CMSC420 Advanced Data Structures

Michael Li

May 25, 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)

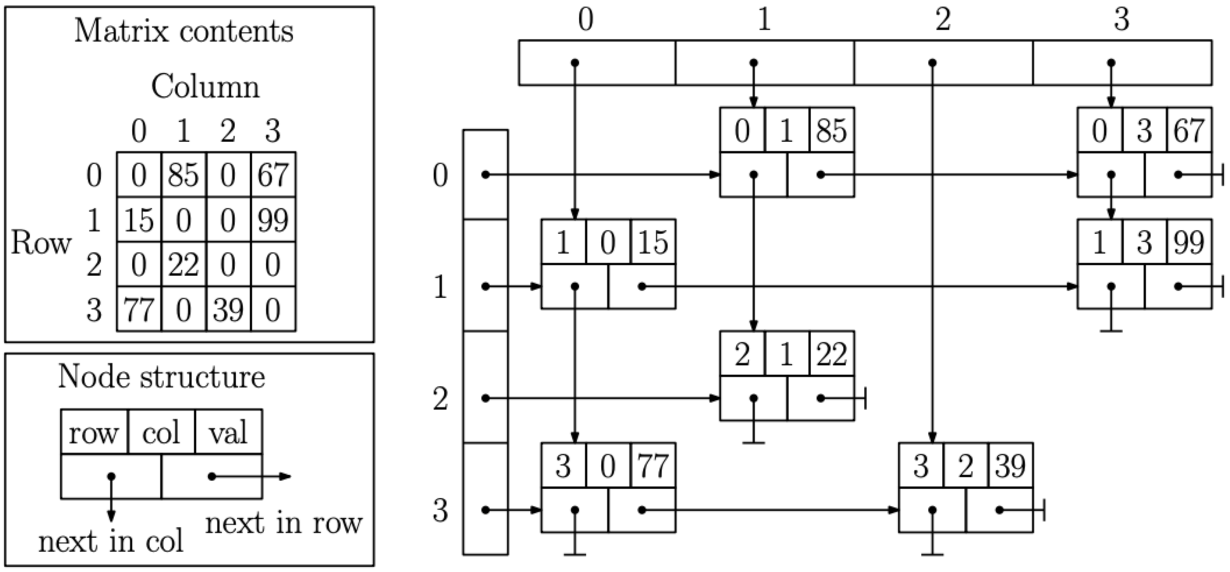


Fig. 2: Sparse matrix representation using a multilist structure.

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

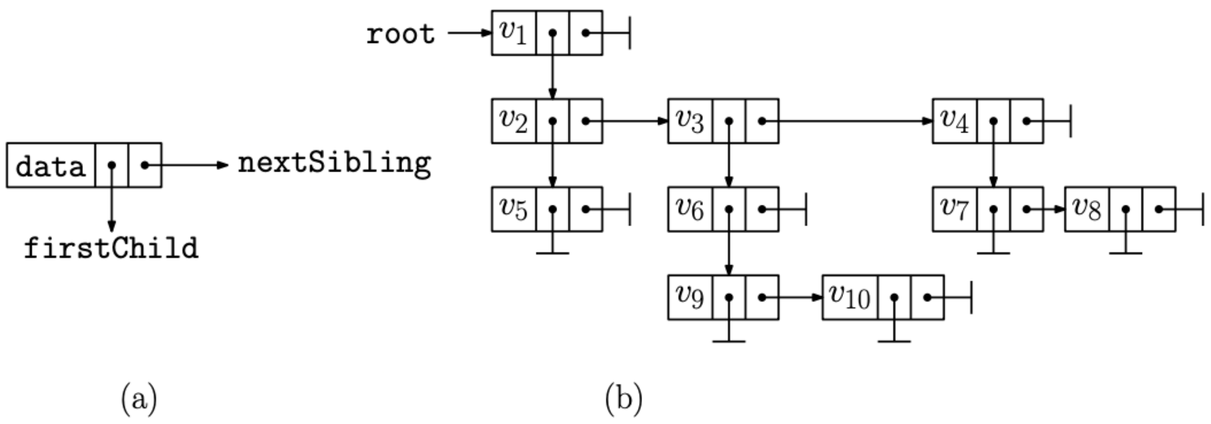


Fig. 3: Standard (binary) representation of rooted trees.

