Lecture 5

Hashing and binary search trees
Today’s plan

1. Sets and dictionaries
2. Chained hash tables
3. Designing hash functions
4. Open addressing
5. Perfect hashing
6. Red-black binary trees
Sets and dictionaries

Set

\[ \exists \ x ? \]

Dictionary

What’s the phone number of \( x \)?

- **Python:** lists, dictionaries
- **C++:** set, map
Array-based implementation

- **Assumption:** no two objects have equal keys

- Space $O(|U|)$, search time $O(1)$, insert / delete time $O(1)$
Chained hash tables

hash function representing an image by a low-res image
Chained hash tables

$U$ - universe of keys

hash function $h$

Collision!
Chained hash tables

- universe of keys

\[ U \]

used keys

\[ k_3, k_1, k_4, k_1 \]

hash function \( h \)

- Insert(key, value): insert (key, value) at the head of the list \( T[h(key)] \)
- Time: \( O(1) \)
Chained hash tables

$U$ - universe of keys

- **Search(key, value):** search (key, value) in the list $T[h(key)]$
- **Time:** $O(|T[h(key)]|)$
Chained hash tables

$U$ - universe of keys

- **Delete(key, value):** delete (key, value) from the list $T[h(key)]$
- **Time:** $O(|T[h(key)]|)$
Analysis

- Time of search and delete depends on the length of the list.

- The length of the list depends on how well the keys are distributed by the hash function.

- In the worst case, search and delete cost $O(n)$ time, where $n$ is the number of keys used.

- However, on average chained hash tables demonstrate much better behaviour (if $h$ satisfies a certain assumption).
Sample space: Finite set $S$ of events in which we are interested.

Probability distribution: Function $\Pr : S \to \mathbb{R}$ such that $0 \leq \Pr[s] \leq 1$ for all $s \in S$ and $\sum_{s \in S} \Pr[s] = 1$.

Random variable: A real valued function of $S$. That is, a function $X : S \to \mathbb{R}$.

Expected value of a random variable: The expected value of a random variable $X$ is

$$ E[X] = \sum_{s \in S} \Pr(s) \cdot X(s). $$
We assume to be given a probability distribution on the universe \( U \) of keys.

This induces a distribution on the \( n \)-tuples of keys.

We want to upper-bound

\[
E[T_{\text{search}}(n)] = \sum_{u_1, \ldots, u_n \in U} \text{(worst-case search time for } u_1, \ldots, u_n) \cdot \Pr[k_1 = u_1, \ldots, k_n = u_n]
\]
Analysis

Simple Uniform Hashing Assumption (SUHA):

“$h$ equally distributes the keys into the table slots”

**Theorem.** Assuming SUHA and that $h(x)$ can be computed in $O(1)$ time, $E[T_{search}(n)] = O(1 + n/|T|)$. 
Simple Uniform Hashing Assumption (SUHA):

∀ y ∈ \{1,2,\ldots,|T|\}, there is \Pr[h(x) = y] = 1/|T|, and

∀ y_1, y_2 ∈ \{1,2,\ldots,|T|\}, there is \Pr[h(x_1) = y_1, h(x_2) = y_2] = (1/|T|)^2

**Theorem.** Under SUHA and assuming that \( h(x) \) can be computed in \( O(1) \) time, \( \mathbb{E}[T_{\text{search}}(n)] = O(1 + n/|T|) \).
Analysis

Suppose that a random variable $X$ can only have values $0, 1, 2, \ldots, t$.

**Notation:** For each $i$, define $\Pr[X = i] = \sum_{s \in S, X(s) = i} \Pr[s]$.

**Claim.** If the only possible values for $X$ are $0, 1, 2, \ldots, t$ then

$$E[X] = \sum_{i=0}^{t} i \cdot \Pr[X = i]$$
Analysis

If a random variable $X$ is a sum of $t$ other random variables, $X = X_1 + X_2 + \ldots + X_t$, then

$$E[X] = E[X_1 + X_2 + \ldots + X_t] = E[X_1] + E[X_2] + \ldots + E[X_t]$$

**Application:** We can find the expected value of $X$ by finding the expected values of each of $X_1, X_2, \ldots, X_t$ and then adding these together.
Analysis

1. Unsuccessful search

Suppose that: $k_1, k_2, \ldots, k_n$ are keys in the dictionary, and we perform an unsuccessful search for a key $k$.

