7 Dictionary

Dictionary:

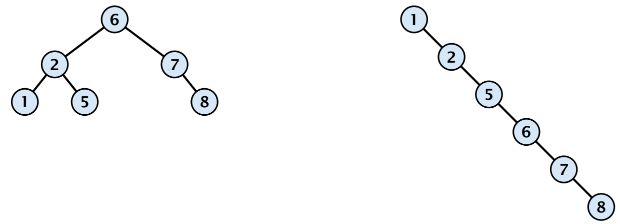
- $S.\text{insert}(x)$: Insert an element x.
- $S.\text{delete}(x)$: Delete the element pointed to by x.
- $S.\text{search}(k)$: Return a pointer to an element e with $\text{key}[e] = k$ in S if it exists; otherwise return null.

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 $\text{key}[v]$ 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:



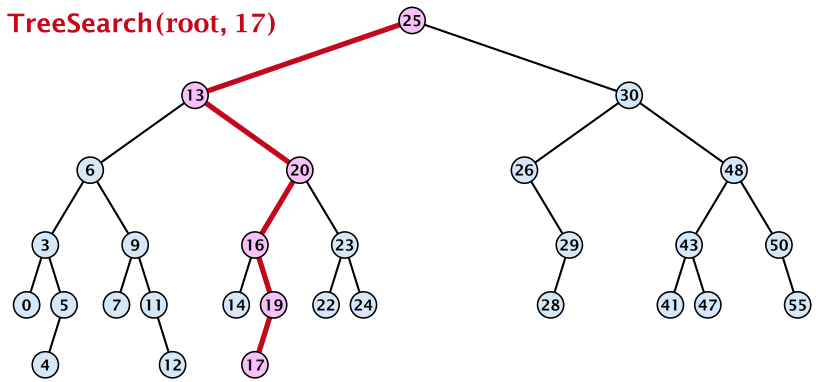
7.1 Binary Search Trees

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

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

Binary Search Trees: Searching

TreeSearch(root, 17)

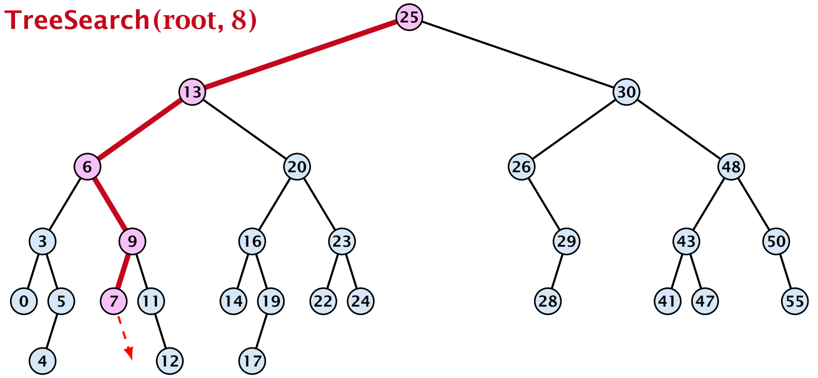


Algorithm 1 TreeSearch(x, k)

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

Binary Search Trees: Searching

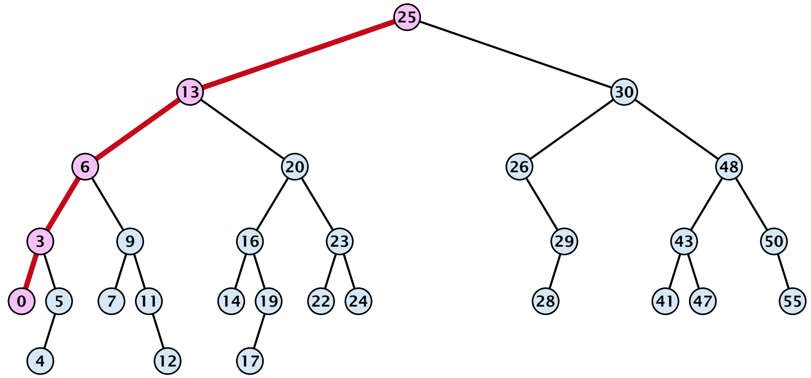
TreeSearch(root, 8)



Algorithm 1 TreeSearch(x, k)

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

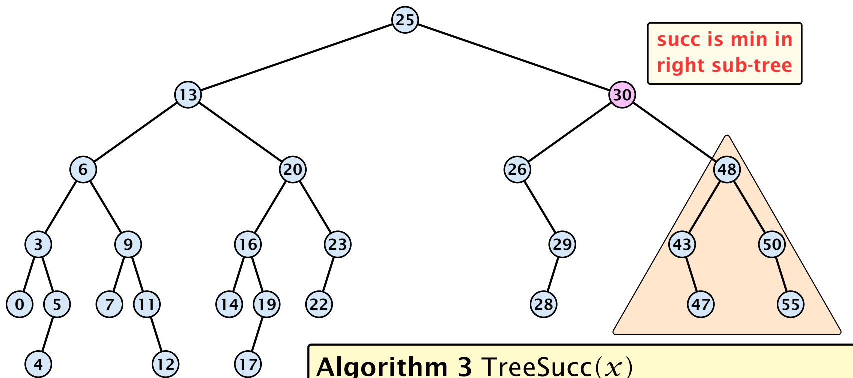
Binary Search Trees: Minimum



Algorithm 2 TreeMin(x)

1: **if** $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x
2: **return** TreeMin$(\text{left}[x])$

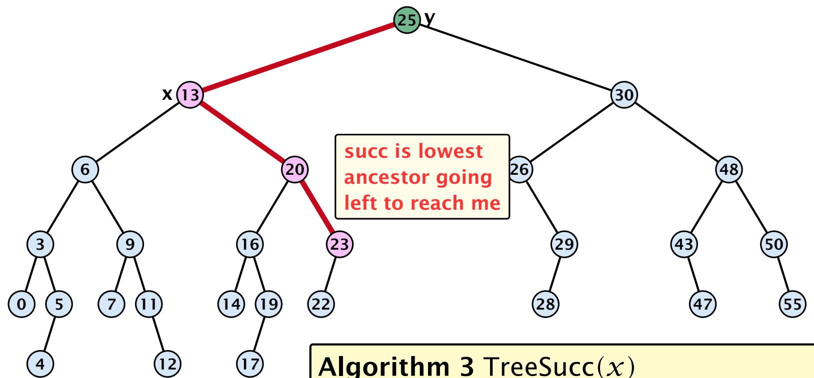
Binary Search Trees: Successor



Algorithm 3 TreeSucc(x)

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

Binary Search Trees: Successor



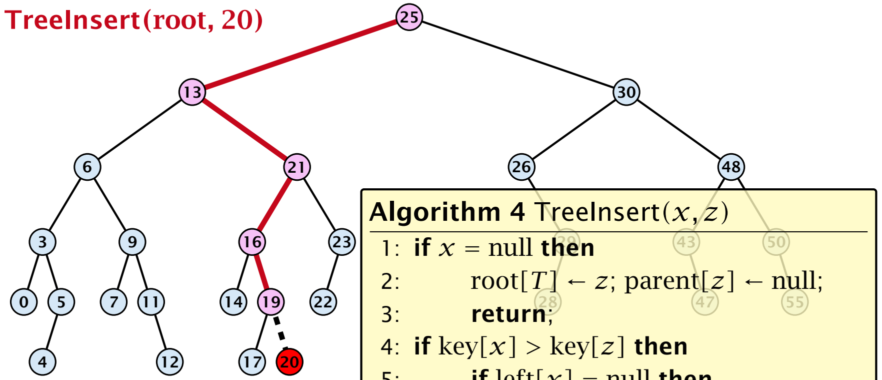
Algorithm 3 TreeSucc(x)

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

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

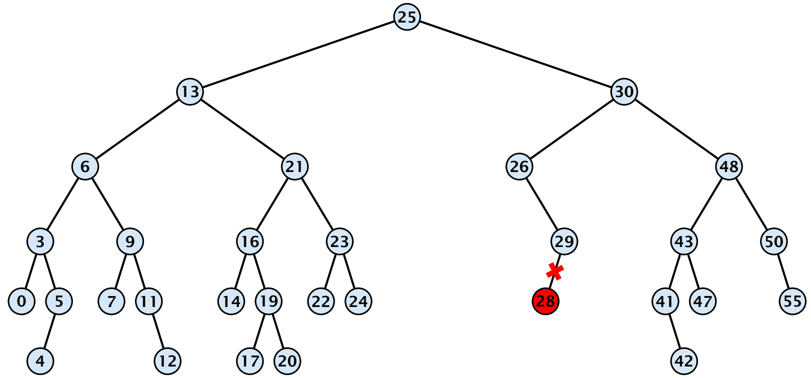


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Algorithm 4 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Delete

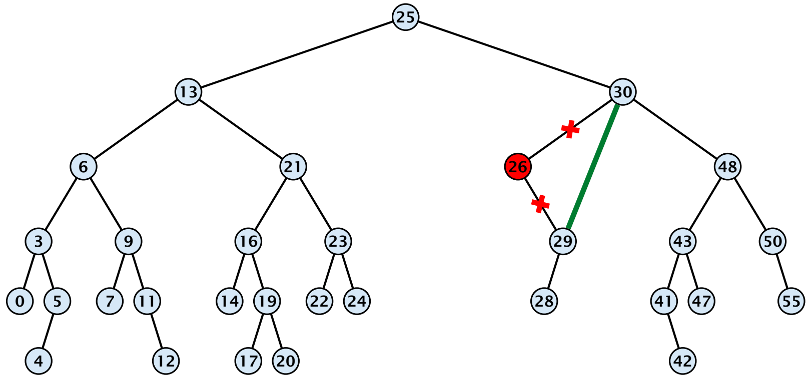


Case 1:

Element does not have any children

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

Binary Search Trees: Delete

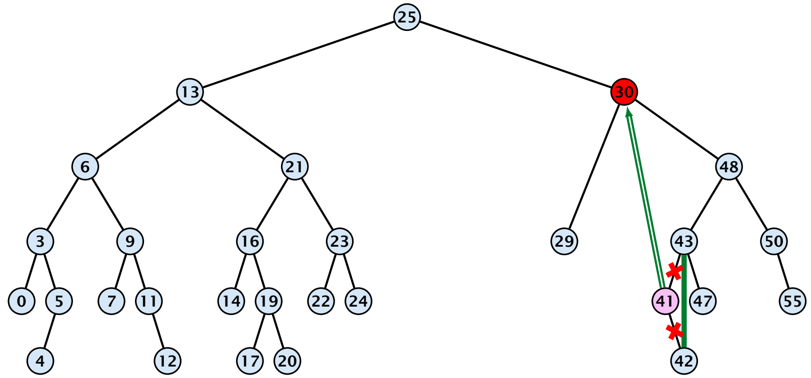


Case 2:

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

(lines 6–12: fix pointer to x)

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.

Binary Search Trees (BSTs)

Bibliography

[MS08] Kurt Mehlhorn, Peter Sanders:
Algorithms and Data Structures — The Basic Toolbox,
Springer, 2008

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
Introduction to Algorithms (3rd ed.),
MIT Press and McGraw-Hill, 2009

Binary search trees can be found in every standard text book. For example Chapter 7.1 in [MS08] and Chapter 12 in [CLRS90].

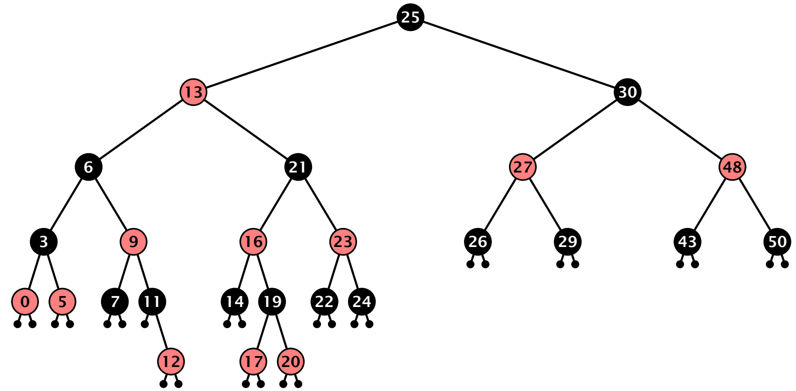
7.2 Red Black Trees

Definition 1
A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, 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 2
A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 3
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 4
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.