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

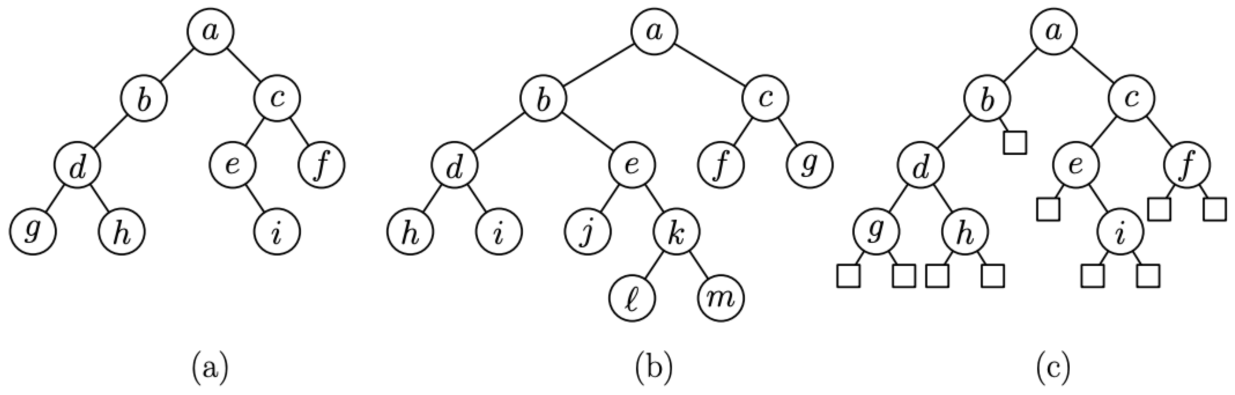


Fig. 4: Binary trees: (a) standard definition, (b) full binary tree, (c) extended binary tree.

```
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

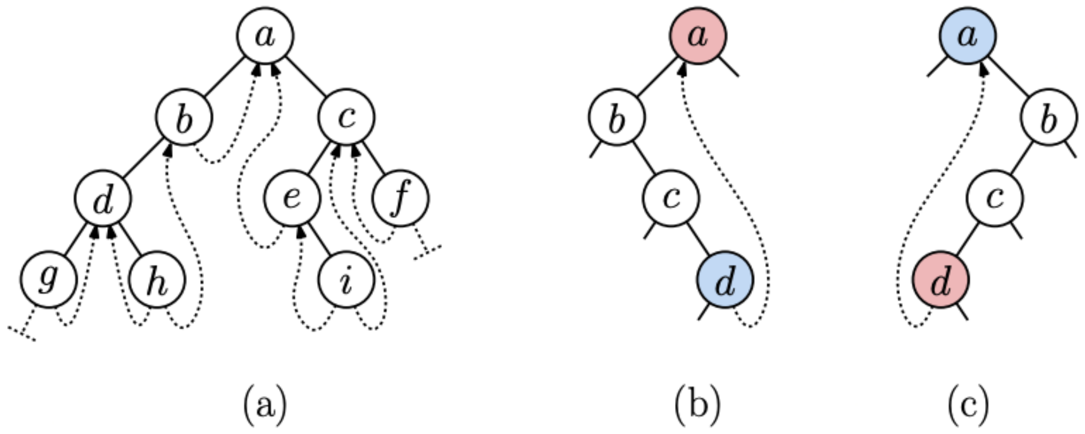


Fig. 6: A Threaded Tree.

