7 Dictionary

Dictionary:

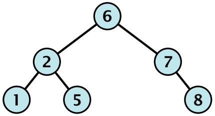
- ***S*.insert(*x*)**: Insert an element x.
- ***S*.delete(*x*)**: Delete the element pointed to by x.
- ***S*.search(*k*)**: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

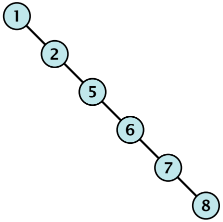
7.1 Binary Search Trees

An (internal) binary search tree stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node v have a smaller key-value than $\mathrm{key}[v]$ and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(External Search Trees store objects only at leaf-vertices)

Examples:



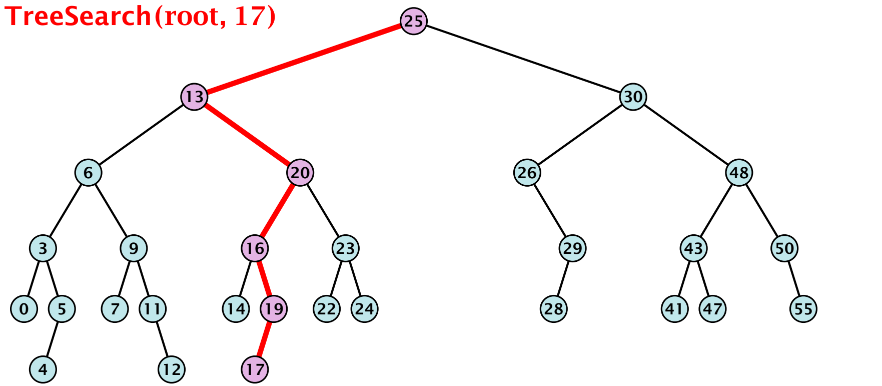


7.1 Binary Search Trees

We consider the following operations on binary search trees. Note that this is a super-set of the dictionary-operations.

- $T.\mathrm{insert}(x)$
- $T.\mathrm{delete}(x)$
- $T.\mathrm{search}(k)$
- $T.\mathrm{successor}(x)$
- $T.\mathrm{predecessor}(x)$
- $T.\mathrm{minimum}()$
- $T.\mathrm{maximum}()$

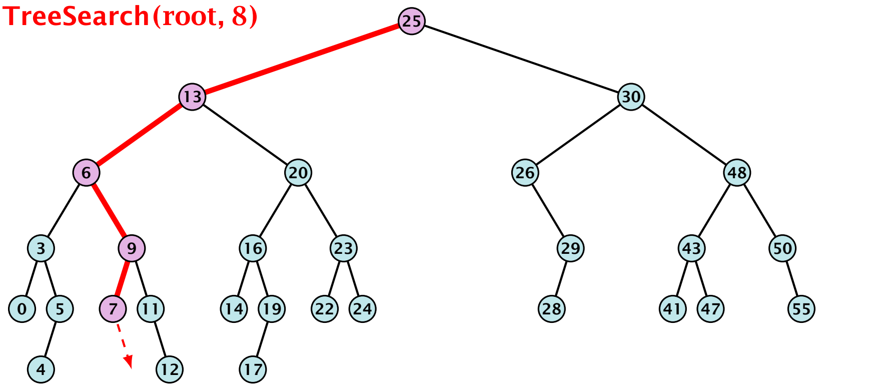
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
1: if $x$ = null or $k = \text{key}[x]$ return $x$
2: if $k < \text{key}[x]$ return TreeSearch(left$[x], k$)
3: else return TreeSearch(right$[x], k$)
```

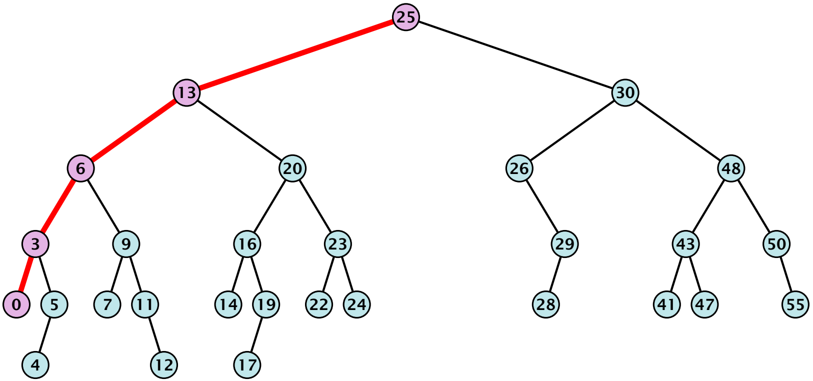
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
1: if $x$ = null or $k = \text{key}[x]$ return $x$
2: if $k < \text{key}[x]$ return TreeSearch(left$[x], k$)
3: else return TreeSearch(right$[x], k$)
```

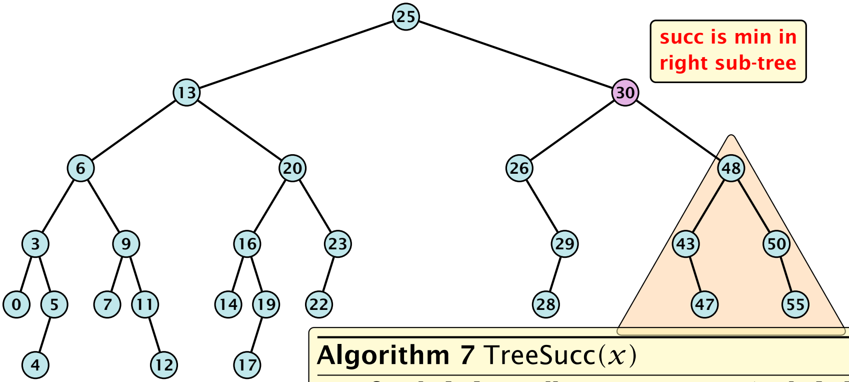
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
1: if $x$ = null or left[$x$] = null return $x$
2: if $k <$ key[$x$] return TreeSearch(left[$x$], $k$)
3: else return TreeMin(left[$x$])
```

Binary Search Trees: Successor

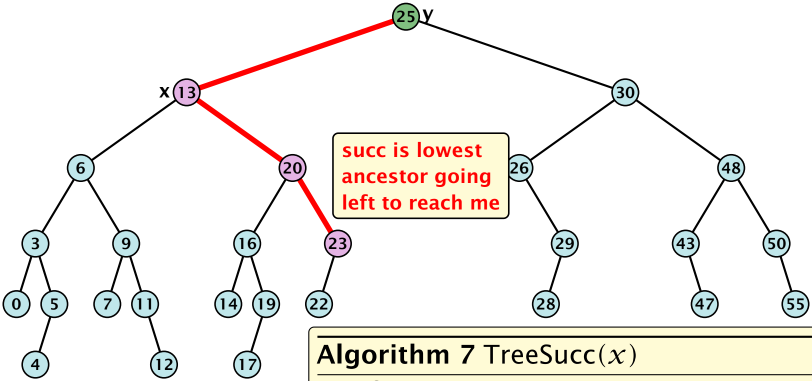


succ is min in right sub-tree

Algorithm 7 TreeSucc(x)

```
1: if right[$x$] $\neq$ null return TreeMin(right[$x$])
2: $y \leftarrow$ parent[$x$]
3: while $y \neq$ null and $x =$ right[$y$] do
4:     $x \leftarrow y$; $y \leftarrow$ parent[$x$]
5: return $y$;
```

Binary Search Trees: Successor



succ is lowest ancestor going left to reach me

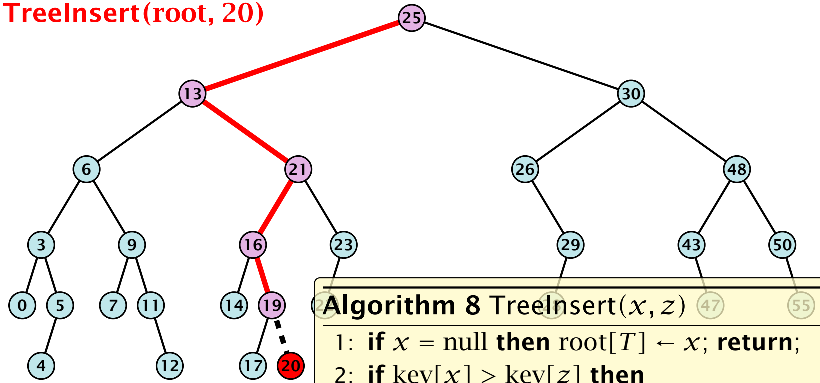
Algorithm 7 TreeSucc(x)

```
1: if right[$x$] $\neq$ null return TreeMin(right[$x$])
2: $y \leftarrow$ parent[$x$]
3: while $y \neq$ null and $x =$ right[$y$] do
4:     $x \leftarrow y$; $y \leftarrow$ parent[$x$]
5: return $y$;
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

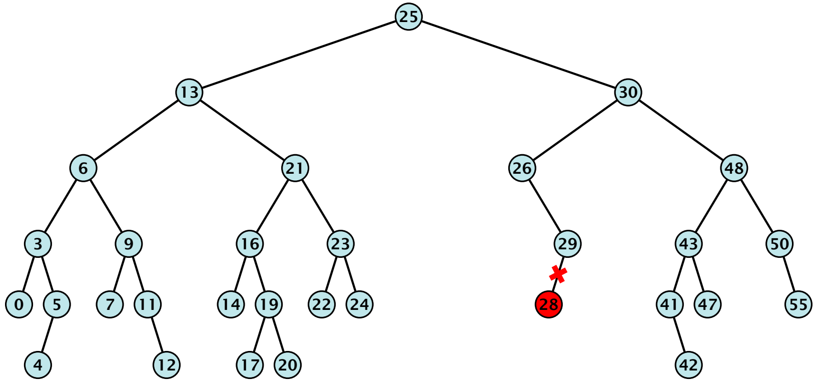


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Algorithm 8 TreeInsert(x, z)

```
1: if $x$ = null then root$[T] \leftarrow x$; return;
2: if key$[x] >$ key$[z]$ then
3:     if left$[x]$ = null then left$[x] \leftarrow z$;
4:     else TreeInsert(left$[x], z$);
5: else
6:     if right$[x]$ = null then right$[x] \leftarrow z$;
7:     else TreeInsert(right$[x], z$);
8: return
```

