Fundamental Algorithms

Chapter 3: Advanced Search Structures

Sevag Gharibian
(based on slides of Christian Scheideler)

WS 2019
Search Structure
Search Structure

insert(15)
Search Structure

delete(20)
Search Structure

search(7) gives 8 (closest successor)
Search Structure

S: set of elements
Every element e identified by key(e).

Operations:
• \textbf{S.insert}(e: Element): S:=S\cup\{e\}
• \textbf{S.delete}(k: Key): S:=S\setminus\{e\}, where e is the element with key(e)=k (note: now given key, not pointer to e!)
• \textbf{S.search}(k: Key): outputs e \in S with minimal key(e) so that key(e) \geq k
Static Search Structure

1. Store elements in sorted array.

**search**: via binary search (in $O(\log n)$ time)
Binary Search

Input: number $x$ and sorted array $A[1],...,A[n]$

Algorithm BinarySearch:

$l:=1; r:=n$

while $l < r$ do

$\quad m:=(r+l) \div 2$

$\quad$ if $A[m] = x$ then return $m$

$\quad$ if $A[m] < x$ then $l:=m+1$

$\quad\quad$ else $r:=m$

return $l$
Dynamic Search Structure

**insert und delete Operations:**

Sorted array difficult to update!

Worst case: $\Theta(n)$ time
Search Structure

2. Sorted List (with an $\infty$-Element)

Problem: insert, delete and search take $\Theta(n)$ time in the worst case (why for insert/delete?)

Observation: If search could be implemented efficiently, then also all other operations
Search Structure

Idea: add navigation structure that allows search to run efficiently
Binary Search Tree (ideal)

search(12)
Binary Search Tree

Search tree invariant:

For all keys $k'$ in $T_1$ and $k''$ in $T_2$: $k' \leq k < k''$
Binary Search Tree

Formally: for every tree node v let

• $\text{key}(v)$ be the key stored at $v$
• $d(v)$ the number of children (degree) of $v$

• Search tree invariant: (as above)

• Degree invariant:
  All tree nodes have exactly two children
  (as long as the number of elements in the list is $>0$, recall presence of $\infty$ node)

• Key invariant:
  For every element $e$ in the list there is exactly one tree node $v$ with $\text{key}(v) = \text{key}(e)$. 
Binary Search Tree

- **Search tree invariant:** (as before)

- **Degree invariant:**
  All tree nodes have exactly two children (as long as the number of elements is $>0$)

- **Key invariant:**
  For every element $e$ in the list there is exactly one tree node $v$ with $\text{key}(v) = \text{key}(e)$.

From the search tree and key invariants it follows that for every left subtree $T$ of a node $v$, the rightmost list element $e$ under $T$ satisfies $\text{key}(v) = \text{key}(e)$. (Why?)
search(x) Operation

For all keys k´ in T₁ and k´´ in T₂: k´ ≤ k < k´´

Search strategy:
• Start at the root, v, of the search tree
• while v is a tree node:
  – if x ≤ key(v) then let v be the left child of v,
    otherwise let v be the right child of v
• Output (list node) v
search(x) Operation

Correctness of search strategy:

- For every left subtree \(T\) of a node \(v\), the rightmost list element \(e\) under \(T\) satisfies \(\text{key}(v) = \text{key}(e)\).
- If \(\text{search}(x)\) enters \(T\), since \(\text{key}(v) \geq x\), there is an element \(e\) in the list below \(T\) with \(\text{key}(e) \geq x\).
Search(9)
Insert and Delete Operations

Strategy:

• **insert(e):**
  First, execute \texttt{search(key(e))} to obtain a list element \(e'\). If \(\text{key}(e) = \text{key}(e')\), replace \(e'\) by \(e\), otherwise insert \(e\) between \(e'\) and its predecessor in the list and add a new search tree leaf leading to \(e\) (left) and \(e'\) (right) with key \(\text{key}(e)\).