```
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)

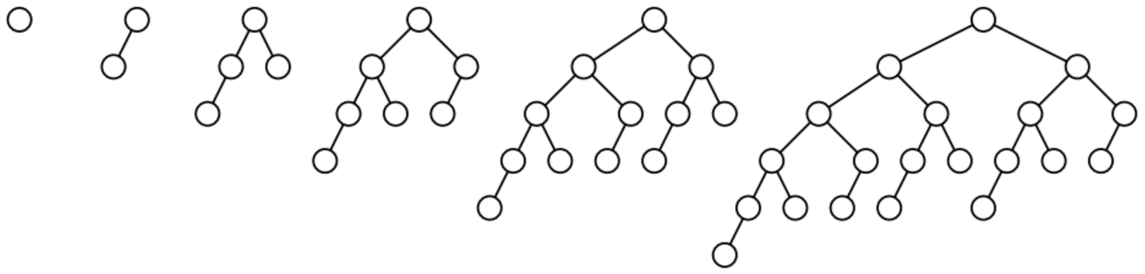
```
BinaryNode delete(Key x, BinaryNode p) {
  if (p == null) throw KeyNotFoundException;
  else
    if (x < p.data)
      x.left = delete(x, p.left);
    else if (x > p.data)
      x.right = delete(x, p.right)
    else if (p.left == null || p.right == null)
      if (p.left == null) return p.right;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

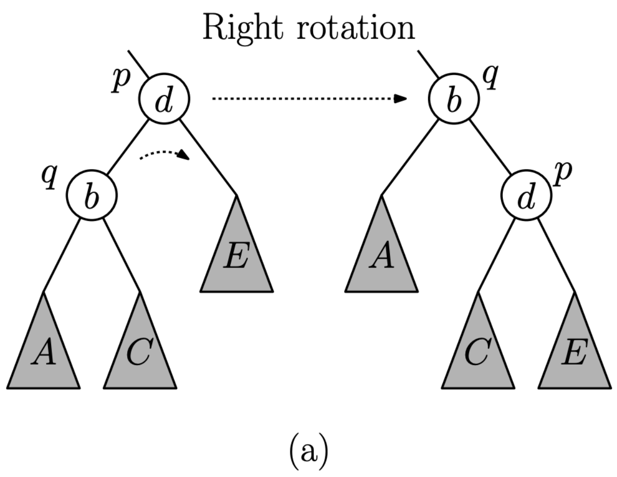


Fig. 3: (Single) Rotations. (Triangles denote subtrees, which may be null.)

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

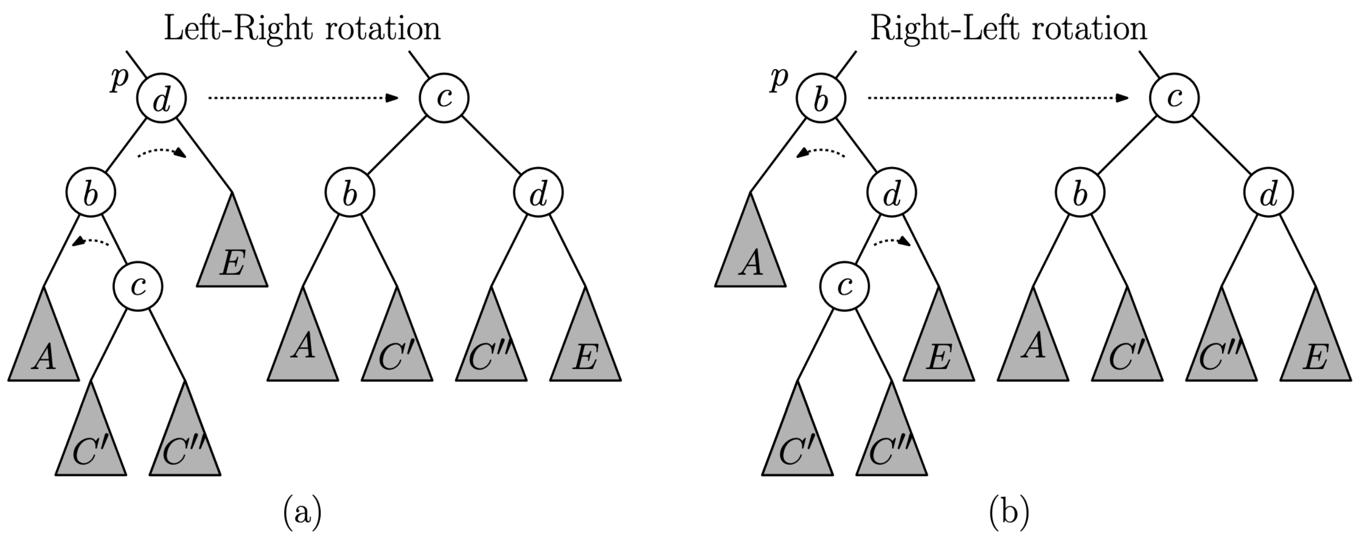


Fig. 4: Double rotations (`rotateLeftRight(p)` and `RotateRightLeft(p)`).

