catch(...){
    printf("Assignment::SolveProblem() AAAA!"ennai);
}
ADD SLIDES ON DISJOINT SETS
2-3 Tree

Outline

- Balanced Search Trees
  - 2-3 Trees
  - 2-3-4 Trees
Why care about advanced implementations?

Same entries, different insertion sequence:

→ Not good! Would like to keep tree balanced.
B-TREE

• B-tree keeps data sorted and allows searches, sequential access, insertions, and deletions in $\log(n)$.
• The B-tree is a generalization of a BST (node can have more than two children)
• Unlike balanced BST, the B-tree is optimized for systems that read and write.
• Used in databases and filesystems.
2-3 Trees

Features

- each internal node has either 2 or 3 children
- all leaves are at the same level
2-3 Trees with Ordered Nodes

2-node

(a)

S

Search keys < S

Search keys > S

3-node

(b)

S

L

Search keys < S

Search keys > S

Search keys > L

Search keys > S and < L

• leaf node can be either a 2-node or a 3-node
Example of 2-3 Tree
What did we gain?

(a) 

(b) 

What is the time efficiency of searching for an item?
Gain: Ease of Keeping the Tree Balanced

Binary Search Tree

both trees after inserting items 39, 38, ... 32

2-3 Tree
Inserting Items

Insert 39
Inserting Items

Insert 38

insert in leaf

divide leaf
and move middle
value up to parent

result

(a)  
30  
10 20 38 39 40

(b)  
30 39  
10 20 38 40

(c)  
50  
30 39  
10 20 38 40  
70 80 90 60 80 100
Inserting Items

Insert 37

Diagram showing a binary tree with nodes containing numbers: 10, 20, 30, 37, 38, 40, 50, 60, 70, 80, 90, 100.
Inserting Items

**Insert 36**

- **insert in leaf**

(a)  

```
10 20 36 37 38 40
```

- **divide leaf and move middle value up to parent**

(b)  

```
  50
  /  \  ...
 30  37  39
   / \  / \  /  \  /   \  /     \  /     \  /       \  /         \  /           \  /             \  /               \  /                 \  /                     \  /                         \  /
10 20 36 38 40
```

overcrowded node
Inserting Items

... still inserting 36

divide overcrowded node,
move middle value up to parent,
attach children to smallest and largest

result

(c) 37 50
   /   \
  30   39
 /     /     \
10 20 36 38 40

(d) 37 50
   /   \
  30   39
 /     /     \
10 20 36 38 40 60 80 100
Inserting Items

After Insertion of 35, 34, 33
Inserting so far

(a) 

(b) 

Slide 18
Inserting so far

(a)

(b)
Inserting Items

How do we insert 32?
Inserting Items

- creating a new root if necessary
- tree grows at the root
Inserting Items

Final Result
Deleting Items

Delete 70

(a) Swap with inorder successor

10 20 30 40

50

70 80 90 100
Deleting Items

Deleting 70: swap 70 with inorder successor (80)

(a)

Swap with inorder successor
Deleting Items

Deleting 70: ... get rid of 70

(b) 80 90
   60
Delete value from leaf

(c) 80 90
   60 100
Merge nodes by deleting empty leaf and moving 80 down

(d) 90
   60 80
   100
Deleting Items

Result

(e)
Deleting Items

Delete 100

(e)
Deleting Items

Deleting 100

(a) 90
  60  80
  Delete value from leaf

(b) 90
  60  80
  Doesn’t work

(c) 80
  60  90
  Redistribute
Deleting Items

Result

(d)
Deleting Items

Delete 80
Deleting Items

Deleting 80 ...

(a)

Swap with inorder successor
Deleting Items

Deleting 80 ...

(b) 60 90
Delete value from leaf

(c) 30
10 20 40 60 90
Merge by moving 90 down and removing empty leaf

Node becomes empty
Deleting Items

Deleting 80 ...

