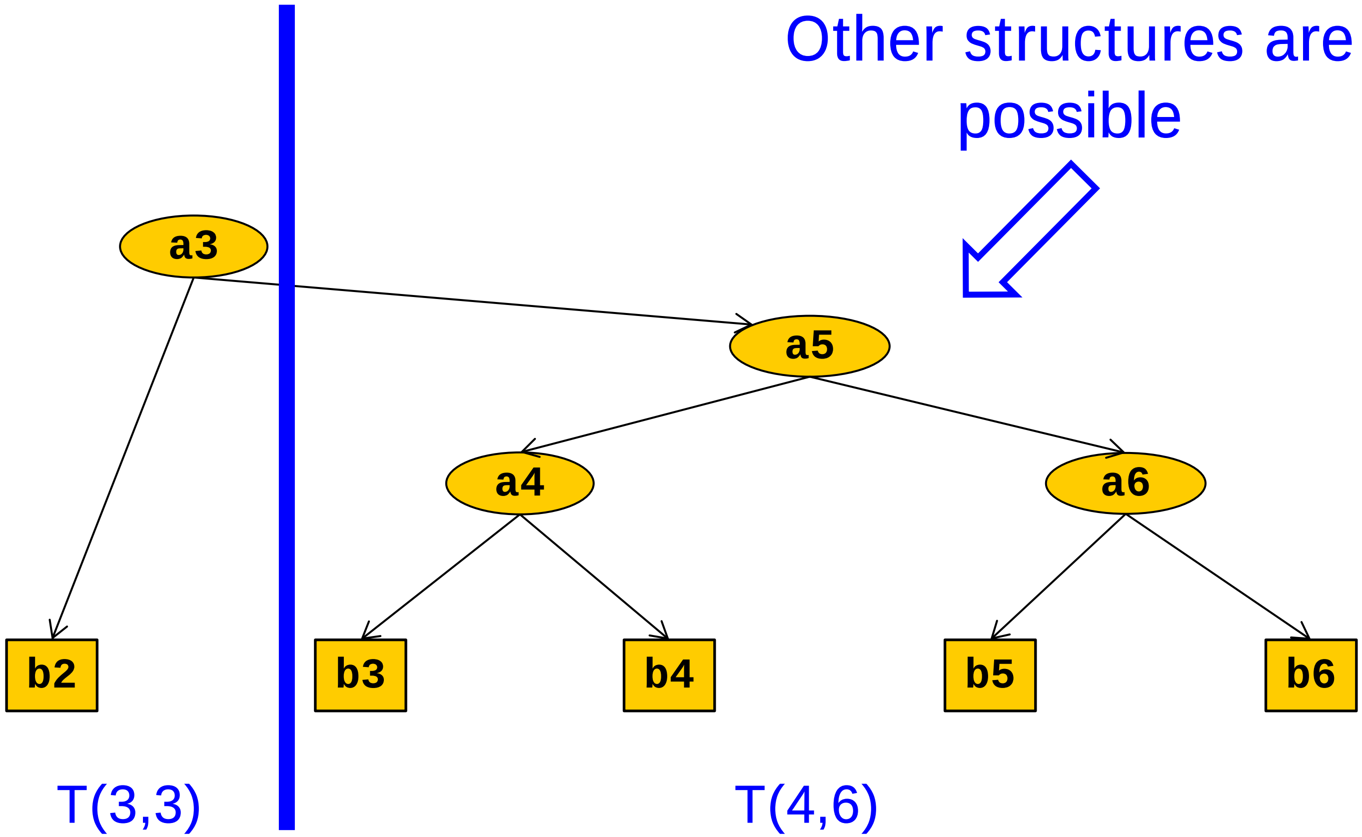
- Bad news: There are exponentially many search trees
  - Proof omitted
- Implication: We cannot enumerate all search trees, compute their cost, and then choose the cheapest
- Good news: We don't need to look at all possible search trees
- We can use a divide & conquer approach

