CMSC420 Advanced Data Structures

Michael Li

May 24, 2020

Contents

1 Lists

```
init() => initializes list
get(i) => returns element at index i
set(i, x) => sets ith element to x
length() => returns number of elements in the list
insert(i, x) => insert x prior to element a_{i} (shifts indices after)
delete(i) => deletes ith element (shift indices after)
```

Sequential Allocation (Array): when array is full, increase its size but a constant factor (e.g. 2). Amortized array operations still O(1)

Linked Allocation (Linked List)

Stack(push, pop): on on end of the list
Queue(enqueue, dequeue): insert at tail (end) and remove from head (start)
Deque(combo stack and queue): can isnert and remove from either ends of list
Multilist: multiple lists combined 1 aggregate structure (e.g. ArrayList)

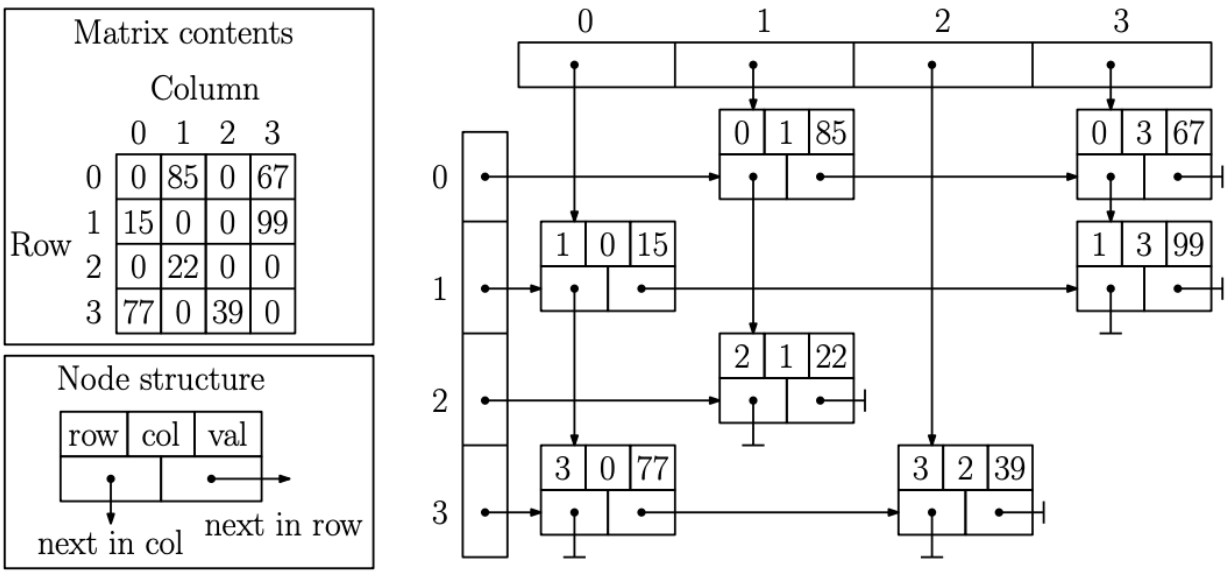


Fig. 2: Sparse matrix representation using a multilist structure.

Sparse Matrix: create 2n linked lists for each row and col
Each entry stores a row index, col index, value, next row ptr, and next col ptr

2 Trees

Free Tree: connected, undirected graph with no cycles (like MST)
Root Tree: each non-leaf node has ≥ 1 children and a single parent (except root)
Aborescence = out-tree Anti-arborescence = in-tree
Depth = max # of edges of path from root to a node

One way to represent tree is to have a pointer to first child and then a pointer to next sibling

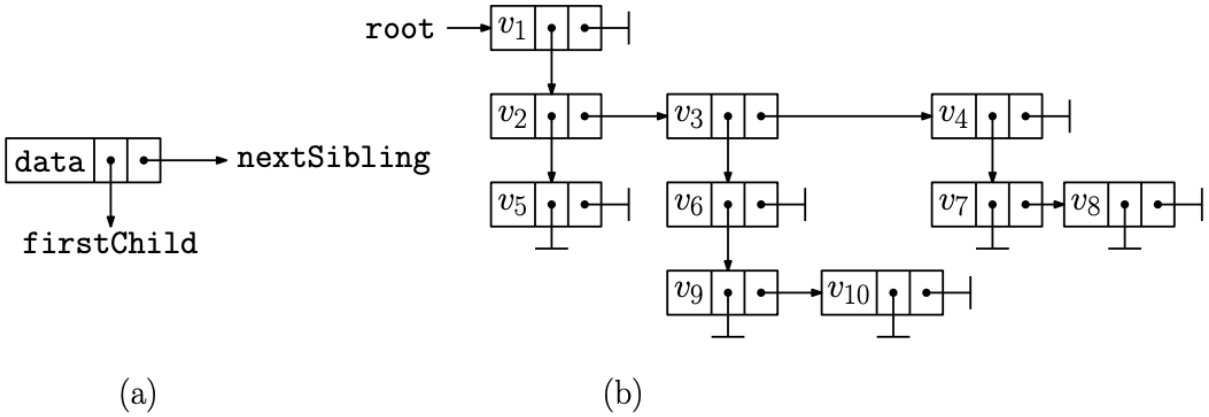


Fig. 3: Standard (binary) representation of rooted trees.

