Why care about advanced implementations?
Same entries, different insertion sequence:

Not good! Would like to keep tree balanced.

Balanced binary tree
- The disadvantage of a binary search tree is that its height can be as large as $N-1$
- This means that the time needed to perform insertion and deletion and many other operations can be $O(N)$ in the worst case
- We want a tree with small height
- A binary tree with $N$ node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree $O(\log N)$
- Such trees are called balanced binary search trees. Examples are 2-3 tree, red-black tree, B-tree.

2-3 Trees
- A node contains one or two key values (called 2-nodes or 3-nodes, respectively)
- Every internal 2-node has two children
- Every internal 3-node has three children
- All leaves are at the same level.

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

Example of 2-3 Tree
2-3 Trees with Ordered Nodes

- leaf node can be either a 2-node or a 3-node

Searching a 2-3 Tree

```
TreeItemType searchItem(TwoThreeTree ttt, KeyType key) {
    // Pre: ttt != NIL
    // Post: return the item with key in the tree ttt if it exists; NIL otherwise.
    if (key is inside ttt’s node) return the item containing key;
    else if (ttt is a leaf) return NIL;
    else if (key < key of first item) return searchItem(firstChild of ttt, key);
    else if (ttt is a 2-node || key < key of second item) return searchItem(secondChild of ttt, key);
    else return searchItem(thirdChild of ttt, key);
}
```

Traversing a 2-3 Tree

```
static void inorder(TwoThreeTree ttt) {
    if (ttt is a leaf) visit the data item of ttt;
    else if (ttt has two data items) {
        inorder(left subtree of ttt);
        visit the first data item;
        inorder(middle subtree of ttt);
        visit the second data item;
        inorder(right subtree of ttt);
    } else {
        inorder(left subtree of ttt);
        visit the data item;
        inorder(right subtree of ttt);
    }
}
```

Inserting Items

Gain: Ease of Keeping the Tree Balanced

```
Gain: Ease of Keeping the Tree Balanced
```

What did we gain?

```
What is the time efficiency of searching for an item?
Because every internal node has at least two children, a tree containing N nodes can have a height of at most log_2(N).
```

Gain:
Ease of Keeping the Tree Balanced

```
Gain: Ease of Keeping the Tree Balanced
```

Inserting Items

```
Inserting Items
```

What is the time efficiency of searching for an item?
Because every internal node has at least two children, a tree containing N nodes can have a height of at most log_2(N).

Let h be the height of 2-3 Tree with N nodes and n be the number of perfect binary tree of height h, then N >= n = 2^h+1 – 1, so h <= log(N+1) – 1.