Binary Search Trees: Delete

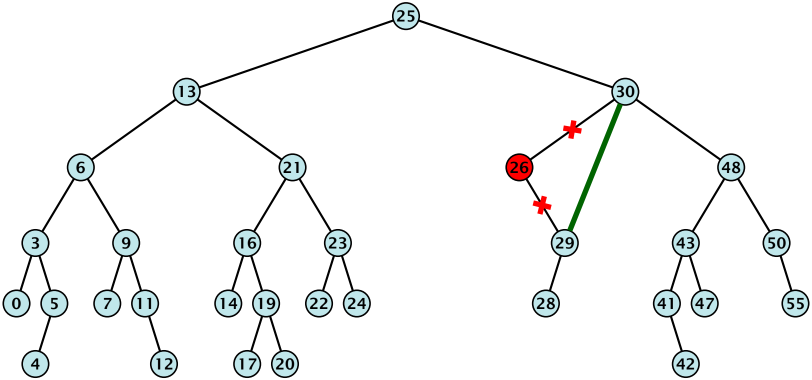


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

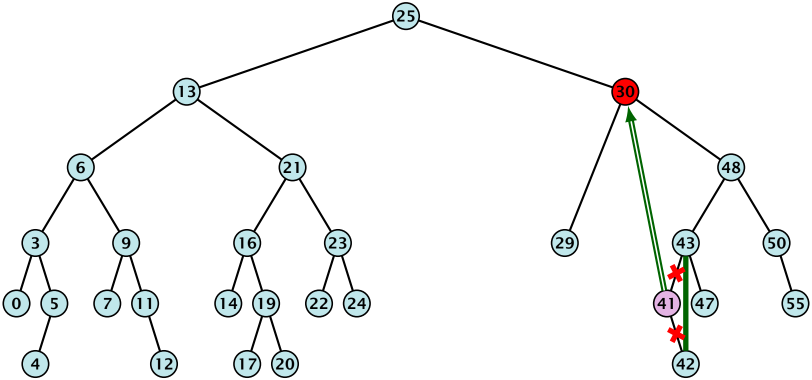


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete



Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

Algorithm 9 TreeDelete(z)

```
if left[$z$] = null or right[$z$] = null
    then $y \leftarrow z$ else $y \leftarrow$ TreeSucc$(z)$;          select $y$ to splice out
if left[$y$] $\neq$ null
    then $x \leftarrow$ left[$y$] else $x \leftarrow$ right[$y$];     $x$ is child of $y$ (or null)
if $x \neq$ null then parent[$x$] $\leftarrow$ parent[$y$];          parent[$x$] is correct
if parent[$y$] = null then
    root[$T$] $\leftarrow x$
else
    if $y$ = left[parent[$x$]] then                                  fix pointer to $x$
        left[parent[$y$]] $\leftarrow x$
    else
        right[parent[$y$]] $\leftarrow x$
if $y \neq z$ then copy $y$-data to $z$
```

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

7.2 Red Black Trees

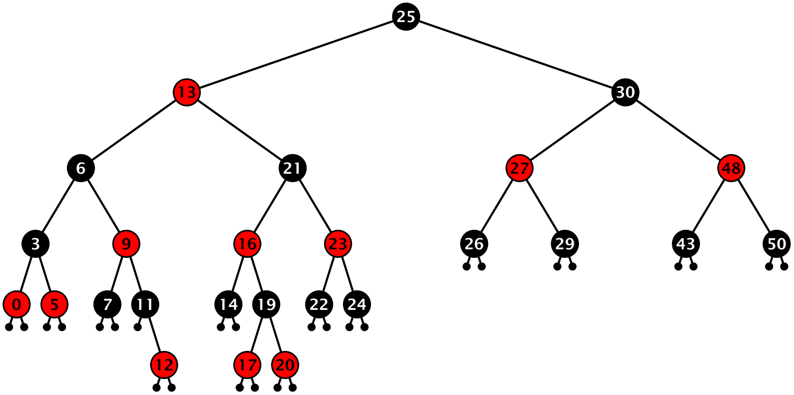
Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a colour, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

Red Black Trees: Example



7.2 Red Black Trees

Lemma 12

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 13

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 14

A sub-tree of black height $\mathrm{bh}(v)$ in a red black tree contains at least $2^{\mathrm{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case $(\mathrm{height}(v) = 0)$

- If $\mathrm{height}(v)$ (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\mathrm{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\mathrm{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\mathrm{bh}(c_i) = \mathrm{bh}(v)$ or $\mathrm{bh}(c_i) = \mathrm{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\mathrm{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\mathrm{bh}(v)-1} - 1) + 1 \geq 2^{\mathrm{bh}(v)} - 1$ vertices.

□

7.2 Red Black Trees

Proof of Lemma 12.

Let h denote the height of the red-black tree, and let p denote a path from the root to the furthest leaf.

At least half of the node on p must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \geq n$.

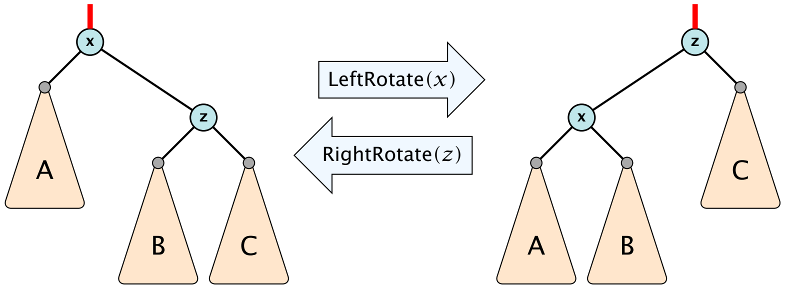
Hence, $h \leq 2 \log n + 1 = \mathcal{O}(\log n)$. $\square$

7.2 Red Black Trees

We need to adapt the insert and delete operations so that the red black properties are maintained.

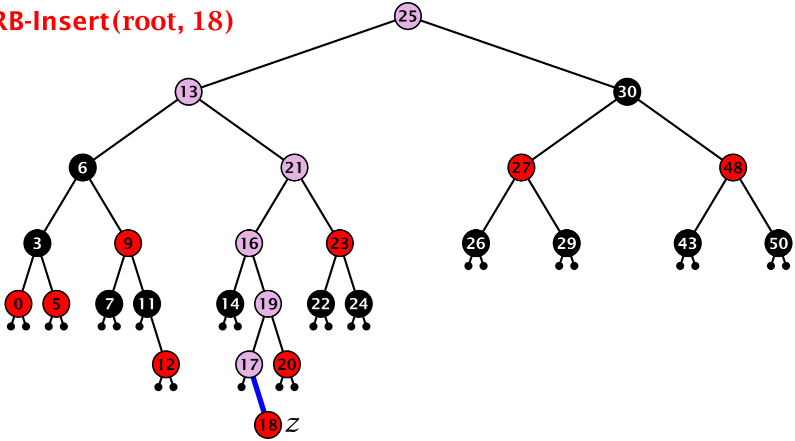
Rotations

The properties will be maintained through rotations:



Red Black Trees: Insert

RB-Insert(root, 18)



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red
    (most important case)
  - or the parent does not exist
    (violation since root must be black)

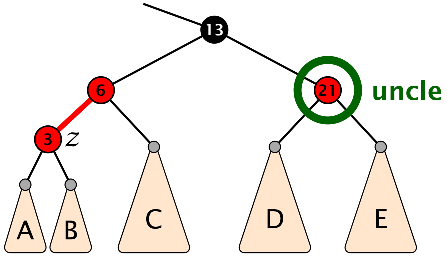
If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