• **delete(k):**
  First, execute \texttt{search(k)} to obtain a list element \(e\). If \(\text{key}(e) = k\), then delete \(e\) from the list and the parent \(v\) of \(e\) from the search tree, and relabel tree node \(w\) with key \(\text{key}(w) = k\) as \(\text{key}(w) := \text{key}(v)\).
Insert(5)
Insert(12)
Insert(12)
Delete(1)
Delete(1)
Delete(14)
Delete(14)
Binary Search Tree

**Problem:** binary tree can degenerate!

**Example:** numbers are inserted in sorted order
Pop quiz

Q1: What is the worst case runtime for binary search on a sorted array?
   \[O(\log n)\].

Q2: What is the worst case runtime for searching in a binary search tree?
   \[O(n)\]! (see e.g. previous slide)
Search Trees

Problem: binary tree can degenerate!

Solutions:
- Splay tree
  (very effective heuristic)
- (a,b)-tree
  (guaranteed well balanced)
- Patricia trie
Splay Tree

**Usually:** Implementation as *internal* search tree (i.e., elements directly integrated into tree and not in an extra list)

**Here:** Implementation as *external* search tree (like for the binary search tree above)
Why Splay Trees?

• Self-adjusting binary search tree
• Invented by Sleator and Tarjan (1985)
• Pros:
  – Recently accessed elements quick to access again. (Great for caches, garbage collection!)
  – Low amortized costs
• Cons:
  – Can still have highly unbalanced trees, hence worst-case linear time search.
Splay Tree

search(19)

Idea: add shortcut pointer to list element ⇒ accelerates search
Splay Tree

Ideas:
1. Add shortcut pointers in tree to list elements
2. For every search(k) operation, move pred(k) (the closest predecessor of k in T) to the root (why?)

Movement for (2): via Splay operation

For simplicity: we focus on search(k) for keys k already in the search tree.
Splay Operation

Movement of key $x$ to the root: 3 cases.

Case 1:

1a. $x$ is a left child of the root:

Before:

```
        x
       / \  
      y   C
     /     
    A      B
```

After:

```
        y
       /   
      A     C
     /     
    B      x
```

zig
Splay Operation

Movement of key $x$ to the root: 3 cases

Case 1:

1b. $x$ is a *right* child of the root:

![Diagram showing the splay operation for a right child](attachment:image.png)
Splay Operation

Case 2:

2a. x has father and grand father to the right

24.10.2019

Chapter 3
Case 2:

2b. \( x \) has father and grand father to the \textit{left}
Splay Operation

Case 3:

3a. x: father left, grand father right

A
B
C

D

y
x
z

zig-zag

A
B
C
D

y
x
z
Splay Operation

Case 3:

3b. x: father right, grand father left

```
\begin{array}{c}
\text{Case 3:} \\
3b. \ x: \text{ father right, grand father left} \\
\end{array}
```
Splay Operation

Example:

zig-zag operation (3a)
Splay Operation
Splay Operation

Examples:

zig-zig, zig-zag, zig-zag, zig

zig-zig, zig-zag, zig-zig, zig
Splay Operation

Observation: Tree can still be highly imbalanced! But amortized costs are low.
Splay Operation

search(k)-operation:
• Move downwards from the root (as in standard binary tree) till \( \text{pred}(k) \) found in search tree (which can be checked via shortcut to the list) or the list is reached
• call \( \text{splay}(\text{pred}(k)) \), output next successor, \( \text{succ}(k) \) (recall we assume \( k \) exists in tree for simplicity: \( \text{pred}(k) = \text{succ}(k) = k \))

Amortized Analysis:
• Note: runtime of \( \text{search}(k) \) is \( O(\text{runtime of splay}(\text{pred}(k))) \).
• Our goal: bound runtime of \( m \) Splay operations on arbitrary binary search tree with \( n \) elements (\( m > n \))
Splay Operation

- **Weight of node** $x$: $w(x) > 0$
- **Tree weight** of tree $T$ with root $x$: $tw(x) = \sum_{y \in T} w(y)$
- **Rank** of node $x$: $r(x) = \log(tw(x))$
- **Potential** of tree $T$: $\phi(T) = \sum_{x \in T} r(x)$

**Lemma 3.1:** Let $T$ be a Splay tree with root $x$ and $u$ be a node in $T$. The amortized cost for $\text{splay}(u, T)$ is at most $1 + 3(r(x) - r(u))$. 
Splay Operation

(Recall: Amortized cost \( A_X(s) := T_X(s) + (\phi(s') - \phi(s)) \))

Proof of Lemma 3.1:
Induction over the sequence of rotations.