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)

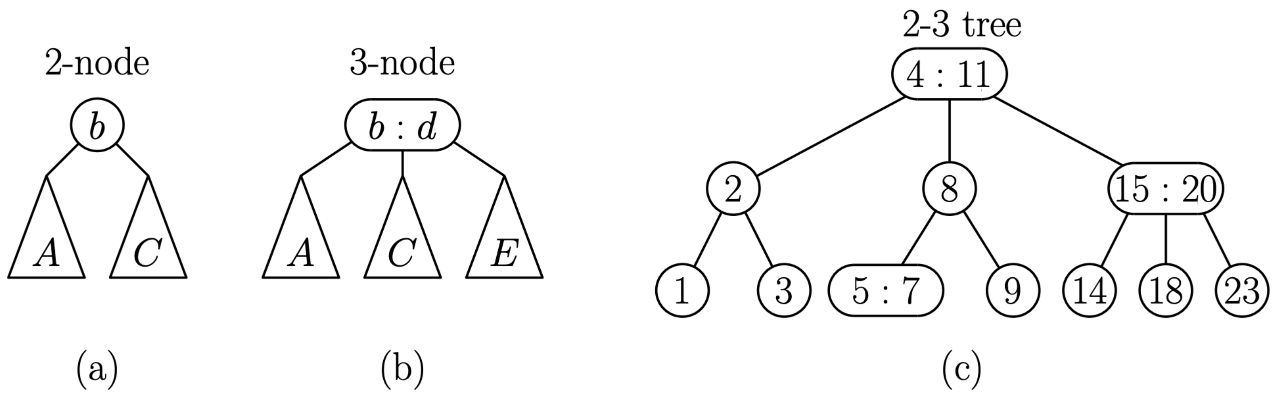


Fig. 1: (a) 2-node, (b) 3-node, and (c) a 2-3 tree.

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

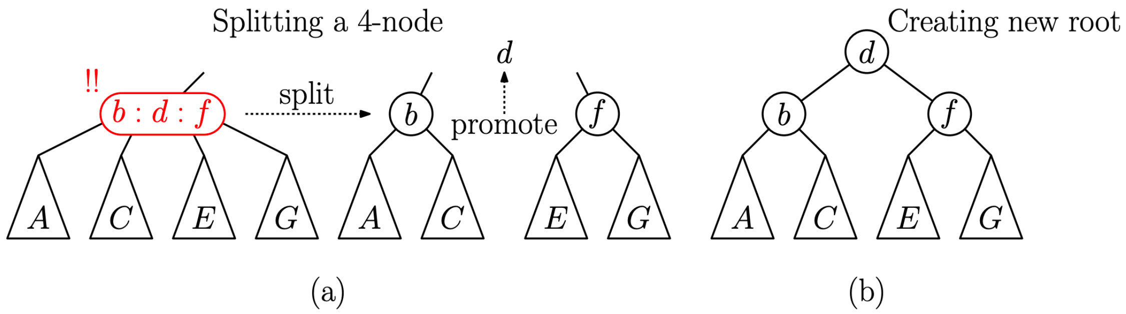


Fig. 2: 2-3 tree insertion: (a) splitting a 4-node into two 2-nodes and (b) creating a new root.