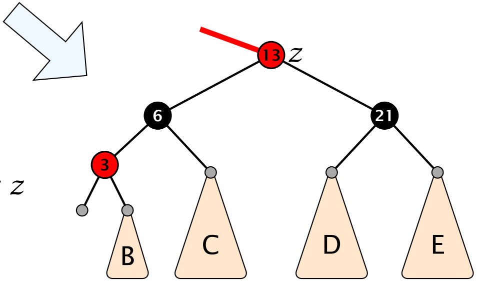
Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
1:  while $\text{parent}[z] \neq \text{null}$ and $\text{col}[\text{parent}[z]] = \text{red}$ do
2:      if $\text{parent}[z] = \text{left}[\text{gp}[z]]$ then          z in left subtree of grandparent
3:          $\textit{uncle} \leftarrow \text{right}[\text{grandparent}[z]]$
4:          if $\text{col}[\textit{uncle}] = \text{red}$ then          Case 1: uncle red
5:              $\text{col}[\text{p}[z]] \leftarrow \text{black}; \text{col}[u] \leftarrow \text{black};$
6:              $\text{col}[\text{gp}[z]] \leftarrow \text{red}; z \leftarrow \text{grandparent}[z];$
7:          else          Case 2: uncle black
8:              if $z = \text{right}[\text{parent}[z]]$ then          2a: z right child
9:                  $z \leftarrow \text{p}[z]$; LeftRotate($z$);
10:             $\text{col}[\text{p}[z]] \leftarrow \text{black}; \text{col}[\text{gp}[z]] \leftarrow \text{red};$          2b: z left child
11:             RightRotate($\text{gp}[z]$);
12:     else same as then-clause but right and left exchanged
13: $\text{col}(\text{root}[T]) \leftarrow \text{black};$
```

Case 1: Red Uncle

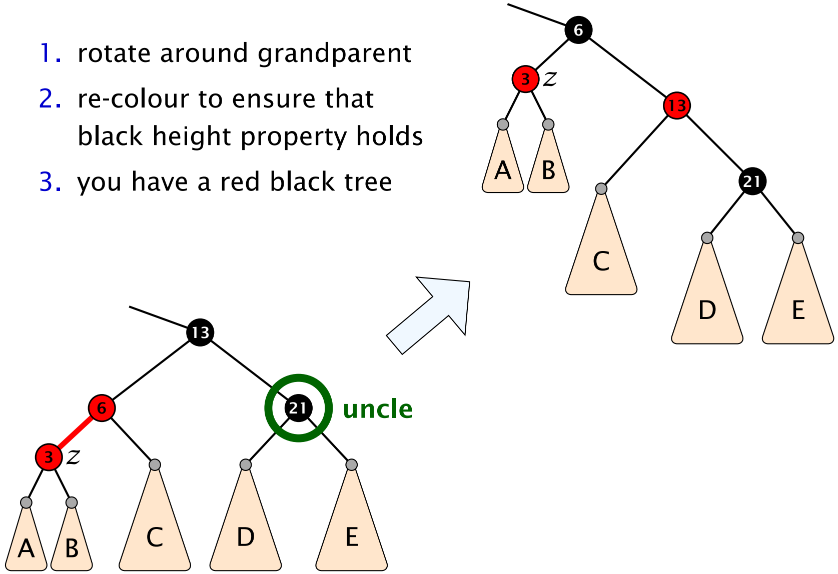




1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

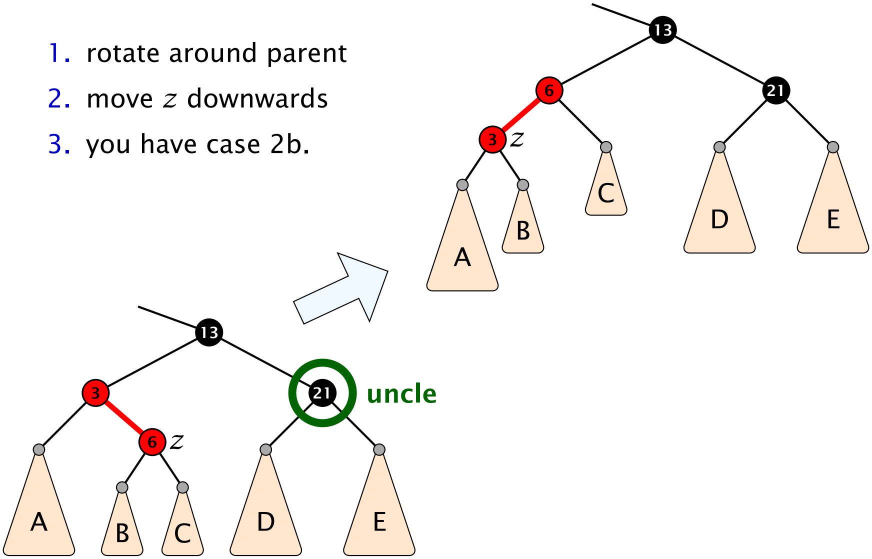
Case 2b: Black uncle and z is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have case 2b.



Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing step one $\mathcal{O}(\log n)$ times and every other step at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colourings and at most 2 rotations.

Red Black Trees: Delete

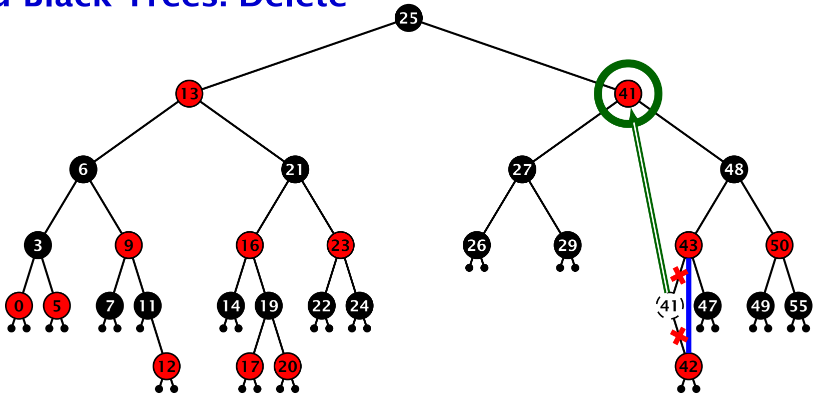
First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

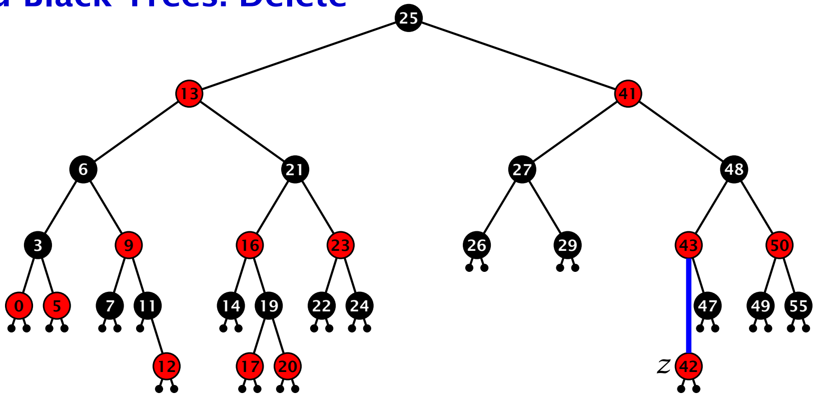


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete



Delete:

- deleting black node messes up black-height property
- if z is red, we can simply color it black and everything is fine
- the problem is if z is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

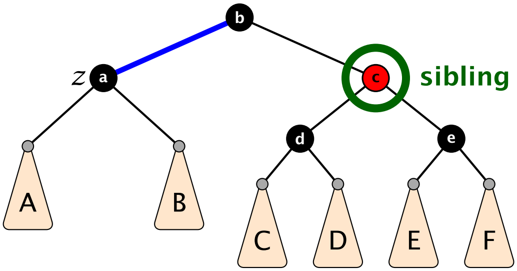
Red Black Trees: Delete

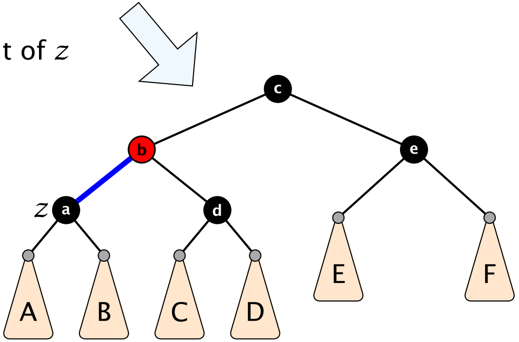
Invariant of the fix-up algorihtm

- the node z is black
- if we "assign" a fake black unit to the edge from z to its parent then the black-height property is fulfilled

Goal: make rotations in such a way that you at some point can remove the fake black unit from the edge.

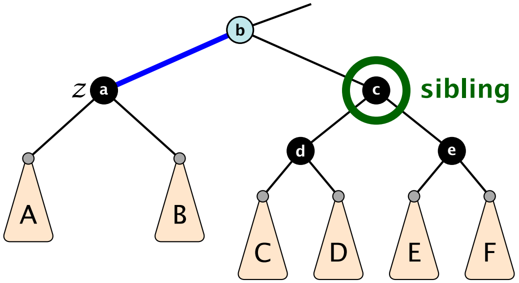
Case 1: Sibling of z is red



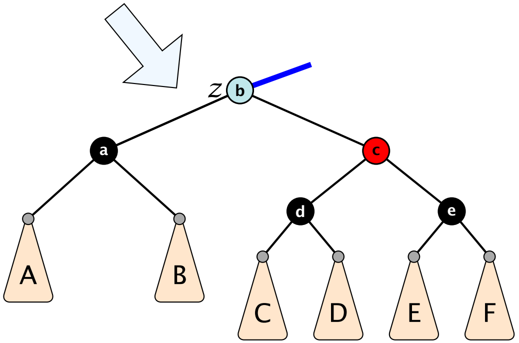


1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

Case 2: Sibling is black with two black children



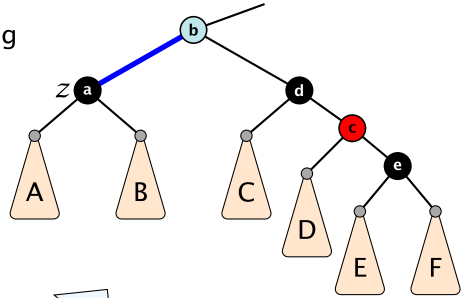
Here b is either black or red. If it is red we are in a special case that directly leads to a red-black tree.

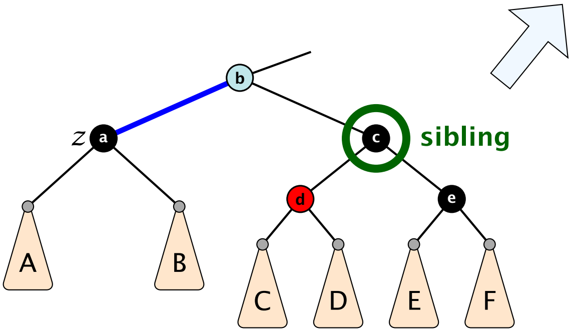


1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

Case 3: Sibling black with one black child to the right

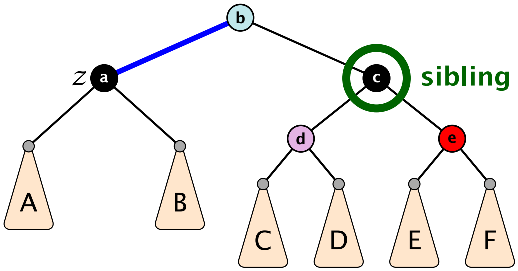
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)





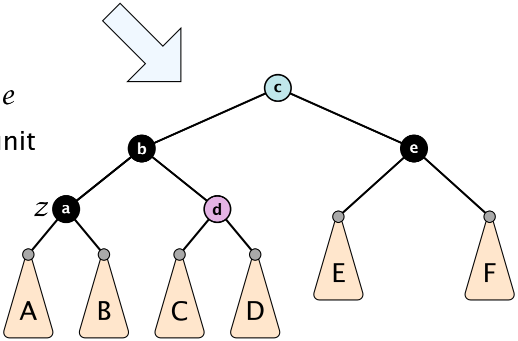
Again the blue color of b indicates that it can either be black or red.

Case 4: Sibling is black with red right child



- Here b and d are either red or black but have possibly different colors.
- We recolor c by giving it the color of b.

1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid red black tree



Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $O(\log n)$ re-colourings and at most 3 rotations.

7.3 AVL-Trees

Definition 15

AVL-trees are binary search trees that fulfill the following balance condition. For every node v

$$|\text{height}(\text{left sub-tree}(v)) - \text{height}(\text{right sub-tree}(v))| \leq 1 \ .$$

Lemma 16

An AVL-tree of height h contains at least $F_{h+2} - 1$ and at most $2^h - 1$ internal nodes, where F_n is the n-th Fibonacci number ($F_0 = 0$, $F_1 = 1$), and the height is the maximal number of edges from the root to an (empty) dummy leaf.

Proof.

The upper bound is clear, as a binary tree of height h can only contain

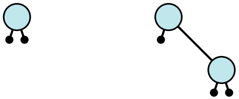
$$\sum_{j=0}^{h-1} 2^j = 2^h - 1$$

internal nodes.

Proof (cont.)

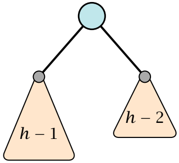
Induction (base cases):

1. an AVL-tree of height $h = 1$ contains at least one internal node, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. an AVL tree of height $h = 2$ contains at least two internal nodes, $2 \geq F_4 - 1 = 3 - 1 = 2$



Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

$$f_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Then

$$\begin{aligned} f_1 &= 2 && = F_3 \\ f_2 &= 3 && = F_4 \\ f_h - 1 &= 1 + f_{h-1} - 1 + f_{h-2} - 1\,, && \text{hence} \\ f_h &= f_{h-1} + f_{h-2} && = F_{h+2} \end{aligned}$$

7.3 AVL-Trees

Since

$$F(k) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^k \quad ,$$

an AVL-tree with n internal nodes has height $\Theta(\log n)$.

7.3 AVL-Trees

We need to maintain the balance condition through rotations.

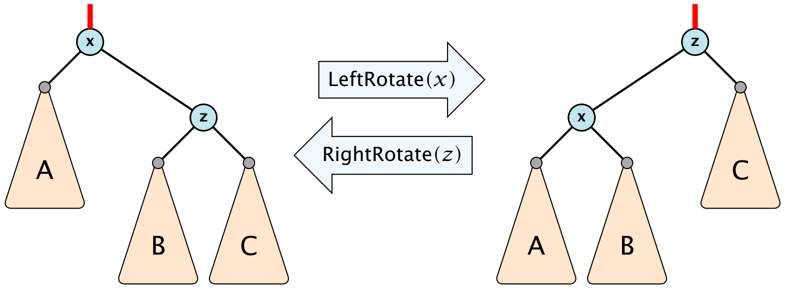
For this we store in every internal tree-node v the balance of the node. Let v denote a tree node with left child c_ℓ and right child c_r.

$$\text{balance}[v] := \text{height}(T_{c_\ell}) - \text{height}(T_{c_r}) \ ,$$

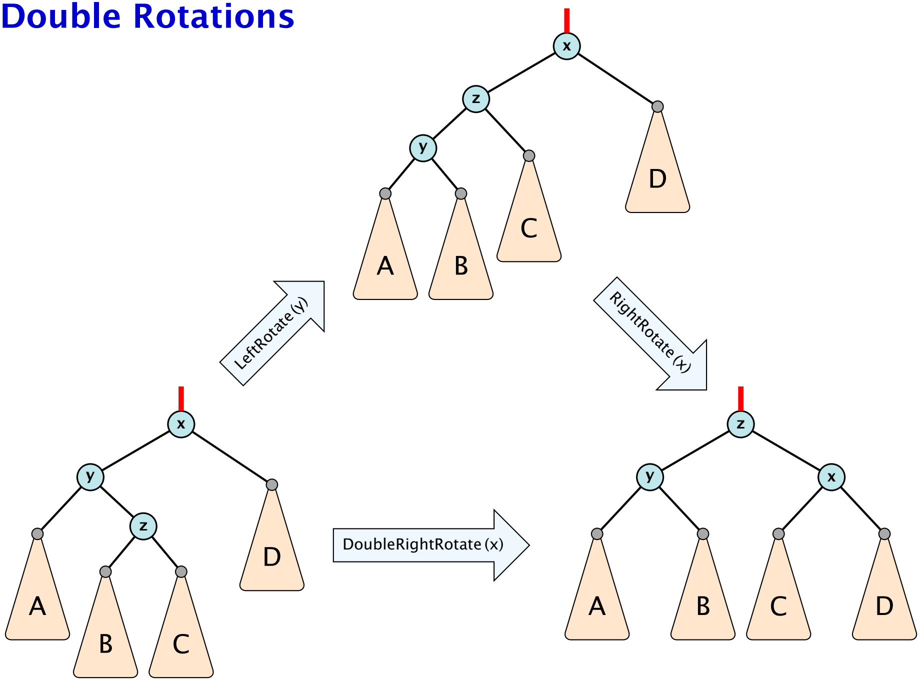
where T_{c_ℓ} and T_{c_r}, are the sub-trees rooted at c_ℓ and c_r, respectively.

Rotations

The properties will be maintained through rotations:

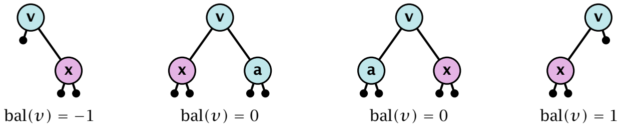


Double Rotations



AVL-trees: Insert

- Insert like in a binary search tree.
- Let v denote the parent of the newly inserted node x.
- One of the following cases holds:



- If $\text{bal}[v] \neq 0$, T_v has changed height; the balance-constraint may be violated at ancestors of v.
- Call fix-up$(\text{parent}[v])$ to restore the balance-condition.

AVL-trees: Insert

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at the node c fulfills $\mathrm{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Insert