- \( r \) and \( tw \): rank and weight before the rotation
- \( r' \) and \( tw' \): rank and weight after the rotation

Case 1:

Amortized cost:

\[
\leq 1 + r'(u) + r'(v) - r(u) - r(v) \leq 1 + r'(u) - r(u) \text{ since } r'(v) \leq r(v) \\
\leq 1 + 3(r'(u) - r(u)) \text{ since } r'(u) \geq r(u)
\]
Splay Operation

Case 2:

Amortized cost:
\[ \leq 2 + r'(u) + r'(v) + r'(w) - r(u) - r(v) - r(w) \]
\[ = 2 + r'(v) + r'(w) - r(u) - r(v) \quad \text{since } r'(u) = r(w) \]
\[ \leq 2 + r'(u) + r'(w) - 2r(u) \quad \text{since } r'(u) \geq r'(v) \text{ and } r(v) \geq r(u) \]
Splay Operation

Case 2:

Claim: It holds that
\[ 2 + r'(u) + r'(w) - 2r(u) \leq 3(r'(u) - r(u)) \]
i.e.
\[ r(u) + r'(w) \leq 2(r'(u) - 1) \]
Case 2:

Claim: It holds that

\[ r(u) + r'(w) \leq 2(r'(u) - 1) \]

• Observe: There exist \(0 < x, y < 1\) and scaling factor \(c > 0\) with \(r(u) = \log(c \cdot x)\), \(r'(w) = \log(c \cdot y)\), and \(r'(u) \geq \log(c(x+y))\).

• Hence, the claim holds if \(\log(c \cdot x) + \log(c \cdot y) \leq 2(\log(c(x+y)) - 1)\) for all \(0 < x, y < 1\) and \(c > 0\).
Splay Operation

Case 2:

For all $0 < x, y < 1$ and $c > 0$ holds:

$$\log(c \cdot x) + \log(c \cdot y) \leq 2(\log(c(x+y)) - 1)$$

$$\iff \log(x) + \log(y) \leq 2(\log(x+y) - 1)$$

WLOG set $c$ so that $c(x+y) = 1$. Let $x' = c \cdot x$ and $y' = c \cdot y$. 

![Diagram of splay operation](image)
Splay Operation

Case 2:

• To show: for all $0 < x', y' \leq 1$, with $x' + y' = 1$:
  \[
  \log(x') + \log(y') \leq 2(\log(1) - 1) = -2
  \]
• Or more generally: show for $f(x, y) = \log(x) + \log(y)$ that
  \[
  f(x, y) \leq -2 \text{ for all } x, y > 0 \text{ with } x + y \leq 1
  \]
Splay Operation

Lemma 3.2: In the area $x,y>0$ with $x+y \leq 1$, the function $f(x,y)=\log x + \log y$ has its maximum at $(\frac{1}{2},\frac{1}{2})$.

Proof:

• Reduce to univariate problem:
  – $\log x$ is monotonically increasing. Hence, WLOG maximum satisfies $x+y=1$, $x,y>0$.
  – Consider determining the maximum for $g(x) = \log x + \log (1-x)$

• High school calculus: (note base of log WLOG is e)
  – The only root of $g'(x) = \frac{1}{x} - \frac{1}{1-x}$ is at $x=1/2$.
  – For $g''(x)= -(1/x^2 + 1/(1-x)^2))$ it holds that $g''(1/2)<0$.

• Hence, $f$ has its maximum at $(\frac{1}{2},\frac{1}{2})$.
Case 2:

Hence, it holds that $f(x,y) \leq -2$ for all $x,y > 0$ with $x+y \leq 1$, which implies the claim that $r(u) + r'(w) \leq 2(r'(u) - 1)$, which was equivalent to obtaining upper bound $3(r'(u) - r(u))$. 