(d) Root becomes empty

Merge: move 50 down, adopt empty leaf’s child, remove empty node

(e) Remove empty root
Deleting Items

Final Result

comparison with binary search tree
Deletion Algorithm I

Deleting item $I$:

1. Locate node $n$, which contains item $I$
2. If node $n$ is not a leaf $\rightarrow$ swap $I$ with inorder successor $\rightarrow$ deletion always begins at a leaf
3. If leaf node $n$ contains another item, just delete item $I$ else try to redistribute nodes from siblings (see next slide) if not possible, merge node (see next slide)
Deletion Algorithm II

Redistribution  
A sibling has 2 items:  
→ redistribute item between siblings and parent

Merging  
No sibling has 2 items:  
→ merge node  
→ move item from parent to sibling
Deletion Algorithm III

Redistribution
Internal node $n$ has no item left
$\rightarrow$ redistribute

Merging
Redistribution not possible:
$\rightarrow$ merge node
$\rightarrow$ move item from parent to sibling
$\rightarrow$ adopt child of $n$

If $n$'s parent ends up without item, apply process recursively
Deletion Algorithm IV

If merging process reaches the root and root is without item \(\Rightarrow\) delete root

![Diagram showing deletion process](image-url)
Operations of 2-3 Trees

all operations have time complexity of log n
2-3-4 Trees

- similar to 2-3 trees
- 4-nodes can have 3 items and 4 children

4-node

```
  S M L
```

- Search keys < S
- Search keys > S and < M
- Search keys > M and < L
2-3-4 Tree Example
2–3–4 Trees and Red-Black Trees

• 2–3–4 trees are an isometry of red-black trees
  – for every 2–3–4 tree, there exists red–black tree with data elements in the same order.
  – operations on 2–3–4 trees that cause node expansions, splits and merges are equivalent to the color-flipping and rotations in red–black trees.

• 2–3–4 trees, difficult to implement in most programming languages so RB-trees tend to be used instead.
2-3-4 Tree: Insertion

Insertion procedure:
• similar to insertion in 2-3 trees
• items are inserted at the leafs
• since a 4-node cannot take another item, 4-nodes are split up during insertion process

Strategy
• on the way from the root down to the leaf: split up all 4-nodes "on the way"
→ insertion can be done in one pass
 (remember: in 2-3 trees, a reverse pass might be necessary)
2-3-4 Tree: Insertion

Inserting 60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 100
2-3-4 Tree: Insertion

Inserting 60, 30, 10, 20 ...

(a) 10 30 60
(b) 30 10 60
(c) 30 10 20 60

... 50, 40 ...

2-3-4 Tree: Insertion

Inserting 50, 40 ...

\[ \begin{array}{c}
30 \\
10 \ 20 \ 40 \ 50 \ 60 \\
\end{array} \]

... 70, ...
2-3-4 Tree: Insertion

Inserting 70 ...

... 80, 15 ...
2-3-4 Tree: Insertion

Inserting 80, 15 ...

... 90 ...

Slide 48
2-3-4 Tree: Insertion

Inserting 90 ...

(a)

(b)

... 100 ...

Slide 49
2-3-4 Tree: Insertion

Inserting 100 ...
2-3-4 Tree: Insertion Procedure

Splitting 4-nodes during Insertion
2-3-4 Tree: Insertion Procedure

Splitting a 4-node whose parent is a 2-node during insertion

(a)

(b)
2-3-4 Tree: Insertion Procedure

Splitting a 4-node whose parent is a 3-node during insertion

(a) P Q
    S M L
    a b c d
e f

M P Q
    S L
e f
    a b c d

(b) P Q
    S M L
    b c d e
    a f

P M Q
    S L
    a f
    b c d e

(c) P Q
    S M L
    c d e f

P Q M
    S L
c d e f
    a b
2-3-4 Tree: Deletion

Deletion procedure:

• similar to deletion in 2-3 trees
• items are deleted at the leaves
  → swap item of internal node with inorder successor
• note: a 2-node leaf creates a problem

Strategy (different strategies possible)

• on the way from the root down to the leaf:
  turn 2-nodes (except root) into 3-nodes
  → deletion can be done in one pass
  (remember: in 2-3 trees, a reverse pass might be necessary)
Turning a 2-node into a 3-node ...

Case 1: an adjacent sibling has 2 or 3 items
  → "steal" item from sibling by rotating items and moving subtree

```
  30  50
 /     \
10  20  40
  \
   25

  20  50
 /     \
10     30  40
    \
     25
```

2-3-4 Tree: Deletion

Turning a 2-node into a 3-node ...

Case 2: each adjacent sibling has only one item
   -> "steal" item from parent and merge node with sibling
   (note: parent has at least two items, unless it is the root)
Delete 32, 35, 40, 38, 39, 37, 60
Red-Black tree

• Recall binary search tree
  – Key values in the left subtree $\leq$ the node value
  – Key values in the right subtree $\geq$ the node value

• Operations:
  – insertion, deletion
  – Search, maximum, minimum, successor, predecessor.
  – $O(h)$, $h$ is the height of the tree.
Red-black trees

• Definition: a binary tree, satisfying:
  1. Every node is red or black
  2. The root is black
  3. Every leaf is NIL and is black
  4. If a node is red, then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

• Purpose: keep the tree balanced.