$\square$

7.2 Red Black Trees

Proof of Lemma 2.

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, $2^{h/2} - 1 \le n$.

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

7.2 Red Black Trees

Definition 1

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

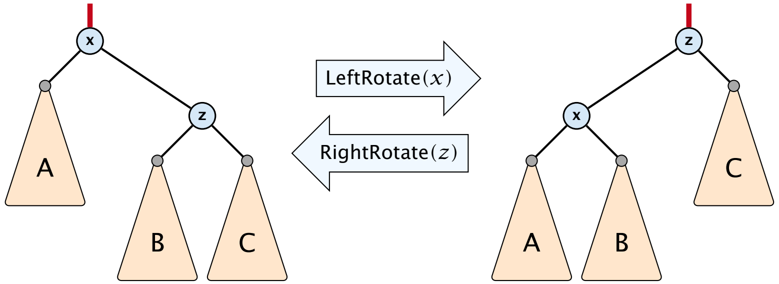
The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data.

7.2 Red Black Trees

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

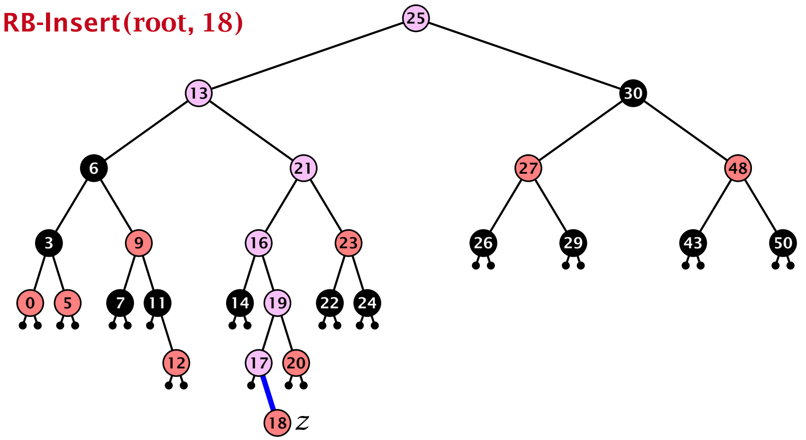
Rotations

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)

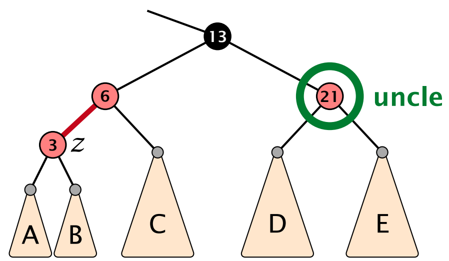
If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

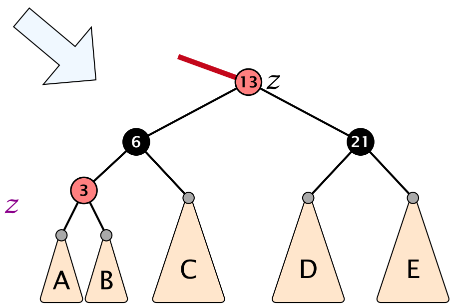
Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then          z in left subtree of grandparent
        uncle ← right[grandparent[z]]
        if col[uncle] = red then              Case 1: uncle red
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else                                  Case 2: uncle black
            if z = right[parent[z]] then      2a: z right child
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;   2b: z left child
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```


Case 1: Red Uncle

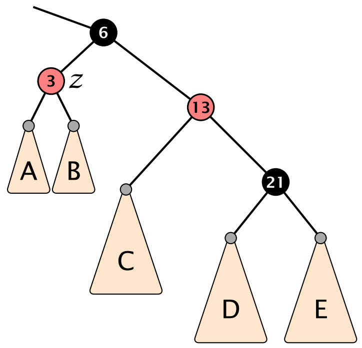


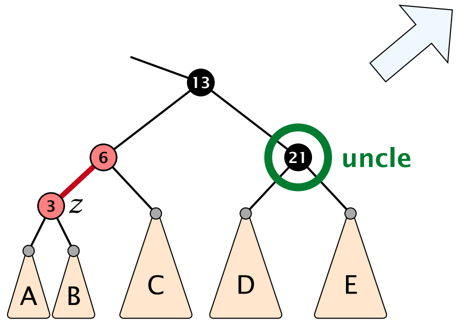
1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress



Case 2b: Black uncle and z is left child

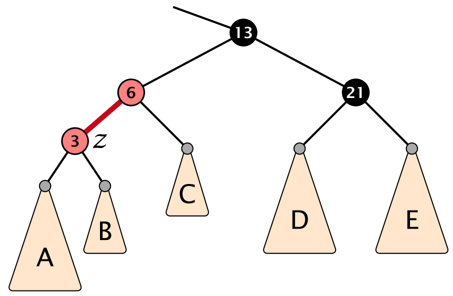
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

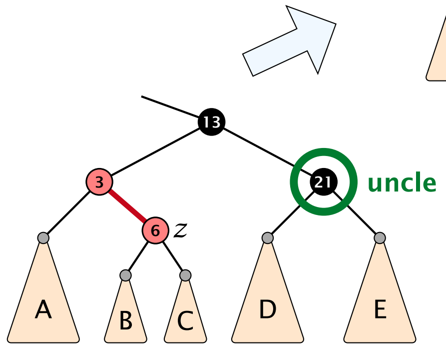




Case 2a: Black uncle and z is right child

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 Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

Red Black Trees: Delete

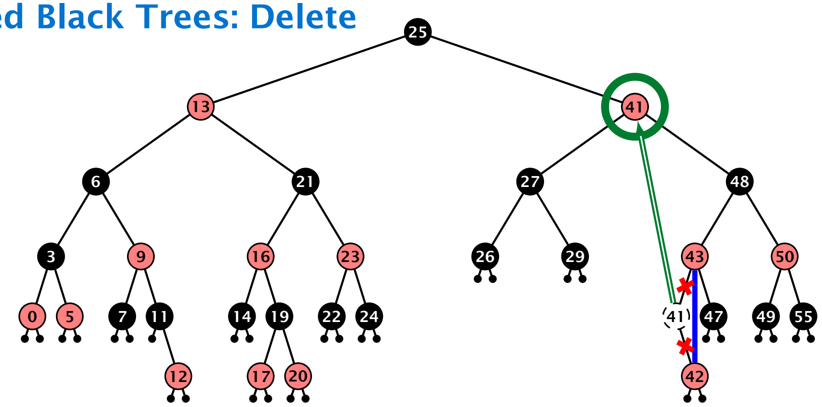
First do a standard delete.

If the spliced out node x was red everything 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

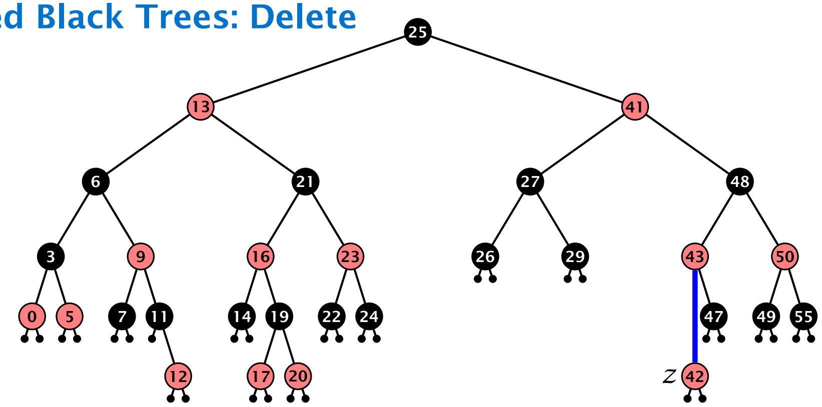


Case 3:

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.

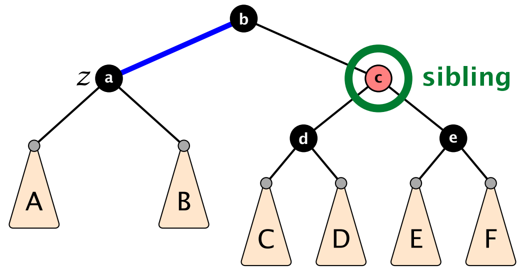
Red Black Trees: Delete

Invariant of the fix-up algorithm

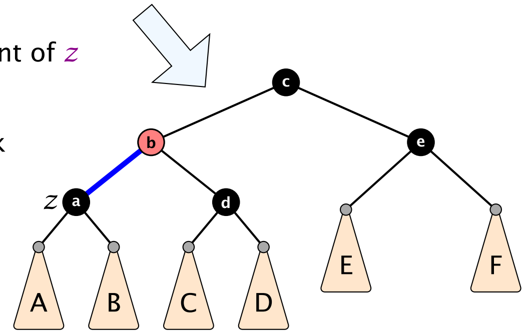
- the node z is black
- if we "assign" a fake black unit to the edge from z to its parent then the black-height property is fulfilled

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

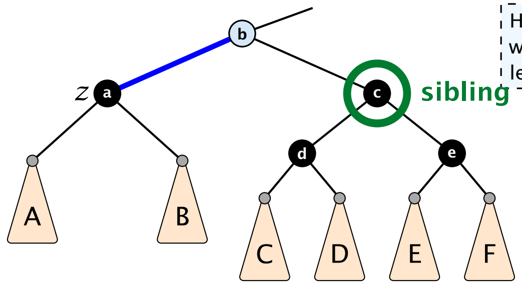
Case 1: Sibling of z is red



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

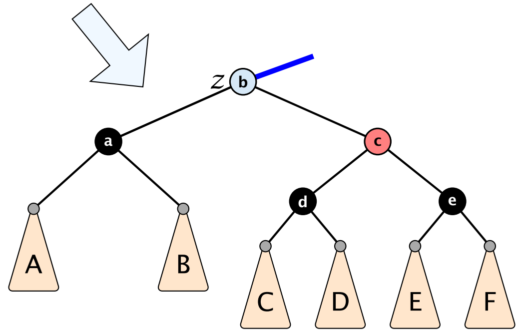


Case 2: Sibling is black with two black children



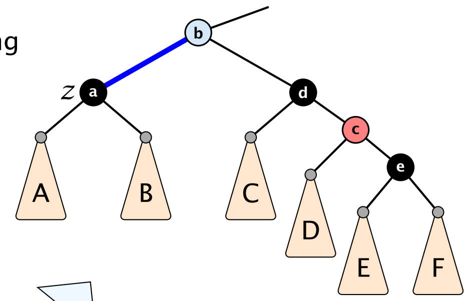
Here b is either black or red. If it is red we are in a special case that directly leads to a red-black tree.

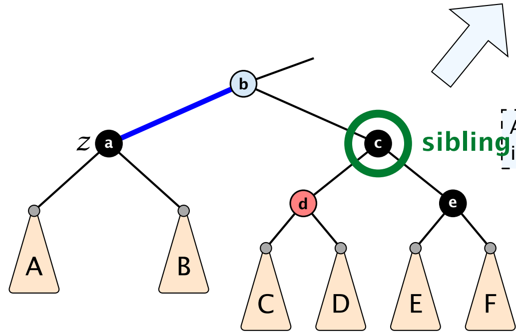
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



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

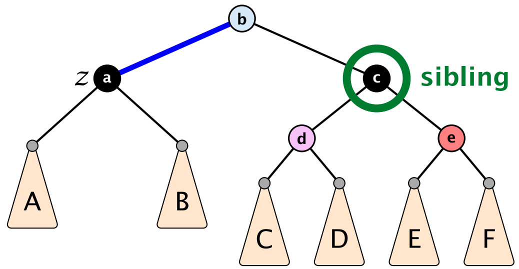
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)





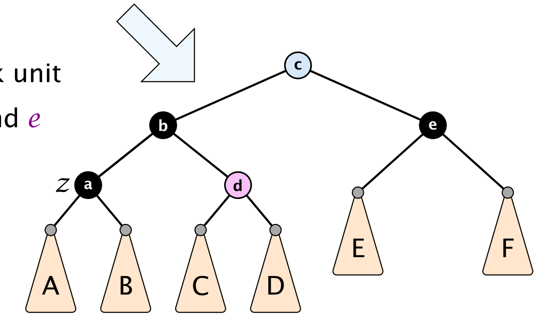
Again the blue color of b indicates that it can either be black or red.

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. remove the fake black unit
3. recolor nodes b, c, and e
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 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most 3 rotations.