24.10.2019

Chapter 3
Splay Operation

Case 3:

Amortized cost:
\[
\leq 2 + r'(u) + r'(v) + r'(w) - r(u) - r(v) - r(w)
\]
\[
\leq 2 + r'(v) + r'(w) - 2r(u) \quad \text{since} \ r'(u) = r(w) \quad \text{and} \ r(u) \leq r(v)
\]
\[
\leq 2(r'(u) - r(u)) \quad \text{because}...
\]
Splay Operation

Case 3:

...it holds that:

$$2 + r'(v) + r'(w) - 2r(u) \leq 2(r'(u) - r(u))$$

$$\Leftrightarrow 2r'(u) - r'(v) - r'(w) \geq 2$$

$$\Leftrightarrow r'(v) + r'(w) \leq 2(r'(u) - 1),$$ which can be shown to hold.
Splay Operation

Proof of Lemma 3.1: (Follow-up)
Induction over the sequence of rotations.
• r and tw : rank and weight before the rotation
• r’ und tw’: rank and weight after the rotation
• For every rotation (i.e. zig, zig-zig, or zig-zag), the amortized cost is \(\leq 1 + 3(r’(u) - r(u))\) (case 1) resp. \(3(r’(u) - r(u))\) (cases 2 and 3)
• Summation of the costs gives at most (x: root)
  \[1 + \sum_{\text{Rotations}} 3(r’(u) - r(u)) = 1 + 3(r(x) - r(u))\]
  – 1. Why do we only add 1 before the summation?
  – 2. Why do we get a telescoping series above?
Splay Operation

- **Tree weight** of tree $T$ with root $x$:
  \[ \text{tw}(x) = \sum_{y \in T} w(y) \]
- **Rank** of node $x$: \( r(x) = \log(\text{tw}(x)) \)
- **Potential** of tree $T$: \( \phi(T) = \sum_{x \in T} r(x) \)

**Lemma 3.1:** Let $T$ be a Splay tree with root $x$ and $u$ be a node in $T$. The amortized cost for $\text{splay}(u, T)$ is at most
\[ 1 + 3(r(x) - r(u)) = 1 + 3 \cdot \log(\text{tw}(x)/\text{tw}(u)). \]

**Corollary 3.3:** Let $W = \sum_x w(x)$ and $w_i$ be the weight of key $k_i$ in the $i$-th search call (recall we assume $k_i$ is in tree). For $m$ search operations, the amortized cost is $O(m + \sum_{i=1}^m \log (W/w_i))$. 
**Splay Tree**

**Theorem 3.4:** The runtime for \( m \) successful search operations in a Splay tree \( T \) with \( n \) elements is at most \( O(m+(m+n)\log n) \).

**Proof:**
- Let \( w(x) = 1 \) for all nodes \( x \) in \( T \).
- Then \( W=n \) and \( r(x) \leq \log W = \log n \) for all \( x \) in \( T \).
- For sequence \( F \) of operations, total runtime satisfies \( T(F) \leq A(F) + \phi(s_0) \) for any amortized cost function \( A \) and any initial state \( s_0 \) (Recall: \( A_X(s) := T_X(s) + (\phi(s') - \phi(s)) \))
- \( \phi(s_0) = \sum_{x \in T} r_0(x) \leq n \log n \)
- Hence, Corollary 3.3 implies Theorem 3.4.
Splay Tree

Suppose we have a probability distribution for the search requests, where each key in tree is searched for at least once.

- $p(x)$: probability of searching for key $x$
- $H(p) = \sum_x p(x) \cdot \log(1/p(x))$: entropy of $p$

**Theorem 3.5**: The expected runtime for $m$ successful search operations in a Splay tree $T$ with $n$ elements is at most $O(m \cdot (1 + H(p)))$.

**Proof**: Follows from proof of Theorem 3.4 with $w(x) = p(x)$ for all $x$, and assuming each item $x$ is searched for $m \cdot p(x)$ times.

Note: This proof requires us to relax our requirement that the potential function $\phi$ is non-negative. Why?
Splay Tree

Something amazing:

For a \textit{fixed} optimal Binary Search Tree where each key $x$ in tree is searched for with probability $p(x)$, one can show expected cost of a successful search is $\Omega(H(p))$ \textit{(entropy bound)}.

Our Theorem 3.5 says Splay Trees are almost optimal, in that the cost per search scales as $O(1+H(p))$!

Note: $0 \leq H(p) \leq \log n$

Question: How does this $O(1+H(p))$ support the idea that Splay trees would be good for applications like caching?
Splay Tree

So far, we assumed all searches were successful, i.e. the key we were searching for was in the tree.