Search Item

```
Searching a 2-3 Tree
```

```
Inserting Items

Insert 38
1) insert 38 in leaf
2) divide leaf and move middle value up to parent
3) result

Insert 37

Insert 36
insert in leaf
divide leaf and move middle value up to parent

... still inserting 36
divide overcrowded node, move middle value up to parent, attach children to smallest and largest

Result

After Insertion of 35, 34, 33

Inserting so far
Inserting Items

How do we insert 32?

Inserting so far

How do we insert 32?

Final Result

Inserting Items

→ creating a new root if necessary
→ tree grows at the root

New root

Inserting Items

Deleting Items

Delete 70

Deleting Items

Deleting 70: swap 70 with inorder successor (80)

Swap with inorder successor

Swap with inorder successor
Deleting Items

Deleting 70: ... get rid of 70

Delete value from leaf

Merge nodes by deleting empty leaf and moving 80 down

Deleting Items

Result

Delete 100

Delete value from leaf

 Doesn’t work

Redistribute

Deleting Items

Deleting 80
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 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

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

Search keys < \( S \)
Search keys > \( S \) and < \( M \)
Search keys > \( M \) and < \( L \)

2-3-4 Tree Example

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"
  \( \rightarrow \) 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 ...

2-3-4 Tree: Insertion

Inserting 50, 40 ...

2-3-4 Tree: Insertion

Inserting 70 ...

2-3-4 Tree: Insertion

Inserting 80, 15 ...

2-3-4 Tree: Insertion

Inserting 90 ...

2-3-4 Tree: Insertion

Inserting 100 ...
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

2-3-4 Tree: Insertion Procedure

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

2-3-4 Tree: Deletion

Deletion procedure:
- similar to deletion in 2-3 trees
- items are deleted at the leafs
  - 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)

Red-Black Tree

- A red-black tree is a binary search such that each node has a color of either red or black.
  - The root is black.
  - Every path from a node to a leaf contains the same number of black nodes.
  - If a node is red then its parent must be black.

Class BinaryNode

KeyType: Key

Boolean: isRed

BinaryNode: LeftChild

BinaryNode: RightChild

BinaryNode: parent

Constructor(KeyType: key)

Key = key

isRed = true

End Constructor

End Class
The root is black. The parent of any red node must be black.

Maintain the Red Black Properties in a Tree

- Insertions
- Must maintain rules of Red Black Tree.
- New Node always a leaf
  - can’t be black or we will violate rule of the same # of blacks along any path
  - therefore the new leaf must be red
  - if parent is black, done (trivial case)
  - if parent red, things get interesting because a red leaf with a red parent violates no double red rule.

Algorithm: Insertion

A red-black tree is a particular binary search tree, so create a new node as red and insert it as in normal search tree.

What property may be violated? The parent of a red node must be black.

Example of Inserting Sorted Numbers

- 1 2 3 4 5 6 7 8 9 10

Insert 1. A leaf so red. Realize it is root so recolor to black.

Example:

1

Insert 2

1

make 2 red. Parent is black so done.

Example:

1

Insert 3

1

Insert 3. Parent is red. Parent's sibling is black (null) 3 is outside relative to grandparent. Rotate parent and grandparent

Example:

1

2

3
On way down see 2 with 2 red children. Recolor 2 red and children black. Realize 2 is root so color back to black.

When adding 4 parent is black so done.

5's parent is red. Parent's sibling is black (null). 5 is outside relative to grandparent (3) so rotate parent and grandparent then recolor.

On way down see 4 with 2 red children. Make 4 red and children black. 4's parent is black so no problem.

6's parent is black so done.

7's parent is red. Parent's sibling is black (null). 7 is outside relative to grandparent (5) so rotate parent and grandparent then recolor.
Finish insert of 7

On way down see 6 with 2 red children. Make 6 red and children black. This creates a problem because 6's parent, 4, is also red. Must perform rotation.

Still Inserting 8

Still Inserting 8

Recolored now need to rotate

Finish inserting 8

Recolored now need to rotate

Insert 9

On way down see 4 has two red children so recolor 4 red and children black. Realize 4 is the root so recolor black

Finish Inserting 9

After rotations and recoloring
Algorithm: Insertion
We have detected a need for balance when X is red and his parent too.
• If X has a red uncle: color the parent and uncle black, and grandparent red. Then replace X by grandparent to see if new X’s parent is red.
• If X is a left child and has a black uncle: color the parent black and the grandparent red, then rightRotate(X.parent.parent).

Algorithm: Insertion
We have detected a need for balance when X is red and its parent, too.