If we do not include comparisons to the null pointer, then the number of comparisons for an unsuccessful search for $k$ is

$$X_1 + X_2 + \ldots + X_n$$

where $X_i = \begin{cases} 1, & \text{if } h(k) = h(k_i); \\ 0, & \text{otherwise.} \end{cases}$

The expected time is $E[X] = \sum_i E[X_i] = \sum_i \Pr[X_i = 1] = n/|T|$ by SUHA.
2. Successful search

Suppose keys were introduced in order $k_1, k_2, \ldots, k_n$.

Consider a successful search for $k_i$.

$k_i$ appears before any of $k_1, k_2, \ldots, k_{i-1}$ that are in the same linked list, and after any of $k_{i+1}, k_{i+2}, \ldots, k_n$ that are in the same linked list.
2. Successful search

Number of comparisons to search for \( k_i \) is, therefore,

\[
Y_i = 1 + X_{i+1} + X_{i+2} + \ldots + X_n
\]

where \( X_j = \begin{cases} 
1 & \text{if } h(k_j) = h(k_i); \\
0 & \text{otherwise.} 
\end{cases} \)

Under SUHA, \( E[X_j] = 1/|T| \). We assume that each key in the table is equally likely to be searched for.

By linearity of expectations, the expected search time is

\[
\frac{1}{n} \sum_{i=1}^{n} E[Y_i] = 1 + \frac{1}{n} \sum_{i=1}^{n} (n - i)/|T| = O(1 + n/|T|)
\]
Designing hash functions
Designing hash functions

• Good hash functions must distribute the keys evenly

• If we do not know the distribution of the keys, it can be hard to achieve

• In practice, various heuristics are used, and we will consider several of them

• We can assume that keys are integers
Heuristic hash functions

1. **Division method:** \( h(k) = k \pmod{m} \). It is better to choose \( m \) to be a prime number, and avoid \( m = 2^p \) (as for this value of \( m \) the function always returns the \( p \) least significant bits of the keys)

2. **Multiplication method:** \( h(k) = \lfloor m \cdot (k \cdot A \pmod{1}) \rfloor \), where \( kA \pmod{1} := k \cdot A - \lfloor k \cdot A \rfloor \).

   Usually, \( m \) is chosen to be \( 2^p \) - easy to compute.
Universal hashing

If we fix the hash function $h$, an adversary can always find a probability distribution on the universe of keys for which our function will be “bad”

Let $H = \{ h : U \rightarrow [0,m-1] \}$ be a finite family of hash functions. It is called universal if

$$\forall k_1 \neq k_2 \in U, \ | \{ h \in H : h(k_1) = h(k_2) \} | \leq |H|/m$$

In other words: if we choose $h \in H$ at random, the probability of collision for the keys $k_1, k_2$ is at most $1/m$. 
Universal hashing

Let $H = \{h : U \to [0,m - 1]\}$ be a finite family of hash functions. It is called universal if

$$\forall k_1 \neq k_2 \in U, |\{h \in H : h(k_1) = h(k_2)\}| \leq |H|/m$$

**Theorem.** Let $h$ be a hash function chosen uniformly at random from a universal family of hash functions. Suppose that $h(k)$ can be computed in constant time and that the hash table contains $n$ keys. Then the expected search time is $O(1 + n/m)$.

**Proof:** Analogous to the case when $h$ satisfies SUHA.
Universal hashing

We will now construct a universal family of hash functions.

Let $p$ be a prime number such that $[0, p - 1] \supseteq U$. Define

$$H = \{ h_{a,b}(k) = ((ak + b) \mod p) \mod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p \}$$

**Theorem.** $H$ is a universal family of hash functions.
Universal hashing

\[ H = \{ h_{a,b}(k) = ((ak + b) \mod p) \mod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p \} \]

**Theorem.** \( H \) is a universal family of hash functions.

Let \( \ell_1 = (ak_1 + b) \mod p \) and \( \ell_2 = (ak_2 + b) \mod p \). We have \( \ell_1 \neq \ell_2 \). The number of such pairs is \( p(p - 1) \). Moreover,

\[
\begin{align*}
a &= ((\ell_1 - \ell_2)((k_1 - k_2)^{-1}) \mod p) \mod p) \\
b &= (\ell_1 - ak_1) \mod p
\end{align*}
\]

