Datenstrukturen und Algorithmen

Exercise 6

FS 2019

Program of today

1. Feedback of last exercise
2. Repetition theory
3. Programming exercise

Feedback

Open hashing:

- $h'(k) = \lceil \ln(k+1) \rceil \bmod q$

Feedback

Open hashing:

- $h'(k) = \lceil \ln(k+1) \rceil \bmod q \rightarrow$ not suitable: $(k = 0) \mapsto 0$

Feedback

Open hashing:

- $h'(k) = \lceil \ln(k+1) \rceil \bmod q$ → not suitable: $(k = 0) \mapsto 0$
- $s(j,k) = k^j \bmod p$

Feedback

Open hashing:

- $h'(k) = \lceil \ln(k+1) \rceil \bmod q \rightarrow$ not suitable: $(k=0) \mapsto 0$
- $s(j,k) = k^j \bmod p \rightarrow$ not suitable: $(k=0) \mapsto 0, (k=1) \mapsto 1$

Feedback

Open hashing:

- $h'(k) = \lceil \ln(k+1) \rceil \bmod q \rightarrow$ not suitable: $(k=0) \mapsto 0$
- $s(j,k) = k^j \bmod p \rightarrow$ not suitable: $(k=0) \mapsto 0, (k=1) \mapsto 1$
- $s(j,k) = ((k \cdot j) \bmod q) + 1$

Feedback

Open hashing:

- $h'(k) = \lceil \ln(k+1) \rceil \bmod q \rightarrow$ not suitable: $(k=0) \mapsto 0$
- $s(j,k) = k^j \bmod p \rightarrow$ not suitable: $(k=0) \mapsto 0, (k=1) \mapsto 1$
- $s(j,k) = ((k \cdot j) \bmod q) + 1 \rightarrow$ not suitable: 1 if k is multiple of q, and range $p-q$ is not covered

Feedback

Coocoo hashing

- $h_1(k) = k \bmod 5$, $h_2(k) = \lfloor k/5 \rfloor \bmod 5$
- add 27, 2, 32

```
T_1: __, __, 27, __, __     T_2: __, __, __, __, __

T_1: __, __,  2, __, __     T_2: 27, __, __, __, __

T_1: __, __, 27, __, __     T_2:  2, 32, __, __, __
```

Feedback

Coocoo hashing

- $h_1(k) = k \bmod 5$, $h_2(k) = \lfloor k/5 \rfloor \bmod 5$
- add 7: infinite loop

```
     T_1: __, __, 27, __, __    T_2:  2, 32, __, __, __
 7:  T_1: __, __,  7, __, __    T_2: 27, 32, __, __, __
 2:  T_1: __, __,  2, __, __    T_2: 27,  7, __, __, __
32:  T_1: __, __, 32, __, __    T_2:  2,  7, __, __, __
27:  T_1: __, __, 27, __, __    T_2:  2, 32, __, __, __
 7:   ...
```

Feedback

Finding a Sub-Array

```
// calculating hash_a, hash_b, c_to_k
It1 window_end = from;
for(It2 current = begin; current != end;
    ++current, ++window_end) {
  if(window_end == to) return to;
  hash_b = (C * hash_b % M + *current) % M;
  hash_a = (C * hash_a % M + *window_end) % M;
  c_to_k = c_to_k * C % M;
}
```

Feedback

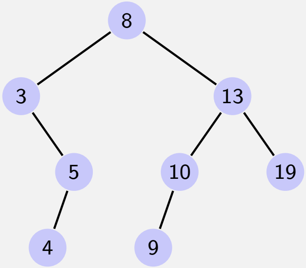
Finding a Sub-Array

```
// looking for b and updating hash_a
for(It1 window_begin = from; ;
    ++window_begin, ++window_end) {
  if(hash_a == hash_b)
    if(std::equal(window_begin, window_end, begin, end))
      return window_begin;
  if(window_end == to) return to;
  hash_a = (C * hash_a % M + *window_end
            + (M - c_to_k) * *window_begin % M) % M;
}
```

2. Repetition theory

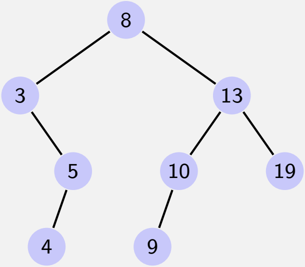
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.