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

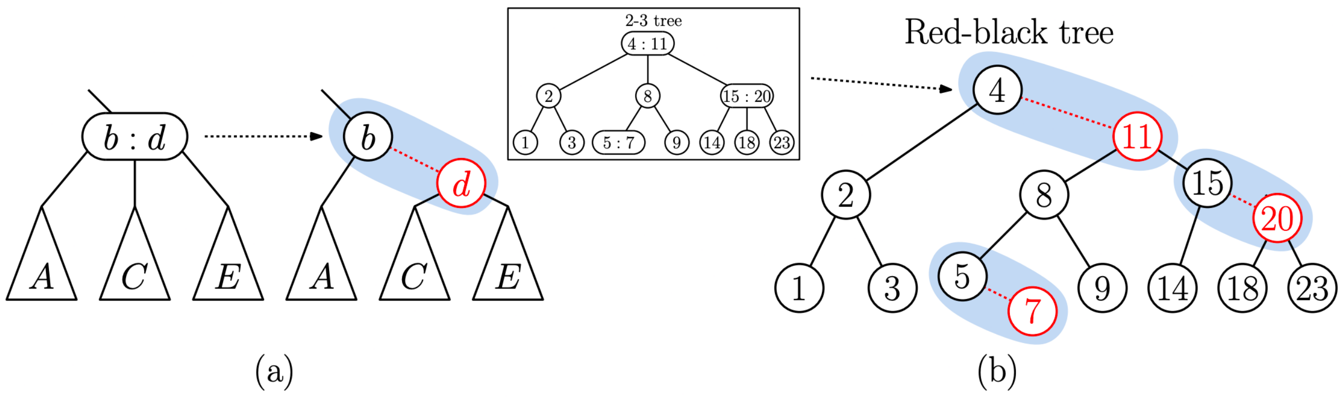


Fig. 6: Representing the 2-3 tree of Fig. 1 as an equivalent binary tree.

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

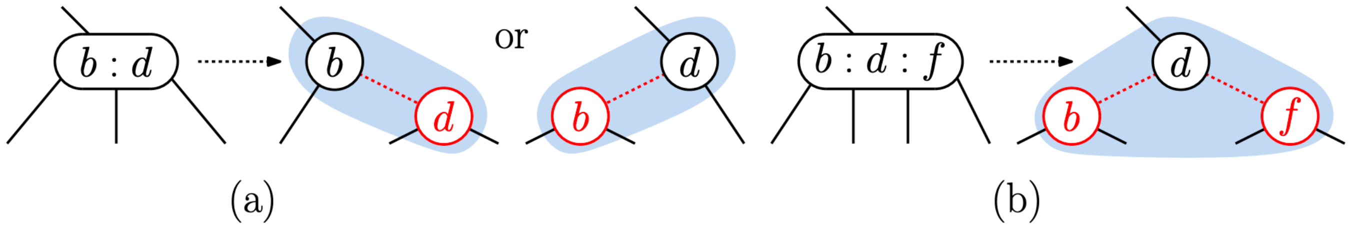


Fig. 7: Color combinations allowed by the red-black tree rules.

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

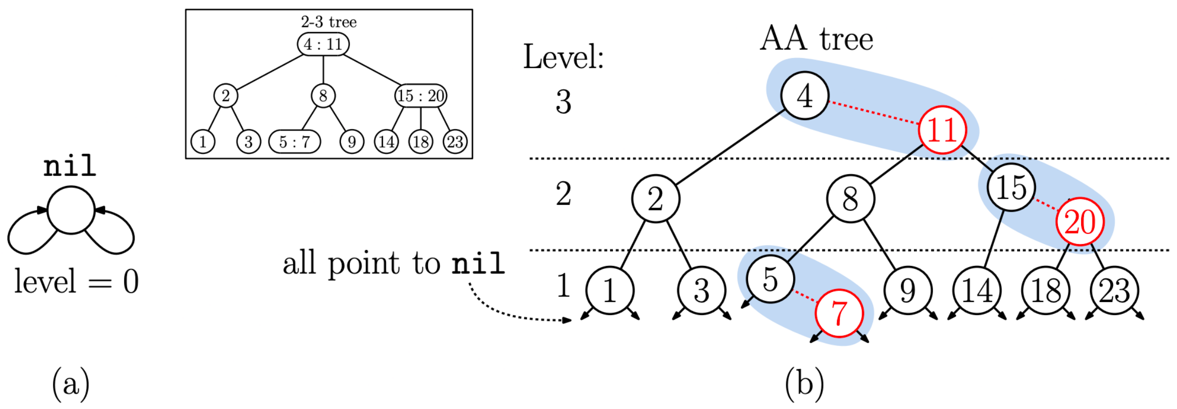


Fig. 8: AA trees: (a) the **nil** sentinel node, (b) the AA tree for the 2-3 tree of Fig. 1.