Hence, there is one-to-one mapping between \((a, b)\) and \((\ell_1, \ell_2)\).
Universal hashing

\[ H = \{ h_{a,b}(k) = ((ak + b) \mod p) \mod m : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p \} \]

**Theorem.** \( H \) is a universal family of hash functions.

The number of \( h \in H \) such that \( h(k_1) = h(k_2) \) is equal to

\[ | \{(\ell_1, \ell_2) : \ell_1 \neq \ell_2 \in \mathbb{Z}_p, \ell_1 = \ell_2 \pmod{m}\} | \leq p(p - 1)/m = |H|/m \]

q.e.d.
Open addressing
Open addressing

- Elements are stored in the table

- Insertion($x$): probe the hash table until we find $x$ or an empty slot. If we find an empty slot, insert $x$ there.

- To define which slots to probe, we use a hash function that depends on the key and the probe number

\[
h(x,0) \qquad h(x,1) \qquad h(x,2)
\]

- we insert $x$ here
Open addressing

• Elements are stored in the table

• Search($x$): **probe** the hash table until either we find $x$ (return YES) or an empty slot (return NO)

• To define which slots to probe, we use a hash function that depends on the key and the probe number

\[
\begin{align*}
    h(43,0) & \downarrow & h(43,1) & \downarrow & h(43,2) & \downarrow \\
        10 & & 7 & 2 & 43 & 
\end{align*}
\]
Which hash function to use?

• In the analysis, we will assume that \( h \) is uniform, i.e. the probe sequence of a key is equally likely to be any of the \( m! \) permutations of the slots.

• Gives good results, but hard to implement.

• In practice, it’s common to use heuristics.

\[
\begin{align*}
\text{h}(43,0) & \downarrow \quad \text{h}(43,1) \downarrow \quad \text{h}(43,2) \downarrow \\
10 & \quad 7 & \quad 2 & \quad 43
\end{align*}
\]
Heuristic hash functions

$h', h'' : U \rightarrow \{0, 1, \ldots, m - 1\}$ - auxiliary hash functions

**Linear probing:** $h(k, i) = (h'(k) + i) \mod m$

Easy to implement, but suffers from primary clustering.

**Quadratic probing:** $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$

Must choose the constants $c_1, c_2$ carefully. Suffers from secondary clustering.

**Double hashing:** $h(k, i) = (h'(k) + ih''(k)) \mod m$

To use the whole table, $h''(k)$ must be relatively prime to $m$, e.g. $h''(k)$ is always odd, $m = 2^i$. 
Analysis of open-address hashing

**Theorem.** Given an open-address hash-table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$, assuming uniform hashing.

Unsuccessful search($x$) = every probed slot except the last one is occupied and does not contain $x$; the last slot is empty.

$A_i$ - $i$th probe occurs and the slot is occupied

$\Pr[\# \text{ of probes } \geq i] = \Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}] =$

$= \Pr[A_1] \cdot \Pr[A_2 | A_1] \cdot \Pr[A_3 | A_1 \cap A_2] \ldots \Pr[A_{i-1} | A_1 \cap A_2 \cap \ldots \cap A_{i-2}]$

(by induction)
Analysis of open-address hashing

**Theorem.** Given an open-address hash-table with load factor \( \alpha = \frac{n}{m} < 1 \), the expected number of probes in an unsuccessful search is at most \( \frac{1}{1 - \alpha} \), assuming uniform hashing.

We must estimate

\[
\Pr[A_1] \cdot \Pr[A_2 | A_1] \cdot \Pr[A_3 | A_1 \cap A_2] \ldots \Pr[A_{i-1} | A_1 \cap A_2 \cap \ldots \cap A_{i-2}]
\]

We have:

\[
\Pr[A_1] = \frac{n}{m} \text{ (} n \text{ cells out of } m \text{ are occupied)}
\]

\[
\Pr[A_2 | A_1] = \frac{n-1}{m-1} \text{ (we can hit one of the remaining } n - 1 \text{ elements in } m - 1 \text{ cells)}
\]

\[
\ldots
\]

\[
\Pr[A_{i-1} | A_1 \cap A_2 \cap \ldots \cap A_{i-2}] = \frac{n-i+2}{m-i+2}
\]
Analysis of open-address hashing

**Theorem.** Given an open-address hash-table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$, assuming uniform hashing.