Binary Tree: rooted, ordered tree where each non-leaf node has 2 possible children (left, right)
Full Tree: All nodes either have 0 children or 2 children
Can make full binary tree by extending tree by adding external nodes to replace all empty subtrees

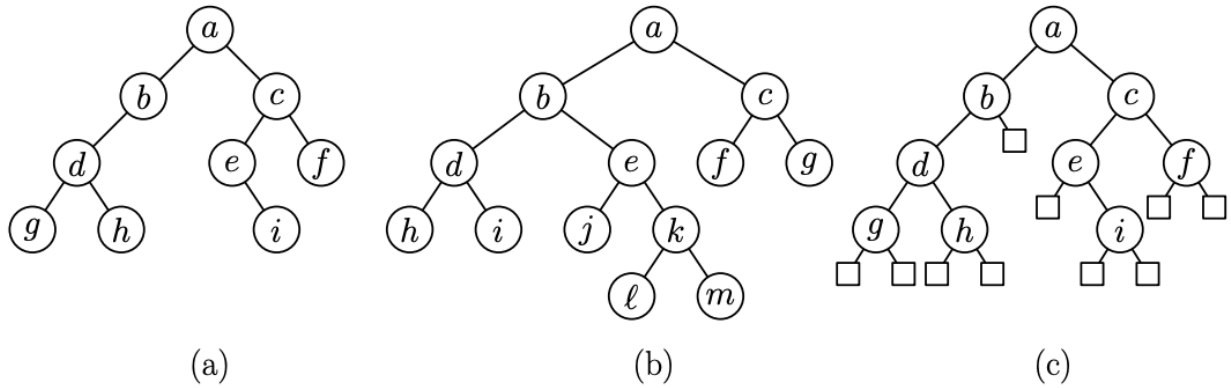


Fig. 4: Binary trees: (a) standard definition, (b) full binary tree, (c) extended binary tree.

```
class BinaryTreeNode<E> {
  private E entry;
  private BinaryTreeNode<E> left;
  private BinaryTreeNode<E> right;
  ...
}
```

In-order traversal: left, root, right
Pre-order traversal: root, left, right
Post-order traversal: left, right, root

If there are n internal nodes in an extended tree, there are n+1 external nodes

Proof by induction: Extended tree binary tree with n internal nodes has n+1 external nodes has 2n+1 total nodes

Let x(n) = number of external nodes given n internal nodes and prove x(n) = n + 1

Base Case x(0) = 1 a tree with no internal nodes has 1 external node

IH: Assume x(i) = i + 1 for all i $\leq$ n - 1

IS: let n_L and n_R be the number of nodes in Left and Right subtrees

x(n) = $(n_L + 1) + (n_R + 1) = (1 + n_L + n_R) + 1$ = n + 1 external nodes

so n + 1 (external) + n (internal) = 2n + 1

Moreover, about 1/2 of nodes of extended Binary Tree are leaf nodes

Threaded Binary Tree: Give null pointers information about where to traverse next

If left-child = null then stores reference to node's inorder predecessor

If right-child = null then stores references to node's inorder successor

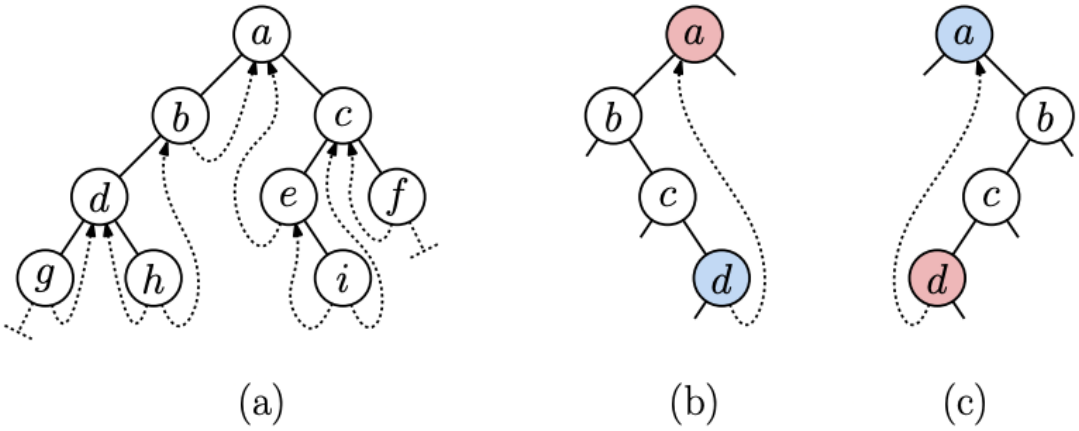


Fig. 6: A Threaded Tree.

```
BinaryTreeNode inOrderSuccessor(BinaryTreeNode v) {
BinaryTreeNode u = v.right;
if(v.right.isThread) return u;
while(!u.left.isThread) u = u.left;
return u;
}
```

if v's right-child is a thread, then we follow thread.

