Next, we discuss data structures that build approximately balanced binary trees.

- Red-black trees & binary
- 2-3-4 Trees, B-trees, & not binary
- AVL trees (1962, 1st one)
- Skip lists, Treaps (<1996 most recent)

Red-black trees - all operations in $O(\log n)$ time

A BST is a red-black tree if it satisfies the following properties:

1. Each node is red or black
2. Root is black
3. Leaves (NIL) are black
4. Parent of every red node is black (so no 2 consecutive reds)
5. All paths from node to leaf have same number of black nodes

$\text{Black-height}(x) = \text{number of black nodes on any path from root not including } x \text{ to a leaf}$

All internal nodes have exactly 2 children, leaves no children

Example

![Red-black tree example diagram]
the properties force the tree to have \( \log \) \( h \), and make properties easy to maintain.

Tricky to maintain properties as we do updates.

Valid binary search tree

order traversal is sorted (red)

Theorem: any red-black tree with \( n \) nodes has \( h \leq 2 \ln(n+1) = O(\log n) \)

pf: base by induction.

pf needs more intuition.
Take red nodes + merge with their parents

In our example we get:

```
    7
   / \
  3   8/10/11
   \   /
    22/26
```

This is a 2-3-4 tree. Each parent has 2, 3 or 4 kids
(Foreshadowing of what is to come)

- In a 2-3-4 tree all leaves have same depth (\(= \log_{4/3} \text{root}\))
- Every internal node has between 2 - 4 children

Number of leaves in a binary tree with
\(n\) internal nodes is \(n+1\)

Also \(n+1\) leaves in the 2-3-4 tree. (because we didn’t change the)
\(H\) of leaves

A 2-3-4 tree of height \(h\) has between \(2^h\) and
\(4^h\) leaves.

\[ 2^h \leq n+1 \leq 4^h \]

\[ h' \leq \log(n+1) \leq 2h' \]

Now relate \(h\) (\(h'\) of red-black tree) to \(h'\).
Using property 4, we know \(h \leq 2h'\)

\[ h \leq ah' \leq a \log(n+1) \]
Later we'll see how to manipulate 2-3-4 trees directly.

Today we'll do red-black trees.

Recap: we know red-black trees are always balanced, if we maintain red-black tree properties all operations will be efficient: \( \mathcal{O}(\text{ht}) = \mathcal{O}(\log n) \)

Ordinary insert/delete on BSTs will not satisfy red-black properties. Updates must therefore modify the tree to preserve red-black properties.

Types of operations we'll do:
- BST operations
- Color changes
- Change links via rotations (this is the only way we'll change links)

Rotations:

\[ \text{Right Rotate}(B) \]

\[ \text{Left Rotate}(A) \]

Check: it preserves BST property.

If 4 nodes \( x, y, \) and \( z \) in left subtree all \( x \)
```plaintext
a < b < c  

in left tree  a < A < b < B < c  
same in right tree.

- Reverses operation.
- So left rotate (A) or opposite

- Constant time to perform a rotation

**INSERTION**  (RB-insert)

Idea:  - Tree insert (x)
       - Make x red
       - Problem if x's parent is red (Prop 4 may be violated)
       - But balance property still holds
       - Move violation of Prop 4 up the tree via recoloring until we can fix violation using a rotation (plus recoloring)

Example insert 15

```

[Diagram of red-black tree with nodes labeled 7, 3, 8, 10, 11, 22, 26, and 15, showing the insertion process and violating Prop 4, with steps for fixing it through rotations and recoloring.]
(case 1)

Violation moved up the tree — between 10 + 18

Look at grandparent of 10.
Can’t do same trick since grandparent has one red, one black child.
Instead do a rotation — rotate 18 to the right

Rotate - R+ (18)

Still a violation between 10 + 18
But now it is straight (7-10-18)
Not Zy-Zyzed

Now rotate left (7) & recoloring
General Algorithm (not terribly intuitive, but can figure it out if you restrict to rotations/recoloring)

Try to recolor - pushing up tree
If we get stuck, rotations work (at most 3)

```
RB-insert(T, x)
```

Tree-Insert(T, x)

```
color(x) = red
while x ≠ root[T] and color[x] = red
    if p(x) = left[p[p[x]]] (case A)
        y ← right(p[p[x]])
        if color(y) = red
            then case B
        else if x = right(p[x])
            then case A2
        else p[x] = right[p[p[x]]] (case B)
reverse left + right above
```

Color[root[T]] ← black
3 cases

意味着这个子树有一个黑色根。

所有节点都有相同的黑色高度。

**Case 1.1 (Case A.1)**

- 以 D 为例。
- 因为 A, B, D 都是红色的，
- 且所有节点都有相同的黑色高度。

在 2-3-4 树中，这是不成立的。

使得 A, D 黑色，C 红色。

**NEW**

- 新的 X = C
- 现在 C 可能会违反属性 4。
- C 更接近根。
- 继续使用 X = C。

注意 X 可以是 A 的左或右 - 没有关系。
**Case 4.2**

- $x$ is right child of $A$

- 2-3-4, only 4 children

Do left-rotate ($A$)

By zigzag between $x$ and grandparent

Now zigzag

**Case 4.3**  

Up picture is as above  

Do right-rotate ($C$), then recolor

No violations!  
(check for hw)

Relief case 1 - recall  
Case 2 - zigzag  
Case 3 - zigzag  Final position
Case 2 + 3 are terminal
   (Case 3 - done)
   (Case 2 \rightarrow \text{immediate} \text{ (as 3)})

Case 1 repeats only log n times

So total time is \(O(n \log n)\)

Also - only \(O(1)\) rotations. Most changes are recoloring so very fast in practice.

Also can do other stuff (search) during the recoloring.

Rotation more expensive since nodes have to be locked during rotations.

DELETION - more complicated.
   see book

End lecture 2

Tutorial 2: Deletion in Red/Black Trees

Middle Q: 2-3 trees

\(\text{modified Red/Black}\)

Only 2 red so no recoloring?
1. grandparent of $x$ has a black child, $w$. 4 cases

2. grandparent of $x$ has 2 red children. Recolor. Eliminates violation or moves violation up the tree.Equivalent to a split of
Example Red Black Insert (14)

corresponds to this 2 3 4 tree

3 4 6 8 11 13 14 17

10
5 12
4 8 11 15
3 6 13 14

10
5 12
4 8 15
3 13 14

10
5 12
4 8 15
3 13 14
Deletion in Red-Black Trees

Do binary search to find item to be removed.

If u does not have an external child, find immediate successor v, swap u with v, and then remove u (which now has an external child).

If v was red, (then r black) — done.
If v black and r red, make r black — done.
On v black, r black. Make r double black (creates underflow in unrooted 2-3-4 tree).

To remedy double-black, there are 3 cases.

1. Sibling y of r is black + has red child z, violation.

   x can be black or red.
   y black, z red, r db1 black.
   x & z same color as color of x above.

   rotate y red.

   or

   rotate x red.
Case 2: Sibling y of r is black, both children of y are black.

Eliminates dbl black or propagate it up the tree.

R coloring

Color r black
Color y red
Color x black if it was red, or dbl black if it was black.

Case 3: Sibling y of r is red. Turn it into case 1 or case 2.