Red-Black Trees

Bibliography

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
Introduction to Algorithms (3rd ed.),
MIT Press and McGraw-Hill, 2009

Red black trees are covered in detail in Chapter 13 of [CLRS90].

Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

+ after access, an element is moved to the root; splay(x)
  repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

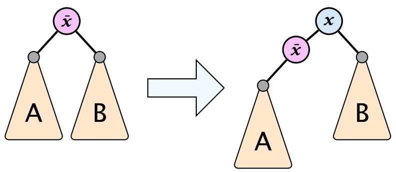
find(x)

- search for x according to a search tree
- let $\bar{x}$ be last element on search-path
- splay$(\bar{x})$

Splay Trees

insert(x)

- search for x; $\bar{x}$ is last visited element during search (successer or predecessor of x)
- splay$(\bar{x})$ moves $\bar{x}$ to the root
- insert x as new root

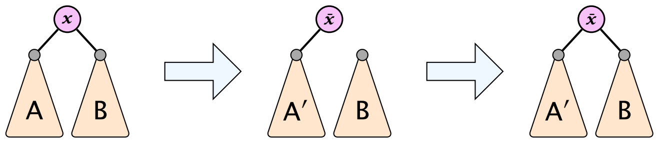


The illustration shows the case when $\bar{x}$ is the predecessor of x.

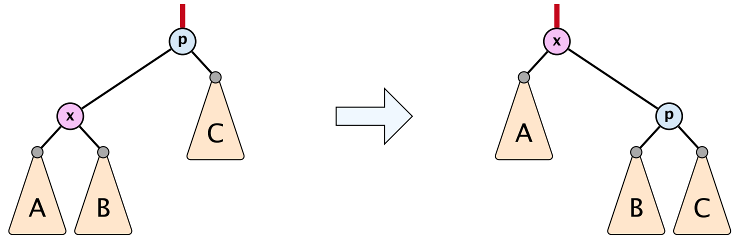
Splay Trees

delete(x)

- search for x; splay(x); remove x
- search largest element $\bar{x}$ in A
- splay($\bar{x}$) (on subtree A)
- connect root of B as right child of $\bar{x}$



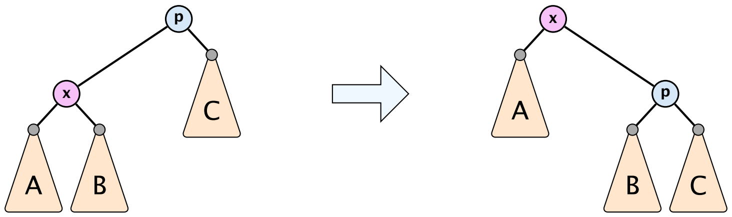
Move to Root



How to bring element to root?

- one (bad) option: moveToRoot(x)
- iteratively do rotation around parent of x until x is root
- if x is left child do right rotation otw. left rotation

Splay: Zig Case

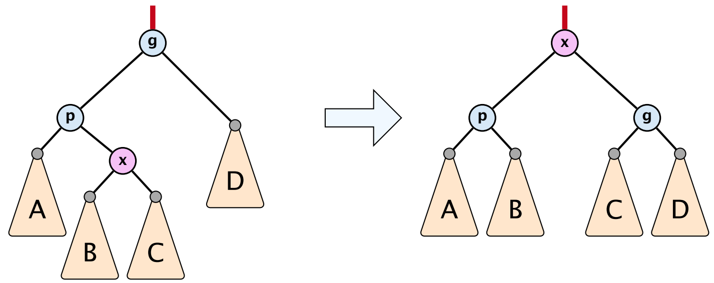


better option splay(x):

- zig case: if x is child of root do left rotation or right rotation around parent

Note that moveToRoot(x) does the same.

Splay: Zigzag Case

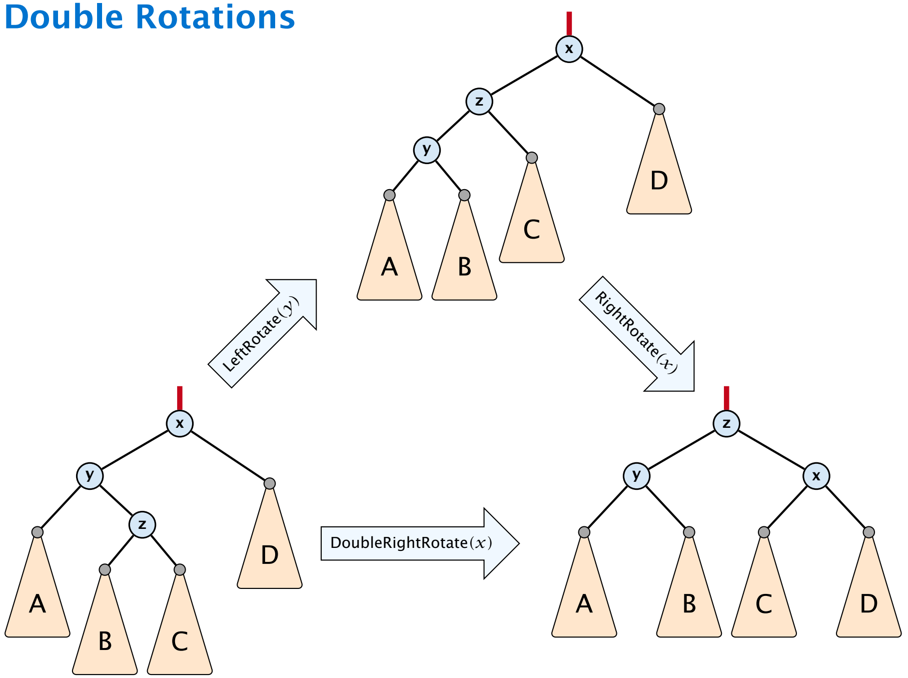


better option splay(x):

- zigzag case: if x is right child and parent of x is left child (or x left child parent of x right child)
- do double right rotation around grand-parent (resp. double left rotation)

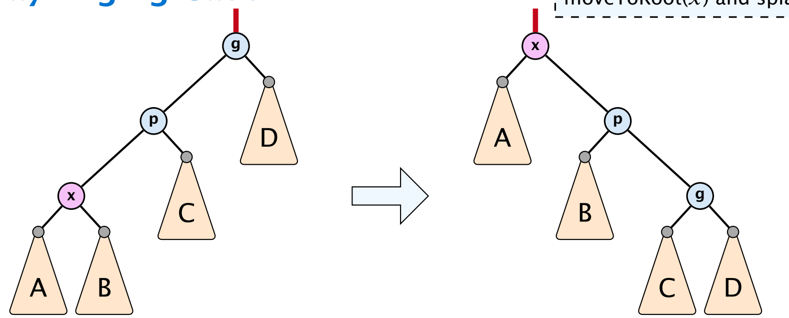
Note that moveToRoot(x) does the same.

Double Rotations



Splay: Zigzig Case

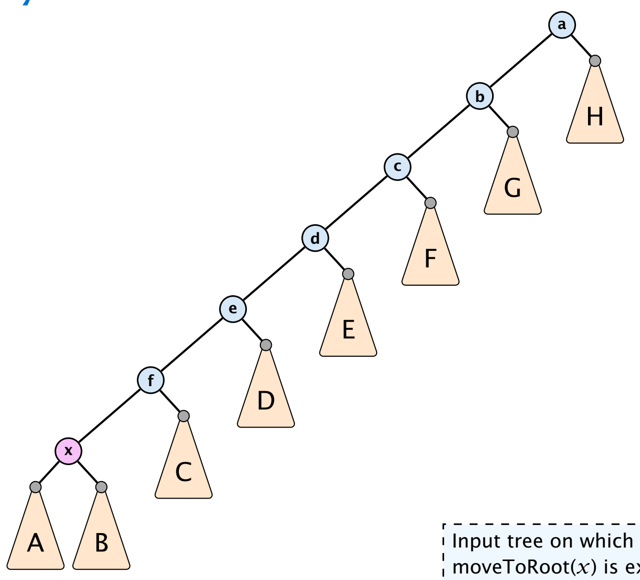
This case is different between moveToRoot(x) and splay(x).



better option splay(x):

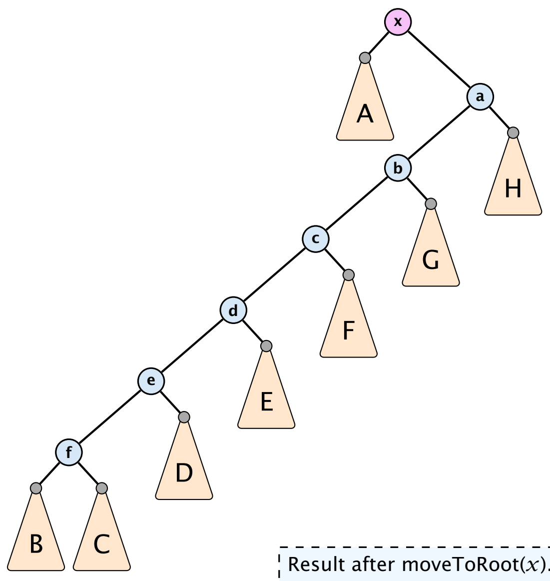
- zigzig case: if x is left child and parent of x is left child (or x right child, parent of x right child)
- do right roation around grand-parent followed by right rotation around parent (resp. left rotations)

Splay vs. Move to Root



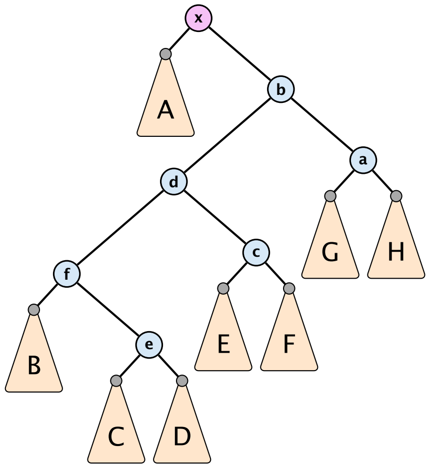
Input tree on which splay(x) and moveToRoot(x) is executed.

Splay vs. Move to Root



Result after moveToRoot(x).

Splay vs. Move to Root



Result after splay(x).

Static Optimality

Suppose we have a sequence of m find-operations. find(x) appears h_x times in this sequence.

The cost of a static search tree T is:

$$\text{cost}(T) = m + \sum_x h_x \, \text{depth}_T(x)$$

The total cost for processing the sequence on a splay-tree is $\mathcal{O}(\text{cost}(T_{\text{min}}))$, where T_{min} is an optimal static search tree.

$\text{depth}_T(x)$ is the number of edges on a path from the root of T to x.

Theorem given without proof.

Dynamic Optimality

Let S be a sequence with m find-operations.

Let A be a data-structure based on a search tree:

- the cost for accessing element x is $1 + \text{depth}(x)$;
- after accessing x the tree may be re-arranged through rotations;

Conjecture:
A splay tree that only contains elements from S has cost $\mathcal{O}(\text{cost}(A, S))$, for processing S.

Lemma 5
Splay Trees have an amortized running time of $\mathcal{O}(\log n)$ for all operations.

Amortized Analysis

Definition 6
A data structure with operations $\mathrm{op}_1(), \dots, \mathrm{op}_k()$ has amortized running times $t_1, \dots, t_k$ for these operations if the following holds.

Suppose you are given a sequence of operations (starting with an empty data-structure) that operate on at most n elements, and let k_i denote the number of occurences of $\mathrm{op}_i()$ within this sequence. Then the actual running time must be at most $\sum_i k_i \cdot t_i(n)$.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Example: Stack

Stack

- $\boldsymbol{S.\mathrm{push}()}$
- $\boldsymbol{S.\mathrm{pop}()}$
- $\boldsymbol{S.\mathrm{multipop}(k)}$: removes k items from the stack. If the stack currently contains less than k items it empties the stack.
- The user has to ensure that pop and multipop do not generate an underflow.

Actual cost:

- $\boldsymbol{S.\mathrm{push}()}$: cost 1.
- $\boldsymbol{S.\mathrm{pop}()}$: cost 1.
- $\boldsymbol{S.\mathrm{multipop}(k)}$: cost $\min\{\mathrm{size}, k\} = k$.

Example: Stack

Use potential function $\Phi(S) =$ number of elements on the stack.