• Other balanced search tree:
  – AVL tree, 2-3-4 tree, Splay tree, Treap
Figure 13.1  A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL’s have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel nil[T], which is always black, and with black-heights omitted. The root’s parent is also the sentinel. (c) The same red-black tree but with leaves and the root’s parent omitted entirely. We shall use this drawing style in the remainder of this chapter.
Fields and property

• Left, right, parent, color, key
• bh(x), black-height of x, the number of black nodes on any path from x (excluding x) to a leaf.
• A red-black tree with n internal nodes has height at most $2\log(n+1)$.
  – Note: internal nodes: all normal key-bearing nodes. External nodes: Nil nodes or the Nil Sentinel.
  – A subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes.
  – By property 4, $bh(root) \geq h/2$.
  – $n > 2^{h/2} - 1$
Some operations in $\log(n)$

- Search, minimum, maximum, successor, predecessor.
- Let us discuss insert or delete.
Left rotation:

\[
y = \text{right}[x]; \quad \text{right}[x] \leftarrow \text{left}[y]; \quad \text{if}(\text{left}[y] \neq \text{nil}) \quad \text{p}[\text{left}[y]] = y; \quad \text{p}[y] = \text{p}[x]; \\
\quad \text{if}(\text{p}[x] = \text{nil}) \quad \text{root} = y; \quad \text{else} \quad \text{right}[\text{p}[x]] = y; \quad \text{left}[\text{p}[x]] = x; \quad \text{left}[y] = x; \quad \text{p}[x] = y;
\]

Right rotation:

\[
x = \text{left}[y]; \quad \text{left}[y] = \text{right}[x]; \quad \text{if}(\text{right}[x] \neq \text{nil}) \quad \text{p}[\text{right}[x]] = x; \quad \text{p}[x] = \text{p}[y]; \quad \text{if}(\text{p}[y] = \text{nil}) \quad \text{root} = x \\
\quad \text{if}(\text{left}[\text{p}[y]] = y) \quad \text{left}[\text{p}[y]] = x; \quad \text{else} \quad \text{right}[\text{p}[y]] = x; \quad \text{right}[x] = y; \quad \text{p}[y] = x;
\]

**Figure 13.2** The rotation operations on a binary search tree. The operation \text{LEFT-ROTATE}(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation \text{RIGHT-ROTATE}(T, y). The letters \( \alpha, \beta, \) and \( \gamma \) represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in \( \alpha \) precede \text{key}[x], which precedes the keys in \( \beta \), which precede \text{key}[y], which precedes the keys in \( \gamma \).
Figure 13.3  An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.
\textbf{RB-INSERT} \((T, z)\)

1 \quad y \leftarrow \text{nil}[T]
2 \quad x \leftarrow \text{root}[T]
3 \quad \textbf{while} x \neq \text{nil}[T]
4 \quad \hspace{1em} \textbf{do} y \leftarrow x
5 \quad \hspace{2em} \textbf{if} \ \text{key}[z] < \text{key}[x]
6 \quad \hspace{3em} \textbf{then} x \leftarrow \text{left}[x]
7 \quad \hspace{3em} \textbf{else} x \leftarrow \text{right}[x]
8 \quad p[z] \leftarrow y
9 \quad \textbf{if} y = \text{nil}[T]
10 \quad \hspace{1em} \textbf{then} \ \text{root}[T] \leftarrow z
11 \quad \textbf{else if} \ \text{key}[z] < \text{key}[y]
12 \quad \hspace{2em} \textbf{then} \ \text{left}[y] \leftarrow z
13 \quad \hspace{2em} \textbf{else} \ \text{right}[y] \leftarrow z
14 \quad \text{left}[z] \leftarrow \text{nil}[T]
15 \quad \text{right}[z] \leftarrow \text{nil}[T]
16 \quad \text{color}[z] \leftarrow \text{RED}
17 \quad \text{RB-INSERT-FIXUP} \(T, z\)
Properties violations