Towards Divide & Conquer

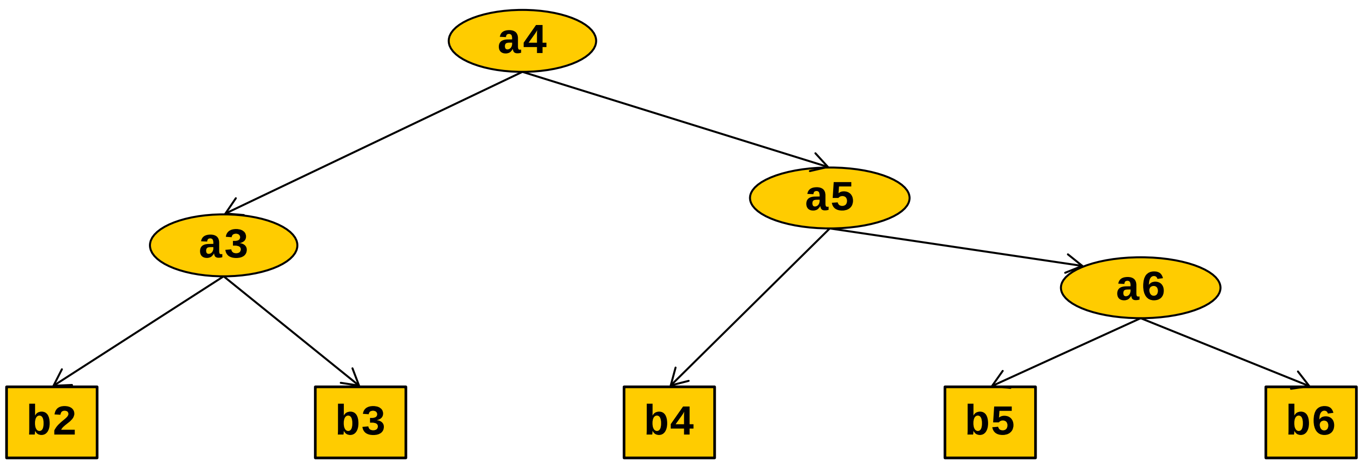
- We can compute P(T) recursively
- Let k_r be root of T and T_l=leftChild(k_r), T_r=rightChild(k_r)
- Clearly: $P(T) = P(T_l) + P(T_r) + a_r + W(T_l) + W(T_r)$
  $= P(T_l) + P(T_r) + W(T)$
- Since W(T) is the same for every possible search tree, the cost of a tree only depends on the cost of its subtrees
- It follows: T is optimal iff T_l and T_r are optimal
- Great! If we can solve the problem for smaller trees (=ranges of keys), we can inductively construct solutions for larger trees