Otherwise go through v's right child and iterate through left-child links

Complete Binary Tree: represented using sequential allocation (array) because no space is wasted

number of nodes is inbetween 2^h and $2^{h+1} - 1$

```
leftChild(i): if(2i <= n) then 2i else null;
rightChild(i): if (2i + 1 <= n) then 2i + 1 else null;
parent(i): if (i >= 2) then [i/2] else null;
```

3 Dictionaries

```
void insert(Key x, Value v) => if key exists, exception is thrown
void delete(Key x) => if key does not exist, exception thrown
Value find(Key x) => return value associated with key or null if not found
```

Array representation:

Unsorted array has O(n) search and delete, O(1) insert although we need O(n) to check for duplicates

Sorted Array has O(logn) search and O(n) insertion and deletion

Binary Search Tree Representation (left < root < right):

```
//Recursive
Value find(Key x, BinaryNode p) {
  if (p == null) return null;
  else if (x < p.key) return find(x, p.left);
  else if (x > p.key) return find(x, p.right);
  else return p.val;
}

//Iterative
Value find(Key x) {
  BinaryNode p = root;
  while(p != null) {
    if (x < p.key) p = p.left;
    else if (x > p.key) p = p.right;
    else return p.value;
  }
  return null;
}
```

O(n) search for degenerate tree, O(logn) search for balanced tree

Can use extended BST to give info that target key is inbetween inorder predecessor and inorder successor

Insert: search for key and if found throw exception else we hit a null and insert there

```
BinaryNode insert(Key x, Value v, BinaryNode p) {
  if (p == null) p = new BinaryNode(x, v, null, null);
  else if (x < p.key) p.left = insert(x, v, p.left);
  else if (x > p.key) p.right = insert(x, v, p.right);
  else throw DuplicateKeyException;
  return p;
}
```

Either tree is empty so return new node or we return the root of the original tree with the added node

O(n) insert for degenerate tree, O(logn) insert for balanced tree

Delete find a replace with inorder successor (aka leftmost on right subtree)

```
BinaryNode delete(Key x, BinaryNode p) {
  if (p == null) throw KeyNotFoundException;
  else
    if (x < p.data)
      x.left = delete(x, p.left);
    else if (x > p.data)
      x.right = delete(x, p.right)
    else if (p.left == null || p.right == null)
      if (p.left == null) return p.right;
      else return p.left;
    else
      r = findReplacement(p);
```

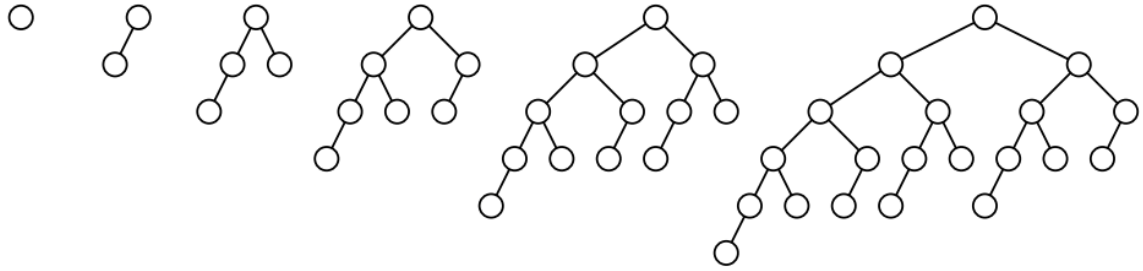
```
      //copy r's contents to p
      p.right = delete(r.key, p.right);
}

BinaryNode findReplacement(BinaryNode p) {
  BinaryNode r = p.right;
  while(r.left != null) r = r.left;
  return ;
}
```

O(n) deletion for degenerate tree, O(logn) deletion for balanced tree height of BST on average will be ln(n)
Proof: for i = 2 to n, insert elements into BST and look at depth of left most node (min value)
chance that a number is the min is $\frac{1}{i}$ so Expected Height is $\sum_{i=2}^{n} \frac{1}{i} \approx \ln(n)$

4 AVL Trees

Balance Condition: For every node in tree, absolute difference between heights of left and right subtrees is at most 1
Worst case height can be shown to be O(logn) using Fibonacci sequence
$F_h \approx \varphi^h \sqrt{5}$ where $\varphi = (1 + \sqrt{5})/2$
let N(h) denote minimum number of nodes in any AVL tree of height h.
N(0) = 1, N(1) = 2, N(h) = $1 + N(h_L) + N(h_R) = 1 + N(h-1) = N(h-2)$
if a given node has height h, one of its subtrees must have height h - 1, and to make it have min # of nodes, the other subtree has height h-2
Now $N(h) = n \geq c\varphi^h \rightarrow h \leq log_\varphi n \rightarrow O(logn)$
Also find method using AVL is O(logn)

Height:	0	1	2	3	4	5
# Nodes:	1	2	4	7	12	20
# Leaves	1	1	2	3	5	8



Rotations are used to main tree's balance by modifying relation between two nodes but preserving the tree's inorder properties

```
BinaryNode rotateRight(BinaryNode p) {
  BinaryNode q = p.left;
  p.left = q.right;
  q.right = p;
  return q;  // q is now root
}
Binary Node rotateLeft(Binary Node p) {
  BinaryNode q = p.right;
  p.right = q.left;
  q.left = p;
  return q;  // q is now root
}
```