Q1: Where in our analysis did this assumption play a role?

Q2: What if we consider the more general case of allowing unsuccessful searches?
Splay Tree – Unsuccessful Searches

• Instead of just successful searches, the Splay tree $T$ should also support the search for the closest successor.

search(23):

$23 \notin [10,14), 23 > 10$

$23 \in [19,28)$:
• output 28
• splay(19)
Splay Tree – Unsuccessful Searches

• To obtain a low amortized time bound, we associate with a key $x$ in $T$ the search range $[x, x_+]$ (including $x$ but excluding $x$), where $x_+$ is closest successor of $x$ in $T$.
• Each search range $[x, x_+]$ is associated with a weight $w([x, x_+])$. Using that, we can revise Corollary 3.3 to:

**Corollary 3.3’**: Let $W = \sum_x w(x)$ and $w_i$ be the weight of the range $[x, x_+]$ containing the $i$-th search key. For $m$ search operations, the amortized cost is $O(m + \sum_{i=1}^{m} \log \left( W/w_i \right))$. 
Splay Tree Operations

Let $T_1$ and $T_2$ be two Splay trees with $\text{key}(x) < \text{key}(y)$ for all $x \in T_1$ and $y \in T_2$.

merge$(T_1, T_2)$:

Take max. element $x < \infty$ in $T_1$ and splay it up to root
Splay Tree Operations

**split(k, T):** returns two trees as follows

- **T**
- **T**
- **T**
- **T**

**k (or pred(k))**

**search(k):**
- causes splay(k)
- or splay(pred(k))

> k
Splay Tree Operations

**insert(e):**
- insert like in binary search tree
- Splay operation to move key(e) to the root

**delete(k):**
- execute search(k) (splays k to the root)
- remove root and execute merge($T_1, T_2$) of the two resulting subtrees
Splay Operations

• $k_-$: closest predecessor $\leq k$ in $T$
• $k_+$: closest successor $> k$ in $T$

Theorem 3.6: The amortized cost of the following operations in the Splay tree are:

• search($k$): $O(1 + \log(W/w([k_-,k_+])))$
• insert($e$): $O(1 + \log(W/w([\text{key}(e),\text{key}(e)_+])))$
• delete($k$): $O(1 + \log(W/w([k,k_+])) + \log((W-w([k,k_+]))/w([k_-,k])))$
Search Trees

Problem: binary tree can degenerate!

Solutions:
• Splay tree
  (very effective heuristic)
• (a,b)-tree
  (guaranteed well balanced)
• Patricia trie
(a,b)-Trees

Problem: how to maintain balanced search tree

Idea:
• All nodes $v$ (except for the root) have degree $d(v)$ with $a \leq d(v) \leq b$, where $a \geq 2$ and $b \geq 2a - 1$ (otherwise this cannot be enforced)
• All leaves have the same depth
(a,b)-Trees

Formally: for a tree node $v$ let

- $d(v)$ be the number of children of $v$
- $t(v)$ be the depth of $v$ (root has depth 0)

- **Form Invariant:**
  For all leaves $v,w$: $t(v)=t(w)$

- **Degree Invariant:**
  For all inner nodes $v$ except for root: $d(v) \in [a,b]$, for root $r$: $d(r) \in [2,b]$ (as long as #elements $>1$)
Lemma 3.10: An \((a,b)\)-tree with \(n\) elements has depth at most \(1 + \lceil \log_a \left( \frac{n}{2} \right) \rceil\)

Proof:

• The root has degree \(\geq 2\) and every other inner node has degree \(\geq a\).