Illustration

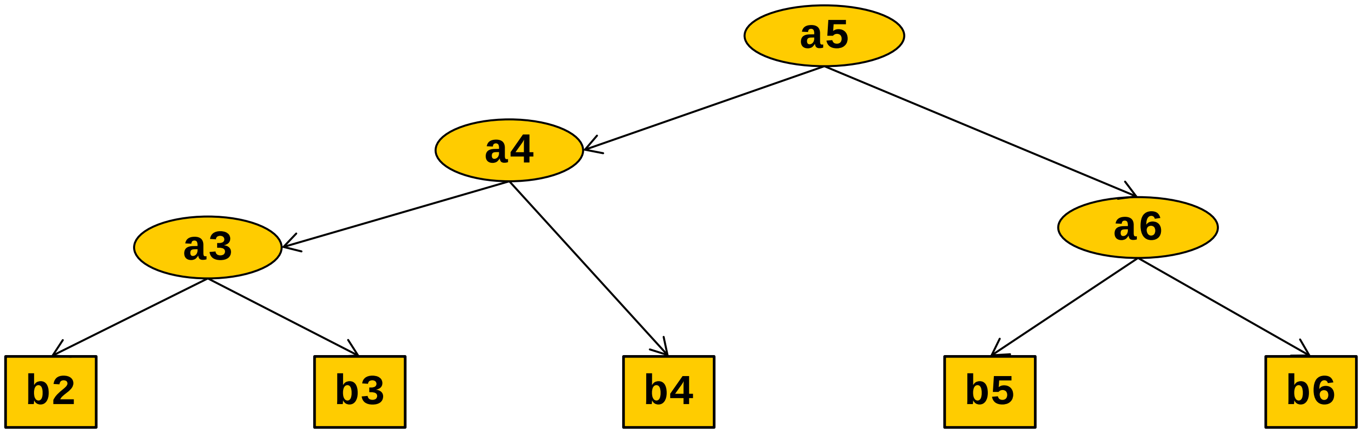
Illustration



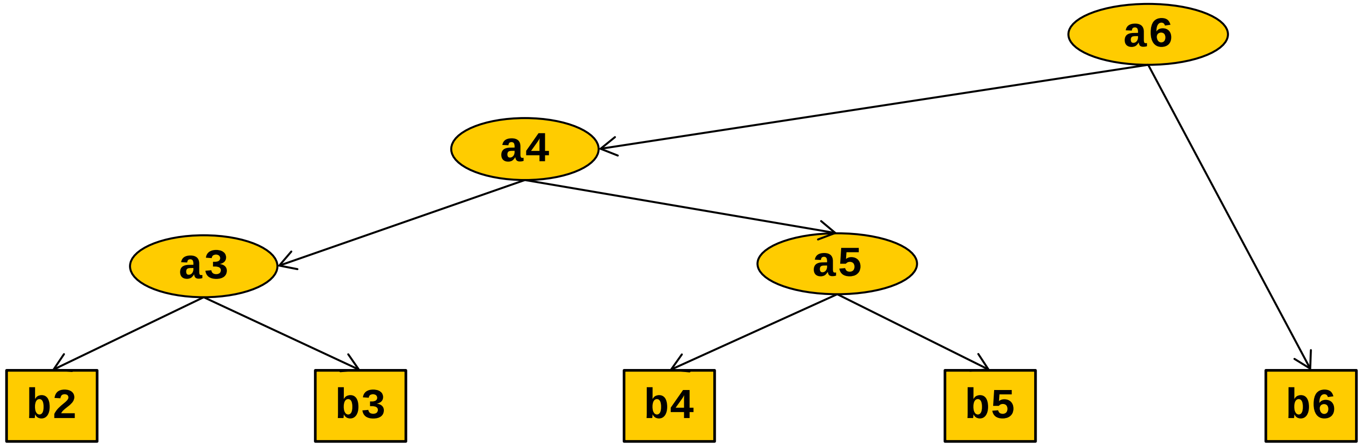
Illustration



Illustration



Illustration

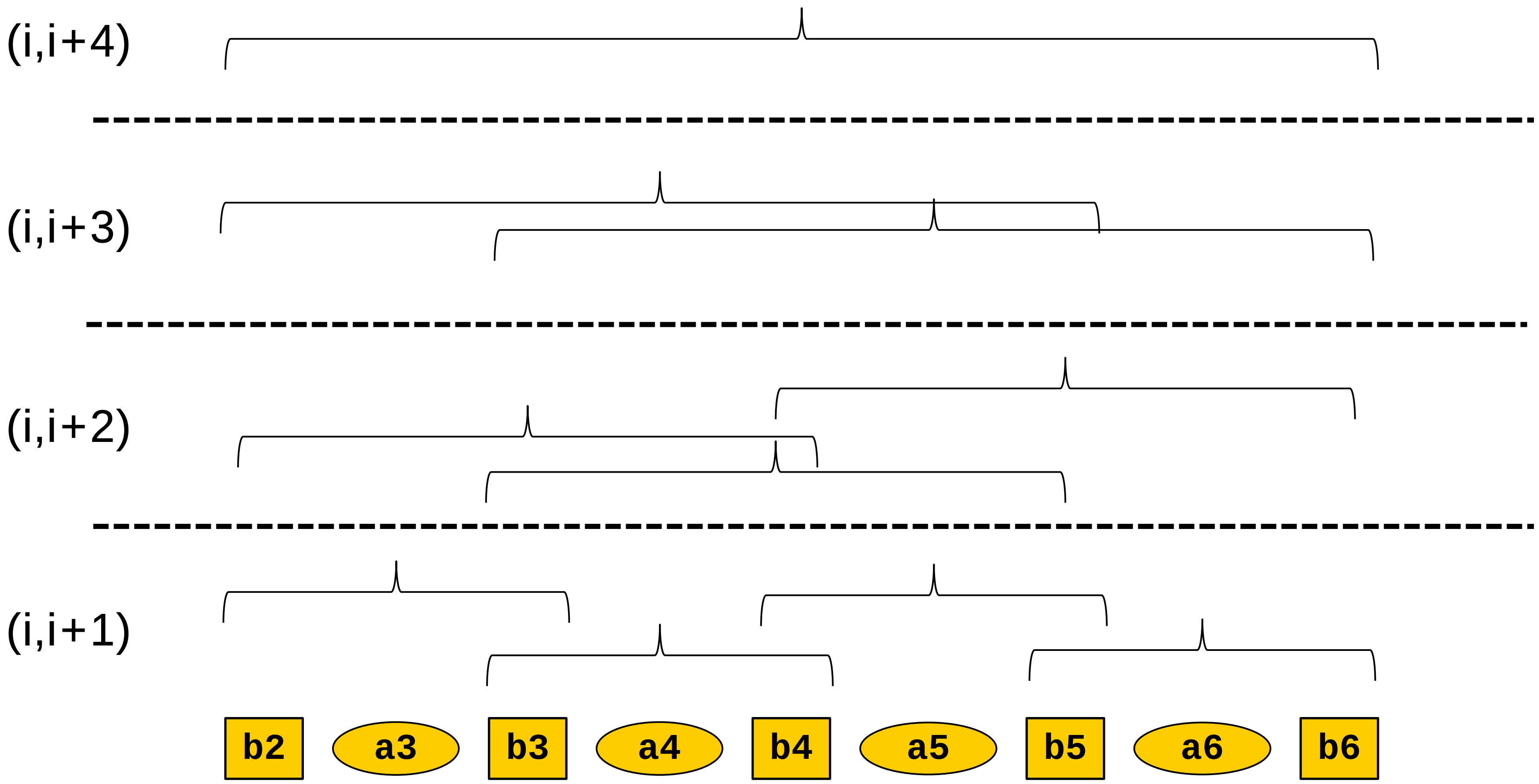


Divide & Conquer

- 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}, ... k_j, $]k_j,k_{j+1}[$ }
- Assume that R(i,j) is represented as subtree T(i,j) of T(1,n)
  - Need not be the case in general; the "left" part of R could lie in a different subtree than the "right" part
- One of the $k_l \in R(i,j)$ must be the root of this subtree
- Thus, k_l divides R(i,j) in two halves R(i,l-1), R(l,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 can find l and the optimal tree T(i,j) in O(j-i)

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

Illustration



Bottom-Up

- We must systematically enumerate smaller T(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 need the P and W-values of enclosed subtrees and we need to find l
- We perform induction over the breadth b of intervals: All intervals of breadth 1, 2 ... n (and we are done)

Induction Start

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

Induction

- General case: $b>1$, subintervals (i,j) with $j-i=b>1$
  - By induction assumption, we know W and P values for all intervals of breadth $\leq b-1$
  - Find the index l for the optimal root of the subtree
  - Then compute