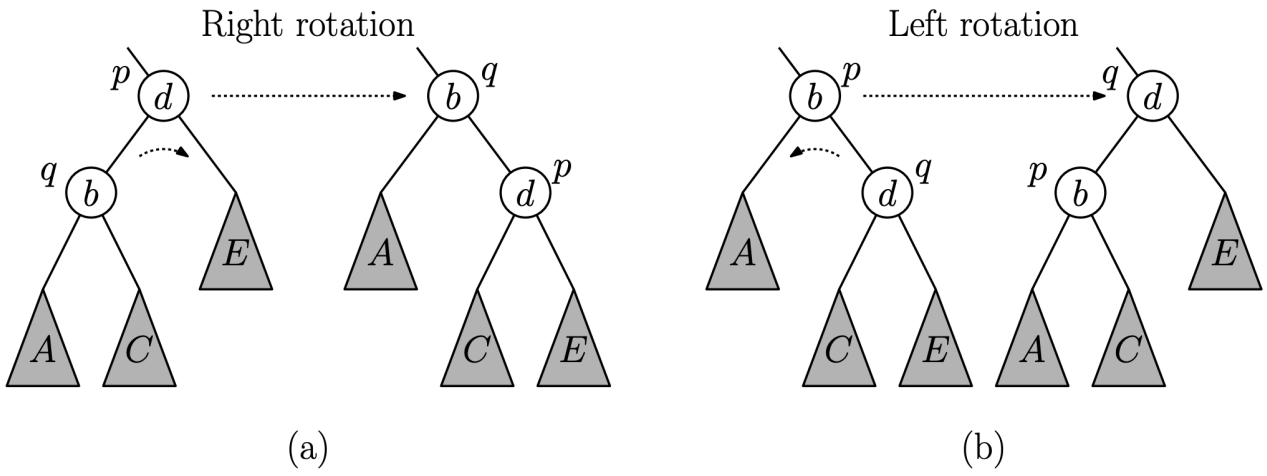


Fig. 3: (Single) Rotations. (Triangles denote subtrees, which may be null.)

Single rotations work when the imbalance occurs on the outer edges of the tree. Need to use double rotations LR or RL to balance inner trees

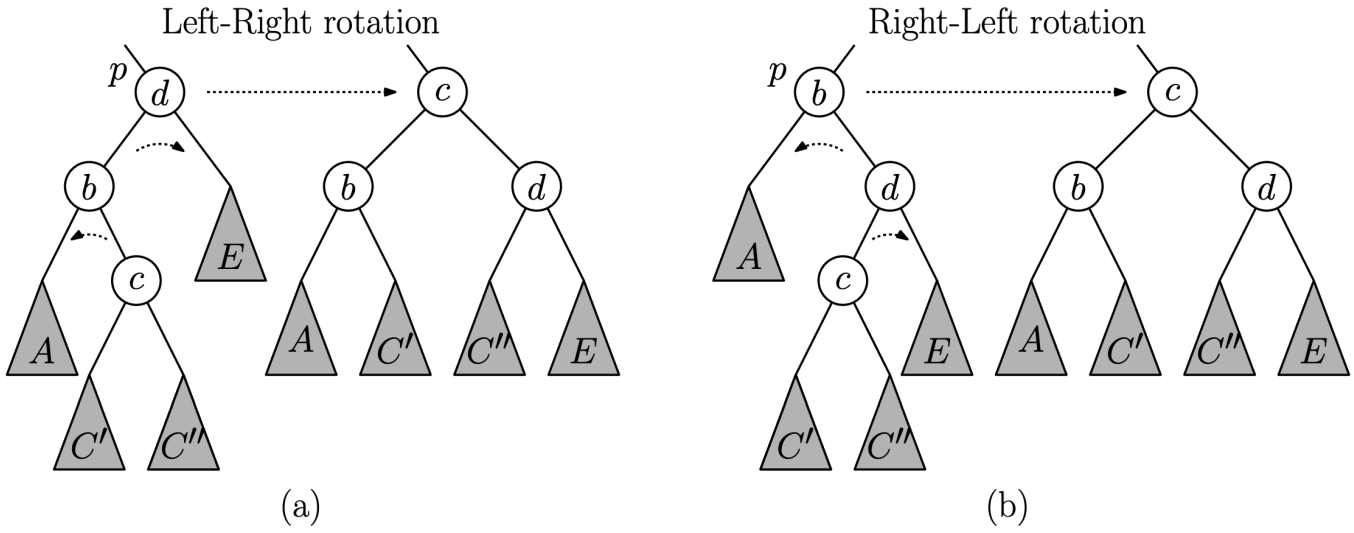


Fig. 4: Double rotations (`rotateLeftRight(p)` and `RotateRightLeft(p)`).

Insertion works similar to BST except we update the heights of subtrees and apply rotations to maintain height
When insertion occurs balance factors of ancestors is altered by ± 1
If a node has a balance factor that violates Balance Property:

- Left-Left substree too deep then rotate right
- Right-Right subtree too deep then rotate left
- Left-Right subtree too deep then rotate left-right
- Right-Left subtree too deep then rotate right-left

```
int height(AvlNode p) return p == null ? -1 : p.height;
void updateHeight(AvlNode p) p.height = 1 + max(height(p.left), height(p.right));
int balanceFactor(AvlNode P) return height.(p.right) - height(p.left);
AvlNode rotateRight(AvlNode p) {
  AvlNode q = p.left;
```

```
  p.left = q.right; // swap inner child
  q.right = p;      // bring q above p
  updateHeight(p);
  updateHeight(q);
  return q;         // q replaces p
}
AvlNode rotateLeft(AvlNode p) {... symmetrical to rotateRight ...}
AvlNode rotateLeftRight(AvlNode p) {
  p.left = rotateLeft(p.left);
  return rotateRight(p);
}
AvlNode rotateRightLeft(AvlNode p) {... symmetrical to rotateLeftRight ...}
AvlNode insert(Key x, Value v, AvlNode p) {
  if (p == null) p = newAvlNode(x, v, null, null);
  else if (x < p .key) p.left = insert(x, v, p.left);
  else if (x > p.key) p.right = insert(x, v, p.right);
  else throw DuplicateKeyException;
  return rebalance(p);
}
AvlNode rebalance(AvlNode p) {
  if (p == null) return p;
  if (balanceFactor(p) < -1) {
    if (height(p.left.left) >= height(p.left.right)) {//left-left heavy
      p = rotateRight(p);
    } else {                                        //left-right heavy
      p = rotateLeftRight(p);
    }
  }
  else if (balanceFactor(p) > 1) {
    if(height(p.right.right) >= height(p.right.left)) {//right-right heavy
      p = rotateLeft(p);
    } else {                                         //right-left heavy
      p = rotateRightLeft(p);
    }
  }
  updateHeight(p);
  return p;
}
```

Deletion works in a similar manner in that we call normal BST delete and then rotate as necessary. However we need to call rebalance on further ancestors to check balance condition

5 2-3 Trees, Red-Black Trees, AA Trees

5.1 2-3 Trees

All leaves are on the same level
nodes can either be 2-node (normal binary tree) or 3-node (2 keys b,d and 3 branches A, C, E where A < b < X < d < E)

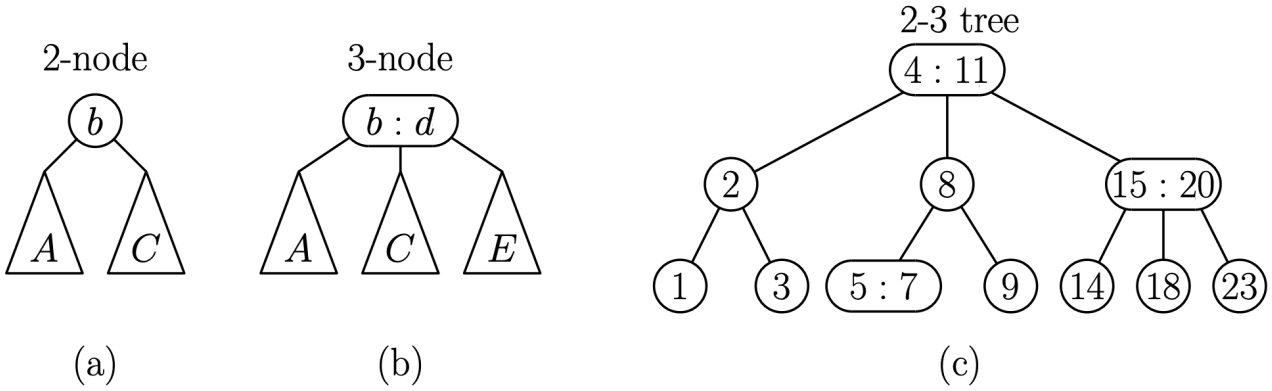


Fig. 1: (a) 2-node, (b) 3-node, and (c) a 2-3 tree.

Recursively defined as:
- empty (null)
- root is 2-node and has two 2-3 subtrees of equal height
- root is 3-node and has three 2-3 subtrees of equal height

Sparsest 2-3 tree is a complete binary tree
Height is O(logn)
Find: recursive descent but when 3-node is reached, compare x with both keys to find which branch to go to
Insertion: search for key and and insert like in a normal tree.
- if parent is a 2-node, now it is a 3-node with a null subtree
- if parent is 3 node then it becomes 4-node and we have to fix it by splitting the 4-node into two 2-nodes and prop the middle term up for recursion
  - Will continue to recurse up until it reaches the root

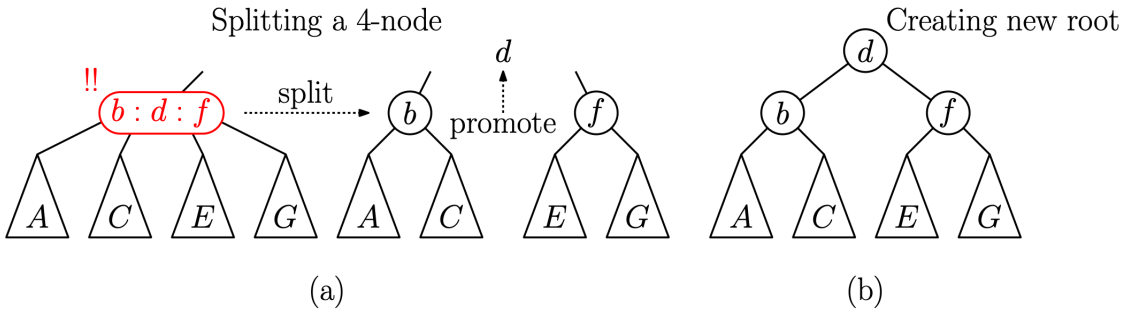


Fig. 2: 2-3 tree insertion: (a) splitting a 4-node into two 2-nodes and (b) creating a new root.