• At depth \(t\) there are at least \(2a^{t-1}\) nodes

• \(n \geq 2a^{t-1} \iff t \leq 1 + \lceil \log_a (n/2) \rceil\)
(a,b)-Trees

(a,b)-Tree-Rule:

For all keys \( k \) in \( T_i \) and \( k' \) in \( T_{i+1} \): \( k \leq s_i < k' \)

Then search operation easy to implement.
Search(9)
Insert(e) Operation

Strategy:

- First search(key(e)) until some \( e' \) found in the list. If \( \text{key}(e') > \text{key}(e) \), insert \( e \) in front of \( e' \), otherwise replace \( e' \) by \( e \).
Insert(e) Operation

Strategy:

- First search(key(e)) until some e´ found in the list. If key(e´)>key(e), insert e in front of e´, otherwise replace e´ by e.
Insert(e) Operation

• Add key(e) and pointer to e in tree node v which is parent of e’. If we still have \(d(v) \in [a,b]\) afterwards, then we are done.
Insert(e) Operation

• If $d(v) > b$, then cut $v$ into two nodes.
  (Example: $a=2$, $b=4$)
Insert(e) Operation

- If after splitting $v$, $d(w) > b$, then cut $w$ into two nodes (and so on, until all nodes have degree $\leq b$ or we reached the root)
Insert(e) Operation

- If for the root \( v \) of \( T \), \( d(v) > b \), then cut \( v \) into two nodes and create a new root node.
Insert(8)

\[a = 2, \ b = 4\]
Insert(8)

\[a=2,\ b=4\]
Insert(8)

\[ a = 2, \ b = 4 \]
Insert(6)

a = 2, b = 4
Insert(6)

a=2, b=4
Insert(7)

\[ a=2, \ b=4 \]
Insert(7)

a=2, b=4
Insert(7)

a=2, b=4
Insert(7)

\[ a = 2, \quad b = 4 \]
Insert Operation

• **Form Invariant:**
  For all leaves \( v, w \): \( t(v) = t(w) \)
  Satisfied by Insert!

• **Degree Invariant:**
  For all inner nodes \( v \) except for the root:
  \[ d(v) \in [a, b] \]
  for root \( r \):
  \[ d(r) \in [2, b] \]

1) Insert splits nodes of degree \( b + 1 \) into nodes of degree \( \lfloor (b+1)/2 \rfloor \) and \( \lceil (b+1)/2 \rceil \). If \( b \geq 2a - 1 \), then both values are at least \( a \).

2) If root has reached degree \( b + 1 \), then a new root of degree 2 is created.
Delete(k) Operation

Strategy:

• First search(k) until some element e is reached in the list. If key(e)=k, remove e from the list, otherwise we are done.
Delete(k) Operation

Strategy:
• First **search**\((k)\) until some element \(e\) is reached in the list. If key\((e)\)=\(k\), remove \(e\) from the list, otherwise we are done.
Delete\((k)\) Operation

- Remove pointer to \(e\) and key \(k\) from the leaf node \(v\) above \(e\). (\(e\) rightmost child: perform key exchange like in binary tree!) If afterwards we still have \(d(v) \geq a\), we are done.
Delete(k) Operation

• Remove pointer to e and key k from the leaf node v above e. (e rightmost child: perform key exchange like in binary tree!) If afterwards we still have \(d(v) \geq a\), we are done.
Delete(k) Operation

• If \( d(v) < a \) and the preceding or succeeding sibling of \( v \) has degree \( > a \), steal an edge from that sibling. (Example: \( a=2 \), \( b=4 \))
Delete(k) Operation

- If \( d(v) < a \) and the preceding and succeeding siblings of \( v \) have degree \( a \), merge \( v \) with one of these. (Example: \( a=3, b=5 \))
Delete(k) Operation

- Perform changes upwards until all inner nodes (except for the root) have degree $\geq a$. If root has degree $<2$: remove root.
Delete(10)

a=2, b=4
Delete(10)

\[ a=2, \ b=4 \]
Delete(14)

$a = 2, b = 4$
Delete(14)

\[a=2, \ b=4\]
Delete(14)

\[a=2, \ b=4\]
Delete(3)

\[ a=2, \ b=4 \]
Delete(3)

\[ a=2, \ b=4 \]
Delete(3)

\[ a=2, \ b=4 \]
Delete(3)

\[ a=2, \ b=4 \]
Delete(1)

\[ a=2, \ b=4 \]
Delete(1)

\[a=2, \ b=4\]
Delete(19)

\[a=2, \; b=4\]
Delete(19)

\[ a = 2, \ b = 4 \]
Delete(19)

$a=2, \ b=4$
Delete(19)

$a = 2, b = 4$
Delete Operation

• Form Invariant:
  For all leaves \( v, w \): \( t(v) = t(w) \)
  Satisfied by Delete!