- Property 1 (each node black or red): hold
- Proper 3: (each leaf is black sentinel): hold.
- Property 5: same number of blacks: hold
- Property 2: (root is black), not, if z is root (and colored red).
- Property 4: (the child of a red node must be black), not, if z’s parent is red.
Case 1,2,3: \( p[z] \) is the left child of \( p[p[z]] \).
Correspondingly, there are 3 other cases in which \( p[z] \) is the right child of \( p[p[z]] \).
Figure 13.4  The operation of RB-INSERT-FIXUP. (a) A node $z$ after insertion. Since $z$ and its parent $p[z]$ are both red, a violation of property 4 occurs. Since $z$’s uncle $y$ is red, case 1 in the code can be applied. Nodes are recolored and the pointer $z$ is moved up the tree, resulting in the tree shown in (b). Once again, $z$ and its parent are both red, but $z$’s uncle $y$ is black. Since $z$ is the right child of $p[z]$, case 2 can be applied. A left rotation is performed, and the tree that results is shown in (c). Now $z$ is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in (d), which is a legal red-black tree.
case 1: z’s uncle is red.

Figure 13.5 Case 1 of the procedure RB-INSERT. Property 4 is violated, since z and its parent $p[z]$ are both red. The same action is taken whether (a) z is a right child or (b) z is a left child. Each of the subtrees $\alpha$, $\beta$, $\gamma$, $\delta$, and $\epsilon$ has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The while loop continues with node z’s grandparent $p[p[z]]$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.
Case 2: z’s uncle is black and z is a right child. Case 3: z’s uncle is black and z is a left child

**Figure 13.6** Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent p[z] are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The while loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

What is the running time of RB_INSERT_FIX? And RB_INSERT?
RB-DELETE(T, z)
1. if left[z] = nil[T] or right[z] = nil[T]
2. then y ← z
3. else y ← TREE-SUCCESSOR(z)
4. if left[y] ≠ nil[T]
5. then x ← left[y]
6. else x ← right[y]
7. p[x] ← p[y]
8. if p[y] = nil[T]
9. then root[T] ← x
10. else if y = left[p[y]]
11. then left[p[y]] ← x
12. else right[p[y]] ← x
13. if y ≠ z
14. then key[z] ← key[y]
15. copy y’s satellite data into z
16. if color[y] = BLACK
17. then RB-DELETE-FIXUP(T, x)
18. return y
RB-DELETE-FIXUP($T, x$)

1  while $x \neq root[T]$ and color[$x$] = BLACK
2      do if $x = left[p[x]]$
3          then $w \leftarrow right[p[x]]$
4                if color[$w$] = RED
5                    then color[$w$] $\leftarrow$ BLACK  ▷ Case 1
6                        color[$p[x]$] $\leftarrow$ RED  ▷ Case 1
7                    LEFT-ROTATE($T, p[x]$)  ▷ Case 1
8                        $w \leftarrow right[p[x]]$  ▷ Case 1
9                if color[$left[w]$] = BLACK and color[$right[w]$] = BLACK
10                    then color[$w$] $\leftarrow$ RED  ▷ Case 2
11                        $x \leftarrow p[x]$  ▷ Case 2
12            else if color[$right[w]$] = BLACK
13                    then color[$left[w]$] $\leftarrow$ BLACK  ▷ Case 3
14                        color[$w$] $\leftarrow$ RED  ▷ Case 3
15                    RIGHT-ROTATE($T, w$)  ▷ Case 3
16                        $w \leftarrow right[p[x]]$  ▷ Case 3
17                        color[$w$] $\leftarrow$ color[$p[x]$]  ▷ Case 4
18                        color[$p[x]$] $\leftarrow$ BLACK  ▷ Case 4
19                        color[$right[w]$] $\leftarrow$ BLACK  ▷ Case 4
20                    LEFT-ROTATE($T, p[x]$)  ▷ Case 4
21                        $x \leftarrow root[T]$  ▷ Case 4
22 else (same as then clause with “right” and “left” exchanged)
23      color[$x$] $\leftarrow$ BLACK
Figure 13.7  The cases in the while loop of the procedure RB-DELETE-FIXUP. Darkened nodes have color attributes BLACK, heavily shaded nodes have color attributes RED, and lightly shaded nodes have color attributes represented by c and c′, which may be either RED or BLACK. The letters α, β, . . . , ζ represent arbitrary subtrees. In each case, the configuration on the left is transformed into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. The only case that causes the loop to repeat is case 2. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. (b) In case 2, the extra black represented by the pointer x is moved up the tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the while loop terminates because the new node x is red-and-black, and therefore the value c of its color attribute is RED. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. (d) In case 4, the extra black represented by x can be removed by changing some colors and performing a left rotation (without violating the red-black properties), and the loop terminates.