We must estimate

$$\Pr[A_1] \cdot \Pr[A_2 | A_1] \cdot \Pr[A_3 | A_1 \cap A_2] \cdots \Pr[A_{i-1} | A_1 \cap A_2 \cap \ldots \cap A_{i-2}] \leq$$

$$\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

Expected number of probes =

$$\sum_{i=1}^{\infty} \Pr[\# \text{ of probes } \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1 - \alpha}$$
Analysis of open-address hashing

**Corollary.** The expected number of probes during insertion($x$) is at most $1/(1 - \alpha)$, assuming uniform hashing.

If we insert $x$, we first run an unsuccessful search for it.
Analysis of open-address hashing

**Theorem.** The expected number of probes during a successful search is at most \( \frac{1}{\alpha} \ln \frac{1}{1 - \alpha} \), assuming uniform hashing and assuming each key in the table is equally likely to be searched for.

A successful search for \( x \) probes the same sequence of slots as insertion(\( x \)).

If \( x \) is the \( i \)-th item inserted into the table, insertion(\( x \)) probes \( \leq \frac{1}{1 - i/m} \) slots in expectation.

Therefore, the expected time of a successful search is at most

\[
\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{m}{n} \ln \frac{m}{m-n} = \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}
\]
Perfect hashing

MR. PERFECT
Perfect hashing

- If the set of keys is **static** (doesn’t change), we can do better
- We call a hashing technique **perfect** if it guarantees constant worst-case search time

\[ \Theta(n) \]

**Theorem.** There is a perfect hashing scheme that stores \( n \) keys in \( \Theta(n) \) space.
Binary search trees

Kayama Matazo
Binary search trees

- **Binary tree**: every node has at most two children
- For every node storing element \( \ell \), the subtree rooted at the left child (if it exists) contains elements \( \leq \ell \), and the subtree rooted at the right child elements \( > \ell \)
- Access, insertion, deletion of a given element: \( O(h) \) time, where \( h \) is the height of the tree.
Binary search trees

Each tree contains 4 elements and has height 4.
Implementations

- 2-3 tree
- AA tree
- AVL tree
- B-tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap
- Weight-balanced tree
- Tango trees

... et cetera
A red-black tree is a **balanced** binary search tree where each node has a color, which can be either Red or Black.

By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the attributes color, key, left, right, and p.

If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.
A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
Lemma. The height of a red-black tree with $n$ nodes is at most $2 \log(n + 1)$.

Let $bh(x)$ be the number of black nodes in the path from $x$ to a leaf.

By induction on $bh(x)$: the subtree rooted at $x$ contains at least $2^{bh(x)} - 1$ nodes.

Hence, if $h$ is the black height of the tree, $n \geq 2^h - 1$ and $h \leq \log(n + 1)$.

Since in any root-to-leaf path the number of red nodes is at most twice the number of black nodes, the lemma follows.
The operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with $n$ keys, take $O(\log n)$ time. Because they modify the tree, the result may violate the red-black properties.

- To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.
- We change the pointer structure through rotation, which is a local operation in a search tree that preserves the binary-search-tree property.
- There are two kinds of rotations: left rotations and right rotations.
Left rotation

When we do a left rotation on a node $x$, we assume that its right child $y$ is not null; $x$ may be any node in the tree.
Right rotation is essentially the reverse operation.
**Insertion**

We can insert a node into an n-node red-black tree in $O(\log n)$ time.

- To do so, we insert the node into the tree T as if it were an ordinary binary search tree.
- Then we color it red.
- To guarantee that the red-black properties are preserved, we then call an auxiliary procedure `RB-INSERT-FIXUP` to recolor nodes and perform rotations.
Insertion

Before:

```
        2
       / \  
      1   4
     / \   /
    3   5  6
   / \   / \
  1   4  3   5
```

After:

```
        2
       / \  
      1   4
     /     /
    3     5
   /   \   /
  1     3   6
```

Because of the insertion operation, the red values are moved to the right and the new value is inserted. Then, the red values are moved back to the correct positions to fix the tree.
RB-INSERT(T, z)
1. y ← T.nil
2. x ← T.root
3. while x ≠ T.nil
4. y = x
5. if z.key < x.key
6. then x ← x.left
7. else x ← x.right
8. z.p = y
9. if y = T.nil
10. then T.root ← z
11. else if z.key < y.key
12. then y.left ← z
13. else
14. y.right ← z
14. z.left ← T.nil
15. z.right ← T.nil
16. z.color ← RED
17. RB-INSERT-FIXUP(T, z)
We now need to show how to restore red-black properties after insertion. Let \( z \) be the node that violates the red-black properties.