```
Algorithm 11 AVL-fix-up-insert($v$)
1: if $\text{balance}[v] \in \{-2, 2\}$ then DoRotationInsert($v$);
2: if $\text{balance}[v] \in \{0\}$ return;
3: AVL-fix-up-insert($\text{parent}[v]$);
```

We will show that the above procedure is correct, and that it will do at most one rotation.

AVL-trees: Insert

Algorithm 12 DoRotationInsert(v)

```
if balance[$v$] = $-2$ then
    if balance[right[$v$]] = $-1$ then
        LeftRotate($v$);
    else
        DoubleLeftRotate($v$);
else
    if balance[left[$v$]] = $1$ then
        RightRotate($v$);
    else
        DoubleRightRotate($v$);
```

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

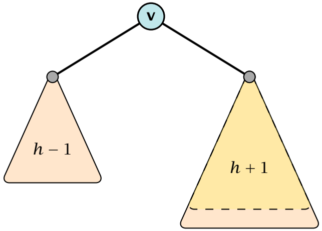
We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

We have the following situation:

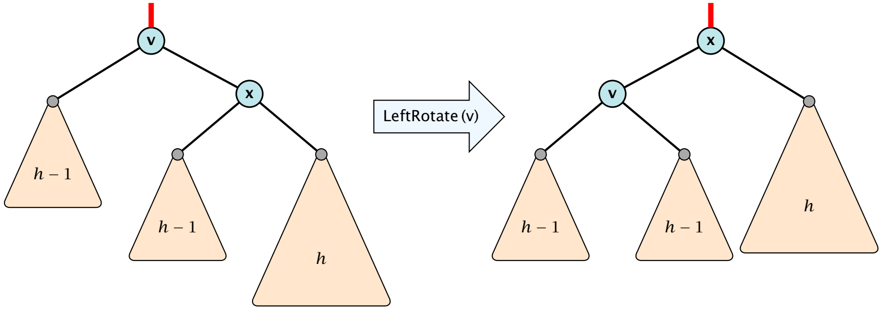


The right sub-tree of v has increased its height which results in a balance of -2 at v.

Before the insertion the height of T_v was $h+1$.

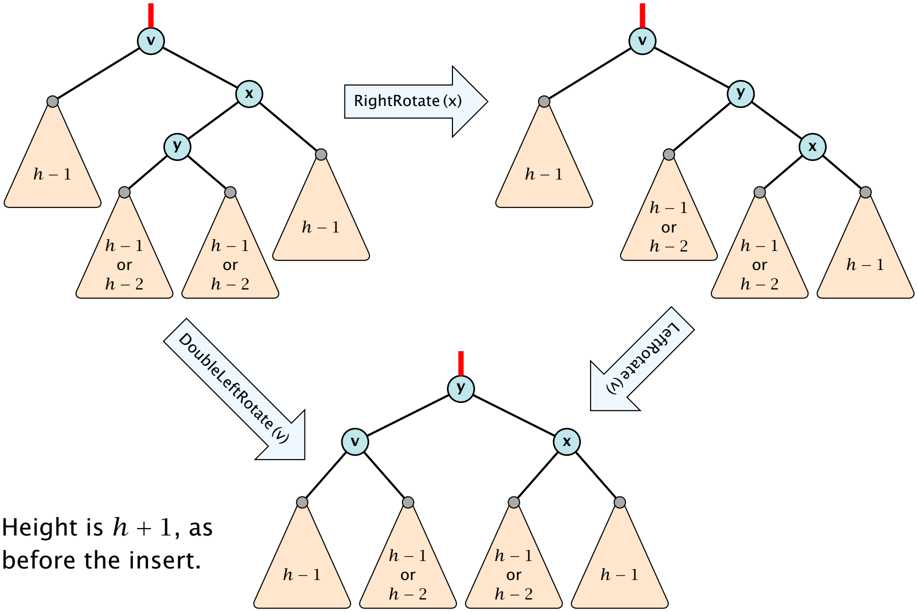
Case 1: balance[right[v]] = −1

We do a left rotation at v



Now, T_v has height $h+1$ as before the insertion. Hence, we do not need to continue.

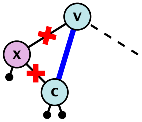
Case 2: balance[right[v]] = 1



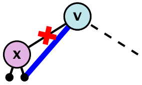
Height is $h+1$, as before the insert.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed— is either a dummy leaf or a node with two dummy leafs as children.



Case 1



Case 2

In both cases $\text{bal}[c] = 0$.

- Call fix-up(v) to restore the balance-condition.

AVL-trees: Delete

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has either decreased its height by one or it has stayed the same (note that this is clear right after the deletion but we have to make sure that it also holds after the rotations done within T_c in previous iterations).
4. The balance at the node c fulfills $\text{balance}[c] = \{0\}$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Delete

Algorithm 13 AVL-fix-up-delete(v)

```
1: if $\text{balance}[v] \in \{-2, 2\}$ then DoRotationDelete($v$);
2: if $\text{balance}[v] \in \{-1, 1\}$ return;
3: AVL-fix-up-delete($\text{parent}[v]$);
```

We will show that the above procedure is correct. However, for the case of a delete there may be a logarithmic number of rotations.

AVL-trees: Delete

Algorithm 14 DoRotationDelete(v)