• If X has a red uncle: color the parent and uncle black, and grandparent red. Then replace X by grandparent to see if new X’s parent is red.
• If X is a left child and has a black uncle: color the parent black and the grandparent red, then rightRotate(X.parent.parent).

rotateRight(G)
Relative to G, X is at left-left positions. rotateRight(G) will exchange of roles between G and P, so P becomes G’s parent. Also must recolor P and G.

After rotateRight(G)
Apparent rule violation? rotateLeft(G) will handle the case when X is at right-right position relative to G.
Algorithm: Insertion

We have detected a need for balance when X is red and his parent too.

- If X has a red uncle: colour the parent and uncle black, and grandparent red.
  Then replace X by grandparent to see if X’s parent is red.
- If X is a left child and has a black uncle, then rotateRight(X.parent.parent)
- If X is a right child and has a black uncle, then rotateLeft(X.parent) and
  rotateRight(X) and grayscale_Rotation(X)

Double Rotation

- What if X is at left right relative to G?
  - a single rotation will not work
- Must perform a double rotation
  - rotate X and P
  - rotate X and G

Properties of Red Black Trees

- If a Red node has any children, it must have two children and they must be Black.
  (Why?)
- If a Black node has only one child that child must be a Red leaf. (Why?)
- Due to the rules there are limits on how unbalanced a Red Black tree may become.
Red-Black Tree vs 2-3-4 Tree

- binary-search-tree representation of 2-3-4 tree
- 3- and 4-nodes are represented by equivalent binary trees
- Each 2-3-4 node generates exactly one black node (on the top), and zero red node for 2-nodes, one red for 3-nodes, and two red ones for 4-nodes.

Red-Black Representation of 4-node

Red-Black Representation of 3-node

Red-Black Tree vs 2-3-4 Tree

- Let $h$ be the height of a 2-3-4 tree and $H$ be the height of the corresponding red-black tree.
- $h = O(\log n)$.
- $h$ is the number of black nodes minus one from the root to any leaf in the corresponding red-black tree.
- $H \leq 2h + 1$, because red nodes cannot be more than black nodes on any path from the root to a leaf.
- Hence $H = O(\log n)$.

Multiway Search Trees

A multiway search tree of order $m$, or an $m$-way search tree, is an $m$-ary tree in which:
1. Each node has up to $m$ children and $m-1$ keys
2. The keys in each node are in ascending order
3. The keys in the first $i$ children are smaller than the $i$th key
4. The keys in the last $m-i$ children are larger than the $i$th key
A 5-Way Search Tree

B-tree
- B-tree is a generalization of 2-3-4 tree with a large number of branches.
- A B-tree of order \( m \) is an \( m \)-way search tree (i.e., a tree where each node may have up to \( m \) children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children.
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least \( \lceil m / 2 \rceil \) children
  4. the root is either a leaf node, or it has from two to \( m \) children
  5. a leaf node contains no more than \( m - 1 \) keys
- The number \( m \) is always odd

An example B-Tree

A B-Tree of Order 1001

B-Tree Insertion Case 1:
A key is placed in a leaf that still has some room

Insert 7
Shift keys to preserve ordering & insert new key.

B-Tree Insertion Case 2:
A key is placed in a leaf that is full

Insert 8
Split the leaf, creating a new leaf, and move half the keys from full leaf to new leaf.
B-Tree Insertion: Case 2

Insert 8

Move median key to parent, and add pointer to new leaf in parent.

B-Tree Insertion: Case 3

The root is full and must be split

Insert 15

In this case, a new node must be created at each level, plus a new root. This split results in an increase in the height of the tree.

B-Tree Insertion: Case 3

The root is full and must be split

Insert 15

Move 12 & 16 up

B+-Tree

A B+-Tree has all keys, with attached records, at the leaf level. Search keys, without attached records, are duplicated at upper levels. A B+-tree also has links between the leaves.

Application: Web Search Engine

A web crawler program gathers information about web pages and stores it in a database for later retrieval by keyword by a search engine such as Google.

- **Search Engine Task:** Given a keyword, return the list of web pages containing the keyword.
- **Assumptions:**
  - The list of keywords can fit in internal memory, but the list of webpages (urls) for each keyword (potentially millions) cannot.
  - Query could be for single or multiple keywords, in which pages contain all of the keywords, but pages are not ranked.

What data structures should be used?
Summary

- 2-3 trees, 2-3-4 trees, red-black trees, and B-tree are all balanced trees, with $O(\log(N))$ heights.
- 2-3-4 trees and red-black trees have one-to-one correspondence.
- Red-black trees are special binary search trees with simple data structure.
- B-trees with large number of branches have small height and are suitable for storing large data sets on slow disks.