Deletion: find and replace target with inorder successor and then delete the leaf
- If parent of leaf is a 3-node then parent becomes a 2-node and done
- If parent is a 2-node then it becomes a 1-node (0 keys, 1 subtree) so we can do
  - Adoption: if sibling is a 3-node then adopt a key and a subtree so we have two 2-nodes
  - Merge: merge 1-node and 2-node and take a key from parent then recurse up. If root is reached, remove it and make a child the root

5.2 Red-Black Trees

Take AVL 3-node and create a 2-node combo by using d, C, E as the right subtree and b, A for left subtree

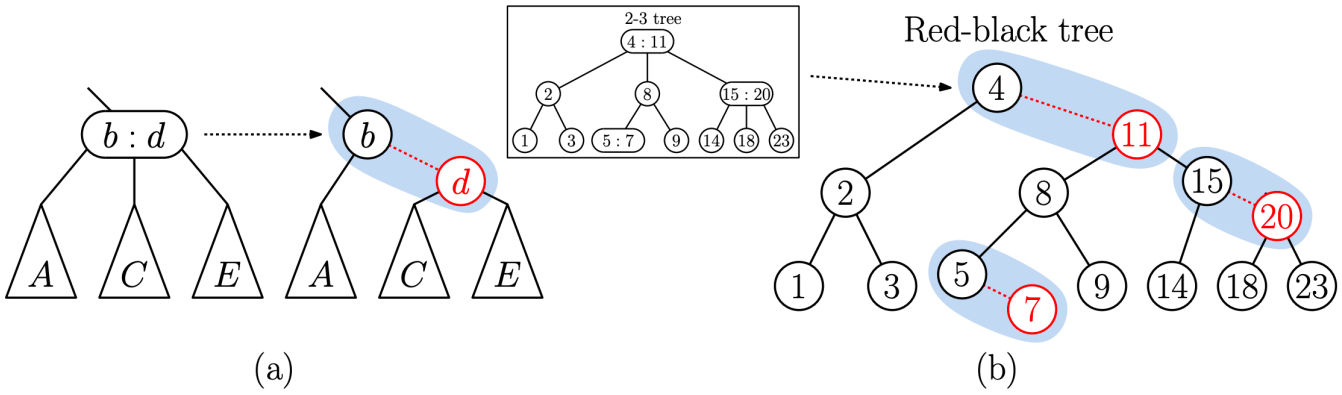


Fig. 6: Representing the 2-3 tree of Fig. 1 as an equivalent binary tree.

Color created right subnode red and all other nodes black, creating a binary search tree
null pointers are labeled black and if a node is red, then both its children are black
Every path from a given node to any of its null descendants contains the same number of black nodes
O(logn) height

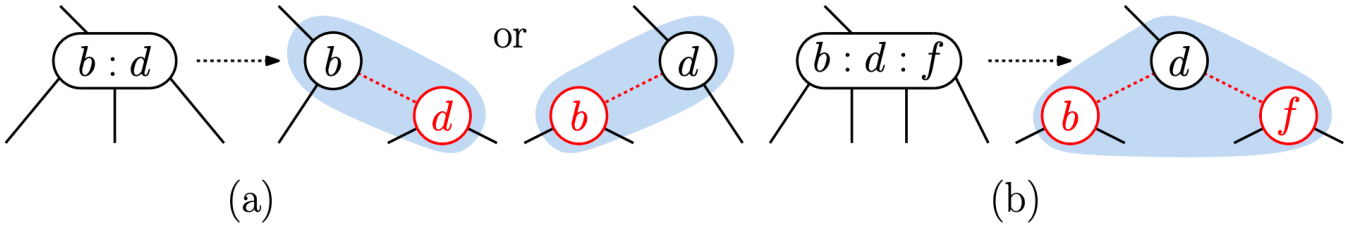


Fig. 7: Color combinations allowed by the red-black tree rules.

Every 2-3 tree corresponds to a red-black tree but converse is not true
Issue with RB tree doesn't distinguish between L and R children so 3-node can be encoded in 2 different ways
Also can't convert a node with 2 red children to 2-3 tree which ends up being a 4-node

5.3 AA Trees

Simplified RB tree where red nodes can only appear as right children of black nodes allowing conversion between 2-3 tree and RB trees

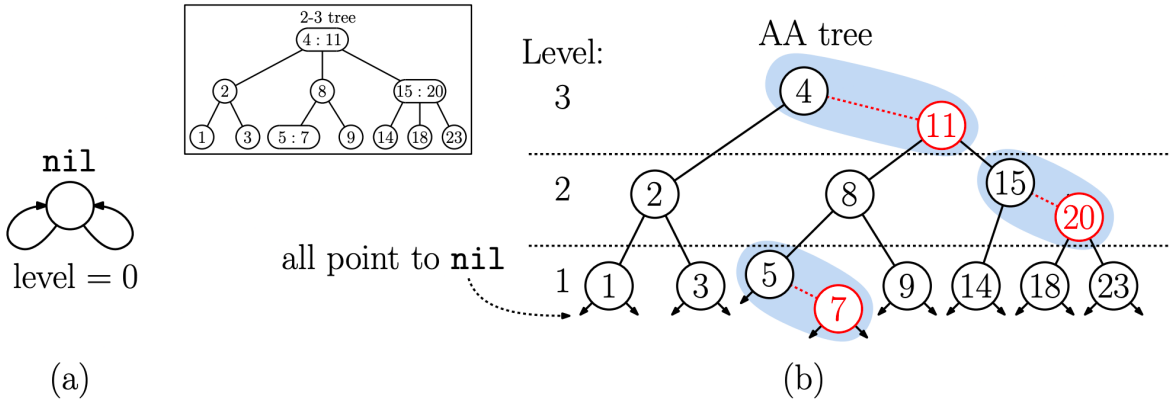


Fig. 8: AA trees: (a) the `nil` sentinel node, (b) the AA tree for the 2-3 tree of Fig. 1.