Amortized cost:

- $\boldsymbol{S.\mathrm{push}()}$: cost

$$\hat{C}_{\mathrm{push}} = C_{\mathrm{push}} + \Delta\Phi = 1 + 1 \leq 2 \ .$$

- $\boldsymbol{S.\mathrm{pop}()}$: cost

$$\hat{C}_{\mathrm{pop}} = C_{\mathrm{pop}} + \Delta\Phi = 1 - 1 \leq 0 \ .$$

- $\boldsymbol{S.\mathrm{multipop}(k)}$: cost

$$\hat{C}_{\mathrm{mp}} = C_{\mathrm{mp}} + \Delta\Phi = \min\{\mathrm{size}, k\} - \min\{\mathrm{size}, k\} \leq 0 \ .$$

Note that the analysis becomes wrong if pop() or multipop() are called on an empty stack.

Example: Binary Counter

Incrementing a binary counter:
Consider a computational model where each bit-operation costs one time-unit.

Incrementing an n-bit binary counter may require to examine n-bits, and maybe change them.

Actual cost:

- Changing bit from 0 to 1: cost 1.
- Changing bit from 1 to 0: cost 1.
- Increment: cost is $k+1$, where k is the number of consecutive ones in the least significant bit-positions (e.g, 001101 has $k=1$).

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1:

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- Changing bit from 1 to 0:

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \le 0 \ .$$

- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \le 2$.

Splay Trees

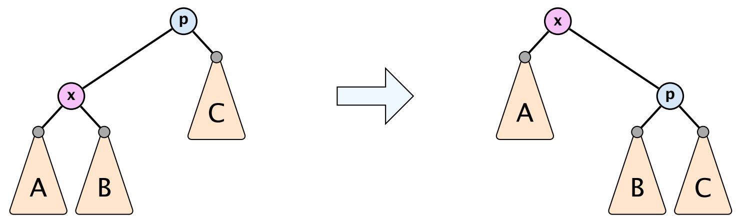
potential function for splay trees:

- size $\mathrm{s}(x) = |T_x|$
- rank $\mathrm{r}(x) = \log_2(s(x))$
- $\Phi(T) = \sum_{v\in T} r(v)$

amortized cost = real cost + potential change

The cost is essentially the cost of the splay-operation, which is 1 plus the number of rotations.

Splay: Zig Case

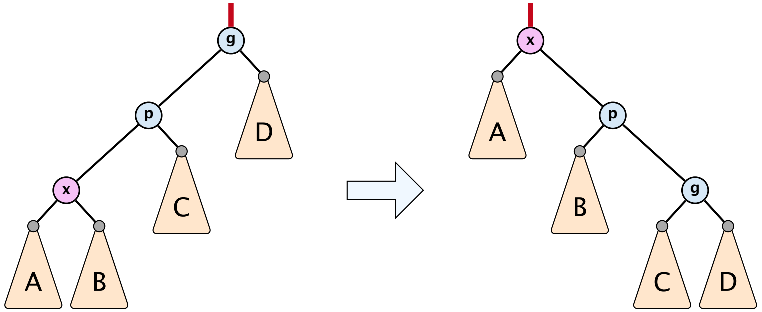


$$\begin{aligned}\Delta\Phi &= r'(x) + r'(p) - r(x) - r(p)\\ &= r'(p) - r(x)\\ &\le r'(x) - r(x)\end{aligned}$$

$$\mathrm{cost}_{\mathrm{zig}} \le 1 + 3(r'(x) - r(x))$$

Splay: Zigzig Case

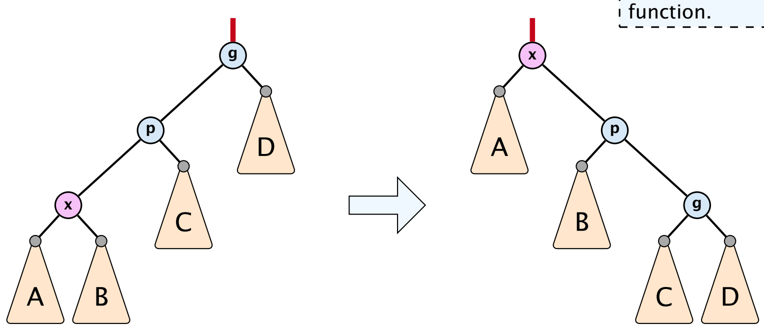
Last inequality follows from next slide.



$$\begin{aligned}\Delta\Phi &= r'(x)+r'(p)+r'(g)-r(x)-r(p)-r(g)\\ &= r'(p)+r'(g)-r(x)-r(p)\\ &\le r'(x)+r'(g)-r(x)-r(x)\\ &= r'(x)+r'(g)+r(x)-3r'(x)+3r'(x)-r(x)-2r(x)\\ &= -2r'(x)+r'(g)+r(x)+3(r'(x)-r(x))\\ &\le -2+3(r'(x)-r(x)) \quad \Rightarrow \text{cost}_{\text{zigzig}} \le 3(r'(x)-r(x))\end{aligned}$$

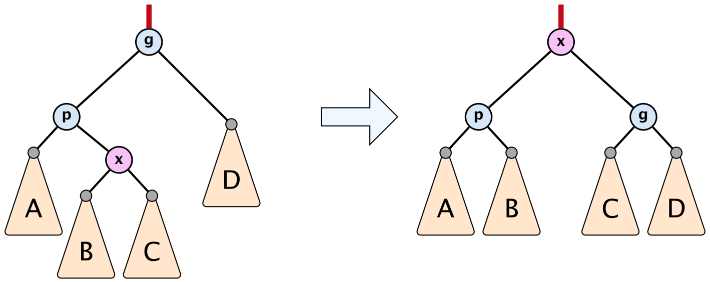
Splay: Zigzig Case

The last inequality holds because log is a concave function.



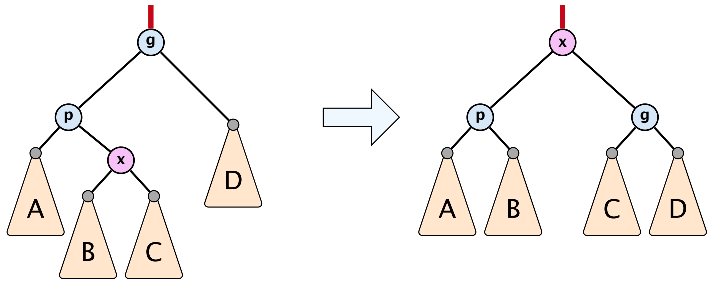
$$\begin{aligned}&\frac{1}{2}\Big(r(x)+r'(g)-2r'(x)\Big)\\ &\quad= \frac{1}{2}\Big(\log(s(x))+\log(s'(g))-2\log(s'(x))\Big)\\ &\quad= \frac{1}{2}\log\Big(\frac{s(x)}{s'(x)}\Big)+\frac{1}{2}\log\Big(\frac{s'(g)}{s'(x)}\Big)\\ &\quad\le \log\Big(\frac{1}{2}\frac{s(x)}{s'(x)}+\frac{1}{2}\frac{s'(g)}{s'(x)}\Big) \le \log\Big(\frac{1}{2}\Big) = -1\end{aligned}$$

Splay: Zigzag Case



$$\begin{aligned}\Delta\Phi &= r'(x)+r'(p)+r'(g)-r(x)-r(p)-r(g)\\ &= r'(p)+r'(g)-r(x)-r(p)\\ &\le r'(p)+r'(g)-r(x)-r(x)\\ &= r'(p)+r'(g)-2r'(x)+2r'(x)-2r(x)\\ &\le -2+2(r'(x)-r(x)) \quad \Rightarrow \text{cost}_{\text{zigzag}} \le 3(r'(x)-r(x))\end{aligned}$$

Splay: Zigzag Case



$$\begin{aligned}&\frac{1}{2}\Big(r'(p)+r'(g)-2r'(x)\Big)\\ &\quad= \frac{1}{2}\Big(\log(s'(p))+\log(s'(g))-2\log(s'(x))\Big)\\ &\quad\le \log\Big(\frac{1}{2}\frac{s'(p)}{s'(x)}+\frac{1}{2}\frac{s'(g)}{s'(x)}\Big) \le \log\Big(\frac{1}{2}\Big) = -1\end{aligned}$$

Harald Räcke
7.3 Splay Trees
6. Feb. 2022
75/218

Amortized cost of the whole splay operation:

$$\leq 1 + 1 + \sum_{\text{steps } t} 3(r_t(x) - r_{t-1}(x))$$
$$= 2 + 3(r(\text{root}) - r_0(x))$$
$$\leq \mathcal{O}(\log n)$$

The first one is added due to the fact that so far for each step of a splay-operation we have only counted the number of rotations, but the cost is 1+#rotations.

The second one comes from the zig-operation. Note that we have at most one zig-operation during a splay.

Splay Trees

Bibliography

??????????????????????????????????????

7.4 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 ℓ-th element; return "error" if the data-structure contains less than ℓ elements.

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

7.4 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.

7.4 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...

7.4 Augmenting Data Structures

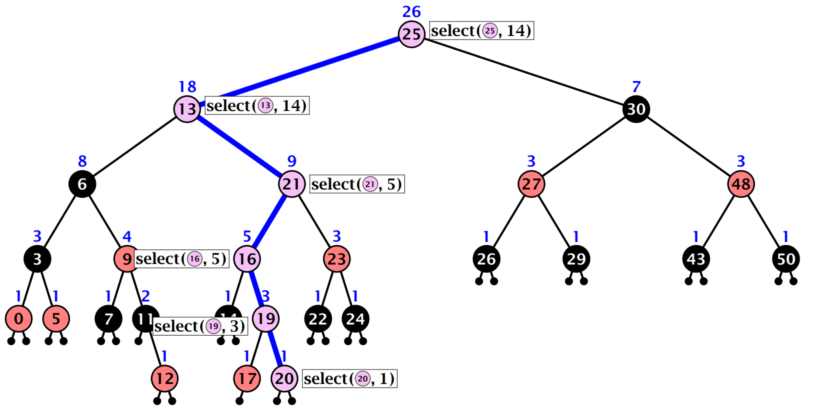
Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

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

Algorithm 1 Select(x, i)

if x = null **then return** error
if left[x] $\neq$ null **then** $r \leftarrow$ left[x].size $+1$ **else** $r \leftarrow 1$
if $i = r$ **then return** x
if $i < r$ **then**
    return Select(left[x], i)
else
    return Select(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

7.4 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?

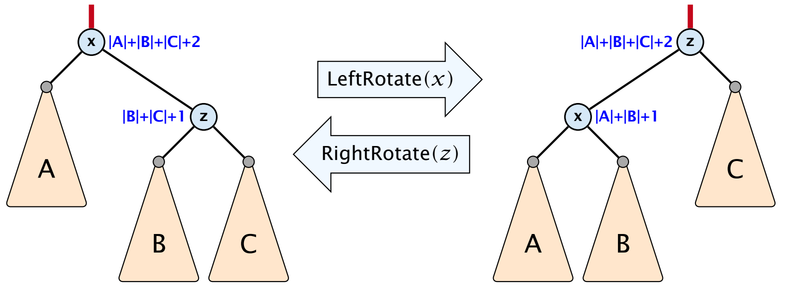
Search(k): Nothing to do.

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.

Augmenting Data Structures

Bibliography

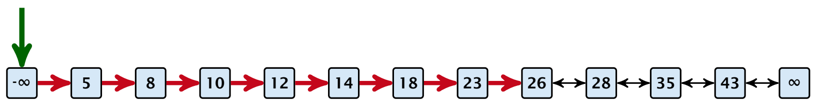
[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein: *Introduction to Algorithms (3rd ed.)*, MIT Press and McGraw-Hill, 2009

See Chapter 14 of [CLRS90].

7.5 Skip Lists

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

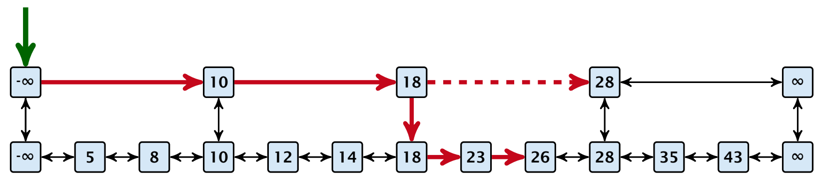
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



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 $r = n^{\frac{1}{k+1}}$. Then

$$\begin{aligned} r^{-k}n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}} \\ &= n^{1-\frac{k}{k+1}} + kn^{\frac{1}{k+1}} \\ &= (k+1)n^{\frac{1}{k+1}} \ . \end{aligned}$$

Choosing $k = \Theta(\log n)$ 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 $L_0,\dots,L_{t-1}$.