• Degree Invariant:
  For all inner nodes \( v \) except for the root: \( d(v) \in [a, b] \), for
  root \( r \): \( d(r) \in [2, b] \)
1) \textbf{Delete} merges node of degree \( a-1 \) with node of
   degree \( a \). Since \( b \geq 2a - 1 \), the resulting node has degree
   at most \( b \).
2) \textbf{Delete} moves edge from a node of degree \( >a \) to a
   node of degree \( a-1 \). Also OK.
3) Root deleted: children have been merged, degree of
   the remaining child is \( \geq a \) (and also \( \leq b \)), so also OK.
More Operations

- **min/max Operation:**
  Pointers to both ends of list: time $O(1)$.

- **Range queries:**
  To obtain all elements in the range $[x, y]$, perform `search(x)` and go through the list till an element $>y$ is found.
  Time $O(\log n + \text{size of output})$. 
n Update Operations

Theorem 3.11: There is a sequence of \( n \) insert and delete operations in a \((2,3)\)-tree that require \( \Omega(n \log n) \) many split and merge Operations.

Proof: Exercise
n Update Operations

Theorem 3.12: Consider an \((a,b)\)-tree with \(b \geq 2a\) that is initially empty. For any sequence of \(n\) insert and delete operations, only \(O(n)\) split and merge operations are needed.

Proof:

Amortized analysis
External (a,b)-Tree

(a,b)-trees well suited for large amounts of data
External (a, b)-Tree

Problem: minimize number of block transfers between internal and external memory

Solution:
• use $b = B$ (block size) and $a = b/2$
• keep highest $(1/2) \cdot \log_a(M/b)$ levels of (a,b)-tree in internal memory (storage needed $\leq M$)
• Lemma 3.10: depth of (a,b)-tree $\leq 1 + \lfloor \log_a(n/2) \rfloor$
• How many levels are not in internal memory?
  $\log_a[n/2] - (1/2) \cdot \log_a(M/b) \leq \log_a[n/(2\sqrt{M})] + O(1)$ (a, b are $O(1)$)
• Cost for insert, delete and search operations: $O(\log_B(n/\sqrt{M}))$ block transfers
External (a,b)-Tree

Problem: minimize number of block transfers between internal and external memory

A better analysis can show (exercise):
• Cost for insert, delete and search operations: 
  \( \sim 2\log_{B/2}(n/M) + 1 \) block transfers (\(+1\): list access)

Example:
• \( n = 100,000,000,000,000 \) keys
• \( M = 16 \) Gbyte (\( \sim 4,000,000,000 \) keys)
• \( B = 256 \) Kbyte (\( \sim 64,000 \) keys)
• \( 2\log_{B/2}(n/M) + 1 \leq 3 \)
Search Trees

Problem: binary tree can degenerate!

Solutions:
• Splay tree
  (very effective heuristic)
• (a,b)-tree
  (guaranteed well balanced)
• Patricia trie
Longest Prefix Search

• All keys are encoded as binary sequence $\{0,1\}^W$
• Prefix of a key $x \in \{0,1\}^W$: arbitrary subsequence of $x$ that starts with the first bit of $x$ (example: $101$ is a prefix of $10110100$)

Problem: given a key $x \in \{0,1\}^W$, find a key $y \in S$ with longest common prefix

Solution: Trie Hashing
Trie

A trie is a search tree over some alphabet \( \Sigma \) that has the following properties:

- Every edge is associated with a symbol \( c \in \Sigma \)
- Every key \( x \in \Sigma^k \) that has been inserted into the trie can be reached from the root of the trie by following the unique path of length \( k \) whose edge labels result in \( x \).

For simplicity: all keys from \( \{0,1\}^W \) for some \( W \in \mathbb{N} \).