Edge between red node and the black parent is called a red edge
Implementation of AA trees also uses a sentinel node nil where every null pointer is replaced with a pointer to nil
In this case, nil.left == nil.right == nil so we don't have to keep doing null checks
Implementation of AA doesn't store colors. Instead stores level of associated node in 2-3 tree
nil = level 0
If black, p.level = q.level (child) + 1
If red, then same level as parent. Now can easily test if node is read by comparing with parent level
Find method works exactly the same as it does for BST
Insertion and Deletion require skew(p) and split(p)

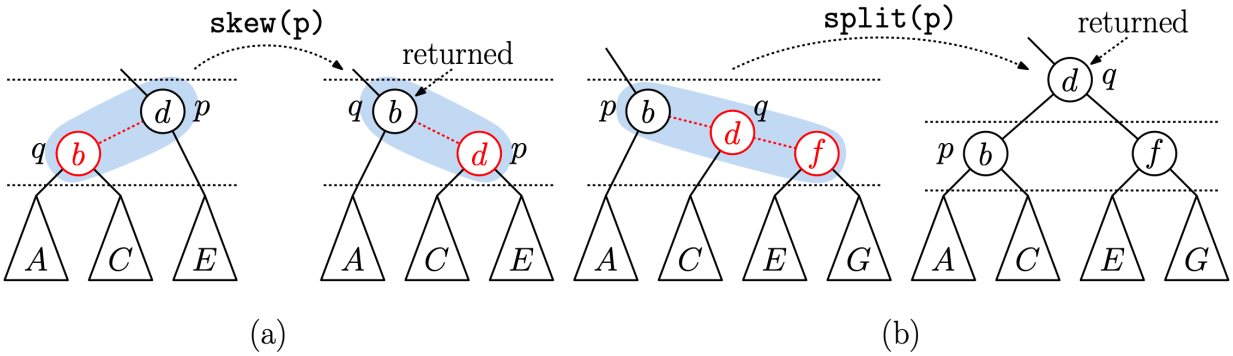


Fig. 9: AA restructuring opertions (a) skew and (b) split. (Afterwards q may be red or black.)

```
AANode skew(AANode p) {
  if (p.left.level == p.level) { // red node to our left?
    AANode q = p.left;          // do right rotation at p
    p.left = q.right;
    q.right = p;
    return q;
  }
  else return p;
}
AANode split(AANode p) {
  if (p.right.right.level == p.level) { //right-right red chain?
```

```
    AANode q = p.right;                // do left rotation at p
    p.right = q.left;
    q.left = p;
    q.level += 1;                      // promote q to higher level
    return q;
  }
  else return p;
}
```

skew(p) if p is black and has a red left child, rotate right

split(p) if p is black and has right-right chain, do a left rotation & promote first red child to next level

Insertion: insert node like in BST except treat it as a red nodethen work back up tree restructuring as we go.

```
AANode insert(Key x, Value v, AANode p) {
  if (p == nil) p = new AANode(x, v, 1, nil, nil) //fell out so create new leaf
  else if (x < p.key) p.left = insert(x, v, p.left);
  else if (x > p.key) p .right = insert(x, v, p.right);
  else throw DuplicateKeyException;
  return split(skew(p));     //restructure (if not needed split and skew return unmodified tree)
}
```