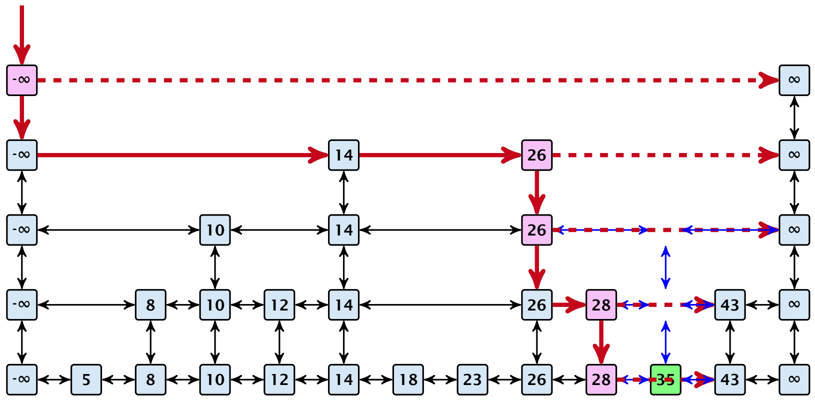
Delete:

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

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

7.5 Skip Lists

Insert (35):



High Probability

Definition 7 (High Probability)

We say a **randomized** algorithm has running time $\mathcal{O}(\log n)$ with high probability if for any constant α the running time is at most $\mathcal{O}(\log n)$ with probability at least $1 - \frac{1}{n^\alpha}$.

Here the $\mathcal{O}$-notation hides a constant that may depend on α.

High Probability

Suppose there are polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ 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 probability that all E_i hold is at least

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

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

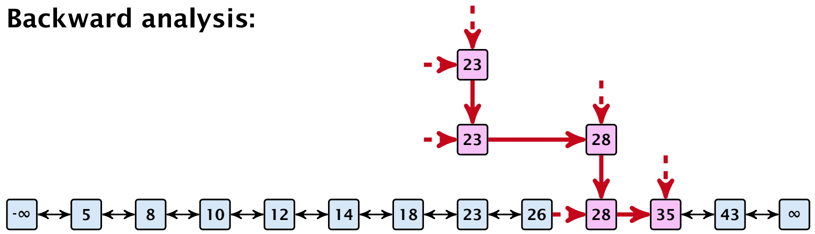
7.5 Skip Lists

Lemma 8

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

7.5 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

Estimation for Binomial Coefficients

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k!\cdot(n-k)!} = \frac{n\cdot\ldots\cdot(n-k+1)}{k\cdot\ldots\cdot 1} \ge \left(\frac{n}{k}\right)^k$$

$$\binom{n}{k} = \frac{n\cdot\ldots\cdot(n-k+1)}{k!} \le \frac{n^k}{k!} = \frac{n^k\cdot k^k}{k^k\cdot k!}$$

$$= \left(\frac{n}{k}\right)^k\cdot\frac{k^k}{k!} \le \left(\frac{n}{k}\right)^k\cdot\sum_{i\ge 0}\frac{k^i}{i!} = \left(\frac{en}{k}\right)^k$$

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

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

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

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

$$\le \left(\frac{2ez}{k}\right)^k 2^{-\beta k}\cdot n^{-\gamma\alpha} \le \left(\frac{2ez}{2^\beta k}\right)^k\cdot n^{-\alpha}$$

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

now choosing $\beta = 6\alpha$ gives

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

for $\alpha \ge 1$.

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 event 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.

Skip Lists

Bibliography

[GT98] Michael T. Goodrich, Roberto Tamassia
Data Structures and Algorithms in JAVA,
John Wiley, 1998

Skip lists are covered in Chapter 7.5 of [GT98].

7.6 van Emde Boas Trees

Dynamic Set Data Structure S:

- $S.\text{insert}(x)$
- $S.\text{delete}(x)$
- $S.\text{search}(x)$
- $S.\text{min}()$
- $S.\text{max}()$
- $S.\text{succ}(x)$
- $S.\text{pred}(x)$

7.6 van Emde Boas Trees

For this chapter we ignore the problem of storing satellite data:

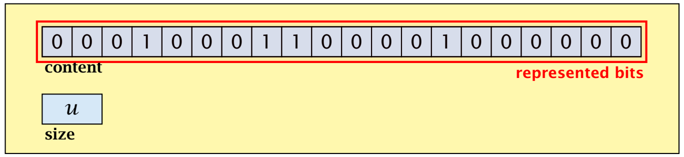
- **$S.\text{insert}(x)$:** Inserts x into S.
- **$S.\text{delete}(x)$:** Deletes x from S. Usually assumes that $x \in S$.
- **$S.\text{member}(x)$:** Returns 1 if $x \in S$ and 0 otw.
- **$S.\text{min}()$:** Returns the value of the minimum element in S.
- **$S.\text{max}()$:** Returns the value of the maximum element in S.
- **$S.\text{succ}(x)$:** Returns successor of x in S. Returns null if x is maximum or larger than any element in S. Note that x needs not to be in S.
- **$S.\text{pred}(x)$:** Returns the predecessor of x in S. Returns null if x is minimum or smaller than any element in S. Note that x needs not to be in S.

7.6 van Emde Boas Trees

Can we improve the existing algorithms when the keys are from a restricted set?

In the following we assume that the keys are from $\{0, 1, \ldots, u-1\}$, where u denotes the size of the universe.

Implementation 1: Array



one array of u bits

Use an array that encodes the indicator function of the dynamic set.

Implementation 1: Array

Algorithm 1 array.insert(x)

content[x] $\leftarrow 1$;

Algorithm 2 array.delete(x)

content[x] $\leftarrow 0$;

Algorithm 3 array.member(x)

return content[x];

- Note that we assume that x is valid, i.e., it falls within the array boundaries.
- Obviously(?) the running time is constant.

Implementation 1: Array

Algorithm 4 array.max()

for (i = size -1; $i \geq 0$; i--) **do**
  if content[i] $= 1$ **then return** i;
return null;

Algorithm 5 array.min()

for ($i = 0$; $i <$ size; i++) **do**
  if content[i] $= 1$ **then return** i;
return null;

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 1: Array

Algorithm 6 array.succ(x)

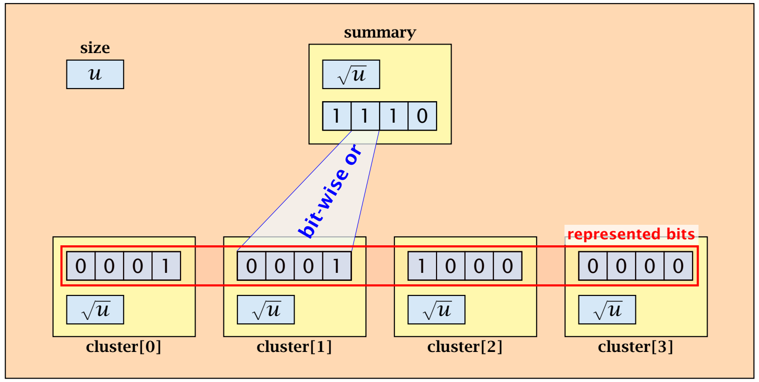
for ($i = x + 1$; $i <$ size; i++) **do**
    if content[i] $= 1$ **then return** i;
return null;

Algorithm 7 array.pred(x)

for ($i = x - 1$; $i \geq 0$; i--) **do**
    if content[i] $= 1$ **then return** i;
return null;

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 2: Summary Array



- $\sqrt{u}$ cluster-arrays of $\sqrt{u}$ bits.
- One summary-array of $\sqrt{u}$ bits. The i-th bit in the summary array stores the bit-wise or of the bits in the i-th cluster.

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

Algorithm 8 member(x)

return cluster[high(x)].member(low(x));

Algorithm 9 insert(x)

cluster[high(x)].insert(low(x));
summary.insert(high(x));

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

Algorithm 10 delete(x)

cluster[high(x)].delete(low(x));
if cluster[high(x)].min() = null **then**
    summary.delete(high(x));

- The running time is dominated by the cost of a minimum computation on an array of size $\sqrt{u}$. Hence, $\mathcal{O}(\sqrt{u})$.

Implementation 2: Summary Array

Algorithm 11 max()

maxcluster ← summary.max();
if *maxcluster* = null **return** null;
offs ← cluster[*maxcluster*].max()
return *maxcluster* ∘ *offs*;

Algorithm 12 min()

mincluster ← summary.min();
if *mincluster* = null **return** null;
offs ← cluster[*mincluster*].min();
return *mincluster* ∘ *offs*;

The operator ∘ stands for the concatenation of two bitstrings. This means if $x = 0111_2$ and $y = 0001_2$ then $x \circ y = 01110001_2$.

- Running time is roughly $2\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 13 succ(x)

m ← cluster[high(x)].succ(low(x))
if $m \neq$ null **then return** high(x) ∘ m;
succcluster ← summary.succ(high(x));
if *succcluster* ≠ null **then**
    offs ← cluster[*succcluster*].min();
    return *succcluster* ∘ *offs*;
return null;

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 14 pred(x)

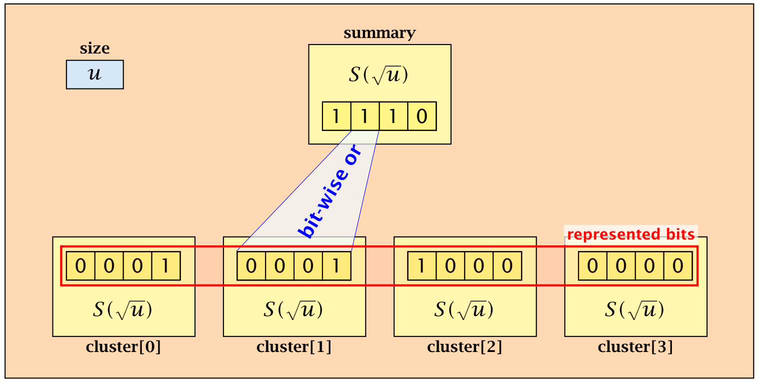
m ← cluster[high(x)].pred(low(x))
if $m \neq$ null **then return** high(x) ∘ m;
predcluster ← summary.pred(high(x));
if *predcluster* ≠ null **then**
    offs ← cluster[*predcluster*].max();
    return *predcluster* ∘ *offs*;
return null;

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 3: Recursion

Instead of using sub-arrays, we build a recursive data-structure.

$S(u)$ is a dynamic set data-structure representing u bits:



Implementation 3: Recursion

We assume that $u = 2^{2^k}$ for some k.

The data-structure $S(2)$ is defined as an array of 2-bits (end of the recursion).

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\mathrm{cluster}[1].\mathrm{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\mathrm{array.min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

Algorithm 15 member(x)

1: **return** cluster[high(x)].member(low(x));

- $T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 16 insert(x)

cluster[high(x)].insert(low(x));
summary.insert(high(x));

- $T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 17 delete(x)

cluster[high(x)].delete(low(x));
if cluster[high(x)].min() = null **then**
    summary.delete(high(x));

- $T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 18 min()

mincluster ← summary.min();
if *mincluster* = null **return** null;
offs ← cluster[*mincluster*].min();
return *mincluster* ∘ *offs*;

- $T_{\min}(u) = 2T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 19 succ(x)

m ← cluster[high(x)].succ(low(x))
if $m \neq$ null **then return** high$(x) \circ m$;
succcluster ← summary.succ(high(x));
if *succcluster* ≠ null **then**
    offs ← cluster[*succcluster*].min();
    return *succcluster* ∘ *offs*;
return null;

- $T_{\text{succ}}(u) = 2T_{\text{succ}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

$T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$:

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$.Then

$$X(\ell) = T_{\text{mem}}(2^\ell) = T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$$
$$= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 = X(\tfrac{\ell}{2}) + 1 \ .$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\log \ell)$, and hence $T_{\text{mem}}(u) = \mathcal{O}(\log\log u)$.

Implementation 3: Recursion

$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1$.

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1$$
$$= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ .$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\ell)$, and hence $T_{\text{ins}}(u) = \mathcal{O}(\log u)$.

The same holds for $T_{\max}(u)$ and $T_{\min}(u)$.

Implementation 3: Recursion

$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u)$.