- 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)

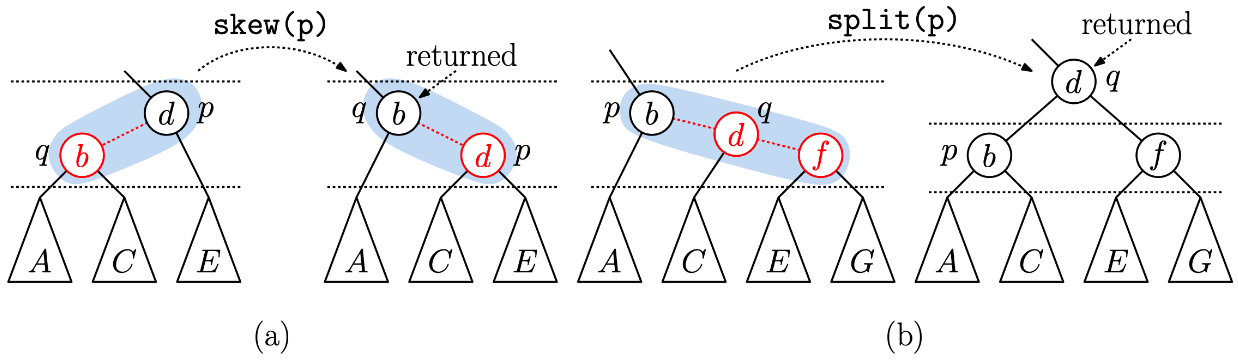


Fig. 9: AA restructuring opertions (a) skew and (b) split. (Afterwards q may be red or black.)

```
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)