```
1: if balance[$v$] = $-2$ then
2:     if balance[right[$v$]] = $-1$ then
3:         LeftRotate($v$);
4:     else
5:         DoubleLeftRotate($v$);
6: else
7:     if balance[left[$v$]] = $\{0, 1\}$ then
8:         RightRotate($v$);
9:     else
10:        DoubleRightRotate($v$);
```

AVL-trees: Delete

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

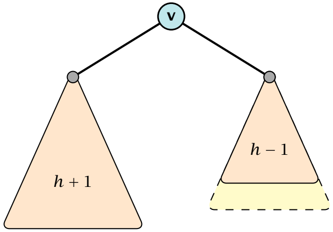
We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- If now $\mathrm{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

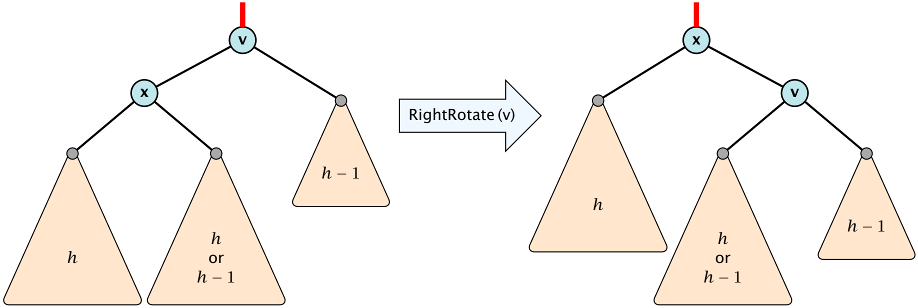
We have the following situation:



The right sub-tree of v has decreased its height which results in a balance of 2 at v.

Before the insertion the height of T_v was $h+2$.

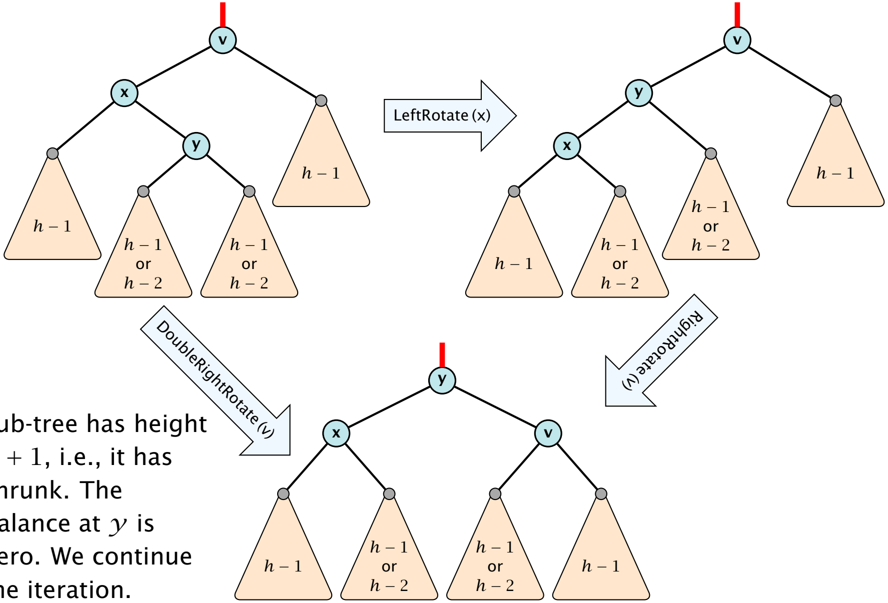
Case 1: balance[left[v]] ∈ {0, 1}



If the middle subtree has height h the whole tree has height $h + 2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

If the middle subtree has height $h - 1$ the whole tree has decreased its height from $h + 2$ to $h + 1$. We do continue the fix-up procedure as the balance at the root is zero.

Case 2: balance[left[v]] = −1



Sub-tree has height $h+1$, i.e., it has shrunk. The balance at y is zero. We continue the iteration.

7.4 (a, b)-trees

Definition 17

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.4 (a, b)-trees

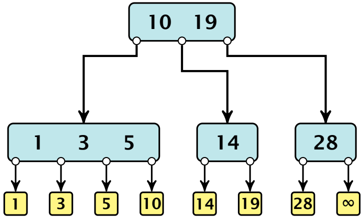
Each internal node v with $d(v)$ children stores $d - 1$ keys $k_1, \dots, k_d - 1$. The i-th subtree of v fulfills

$$k_{i-1} < \text{ key in } i\text{-th sub-tree } \leq k_i \;,$$

where we use $k_0 = -\infty$ and $k_d = \infty$.

7.4 (a, b)-trees

Example 18



7.4 (a, b)-trees

Variants

- The dummy leaf element may not exist; this only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as only $1/2$-full (the requirement of a B-tree).

Lemma 19

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \le n+1 \le b^h$
2. $\log_b(n+1) \le h \le \log_a(\frac{n+1}{2})$

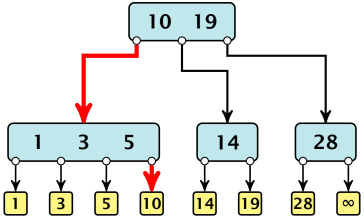
Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

□

Search

Search(8)

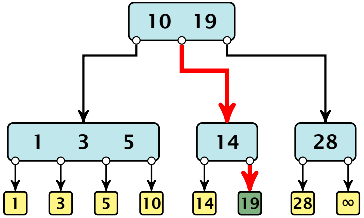


The search is straightforward. It is only important that you need to go all the way to the leaf.

Time: $\mathcal{O}(b \cdot h) = \mathcal{O}(b \cdot \log n)$, if the individual nodes are organized as linear lists.

Search

Search(19)



The search is straightforward. It is only important that you need to go all the way to the leaf.

Time: $\mathcal{O}(b \cdot h) = \mathcal{O}(b \cdot \log n)$, if the individual nodes are organized as linear lists.

Insert

Insert element x:

- Follow the path as if searching for $\text{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\text{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

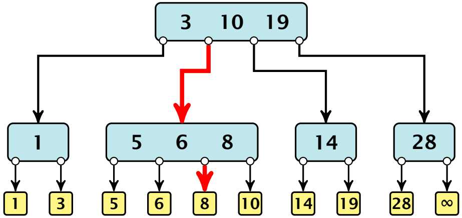
Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

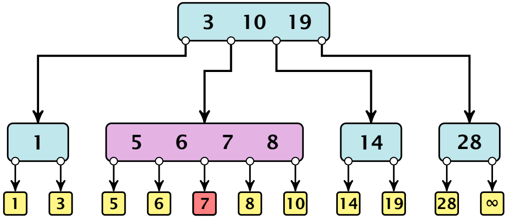
Insert

Insert(7)



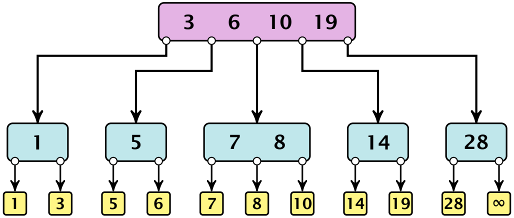
Insert

Insert(7)



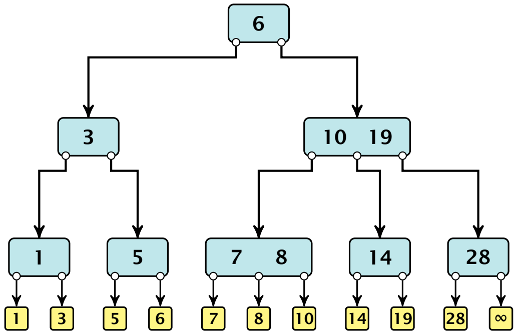
Insert

Insert(7)



Insert

Insert(7)



Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\mathrm{key}[x]$ is contained in v, remove the key from v, and delete the leaf vertex.
- Otherwise delete the key of the predecessor of x from v; delete the leaf vertex; and replace the occurrence of $\mathrm{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- If now the number of keys in v is below $a - 1$ perform Rebalance'(v).

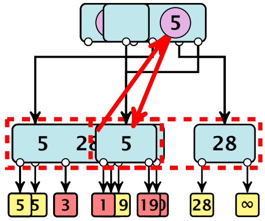
Delete

Rebalance'(v):

- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \le b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

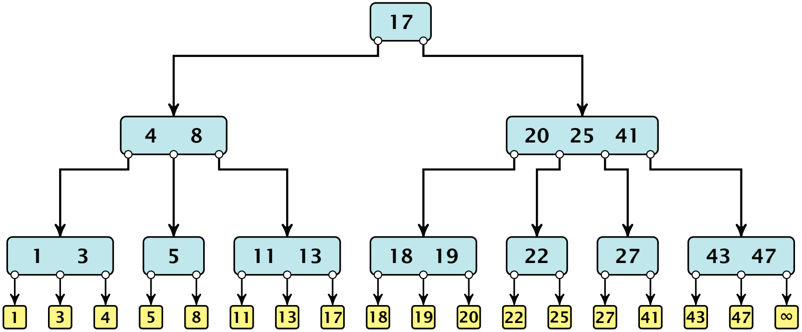
Delete

Delete(10) Delete(14)
Delete(3) Delete(1)
Delete(19)



$(2,4)$-trees and red black trees

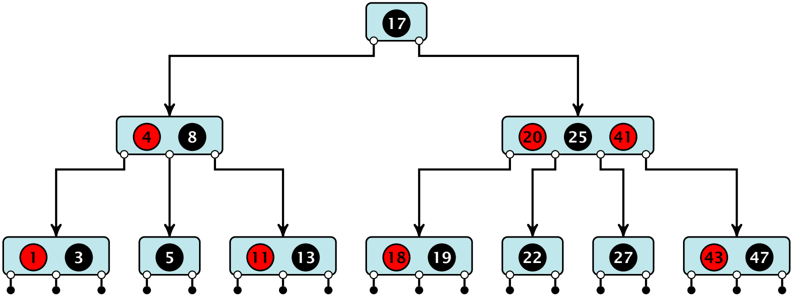
There is a close relation between red-black trees and $(2,4)$-trees:



First make it into an internal search tree by moving the satellite-data from the leaves to internal nodes. Add dummy leaves.

$(2,4)$-trees and red black trees

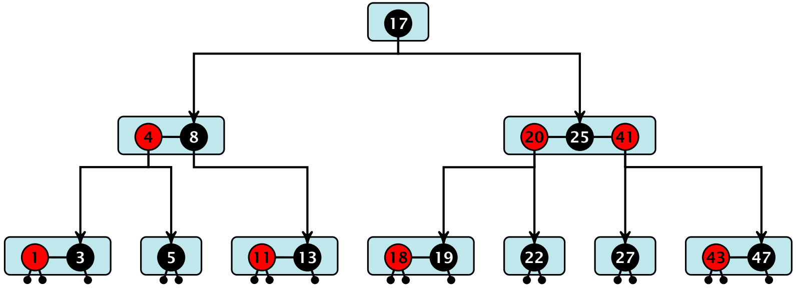
There is a close relation between red-black trees and $(2,4)$-trees:



Then, color one key in each internal node v black. If v contains 3 keys you need to select the middle key otherwise choose a black key arbitrarily. The other keys are colored red.

$(2,4)$-trees and red black trees

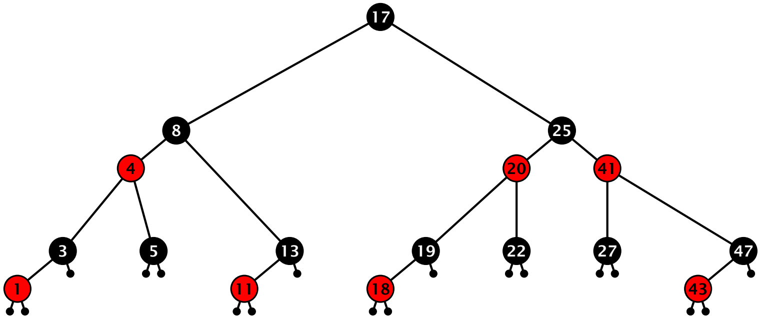
There is a close relation between red-black trees and $(2,4)$-trees:



Re-attach the pointers to individual keys. A pointer that is between two keys is attached as a child of the red key. The incoming pointer, points to the black key.

$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:

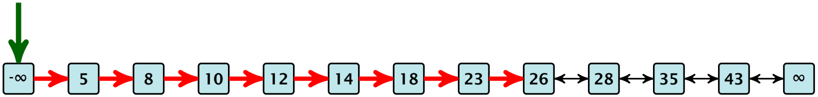


Note that this correspondence is not unique. In particular, there are different red-black trees that correspond to the same $(2,4)$-tree.

7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

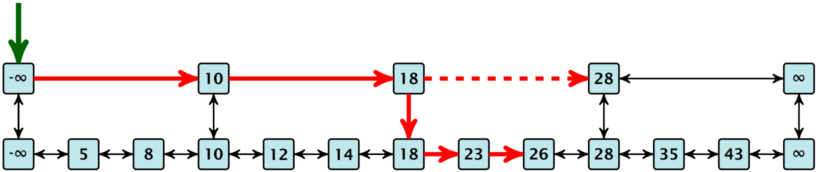
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:



Let $|L_1|$ denote the number of elements in the “express lane”, and $|L_0| = n$ the number of all elements (ignoring dummy elements).

Worst case search time: $|L_1| + \frac{|L_0|}{|L_1|}$ (ignoring additive constants)

Choose $|L_1| = \sqrt{n}$. Then search time $\Theta(\sqrt{n})$.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0,\dots,L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k|+2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil\frac{|L_{k-1}|}{|L_k|+1}\rceil+2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil\frac{|L_{k-2}|}{|L_{k-1}|+1}\rceil+2$ steps.
- ...
- At most $|L_k|+\sum_{i=1}^{k}\frac{L_{i-1}}{L_i}+3(k+1)$ steps.

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$. Choose

$$r = \sqrt[k+1]{n} \quad \Longrightarrow \quad \text{time: } \mathcal{O}(k\sqrt[k+1]{n})$$

Choosing $k = \Theta(\log k)$ gives a logarithmic running time.

7.5 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

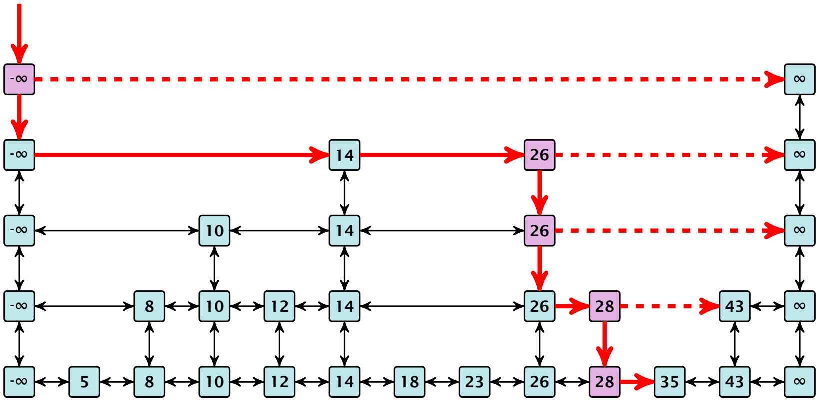
Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

Skip Lists

Insert (35):



7.5 Skip Lists

Lemma 20

A search (and, hence, also insert and delete) in a skip list with n elements takes time $\mathcal{O}(logn)$ with high probability (w. h. p.).

This means for any constant α the search takes time $\mathcal{O}(\log n)$ with probability at least $1 - \frac{1}{n^\alpha}$.

Note that the constant in the $\mathcal{O}$-notation may depend on α.

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

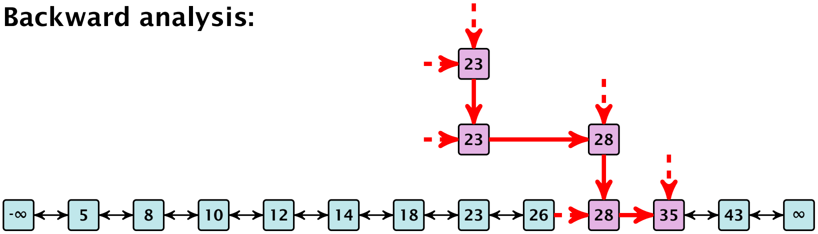
Then the probabilityx that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\leq 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha} \ .\end{aligned}$$

This means $\Pr[E_1 \wedge \cdots \wedge E_\ell]$ holds with high probability.

Skip Lists

Backward analysis:



At each point the path goes up with probability $1/2$ and left with probability $1/2$.

We show that w.h.p:

- A “long” search path must also go very high.
- There are no elements in high lists.

From this it follows that w.h.p. there are no long paths.

7.5 Skip Lists

Let $E_{z,k}$ denote the event that a search path is of length z (number of edges) but does not visit a list above L_k.

In particular, this means that during the construction in the backward analysis we see at most k heads (i.e., coin flips that tell you to go up) in z trials.

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k (2^{-\beta})^k \cdot n^{-\alpha} \leq \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

$$\leq \left(\frac{42\alpha}{64^\alpha}\right)^k n^{-\alpha} \leq n^{-\alpha}$$

for $\alpha \geq 1$.

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \ .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\begin{aligned}\Pr[\text{search requires } z \text{ steps}] &\leq \Pr[E_{z,k}] + \Pr[A_{k+1}] \\ &\leq n^{-\alpha} + n^{-(\gamma-1)}\end{aligned}$$

This means, the search requires at most z steps, w. h. p.

7.6 Augmenting Data Structures

Suppose you want to develop a data structure with:

- **Insert(x):** insert element x.
- **Search(k):** search for element with key k.
- **Delete(x):** delete element referenced by pointer x.
- **find-by-rank(ℓ):** return the k-th element; return “error” if the data-structure contains less than k elements.

Augment an existing data-structure instead of developing a new one.

7.6 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

- Of course, the above steps heavily depend on each other. For example it makes no sense to choose additional information to be stored (Step 2), and later realize that either the information cannot be maintained efficiently (Step 3) or is not sufficient to support the new operations (Step 4).
- However, the above outline is a good way to describe/document a new data-structure.

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.6 Augmenting Data Structures

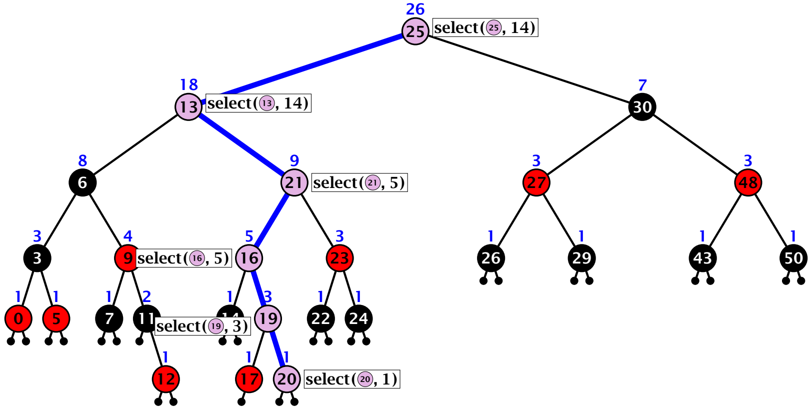
Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

4. How does find-by-rank work?
   Find-by-rank(k) ≔ Select(root, k) with

Algorithm 15 Select(x, i)