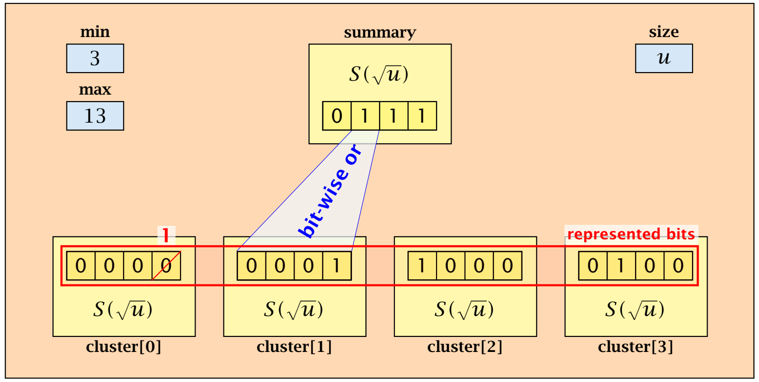
Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$X(\ell) = T_{\text{del}}(2^\ell) = T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + c\log u$$
$$= 2T_{\text{del}}(2^{\frac{\ell}{2}}) + c\ell = 2X(\tfrac{\ell}{2}) + c\ell \ .$$

Using Master theorem gives $X(\ell) = \Theta(\ell\log\ell)$, and hence $T_{\text{del}}(u) = \mathcal{O}(\log u \log\log u)$.

The same holds for $T_{\text{pred}}(u)$ and $T_{\text{succ}}(u)$.

Implementation 4: van Emde Boas Trees



- The bit referenced by min is not set within sub-datastructures.
- The bit referenced by max is set within sub-datastructures (if $\max \neq \min$).

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for min and max are constant time.
- $\text{min} = \text{null}$ means that the data-structure is empty.
- $\text{min} = \text{max} \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\text{min} = \text{max} = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\text{min} = \text{max} = \text{null}$.

Implementation 4: van Emde Boas Trees

Algorithm 20 max()

```
return max;
```

Algorithm 21 min()

```
return min;
```

- Constant time.

Implementation 4: van Emde Boas Trees

Algorithm 22 member(x)

```
if x = min then return 1; // TRUE
return cluster[high(x)].member(low(x));
```

- $T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1 \Longrightarrow T(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 23 succ(x)

```
if min ≠ null ∧ x < min then return min;
maxincluster ← cluster[high(x)].max();
if maxincluster ≠ null ∧ low(x) < maxincluster then
    offs ← cluster[high(x)].succ(low(x));
    return high(x) ∘ offs;
else
    succcluster ← summary.succ(high(x));
    if succcluster = null then return null;
    offs ← cluster[succcluster].min();
    return succcluster ∘ offs;
```

- $T_{\text{succ}}(u) = T_{\text{succ}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{succ}}(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 35 insert(x)

1: **if** min = null **then**
2: min = x; max = x;
3: **else**
4: **if** x < min **then** exchange x and min;
5: **if** x > max **then** max = x;
6: **if** cluster[high(x)].min = null; **then**
7: summary.insert(high(x));
8: cluster[high(x)].insert(low(x));
9: **else**
10: cluster[high(x)].insert(low(x));

- $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{ins}}(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Note that the recusive call in Line 8 takes constant time as the if-condition in Line 6 ensures that we are inserting in an empty sub-tree.

The only non-constant recursive calls are the call in Line 7 and in Line 10. These are mutually exclusive, i.e., only one of these calls will actually occur.

From this we get that $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1$.

Implementation 4: van Emde Boas Trees

- **Assumes that x is contained in the structure.**

Algorithm 36 delete(x)

1: **if** min = max **then**
2: min = max = null;
3: **else**
4: **if** x = min **then** — find new minimum
5: *firstcluster* ← summary.min();
6: *offs* ← cluster[*firstcluster*].min();
7: x ← *firstcluster* ∘ *offs*;
8: min ← x;
9: cluster[high(x)].delete(low(x)); — delete

continued...

Implementation 4: van Emde Boas Trees

Algorithm 36 delete(x)

...continued — fix maximum

10: **if** cluster[high(x)].min() = null **then**
11: summary.delete(high(x));
12: **if** x = max **then**
13: *summax* ← summary.max();
14: **if** *summax* = null **then** max ← min;
15: **else**
16: *offs* ← cluster[*summax*].max();
17: max ← *summax* ∘ *offs*
18: **else**
19: **if** x = max **then**
20: *offs* ← cluster[high(x)].max();
21: max ← high(x) ∘ *offs*;

Implementation 4: van Emde Boas Trees

Note that only one of the possible recusive calls in Line 9 and Line 11 in the deletion-algorithm may take non-constant time.

To see this observe that the call in Line 11 only occurs if the cluster where x was deleted is now empty. But this means that the call in Line 9 deleted the last element in $\text{cluster}[\text{high}(x)]$. Such a call only takes constant time.

Hence, we get a recurrence of the form

$$T_{\text{del}}(u) = T_{\text{del}}(\sqrt{u}) + c \ .$$

This gives $T_{\text{del}}(u) = \mathcal{O}(\log \log u)$.

7.6 van Emde Boas Trees

Space requirements:

- The space requirement fulfills the recurrence

$$S(u) = (\sqrt{u} + 1)S(\sqrt{u}) + \mathcal{O}(\sqrt{u}) \ .$$

- Note that we cannot solve this recurrence by the Master theorem as the branching factor is not constant.
- One can show by induction that the space requirement is $S(u) = \mathcal{O}(u)$. Exercise.

- Let the "real" recurrence relation be

$$S(k^2) = (k+1)S(k) + c_1 \cdot k; \ S(4) = c_2$$

- Replacing $S(k)$ by $R(k) := S(k)/c_2$ gives the recurrence

$$R(k^2) = (k+1)R(k) + ck; \ R(4) = 1$$

where $c = c_1/c_2 < 1$.
- Now, we show $R(k) \leq k - 2$ for squares $k \geq 4$.
  - Obviously, this holds for $k = 4$.
  - For $k = \ell^2 > 4$ with ℓ integral we have

$$\begin{aligned} R(k) &= (1+\ell)R(\ell) + c\ell \\ &\leq (1+\ell)(\ell-2) + \ell \leq k - 2 \end{aligned}$$

  - This shows that $R(k)$ and, hence, $S(k)$ grows linearly.

van Emde Boas Trees

Bibliography

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein: *Introduction to Algorithms (3rd ed.)*, MIT Press and McGraw-Hill, 2009

See Chapter 20 of [CLRS90].

7.7 Hashing

Dictionary:

- $S.\text{insert}(x)$: Insert an element x.
- $S.\text{delete}(x)$: Delete the element pointed to by x.
- $S.\text{search}(k)$: Return a pointer to an element e with $\text{key}[e] = k$ 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:

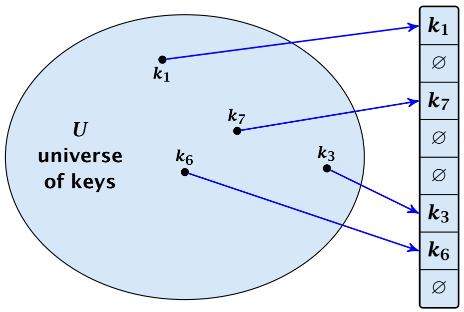
- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \dots, n-1]$ hash-table.
- Hash function $h : U \to [0, \dots, n-1]$.

The hash-function h should fulfill:

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

Direct Addressing

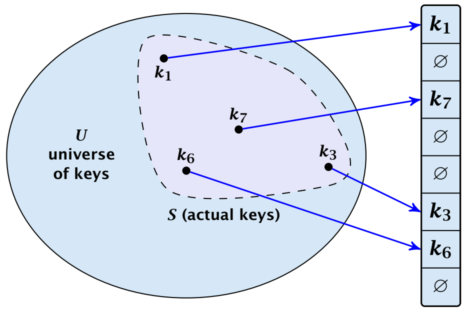
Ideally the hash function maps all keys to different memory locations.



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.

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.



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

Collisions

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.

Collisions

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

Lemma 9
The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

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

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

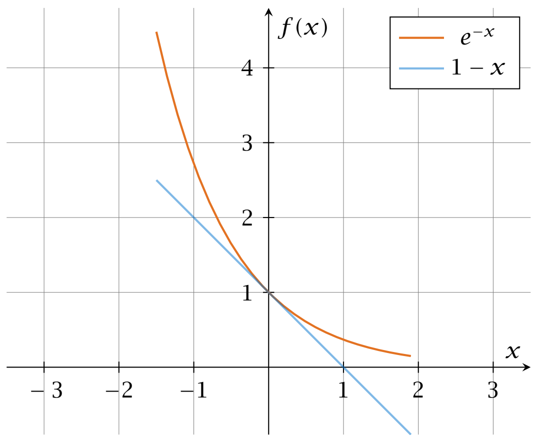
Collisions

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

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

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. ☐

Collisions



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

Resolving Collisions

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

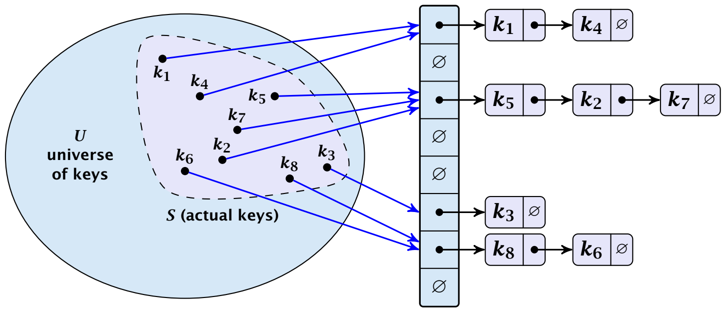
- **open addressing**, aka. closed hashing
- **hashing with chaining**, aka. closed addressing, open hashing.

There are applications e.g. computer chess where you do not resolve collisions at all.

Hashing with Chaining

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

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



Hashing with Chaining

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

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

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 indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

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

Hashing with Chaining

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

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

Hashing with Chaining

Disadvantages:

- pointers increase memory requirements
- pointers may lead to bad cache efficiency

Advantages:

- no à priori limit on the number of elements
- deletion can be implemented efficiently
- by using balanced trees instead of linked list one can also obtain worst-case guarantees.

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 $h(k,0)$,…,$h(k,n-1)$ must form a permutation of $0,\dots,n-1$.

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 $h(k,j)$:

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

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 (teilerfremd); 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 10
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 11
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 cache-miss.

Lemma 12
Let D be the method of double hashing for resolving collisions:

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

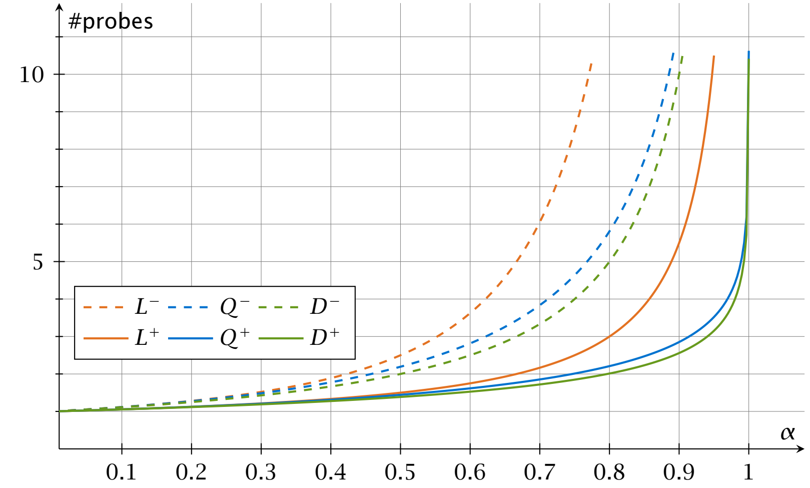
$$D^- \approx \frac{1}{1-\alpha}$$

Open Addressing

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

Open Addressing



Analysis of Idealized Open Address Hashing

We analyze the time for a search in a very idealized Open Addressing scheme.

- The probe sequence $h(k,0), h(k,1), h(k,2), \dots$ is equally likely to be any permutation of $\langle 0, 1, \dots, n-1\rangle$.