**Case 1.** \( z \)'s uncle \( y \) is red  
**Case 2.** \( z \)'s uncle \( y \) is black and \( z \) is a right child  
**Case 3.** \( z \)'s uncle \( y \) is black and \( z \) is a left child
**Case 1.** $z$’s uncle $y$ is red

we moved $z$, the “bad” node, up

color $z.p$ BLACK,
color $y$ BLACK,
color $z.p.p$ RED
Case 2. $z$'s uncle $y$ is black and $z$ is a right child

Case 3. $z$'s uncle $y$ is black and $z$ is a left child
Case 3. $z$'s uncle $y$ is black and $z$ is a left child.

We’re good!

color $z.p$ BLACK,
color $z.p.p$ RED,
right-rotate
RB-INSERT-FIXUP

RB-INSERT-FIXUP(T, z)
1. while z.p.color == RED
2.   if z.p == z.p.p.left
3.     then y ← z.p.p.right
4.     if y.color == RED
5.       then z.p.color ← BLACK //Case 1
6.       y.color ← BLACK //Case 1
7.       z.p.p.color ← RED //Case 1
8.       z ← z.p.p // Case 1
9.     else if z == z.p.right
10.    then z ← z.p //Case 2
11.     LEFT-ROTATE(T, z) //Case 2
12.    z.p.color ← BLACK //Case 3
13.    z.p.p.color ← RED //Case 3
14.     RIGHT-ROTATE(T, z.p.p) // Case 3
15.    else same as then clause with "right" and "left" exchanged
16.    T.root.color ← BLACK
To understand how RB-INSERT-FIXUP works, we shall break our examination of the code into three major steps:

- **First**, we shall determine what violations of the red-black properties are introduced in RB-INSERT when node $z$ is inserted and colored red.
- **Second**, we shall examine the overall goal of the while loop in lines 1–15.
- **Finally**, we shall explore each of the three cases within the while loop’s body and see how they accomplish the goal.
The while loop in lines 1–15 maintains the following three-part invariant. At the start of each iteration of the loop:

1. Node z is red.
2. If z.p is the root, then z.p is black.
3. If there is a violation of the red-black properties, there is at most one violation, and it is of either
   - Property 2: The root is black. Violated when z is the root and is red, in this case, we color z black.
   - Property 4: If a node is red, then both its children are black. Violated when z and z.p are red, in this case, the depth of z can only decrease, this happens $O(\log n)$ times.
Deletion

Like other basic operations on a red-black tree, deletion of a node takes time $O(\log n)$.

Deleting a node from a red-black tree is a bit more complicated than inserting a node.

The procedure for deleting a node from a red-black tree is based on the RB-DELETE procedure.

First, we need to design the TRANSPLANT subroutine that RB-DELETE calls so that it applies to a red-black tree.
Deletion

Case 1: node has no children

Case 2: node has one child

Case 3: node has two children

S - successor of the node to delete
Deletion

Essentially, this procedure puts $v$ in place of $u$.

**RB-TRANSPLANT**($T$, $u$, $v$)
1. **if** $u.p == T.nil$
2. $T.root = v$
3. **else if** $u == u.p.left$
4. $u.p.left = v$
5. **else**
6. $u.p.right = v$
6. $v.p = u.p$
Deletion

**RB-DELETE (T, z)**
1. \( y = z \)
2. \( y\text{-original-color} = y\text{.color} \)
3. **if** \( z\text{.left} == T\text{.nil} \)
   4. \( x = z\text{.right} \)
   5. **RB-TRANSPLANT (T, z, z\text{.right})**
4. **else if** \( z\text{.right} == T\text{.nil} \)
   7. \( x = z\text{.left} \)
   8. **RB-TRANSPLANT(T, z, z\text{.left})**
5. **else**
10. \( y = \text{TREE-MINIMUM(z\text{.right})} \)
11. \( y\text{-original-color} = y\text{.color} \)
12. \( x = y\text{.right} \)
13. **if** \( y\text{.p} == z \)
14. \( x\text{.p} = y \)
15. else
16.   RB-TRANSPLANT(T, y, y.right)
17.   y.right = z.right
18.   y.right.p = y
19.   RB-TRANSPLANT(T, z, y)
20.   y.left = z.left
21.   y.left.p = y
22.   y.color = z.color
23. if y-original-color == BLACK
24.   RB-DELETE-FIXUP(T, x)
Deletion