  }
```

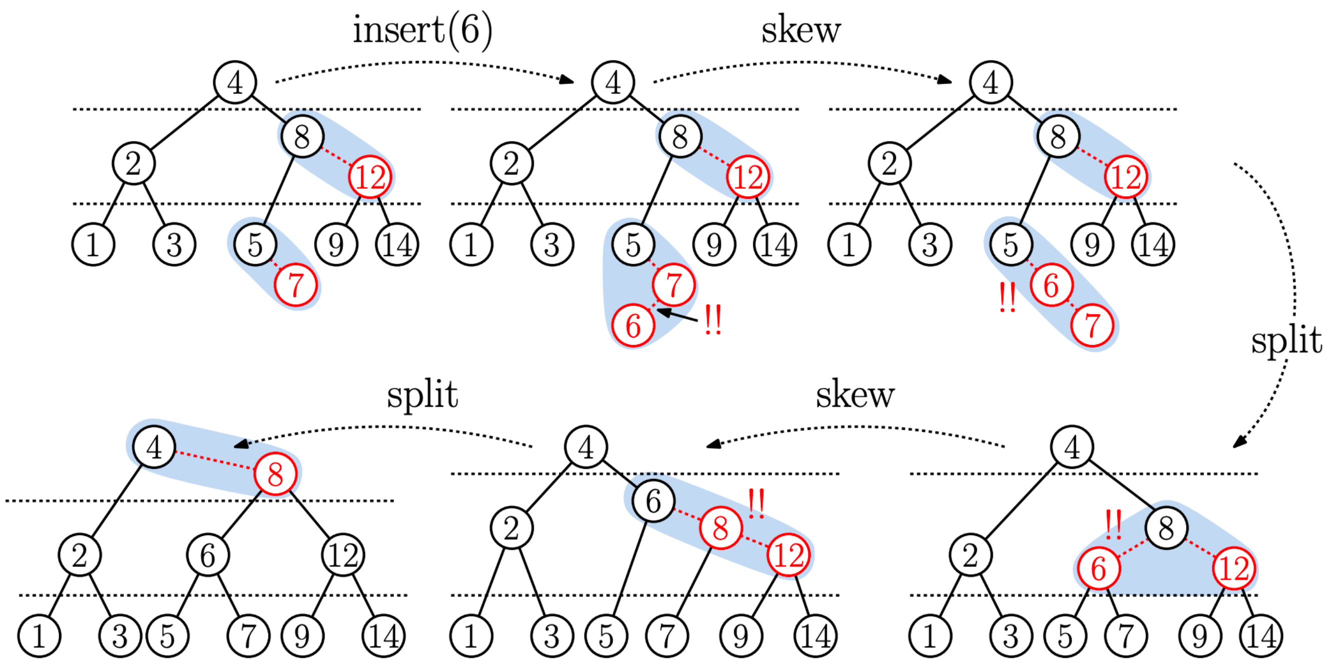


Fig. 11: Example of AA-tree insertion.

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)

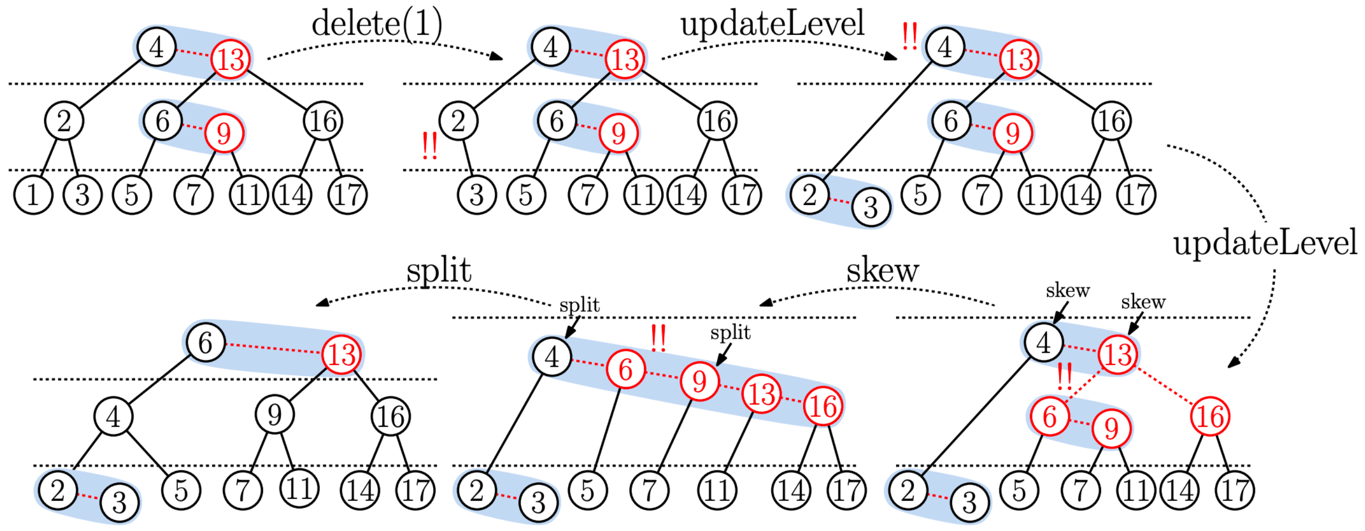


Fig. 12: Example of AA-tree deletion.