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.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}] \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \dots \cdot \Pr[A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

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

Analysis of Idealized Open Address Hashing

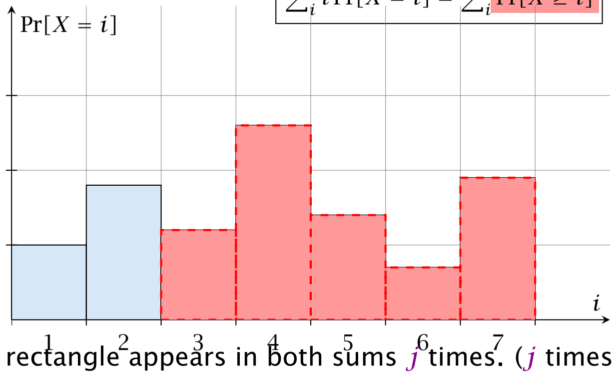
$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} .$$

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

Analysis of Idealized Open Address Hashing

$i = 3$

$\boxed{\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]}$

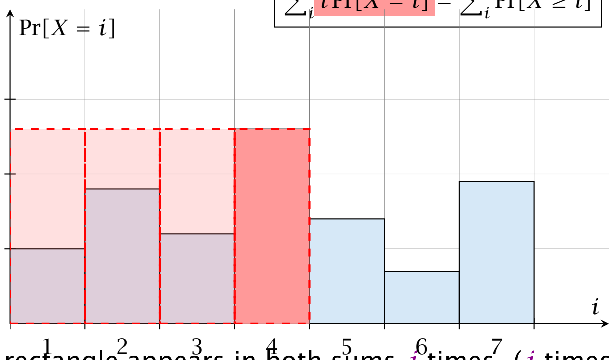


The j-th rectangle appears in both sums j times. (j times in the first due to multiplication with j; and j times in the second for summands $i = 1, 2, \ldots, j$)

Analysis of Idealized Open Address Hashing

$i = 4$

$\boxed{\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]}$



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

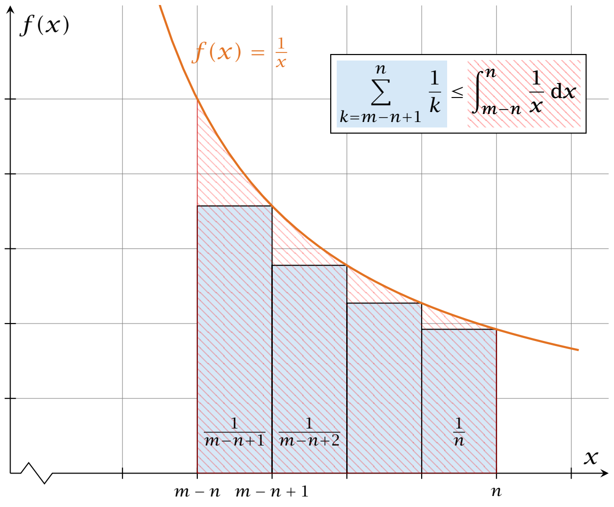
The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

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

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

$$\leq \frac{1}{\alpha} \int_{n-m}^{n} \frac{1}{x} dx = \frac{1}{\alpha} \ln \frac{n}{n-m} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} .$$

Analysis of Idealized Open Address Hashing



Deletions in Hashtables

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.

Deletions in Hashtables

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions for Linear Probing

- For Linear Probing one can delete elements without using deletion-markers.
- Upon a deletion elements that are further down in the probe-sequence may be moved to guarantee that they are still found during a search.

Deletions for Linear Probing

Algorithm 37 delete(p)

```
T[p] ← null
p ← succ(p)
while T[p] ≠ null do
    y ← T[p]
    T[p] ← null
    p ← succ(p)
    insert(y)
```

p is the index into the table-cell that contains the object to be deleted.

Pointers into the hash-table become invalid.

Universal 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 $f : U \to [0, \ldots, n-1]$ 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 $\mathcal{H}$.

Universal Hashing

Definition 13

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ 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 the probability of a collision between two arbitrary elements is at most $\frac{1}{n}$.

Universal Hashing

Definition 14

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

- For any key $u \in U$, and $t \in \{0, \ldots, n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, 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:

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

This requirement clearly implies a universal hash-function.

Universal Hashing

Definition 15
A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ is called k-independent if for any choice of $\ell \leq k$ distinct keys $u_1, \ldots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1, \ldots, t_\ell$:

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

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

Universal Hashing

Definition 16
A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ is called (μ, k)-independent if for any choice of $\ell \leq k$ distinct keys $u_1, \ldots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1, \ldots, t_\ell$:

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

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

Universal Hashing

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

Define

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

Lemma 17
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 $\{0, \ldots, n-1\}$.

Universal 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 (Körper) and, hence, has no zero divisors (nullteilerfrei).

Universal Hashing

- The hash-function does not generate collisions before the $(\bmod\ n)$-operation. Furthermore, every choice (a, b) is mapped to a different pair (t_x, t_y) with $t_x := ax + b$ and $t_y := ay + b$.

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

$$
\begin{aligned}
t_x &\equiv ax + b && \pmod p\\
t_y &\equiv ay + b && \pmod p
\end{aligned}
$$

$$
\begin{aligned}
t_x - t_y &\equiv a(x - y) && \pmod p\\
t_y &\equiv ay + b && \pmod p
\end{aligned}
$$

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

Universal 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.

Universal 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 $\frac{1}{n}$.

Universal Hashing

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

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

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 $\lceil \frac{p}{n} \rceil$.

Universal Hashing

Definition 18
Let $d \in \mathbb{N}$; $q \geq (d+1)n$ be a prime; and let $\bar{a} \in \{0, \dots, q-1\}^{d+1}$.
Define for $x \in \{0, \dots, q-1\}$

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

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

Note that in the previous case we had $d = 1$ and chose $a_d \neq 0$.

Universal Hashing

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.

Universal Hashing

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

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

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

In order to obtain the cardinality of A^ℓ we choose our polynomial by fixing $d+1$ points.

We first fix the values for inputs $x_1, \dots, x_\ell$.
We have

$$|B_1| \cdot \ldots \cdot |B_\ell|$$

possibilities to do this (so that $h_{\bar{a}}(x_i) = t_i$).

- A^ℓ denotes the set of hash-functions such that every x_i hits its pre-defined position t_i.
- B_i is the set of positions that $f_{\bar{a}}$ can hit so that $h_{\bar{a}}$ still hits t_i.

Universal Hashing

Now, we choose $d - \ell + 1$ other inputs and choose their value arbitrarily. We have $q^{d-\ell+1}$ possibilities to do this.

Therefore we 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 $h_{\bar{a}} \in A_\ell$.

Universal Hashing

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

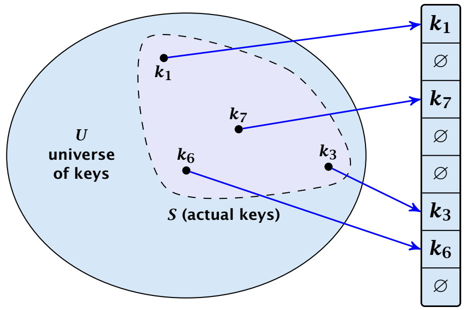
$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \le \frac{(\frac{q+n}{n})^\ell}{q^\ell} \le \left(\frac{q+n}{q}\right)^\ell \cdot \frac{1}{n^\ell}$$
$$\le \left(1+\frac{1}{\ell}\right)^\ell \cdot \frac{1}{n^\ell} \le \frac{e}{n^\ell} \ .$$

This shows that the $\mathcal{H}$ is $(e, d+1)$-universal.

The last step followed from $q \ge (d+1)n$, and $\ell \le d+1$.

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

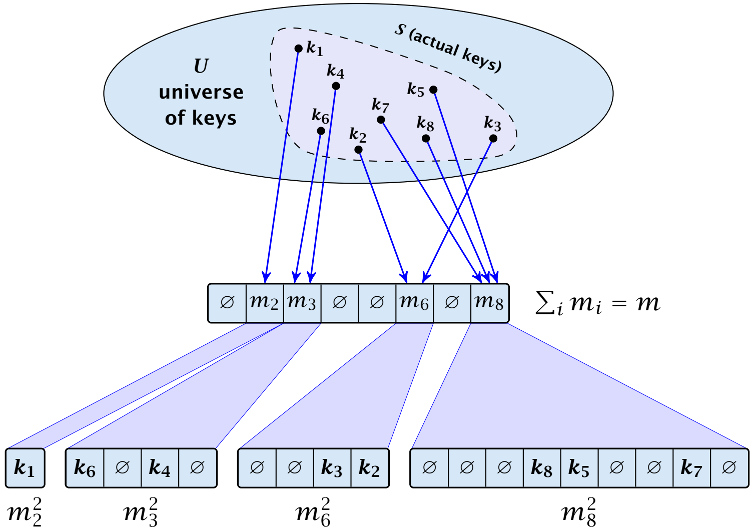
We can find such a hash-function by a few trials.

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



Perfect Hashing

The total memory that is required by all hash-tables is $\mathcal{O}(\sum_j m_j^2)$. Note that m_j is a random variable.

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

The first expectation is simply the expected number of collisions, for the first level. Since we use universal hashing we have

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

Perfect Hashing

We need only $\mathcal{O}(m)$ time to construct a hash-function h with $\sum_j m_j^2 = \mathcal{O}(4m)$, because with probability at least $1/2$ a random function from a universal family will have this property.

Then we construct a hash-table h_j for every bucket. This takes expected time $\mathcal{O}(m_j)$ for every bucket. A random function h_j is collision-free with probability at least $1/2$. We need $\mathcal{O}(m_j)$ to test this.

We only need that the hash-functions are chosen from a universal family!!!

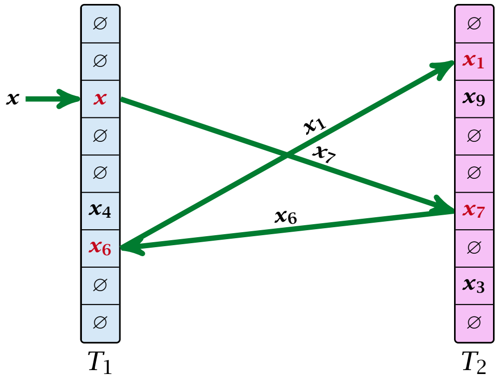
Cuckoo Hashing

Goal:
Try to generate a hash-table with 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 38 Cuckoo-Insert(x)

```
if T1[h1(x)] = x ∨ T2[h2(x)] = x then return
steps ← 1
while steps ≤ maxsteps do
    exchange x and T1[h1(x)]
    if x = null then return
    exchange x and T2[h2(x)]
    if x = null then return
    steps ← steps +1
rehash() // change hash-functions; rehash everything
Cuckoo-Insert(x)
```


Cuckoo Hashing

- We call one iteration through the while-loop a step of the algorithm.
- We call a sequence of iterations through the while-loop without the termination condition becoming true a phase of the algorithm.
- We say a phase is successful if it is not terminated by the maxstep-condition, but the while loop is left because $x = \text{null}$.

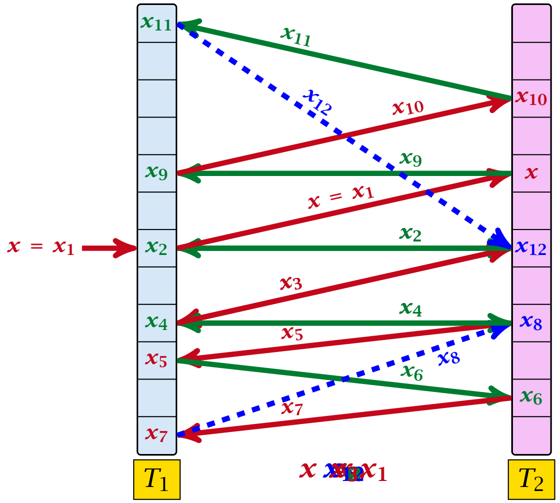
Cuckoo Hashing

What is the expected time for an insert-operation?

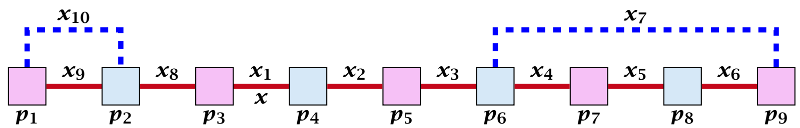
We first analyze the probability that we end-up in an infinite loop (that is then terminated after maxsteps steps).

Formally what is the probability to enter an infinite loop that touches s different keys?