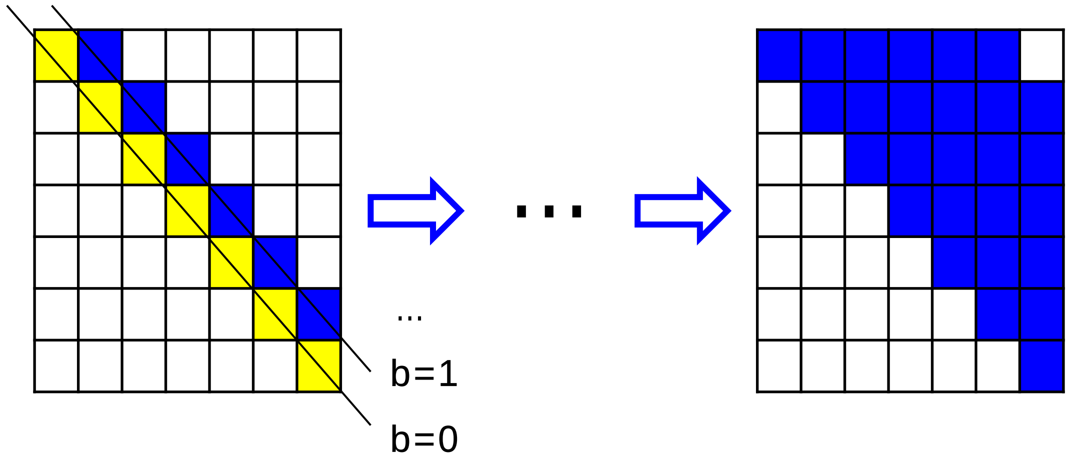
$$W(i,j) = W(i,l-1) + a_l + W(l,j)$$

$$P(i,j) = P(i,l-1) + W(i,j) + P(l,j)$$

- Done

Implementation

- There are only (n+1)*(n+1) different pairs i,j
- We need one two-dimensional quadratic matrix of size (n+1)*(n+1) for W and one for P
  - Since j>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: $O(n^3)$
  - [Can be improved to $O(n^2)$]

```
initialize W(i,i);
initialize P(i,i);
initialize W(i,i+1);
initialize P(i,i+1);
for b = 2 to n do
  for i = 0 to (n-b) do
    j := i+b;
    find optimal l in [i,j];
    W(i,j) := …
    P(i,j) := …
  end for;
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
- But: $O(n^2)$ is quite expensive for any n where such algorithms make sense
- Fortunately, we can compute „almost“ optimal search trees in linear time
  - Not this lecture

Content of this Lecture

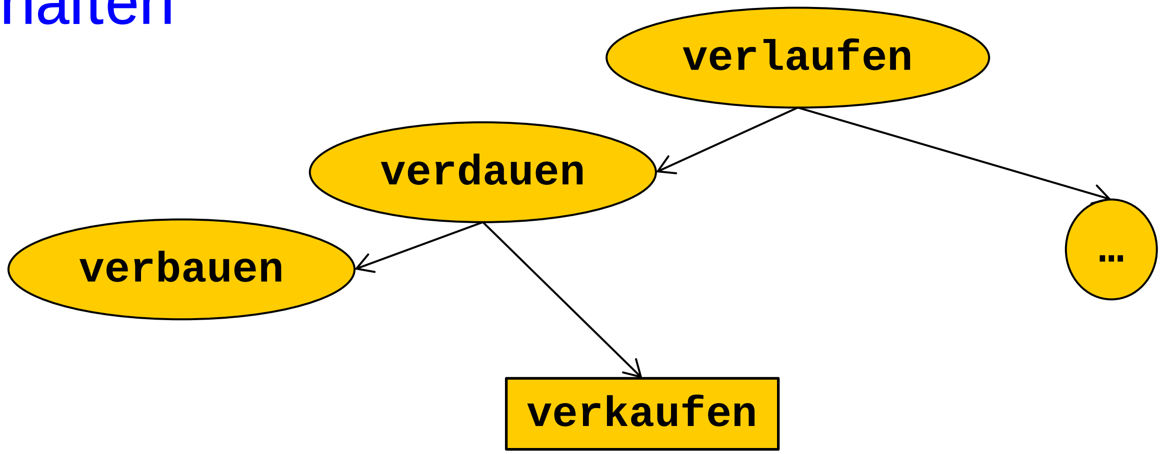
- Optimal Search Trees
- Construction of Optimal Search Trees
- Searching Strings: Tries

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

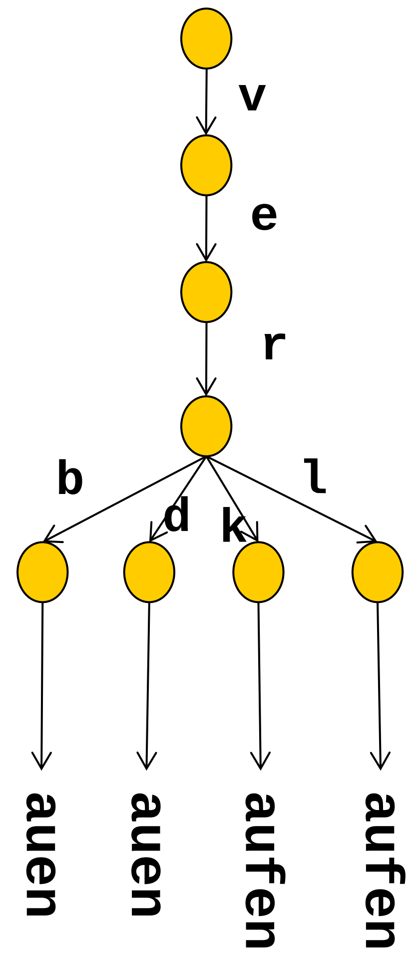
Example

k=„verhalten“



Tries

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



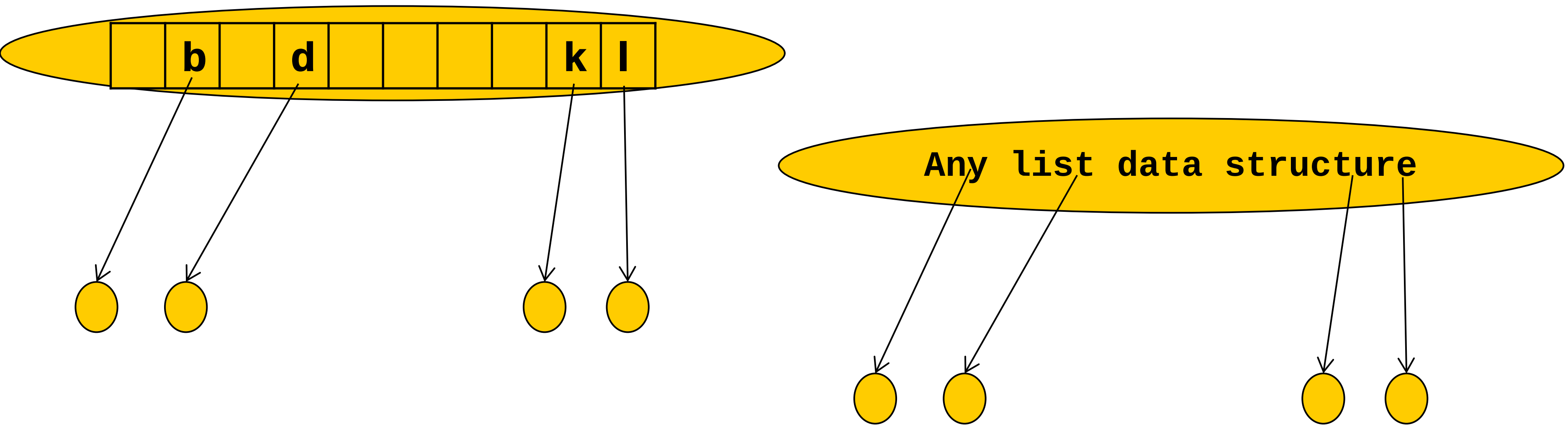
Analysis

- Construction of a trie over K?
  - Let len(K) be the sum of all key lengths in K
  - We start with an empty tree and iteratively add all k
  - Upon adding k, we simply 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(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 len(K) edges and len(K)+1 nodes
  - Shared prefixes make the actual number smaller
- But we also need pointer to children
- To hold our search complexity, choosing the right pointer must be possible in O(1)
- This adds O(len(K)*|Σ|) pointers
- Too much for any non-trivial alphabet
  - Digital tries are a popular data structure in coding theory
  - There, |Σ|=2, so the pointers don't matter much
- Furthermore, most of the pointers will be null
  - Depending on |Σ|, |K|, and lengths of shared prefixes

Alternatives



- Advantage: O(|k|) search
- Disadvantage: Excessive place consumption

- Advantage: O(len(K)) space
- Disadvantage: At least O(|k|*log(|Σ|)) search

Compressed Tries = Patricia Trees

- We can save further space
- A patricia tree is a trie where edges are labeled with (sub-)strings, not with characters
- All sequences S=<node, edge> 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