```
1: if $x = \text{null}$ then return $\text{error}$
2: if $\text{left}[x] \neq \text{null}$ then $r \leftarrow \text{left}[x].\text{size} + 1$ else $r \leftarrow 1$
3: if $i = r$ then return $x$
4: if $i < r$ then
5:     return Select$(\text{left}[x], i)$
6: else
7:     return Select$(\text{right}[x], i - r)$
```

Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

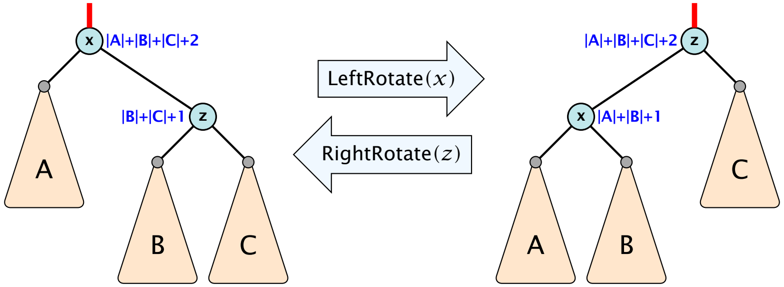
Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

Rotations

The only operation during the fix-up procedure that alters the tree and requires an update of the size-field:



The nodes x and z are the only nodes changing their size-fields.

The new size-fields can be computed locally from the size-fields of the children.

7.7 Hashing

Dictionary:

- $\boldsymbol{S}$**.insert**$(\boldsymbol{x})$: Insert an element x.
- $\boldsymbol{S}$**.delete**$(\boldsymbol{x})$: Delete the element pointed to by x.
- $\boldsymbol{S}$**.search**$(\boldsymbol{k})$: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7.7 Hashing

Definitions:

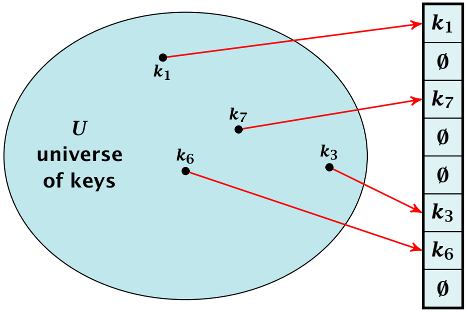
- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0, \dots, n-1]$ hash-table.
- Hash function $h : U \to [0, \dots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

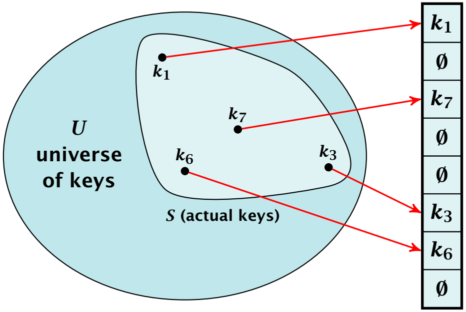
Ideally the hash function maps all keys to different memory locations.



This special case is known as Direct Addressing. It is usually very unrealistic as the universe of keys typically is quite large, and in particular larger than the available memory.

7.7 Hashing

Suppose that we know the set S of actual keys (no insert/no delete). Then we may want to design a simple hash-function that maps all these keys to different memory locations.



Such a hash function h is called a perfect hash function for set S.

7.7 Hashing

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions
Usually the universe U is much larger than the table-size n.

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

7.7 Hashing

Typically, collisions do not appear once the size of the set S of actual keys gets close to n, but already once $|S| \geq \omega(\sqrt{n})$.

Lemma 21

The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

$$1 - e^{-\frac{m(m-1)}{2}} \approx 1 - e^{-\frac{m^2}{2n}} \; .$$

Uniform hashing:

Choose a hash function uniformly at random from all functions $f : U \to [0, \dots, n-1]$.

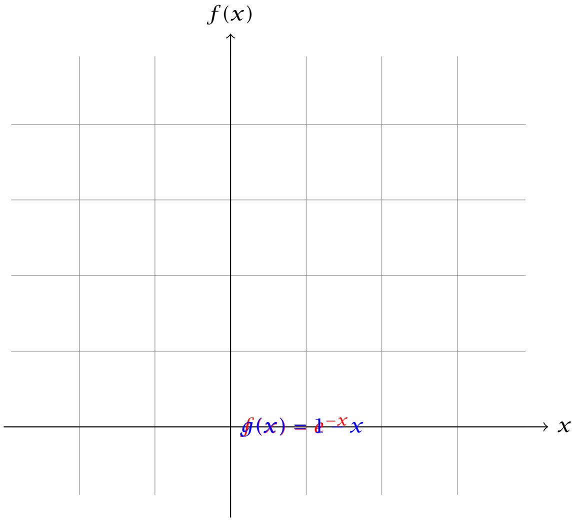
7.7 Hashing

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\begin{aligned}\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\Big(1-\frac{j}{n}\Big) \\ &\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \enspace .\end{aligned}$$

Here the first equality follows since the ℓ-th element that is hashed has a probability of $\frac{n-\ell+1}{n}$ to not generate a collision under the condition that the previous elements did not induce collisions. $\square$



The inequality $1 - x \leq e^{-x}$ is derived by stopping the tayler-expansion of e^{-x} after the second term.

Resolving Collisions

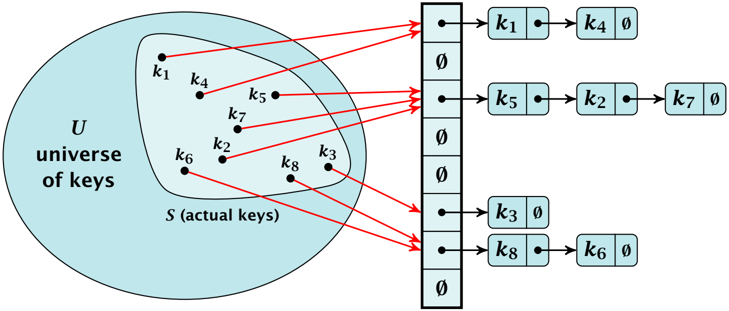
The methods for dealing with collisions can be classified into the two main types

- **open addressing**, aka. closed hashing
- **hashing with chaining.** aka. closed addressing, open hashing.

Hashing with Chaining

Arrange elements that map to the same position in a linear list.

- Access: compute $h(x)$ and search list for $\mathrm{key}[x]$.
- Insert: insert at the front of the list.



7.7 Hashing

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a successful search when using A;
- A^- denotes the average time for an unsuccessful search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$. Hence, if A is the collision resolving strategy “Hashing with Chaining” we have

$$A^- = 1 + \alpha \ .$$

Note that this result does not depend on the hash-function that is used.

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\underbrace{\left(1+\overbrace{\sum_{j=i+1}^{m}X_{ij}}^{\text{keys before } k_i}\right)}_{\text{cost for key } k_i}\right]$$

Hashing with Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right)\\
&= 1+\frac{m-1}{2n} = 1+\frac{\alpha}{2}-\frac{\alpha}{2m}\ .
\end{aligned}
$$

Hence, the expected cost for a successful search is $A^+ \leq 1+\frac{\alpha}{2}$.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k,j)$ that determines the table-position to be examined in the j-th step. The values $h(k,0)$,. . . ,$h(k,n-1)$ form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k,0)$; if it is empty your search fails; otw. continue with $h(k,1)$, $h(k,2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k,n-1)$, and this slot is non-empty then your table is full.

Open Addressing

Choices for $h(k, j)$:

- $h(k,i) = h(k) + i \mod n$. Linear probing.
- $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k,i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n; for quadratic probing c_1 and c_2 have to be chosen carefully).

Linear Probing

- Advantage: Cache-efficiency. The new probe position is very likely to be in the cache.
- Disadvantage: Primary clustering. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

Lemma 22

Let L be the method of linear probing for resolving collisions:

$$L^+ \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$L^- \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Quadratic Probing

- Not as cache-efficient as Linear Probing.
- Secondary clustering: caused by the fact that all keys mapped to the same position have the same probe sequence.

Lemma 23

Let Q be the method of quadratic probing for resolving collisions:

$$Q^+ \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^- \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

Double Hashing

- Any probe into the hash-table usually creates a cash-miss.

Lemma 24

Let A be the method of double hashing for resolving collisions:

$$D^+ \approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

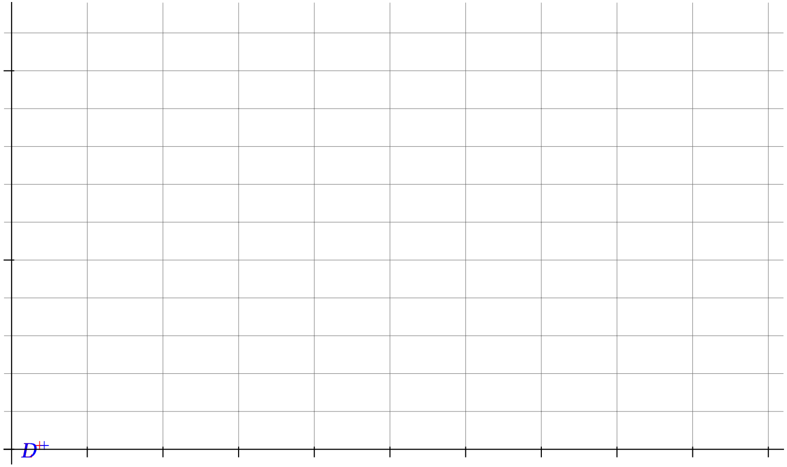
$$D^- \approx \frac{1}{1-\alpha}$$

7.7 Hashing

Some values:

α	Linear Probing		Quadratic Probing		Double Hashing	
	L^+	L^-	Q^+	Q^-	D^+	D^-
0.5	1.5	2.5	1.44	2.19	1.39	2
0.9	5.5	50.5	2.85	11.40	2.55	10
0.95	10.5	200.5	3.52	22.05	3.15	20

7.7 Hashing



Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}] \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

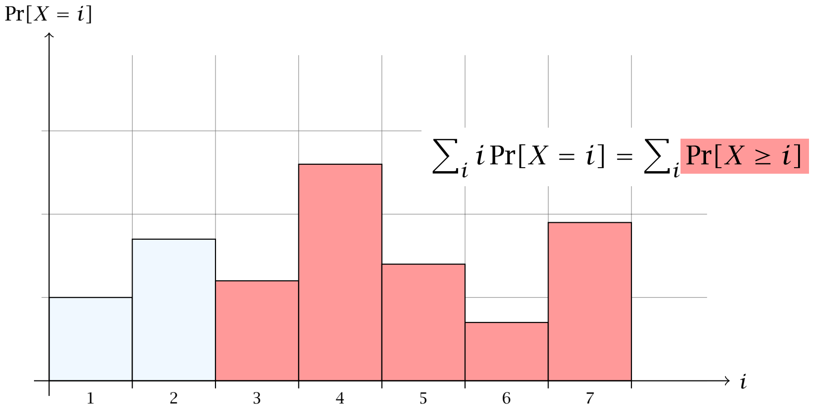
$$\begin{aligned}\Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1} = \alpha^{i-1} \enspace .\end{aligned}$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \enspace .$$

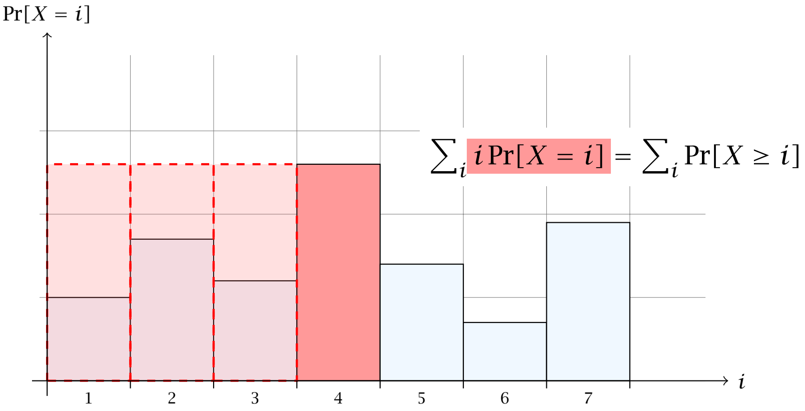
$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

$i = 3$



The j-th rectangle appears in both sums j times. (j times in the first due to multiplication with j; and j times in the second for summands $i = 1, 2, \ldots, j$)

$i = 4$



The j-th rectangle appears in both sums j times. (j times in the first due to multiplication with j; and j times in the second for summands $i = 1, 2, \ldots, j$)

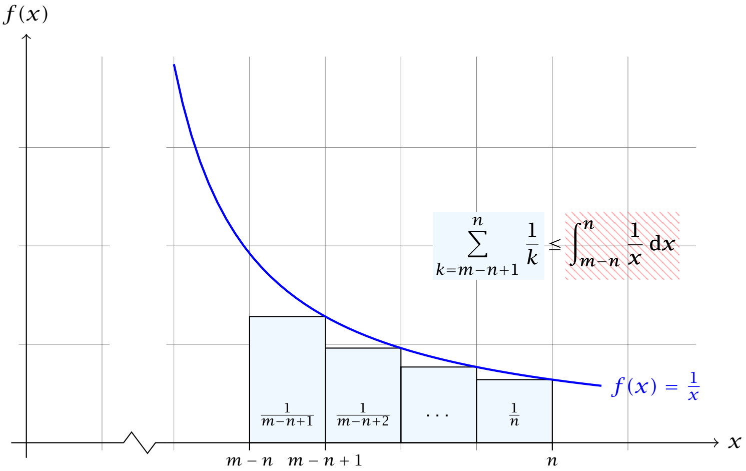
Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\le \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x = \frac{1}{\alpha}\ln\frac{n}{n-m} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \; .$$



$f(x)$
x
$\sum_{k=m-n+1}^{n} \frac{1}{k} \le \int_{m-n}^{n} \frac{1}{x}\,\mathrm{d}x$
$f(x) = \frac{1}{x}$
$\frac{1}{m-n+1}$
$\frac{1}{m-n+2}$
$\dots$
$\frac{1}{n}$
$m-n$
$m-n+1$
n

7.7 Hashing

How do we delete in a hash-table?

- For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- For open addressing this is difficult.

7.7 Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

7.7 Hashing

Definition 25

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\dots,n-1\}$ is called universal if for all $u_1, u_2 \in U$ with $u_1 \neq u_2$

$$\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n} \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Note that this means that $\Pr[h(u_1) = h(u_2)] = \frac{1}{n}$.

7.7 Hashing

Definition 26

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\dots,n-1\}$ is called 2-independent (pairwise independent) if the following two conditions hold

- For any key $u \in U$, and $t \in \{0,\dots,n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, i.e., a key is distributed uniformly within the hash-table.
- For all $u_1, u_2 \in U$ with $u_1 \neq u_2$, and for any two hash-positions t_1, t_2:

$$\Pr[h(u_1) = t_1 \wedge h(u_2) = t_2] \leq \frac{1}{n^2} \ .$$

Note that the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

This requirement clearly implies a universal hash-function.

7.7 Hashing

Definition 27

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\dots,n-1\}$ is called *k-independent* if for any choice of $\ell \leq k$ distinct keys $u_1,\dots,u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1,\dots,t_\ell$:

$$\Pr[h(u_1) = t_1 \wedge \cdots \wedge h(u_\ell) = t_\ell] \leq \frac{1}{n^\ell} \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

7.7 Hashing

Definition 28

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\dots,n-1\}$ is called (μ,k)-independent if for any choice of $\ell \le k$ distinct keys $u_1,\dots,u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1,\dots,t_\ell$:

$$\Pr[h(u_1) = t_1 \wedge \cdots \wedge h(u_\ell) = t_\ell] \le \left(\frac{\mu}{n}\right)^\ell \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

7.7 Hashing

Let $U := \{0, \dots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \dots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \dots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 29

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \dots, n-1\}$.

7.7 Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

  This holds because we can compute a and b when given t_x and t_y:

$$\begin{aligned} t_x &\equiv ax + b &&\pmod{p} \\ t_y &\equiv ay + b &&\pmod{p} \\[1em] t_x - t_y &\equiv a(x - y) &&\pmod{p} \\ t_y &\equiv ay + b &&\pmod{p} \\[1em] a &\equiv (t_x - t_y)(x - y)^{-1} &&\pmod{p} \\ b &\equiv ay - t_y &&\pmod{p} \end{aligned}$$

7.7 Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \dots, p - 1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

7.7 Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

7.7 Hashing

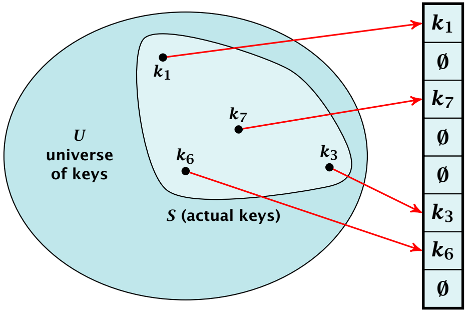
It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

$$\frac{\left\lfloor \frac{p}{n} \right\rfloor^2}{p(p-1)} \le \Pr_{t_x \neq t_y \in \mathbb{Z}_p^2}\left[\begin{array}{c} t_x \bmod n = h_1 \\ \wedge \\ t_y \bmod n = h_2 \end{array} \right] \le \frac{\left\lceil \frac{p}{n} \right\rceil^2}{p(p-1)}$$

Note that the middle is the probability that $h(x) = h_1$ and $h(y) = h_2$. The total number of choices for (t_x, t_y) is $p(p-1)$. The number of choices for t_x (t_y) such that $t_x \bmod n = h_1$ ($t_y \bmod n = h_2$) lies between $\lfloor \frac{p}{n} \rfloor$ and $\lceil \frac{p}{n} \rceil$.

Perfect Hashing

Suppose that we know the set S of actual keys (no insert/no delete). Then we may want to design a simple hash-function that maps all these keys to different memory locations.



Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$\begin{aligned}\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big]\\ &= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]\end{aligned}$$

The first expectation is simply the expected number of collisions, for the first level.

$$= 2\binom{m}{2}\frac{1}{m} + m = 2m - 1$$

Perfect Hashing

We need only $\mathcal{O}(m)$ time to construct a hash-function h with $\sum_j m_j^2 = \mathcal{O}(4m)$.

Then we construct a hash-table h_j for every bucket. This takes expected time $\mathcal{O}(m_j)$ for every bucket.

We only need that the hash-function is universal!!!

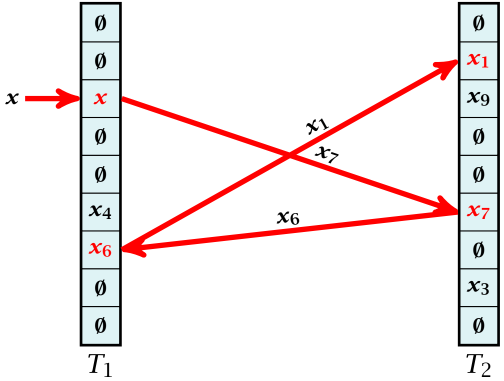
Cuckoo Hashing

Goal:
Try to generate a perfect hash-table (constant worst-case search time) in a dynamic scenario.

- Two hash-tables $T_1[0,\dots,n-1]$ and $T_2[0,\dots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

Cuckoo Hashing

Insert:



Cuckoo Hashing

Algorithm 16 Cuckoo-Insert(x)