Example:
\((0,2,3,5,6)\) with \( W=3 \) results in \((000,010,011,101,110)\)
Trie

Example: (without list at bottom)
Trie

search(4) (4 corresponds to 100):  

Output: 5 (longest common prefix)
Trie

**In general:** a search(x) request follows the edges in the trie as long as their labels form a prefix of x. Once no edge is available any more to follow the bits in x, the request may be forwarded to any leaf y in the subtrie below since all of them have the same longest prefix match with x.
Trie

insert(1) (1 corresponds to 001):
In general: an insert(x) request follows the edges in the trie as long as their labels form a prefix of \( x \). Once no edge is available any more to follow the bits in \( x \), a new path (of length the remaining bits in \( x \)) is created that leads to the new leaf \( x \).
Trie

delete(5):

```
    0
   / \
  0   1
 /     \
0 1     0
 /   \
0 1   1
 /     /
0 1   1
 /     /
0 1   1
```

Nodes: 000, 001, 010, 011, 101, 110
Trie

In general: a delete(x) request follows the edges in the trie down to the leaf x. If x does not exist, the delete operation terminates. Otherwise, x as well as the chain of nodes upwards till the first node with at least two children is deleted.
Problem:
• Longest common prefix search for some $x \in \{0,1\}^W$ can take $\Theta(W)$ time.
• Insert and delete may require $\Theta(W)$ structural changes in the trie.

Improvement: use Patricia trie
A Patricia trie is a compressed trie in which all chains (i.e., maximal sequences of nodes of degree 1) are merged into a single edge whose label is equal to the concatenation of the labels of the merged trie edges.
Trie

Example 1:
Patricia Trie

Example 1:
Example 2:

Trie
Patricia Trie

Example 2:

root stores prefix 0
Patricia Trie

search(4):

```
  0  1
 /   \
0    1
/     /
00  01 10
/   /   /
01  01 10 11
    /   /
   01 10
    /    /
   00 11
```

24.10.2019
Patricia Trie

In general: a search($x$) request follows the edges in the Patricia trie as long as their labels form a prefix of $x$. Once no edge is available any more to follow the bits in $x$, choose the current child $c$ with longest common prefix. Then, the request may be forwarded to any leaf $y$ in the subtrie rooted $c$ at below since all of them have the same longest prefix match with $x$. 

![Diagram of Patricia Trie](image-url)
Patricia Trie

insert(1):
Patricia Trie

Insert(5):
In general: an insert(x) request follows the edges in the Patricia trie as long as their labels form a prefix of x. Once an edge e is reached whose label l(e) does not follow the bits in x, a new tree node is created in the middle of e.
Patricia Trie

In general: an insert(x) request follows the edges in the Patricia trie as long as their labels form a prefix of x. Once an edge e is reached whose label l(e) does not follow the bits in x, a new tree node is created in the middle of e.

Example: l(e)=10010, x=…10110100

\[
\begin{align*}
 & e \\
 & x \\
\end{align*}
\]

\[
\begin{align*}
 & e' \\
 & x \\
\end{align*}
\]

\[
\begin{align*}
 & e'' \\
 & e''' \\
 & x \\
\end{align*}
\]

l(e')=10, l(e'')=010, l(e''')=110100
Patricia Trie

In general: an insert(x) request follows the edges in the Patricia trie as long as their labels form a prefix of x. Once an edge e is reached whose label l(e) does not follow the bits in x, a new tree node is created in the middle of e.

Special case:
Patricia Trie

delete(5):

![Patricia Trie Diagram]
Patricia Trie

delete(6):

```
000 001 010 011
```

```
0 1 0 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```

```
0 1 1 1
```
Patricia Trie

In general: a delete(x) request follows the edges in the Patricia trie down to the leaf x. If x does not exist, the delete operation terminates. Otherwise, x as well as its parent are deleted.

Example: l(e´)=10, l(e´´)=010, l(e´´´)=110100, x=…10110100

![Diagram of Patricia Trie before and after deletion]

24.10.2019 Chapter 3
Patricia Trie

- Search, insert, and delete like in an ordinary binary tree, with the difference that we have labels at the edges.
- Search time still $O(W)$ in the worst case, but just $O(1)$ structural changes.
Patricia Trie

• History:
  – Morrison called them „Patricia trees“, where PATRICIA stands for Practical Algorithm To Retrieve Information Coded in Alphanumeric.
  – Patricia trees are also referred to as radix trees (with radix 2).

Idea (Kniesburges and Scheideler, 2011):
• Can improve search time to $O(\log W)$ using „hashed Patricia tries“. (Will not cover this here.)