Cuckoo Hashing: Insert



Cuckoo Hashing



A cycle-structure of size s is defined by

- $s - 1$ different cells (alternating btw. cells from T_1 and T_2).
- s distinct keys $x = x_1, x_2, \ldots, x_s$, linking the cells.
- The leftmost cell is "linked forward" to some cell on the right.
- The rightmost cell is "linked backward" to a cell on the left.
- One link represents key x; this is where the counting starts.

Cuckoo Hashing

A cycle-structure is active if for every key x_ℓ (linking a cell p_i from T_1 and a cell p_j from T_2) we have

$$h_1(x_\ell) = p_i \qquad \text{and} \qquad h_2(x_\ell) = p_j$$

Observation:
If during a phase the insert-procedure runs into a cycle there must exist an active cycle structure of size $s \geq 3$.

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size s correctly map into their T_1-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_1 is a (μ, s)-independent hash-function.

What is the probability that all keys in the cycle-structure of size s correctly map into their T_2-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_2 is a (μ, s)-independent hash-function.

These events are independent.

Cuckoo Hashing

The probability that a given cycle-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

What is the probability that there exists an active cycle structure of size s?

Cuckoo Hashing

The number of cycle-structures of size s is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1} \ .$$

- There are at most s^2 possibilities where to attach the forward and backward links.
- There are at most s possibilities to choose where to place key x.
- There are m^{s-1} possibilities to choose the keys apart from x.
- There are n^{s-1} possibilities to choose the cells.

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\begin{aligned}\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} &= \frac{\mu^2}{nm}\sum_{s=3}^{\infty} s^3\left(\frac{m}{n}\right)^s \\ &\le \frac{\mu^2}{m^2}\sum_{s=3}^{\infty} s^3\left(\frac{1}{1+\epsilon}\right)^s \le \mathcal{O}\left(\frac{1}{m^2}\right) \ .\end{aligned}$$

Here we used the fact that $(1+\epsilon)m \le n$.

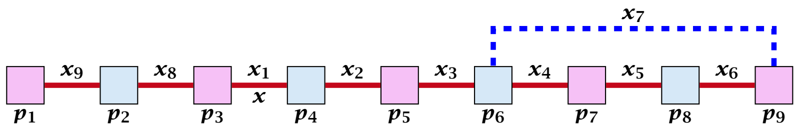
Hence,

$$\Pr[\text{cycle}] = \mathcal{O}\left(\frac{1}{m^2}\right) \ .$$

Cuckoo Hashing

Now, we analyze the probability that a phase is not successful without running into a closed cycle.

Cuckoo Hashing



Sequence of visited keys:

$x = x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_3, x_2, x_1 = x, x_8, x_9, \dots$

Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with x in the order that they are visited during the phase.

Lemma 19

If the sequence is of length p then there exists a sub-sequence of at least $\frac{p+2}{3}$ keys starting with x of distinct keys.

Cuckoo Hashing

Taking $x_1 \to \cdots \to x_i$ twice, and $x_1 \to x_{i+1} \to \dots x_j$ once gives $2i + (j - i + 1) = i + j + 1 \geq p + 2$ keys. Hence, one of the sequences contains at least $(p+2)/3$ keys.

Proof.

Let i be the number of keys (including x) that we see before the first repeated key. Let j denote the total number of distinct keys.

The sequence is of the form:

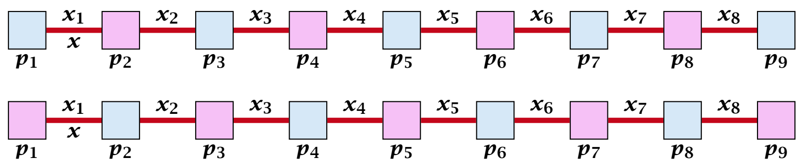
$x = x_1 \to x_2 \to \cdots \to x_i \to x_r \to x_{r-1} \to \cdots \to x_1 \to x_{i+1} \to \cdots \to x_j$

As $r \leq i - 1$ the length p of the sequence is

$$p = i + r + (j - i) \leq i + j - 1 \ .$$

Either sub-sequence $x_1 \to x_2 \to \cdots \to x_i$ or sub-sequence $x_1 \to x_{i+1} \to \cdots \to x_j$ has at least $\frac{p+2}{3}$ elements. □

Cuckoo Hashing



A path-structure of size s is defined by

- $s + 1$ different cells (alternating btw. cells from T_1 and T_2).
- s distinct keys $x = x_1, x_2, \dots, x_s$, linking the cells.
- The leftmost cell is either from T_1 or T_2.

Cuckoo Hashing

A path-structure is active if for every key x_ℓ (linking a cell p_i from T_1 and a cell p_j from T_2) we have

$$h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j$$

Observation:
If a phase takes at least t steps without running into a cycle there must exist an active path-structure of size $(2t + 2)/3$.

Note that we count complete steps. A search that touches $2t$ or $2t + 1$ keys takes t steps.

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

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

$$2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} \le 2\mu^2\left(\frac{m}{n}\right)^{s-1} \le 2\mu^2\left(\frac{1}{1+\epsilon}\right)^{s-1}$$

Plugging in $s = (2t + 2)/3$ gives

$$\le 2\mu^2\left(\frac{1}{1+\epsilon}\right)^{(2t+2)/3-1} = 2\mu^2\left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3} .$$

Cuckoo Hashing

We choose $\text{maxsteps} \ge 3\ell/2 + 1/2$. Then the probability that a phase terminates unsuccessfully without running into a cycle is at most

$$\begin{aligned}
&\Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\quad \le \Pr[\exists \text{ active path-structure of size at least } \tfrac{2\text{maxsteps}+2}{3}] \\
&\quad \le \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \\
&\quad \le \Pr[\exists \text{ active path-structure of size exactly } \ell + 1] \\
&\quad \le 2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{\ell} \le \frac{1}{m^2}
\end{aligned}$$

by choosing $\ell \ge \log\left(\frac{1}{2\mu^2 m^2}\right)/\log\left(\frac{1}{1+\epsilon}\right) = \log\left(2\mu^2 m^2\right)/\log\left(1+\epsilon\right)$

This gives $\text{maxsteps} = \Theta(\log m)$.

Note that the existence of a path structure of size larger than s implies the existence of a path structure of size exactly s.

Cuckoo Hashing

So far we estimated

$$\Pr[\text{cycle}] \le \mathcal{O}\Big(\frac{1}{m^2}\Big)$$

and

$$\Pr[\text{unsuccessful} \mid \text{no cycle}] \le \mathcal{O}\Big(\frac{1}{m^2}\Big)$$

Observe that

$$\begin{aligned}
\Pr[\text{successful}] &= \Pr[\text{no cycle}] - \Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\ge c \cdot \Pr[\text{no cycle}]
\end{aligned}$$

for a suitable constant $c > 0$.

This is a very weak (and trivial) statement but still sufficient for our asymptotic analysis.

Cuckoo Hashing

The expected number of complete steps in the successful phase of an insert operation is:

$$\begin{aligned}&\mathrm{E}[\text{number of steps} \mid \text{phase successful}]\\&\quad= \sum_{t\geq 1}\Pr[\text{search takes at least } t \text{ steps} \mid \text{phase successful}]\end{aligned}$$

We have

$$\begin{aligned}&\Pr[\text{search at least } t \text{ steps} \mid \text{successful}]\\&\quad= \Pr[\text{search at least } t \text{ steps} \wedge \text{successful}]/\Pr[\text{successful}]\\&\quad\leq \frac{1}{c}\Pr[\text{search at least } t \text{ steps} \wedge \text{successful}]/\Pr[\text{no cycle}]\\&\quad\leq \frac{1}{c}\Pr[\text{search at least } t \text{ steps} \wedge \text{no cycle}]/\Pr[\text{no cycle}]\\&\quad= \frac{1}{c}\Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}]\ .\end{aligned}$$

$\Pr[A \mid B] = \dfrac{\Pr[A \wedge B]}{\Pr[B]}$

Cuckoo Hashing

Hence,

$$\begin{aligned}&\mathrm{E}[\text{number of steps} \mid \text{phase successful}]\\&\quad\leq \frac{1}{c}\sum_{t\geq 1}\Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}]\\&\quad\leq \frac{1}{c}\sum_{t\geq 1}2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{(2t-1)/3} = \frac{1}{c}\sum_{t\geq 0}2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{(2(t+1)-1)/3}\\&\quad= \frac{2\mu^2}{c(1+\epsilon)^{1/3}}\sum_{t\geq 0}\Big(\frac{1}{(1+\epsilon)^{2/3}}\Big)^t = \mathcal{O}(1)\ .\end{aligned}$$

This means the expected cost for a successful phase is constant (even after accounting for the cost of the incomplete step that finishes the phase).

Cuckoo Hashing

A phase that is not successful induces cost for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is $q = \mathcal{O}(1/m^2)$ (probability $\mathcal{O}(1/m^2)$ of running into a cycle and probability $\mathcal{O}(1/m^2)$ of reaching $\mathrm{maxsteps}$ without running into a cycle).

A rehash try requires m insertions and takes expected constant time per insertion. It fails with probability $p := \mathcal{O}(1/m)$.

The expected number of unsuccessful rehashes is $\sum_{i\geq 1}p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p)$.

Therefore the expected cost for re-hashes is $\mathcal{O}(m)\cdot\mathcal{O}(p) = \mathcal{O}(1)$.

Formal Proof

Let Y_i denote the event that the i-th rehash occurs and does not lead to a valid configuration (i.e., one of the $m+1$ insertions fails):

$$\Pr[Y_i|Z_i] \leq (m+1)\cdot\mathcal{O}(1/m^2) \leq \mathcal{O}(1/m) =: p\ .$$

Let Z_i denote the event that the i-th rehash occurs:

The 0-th (re)hash is the initial configuration when doing the insert.

$$\Pr[Z_i] \leq \Pr[\wedge_{j=0}^{i-1}Y_j] \leq p^i$$

Let X_i^s, $s \in \{1,\ldots,m+1\}$ denote the cost for inserting the s-th element during the i-th rehash (assuming i-th rehash occurs):

$$\begin{aligned}\mathrm{E}[X_i^s] &= \mathrm{E}[\text{steps} \mid \text{phase successful}]\cdot\Pr[\text{phase sucessful}]\\&\quad+ \mathrm{maxsteps}\cdot\Pr[\text{not sucessful}] = \mathcal{O}(1)\ .\end{aligned}$$

The expected cost for all rehashes is

$$\mathrm{E}\left[\sum_i \sum_s Z_i X_i^s\right]$$

Note that Z_i is independent of X_j^s, $j \geq i$ (however, it is not independent of X_j^s, $j < i$). Hence,

$$\begin{aligned}
\mathrm{E}\left[\sum_i \sum_s Z_i X_s^i\right] &= \sum_i \sum_s \mathrm{E}[Z_i] \cdot \mathrm{E}[X_s^i] \\
&\leq \mathcal{O}(m) \cdot \sum_i p^i \\
&\leq \mathcal{O}(m) \cdot \frac{p}{1-p} \\
&= \mathcal{O}(1) \ .
\end{aligned}$$

Cuckoo Hashing

What kind of hash-functions do we need?

Since maxsteps is $\Theta(\log m)$ the largest size of a path-structure or cycle-structure contains just $\Theta(\log m)$ different keys.

Therefore, it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

How do we make sure that $n \geq (1 + \epsilon)m$?

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

Cuckoo Hashing

Lemma 20

Cuckoo Hashing has an expected constant insert-time and a worst-case constant search-time.

Note that the above lemma only holds if the fill-factor (number of keys/total number of hash-table slots) is at most $\frac{1}{2(1+\epsilon)}$.

The $1/(2(1 + \epsilon))$ fill-factor comes from the fact that the total hash-table is of size $2n$ (because we have two tables of size n); moreover $m \leq (1 + \epsilon)n$.

Hashing

Bibliography

[MS08] Kurt Mehlhorn, Peter Sanders:
Algorithms and Data Structures — The Basic Toolbox,
Springer, 2008

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
Introduction to algorithms (3rd ed.),
MIT Press and McGraw-Hill, 2009

Chapter 4 of [MS08] contains a detailed description about Hashing with Linear Probing and Hashing with Chaining. Also the Perfect Hashing scheme can be found there.

The analysis of Hashing with Chaining under the assumption of uniform hashing can be found in Chapter 11.2 of [CLRS90]. Chapter 11.3.3 describes Universal Hashing. Collision resolution with Open Addressing is described in Chapter 11.4. Chapter 11.5 describes the Perfect Hashing scheme.

Reference for Cuckoo Hashing???