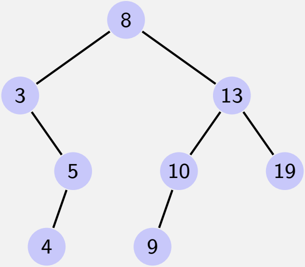
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19



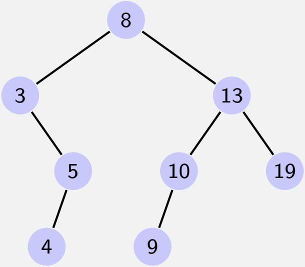
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.



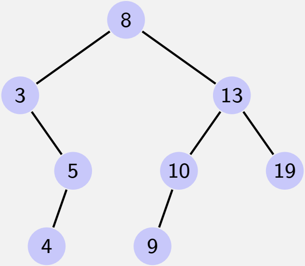
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8



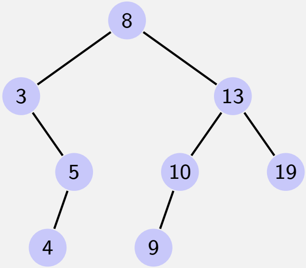
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\text{left}}(v)$, then v, then $T_{\text{right}}(v)$.



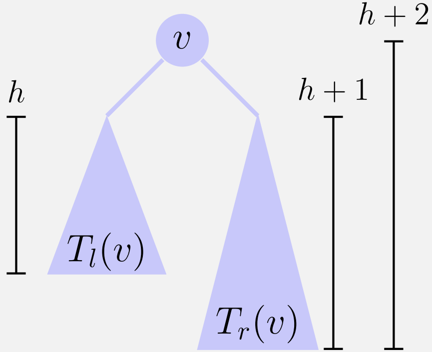
Traversal possibilities

- preorder: v, then $T_{\mathrm{left}}(v)$, then $T_{\mathrm{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\mathrm{left}}(v)$, then $T_{\mathrm{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\mathrm{left}}(v)$, then v, then $T_{\mathrm{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19

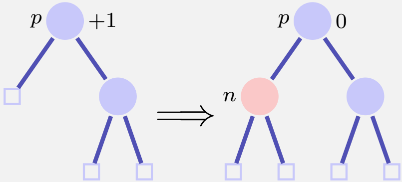


AVL Condition

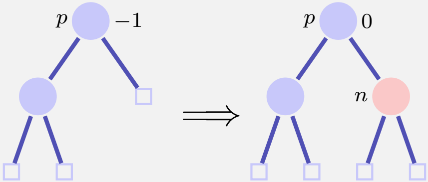
AVL Condition: for each node v *of a tree* $\operatorname{bal}(v) \in \{-1, 0, 1\}$



Balance at Insertion Point



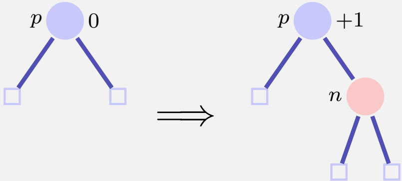
case 1: $\mathrm{bal}(p) = +1$



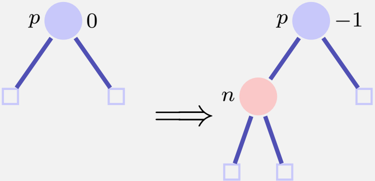
case 2: $\mathrm{bal}(p) = -1$

Finished in both cases because the subtree height did not change

Balance at Insertion Point



case 3.1: $\mathrm{bal}(p) = 0$ right



case 3.2: $\mathrm{bal}(p) = 0$, left

Not finished in both case. Call of `upin(p)`

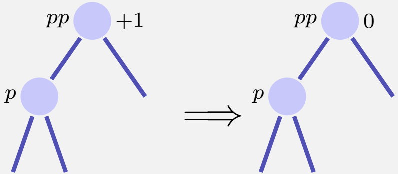
upin(p) - invariant

When `upin(p)` is called it holds that

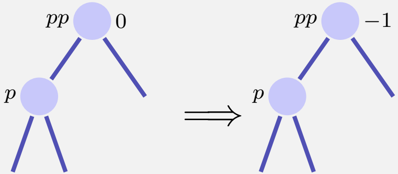
- the subtree from p has grown and
- $\mathrm{bal}(p) \in \{-1, +1\}$

upin(p)

Assumption: p is left son of pp[1]



case 1: $\mathrm{bal}(pp) = +1$, done.



case 2: $\mathrm{bal}(pp) = 0$, **`upin(pp)`**

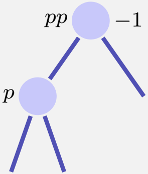
In both cases the AVL-Condition holds for the subtree from pp

---

[1] If p is a right son: symmetric cases with exchange of $+1$ and -1

upin(p)

Assumption: p is left son of pp



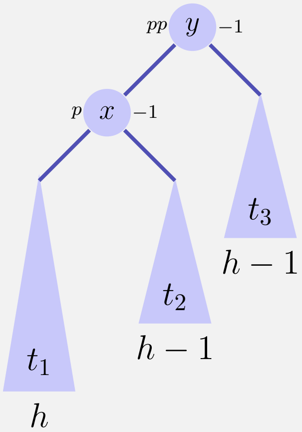
case 3: $\mathrm{bal}(pp) = -1$,

This case is problematic: adding n to the subtree from pp has violated the AVL-condition. Re-balance!