```
1: if $T_1[h_1(x)] = x \vee T_2[h_2(x)] = x$ then return
2: $\text{steps} \leftarrow 1$
3: while $\text{steps} \le \text{maxsteps}$ do
4:         exchange $x$ and $T_1[h_1(x)]$
5:         if $x = \text{null}$ then return
6:         exchange $x$ and $T_2[h_2(x)]$
7:         if $x = \text{null}$ then return
8: rehash() // change table-size and rehash everything
9: Cuckoo-Insert($x$)
```

Cuckoo Hashing

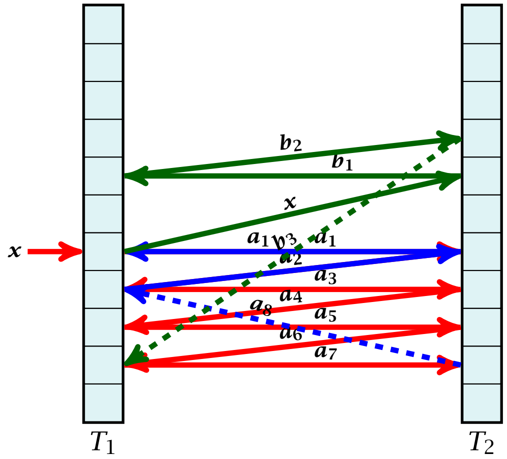
What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

Cuckoo Hashing

Insert:



Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

Observation If we end up in an infinite loop there must exist a cycle-structure that is active for x.

Cuckoo Hashing

A cycle-structure is defined without knowing the hash-functions.

Whether a cycle-structure is active for key x depends on the hash-functions.

Lemma 30

A given cycle-structure of size s is active for key x with probability at most

$$\left(\frac{\mu}{n}\right)^{2(s+1)} ,$$

if we use $(\mu, s+1)$-independent hash-functions.

Cuckoo Hashing

Proof.

All positions are fixed by the cycle-structure. Therefore we ask for the probability of mapping $s+1$ keys (the a-keys, the b-keys and x) to pre-specified positions in T_1, **and** to pre-specified positions in T_2.

The probability is

$$\left(\frac{\mu}{n}\right)^{s+1} \cdot \left(\frac{\mu}{n}\right)^{s+1} \enspace ,$$

since h_1 and h_2 are chosen independently. $\square$

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1,\dots,a_{\ell_a}$ and $b_1,\dots,b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1,\dots,p_{\ell_a}$ and $p'_1,\dots,p'_{\ell_a}$.

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

The probability that there is an active cycle-structure of size s is at most

$$\begin{aligned} s^3(mn)^s \cdot \left(\frac{\mu}{n}\right)^{2(s+1)} &= \frac{s^3}{mn}\left(mn\right)^{s+1}\left(\frac{\mu^2}{n^2}\right)^{s+1} \\ &= \frac{s^3}{mn}\left(\frac{\mu^2 m}{n}\right)^{s+1} \end{aligned}$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$\begin{aligned}
&\Pr[\text{there exists an active cycle-structure}] \\
&\qquad \leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s] \\
&\qquad \leq \sum_{s=2}^{\infty} \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1} \\
&\qquad \leq \frac{1}{mn}\sum_{s=0}^{\infty} s^3\Big(\frac{1}{1+\delta}\Big)^{s} \\
&\qquad \leq \frac{1}{m^2}\cdot \mathcal{O}(1)\enspace .
\end{aligned}$$

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \leq 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if $h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$, $h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$, $h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$, $h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Cuckoo Hashing

Observation:
If the insert takes $t \geq 4\ell$ steps there must either be a left-active or a right-active sub-sequence of length ℓ starting with x.

Cuckoo Hashing

The probability that a given sub-sequence is left-active (right-active) is at most

$$\left(\frac{\mu}{n}\right)^{2\ell} ,$$

if we use (μ, ℓ)-independent hash-functions. This holds since there are ℓ keys whose hash-values (two values per key) have to map to pre-specified positions.

Cuckoo Hashing

The number of sequences is at most $m^{\ell-1}p^{\ell+1}$ as we can choose $\ell-1$ keys (apart from x) and we can choose $\ell+1$ positions $p_0,\dots,p_\ell$.

The probability that there exists a left-active **or** right-active sequence of length ℓ is at most

$$\begin{aligned}\Pr[\text{there exists active sequ. of length } \ell] \\ &\le 2\cdot m^{\ell-1}\cdot n^{\ell+1}\cdot\Big(\frac{\mu}{n}\Big)^{2\ell} \\ &\le 2\Big(\frac{1}{1+\delta}\Big)^{\ell}\end{aligned}$$

Cuckoo Hashing

If the search does not run into an infinite loop the probability that it takes more than 4ℓ steps is at most

$$2\Big(\frac{1}{1+\delta}\Big)^{\ell}$$

We choose $\mathrm{maxsteps} = 4(1+2\log m)/\log(1+\delta)$. Then the probability of terminating the while-loop because of reaching $\mathrm{maxsteps}$ is only $\mathcal{O}(\frac{1}{m^2})$ ($\mathcal{O}(1/m^2)$ because of reaching an infinite loop and $1/m^2$ because the search takes $\mathrm{maxsteps}$ steps without running into a loop).

Cuckoo Hashing

The expected time for an insert under the condition that $\mathrm{maxsteps}$ is not reached is

$$\sum_{\ell\geq 0}\Pr[\text{search takes at least } \ell \text{ steps} \mid \text{iteration successful}]$$

$$\leq \sum_{\ell\geq 0} 8\Bigl(\frac{1}{1+\delta}\Bigr)^{\ell} = \mathcal{O}(1)\enspace .$$

More generally, the above expression gives a bound on the cost in the successful iteration of an insert-operation (there is exactly one successful iteration).

An iteration that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash.

Cuckoo Hashing

The expected number of unsuccessful operations is $\mathcal{O}(\frac{1}{m^2})$.
Hence, the expected cost in unsuccessful iterations is only $\mathcal{O}(\frac{1}{m})$.

Hence, the total expected cost for an insert-operation is constant.

Cuckoo Hashing

What kind of hash-functions do we need?

Since $\mathrm{maxsteps}$ is $\Theta(\log m)$ it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

How do we make sure that $n \geq \mu^2(1+\delta)m$?

- Let $\alpha := 1/(\mu^2(1+\delta))$.
- Keep track of the number of elements in the table. Whenever $m \geq \alpha n$ we double n and do a complete re-hash (table-expand).
- Whenever m drops below $\frac{\alpha}{4}n$ we divide n by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \frac{\alpha}{2}n$. In order for a table-expand to occur at least $\frac{\alpha}{2}n$ insertions are required. Similar, for a table-shrink at least $\frac{\alpha}{4}$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Definition 31

Let $d \in \mathbb{N}$; $q \geq n$ be a prime; and let $\vec{a} \in \{0, \ldots, q-1\}^{d+1}$. Define for $x \in \{0, \ldots, q\}$

$$h_{\vec{a}}(x) := \Big(\sum_{i=0}^{d} a_i x^i \bmod q \Big) \bmod n \ .$$

Let $\mathcal{H}_n^d := \{h_{\vec{a}} \mid \vec{a} \in \{0, \ldots, q\}^{d+1}\}$. The class $\mathcal{H}_n^d$ is $(2, d+1)$-independent.

For the coefficients $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i x^i \Big) \bmod q$$

The polynomial is defined by $d+1$ distinct points.

Fix $\ell \leq d+1$; let $x_1,\dots,x_\ell \in \{0,\dots,q-1\}$ be keys, and let $t_1,\dots,t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i$ for all $i \in \{1,\dots,\ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0,\dots,\lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

$$|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \leq \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}} \in A_\ell$.

Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \leq \left(\frac{2}{n}\right)^\ell$$