• If node $y$ (the successor of $z$ that we move to $z$’s place) is red, no red-black properties are violated.
• If node $y$ is black, we can violate three properties:
  1. **Property 2:** The root is black (when $y$ is the root and we move $y$’s child $x$, possibly red, in its place)
  2. **Property 4:** If a node is red, then both its children are black. (when we move $x$)
  3. **Property 5:** For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. (every root-to-leaf path containing node $y$ now contains one less black node)
We now need to show how to restore red-black properties after insertion. Let $x$ be the node that violates the red-black properties. We will consider four cases:

**Case 1:** $x$’s sibling $w$ is red

**Case 2:** $x$’s sibling $w$ is black, and both children of $w$ are black

**Case 3:** $x$’s sibling $w$ is black, and $w$’s left child is red, $w$’s right child is black

**Case 4:** $x$’s sibling $w$ is black, and $w$’s right child is red
Case 1: x’s sibling w is red

Reduces to Cases 2-4
Case 2: $x$'s sibling $w$ is black, and both children of $w$ are black
**RB-DELETE-FIXUP**

**Case 3:** $x$'s sibling $w$ is black, and $w$'s left child is red, and $w$'s right child is black

Reduces to Case 4
Case 4: x’s sibling w is black, and w’s right child is red
RB-DELETE-FIXUP(T, x)
1. while x != T.root and x.color == BLACK
2.     if x == x.p.left
3.         w = x.p.right
4.     if w.color == RED
5.         w.color = BLACK //case 1
6.         x.p.color = RED //case 1
7.     LEFT-ROTATE(T, x.p) //case 1
8.         w = x.p.right //case 1
9.     if w.left.color == BLACK and w.right.color == BLACK
10.    w.color = RED //case 2
11.    x = x.p //case 2
12. \textbf{else if} \( w \text{.right.color} == \text{BLACK} \)
13. \hspace{1em} \( w \text{.left.color} = \text{BLACK} \) \hspace{1em} //case 3
14. \hspace{1em} \( w \text{.color} = \text{RED} \) \hspace{1em} //case 3
15. \hspace{1em} \text{RIGHT-ROTATE}(T, w) \hspace{1em} //case 3
16. \hspace{1em} \( w = x \text{.p.right} \) \hspace{1em} //case 3
17. \hspace{1em} \( w \text{.color} = x \text{.p.color} \) \hspace{1em} //case 4
18. \hspace{1em} \( x \text{.p.color} = \text{BLACK} \) \hspace{1em} //case 4
19. \hspace{1em} \( w \text{.right.color} = \text{BLACK} \) \hspace{1em} //case 4
20. \hspace{1em} \text{LEFT-ROTATE}(T, x \text{.p}) \hspace{1em} //case 4
21. \hspace{1em} \( x = T \text{.root} \) \hspace{1em} //case 4
22. \textbf{else same as then clause with “right” and “left” exchanged}
23. \( x \text{.color} = \text{BLACK} \)
Open problem: dynamic optimality

• In practice, we access some elements more often than others. Can we use this to make the search faster?

• Standard operations on BSTs: move the pointer to a child / the parent of the pointer, perform a rotation on the pointer and its parent

• For a fixed access sequence $X$, let $OPT(X)$ be denote the number of unit-cost operations made by the fastest BST for $X$

• A BST is $c$-competitive if it executes all sufficiently long sequences $X$ in time at most $c \cdot OPT(X)$.

• Tango trees are $O(\log \log n)$-competitive. Splay trees are conjectured to be $O(1)$-competitive, but we do not know if this is true.
Next lecture

- Pattern matching
  - Naive
  - KMP
- Dictionary look-up (Trie)
- Multiple pattern matching
  - Aho-Corasick
  - Suffix trees

Thank you!