Two cases $\mathrm{bal}(p) = -1$, $\mathrm{bal}(p) = +1$

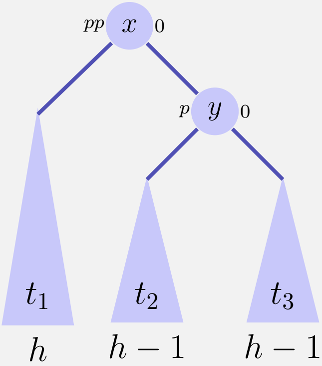
Rotationen

case 1.1 $\mathrm{bal}(p) = -1$. [2]



$\Longrightarrow$

rotation
right

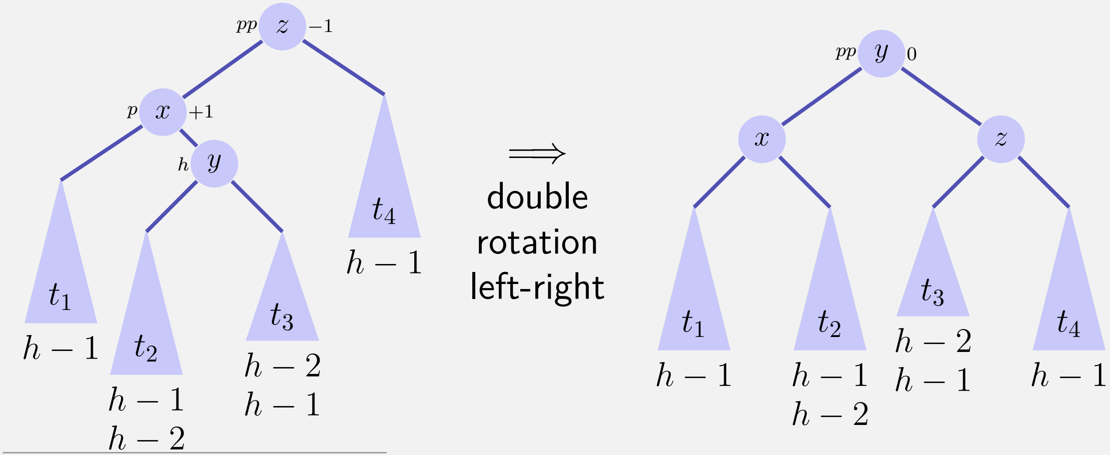


---

[2] p right son: $\mathrm{bal}(pp) = \mathrm{bal}(p) = +1$, left rotation

Rotationen

case 1.2 $\mathrm{bal}(p) = +1$.[3]



$\Longrightarrow$

double rotation left-right

---

[3] p right son: $\mathrm{bal}(pp) = +1$, $\mathrm{bal}(p) = -1$, double rotation right left

3. Programming exercise

Template - AvlNode

```
class AvlNode {
    AvlNode<T> *left = nullptr;
    AvlNode<T> *right = nullptr;
    T value;
    // Returns if subtree height has changed
    bool insert(T x){...}
    // Check if subtree is an AVL tree. Implement it
    // before rotations.
    bool isAvl() {...}
    ...
```

Template - AvlNode

```
...
// Returns if subtree height has changed
// If sub-insert changes height, call upin(..)
bool insert(T x){...}

// See whats wrong, call rotate* if necessary
// Return if AVL condition can be fixed
bool upin(Dir dir){ ... };
void rotateRight(){ ... };
void rotateLeft(){ ... };
...
```

Template - Rotations

- Rotations must update parent's child pointers!

```
// Update this ptr
if(parent == nullptr) root = newroot;
else if(parent->left == this) parent->left = newroot;
else if(parent->right == this) parent->right = newroot
// Update parent pointers
...
```

- Better store pointer to pointer: less cases, no global root

Template - Rotations

- Store which pointer points to this: this_ptr

```
*this_ptr = newroot;
...
// Update parent pointers
x->right = y;
y->this_ptr = &x->right;
```

Questions?