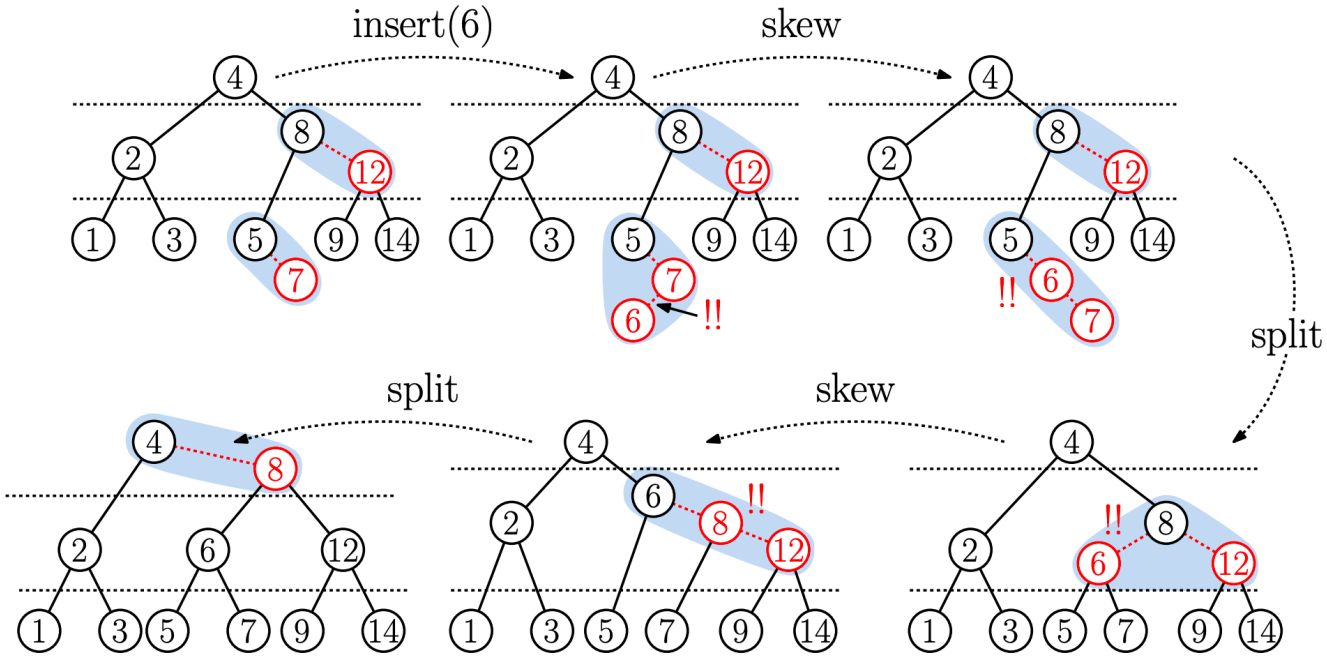


Fig. 11: Example of AA-tree insertion.

If red node inserted as left child then perform skew(p) on parent

If red node inserted as right child of red node, call split(p) on grandparent and then recurse up to fix any issues

Deletion: Replace target node with inorder successor then delete leaf and retrace search path to restructure tree

use updateLevel(p) helper to update level of node p based on children

since every node has at least 1 black node, ideal level for any node is 1 + min of its children

if p is updated and right child is red then we need to update p.right.level = p.level

```
AANode updateLevel(AANode p) {
  int idealLevel = 1 + min(p.left.level, p.right.level);
  if (p.level > idealLevel) {
```

```
    p.level = idealLevel;
    if(p.right.level > idealLevel) p.right.level = idealLevel; //is right child a red node?
  }
}
```

use fixupAfterDelete(p) to make sure any red children are on the right

```
AANode fixupAfterDelete(AANode p) {
  p = updateLevel(p);
  p = skew(p);
  p.right = skew(p.right);
  p.right.right = skew(p.right.right);
  p = split(p);
  p.right = split(p.right);
  return p;
}
```

May need to call up to 3 skew operations (p, p.right, p.right.right) and then 2 splits (p and its right-right grandchild)

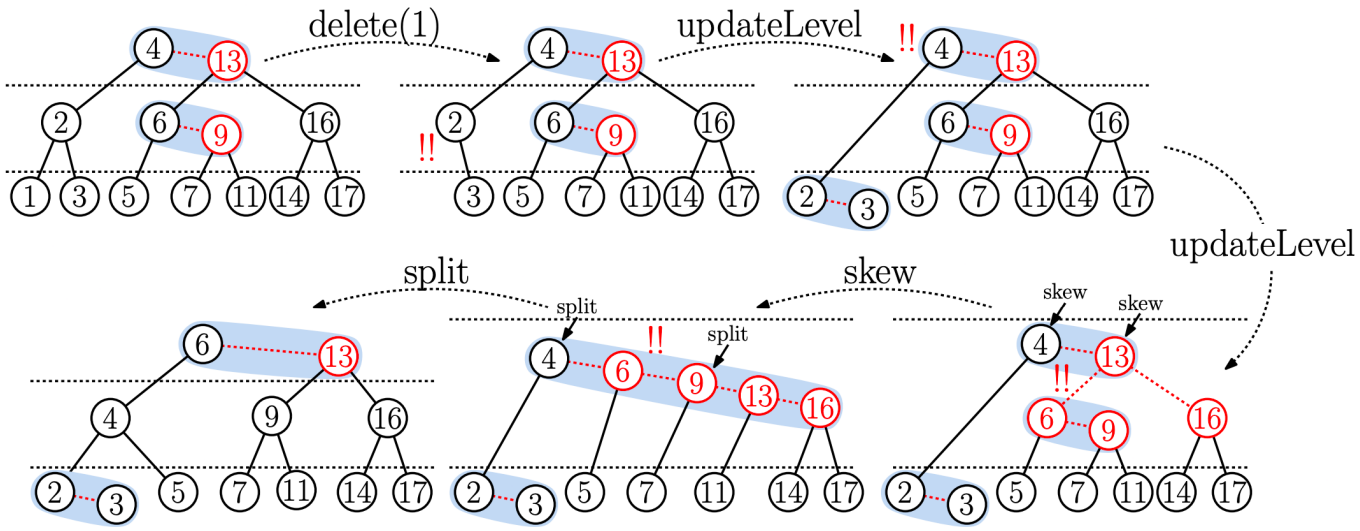


Fig. 12: Example of AA-tree deletion.

```
AANode delete(Key x, AANode p) {
  if (p == nil) throw KeyNotFoundException;
  else {
    if (x < p.key) p.left = delete(x, p.left);
    else if (x > p.key) p.right = delete(x, p.right);
    else {
      if (p.left == nil && p.right == nil) return nil;
      else if (p.left == nil) {       //no left child
        AANode r = inOrderSuccessor(p);
        p.copyContentsFrom(r);
        p.right = delete(r.key, p.right);
      } else {                        //no right child
        AANode r = inOrderPrdecessor(p);
        p.copyContentsFrom(r);
        p.left = delete(r.key, p.left);
      }
```

```
      }
      return fixupAfterDelete(p0:
    }
  }
```