```
    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:
      }
    }
```

6 Treaps and Skip Lists

6.1 Treaps

Intution is that if keys are inserted into BST in random order, then height will be ≈ O(log(n))
Insertion: When a node is inserted, assign it a random priority (p.priority) then sort based on this priority
  Now we have to rotate the tree several times to balance it based on p.priority

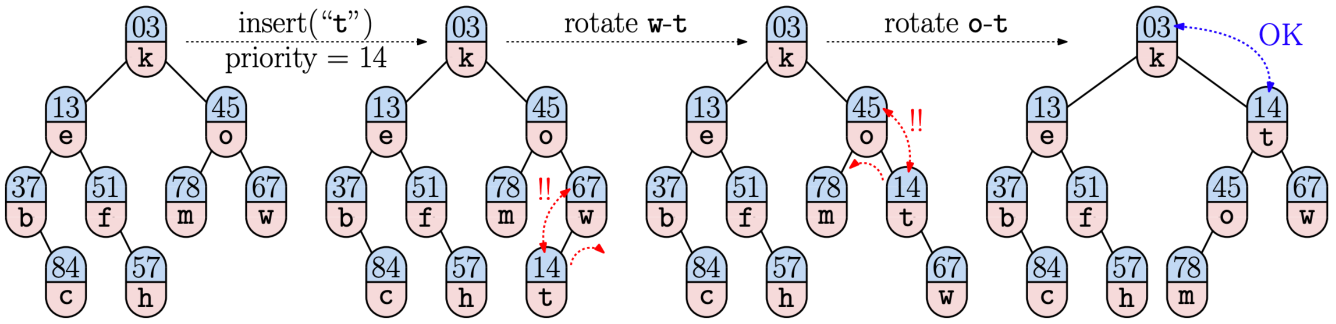


Fig. 2: Treap insertion.

Deletion: 3 cases
  node is leaf just remove it
  node has 1 child then replace node with child
  node has 2 children then its priority to ∞ and apply rotations to sift down to leaf and remove

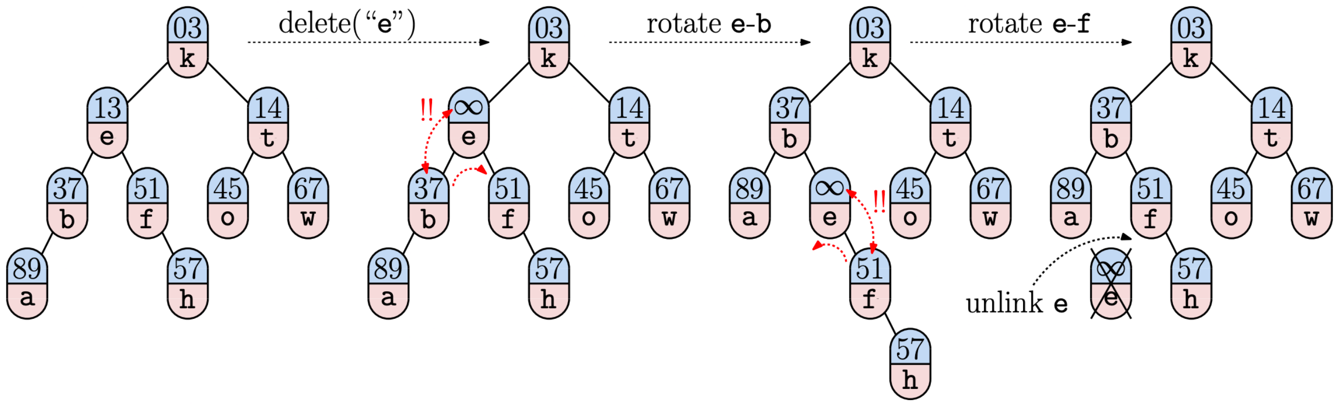


Fig. 3: Treap deletion.

6.2 Skip Lists

Intution is to skip multiple items at a time to speed up searching
Skip Lists made up of multiple levels that are built from
  Taking every other node in the linked list and extending it up to a new linked list with 1/2 as many nodes
  Repeat this extension with 1/2 as many terms until no more terms
  head and tail nodes are always lifted and tail has the key value ∞

process will repeat $\lceil lg(n) \rceil$ times

To find x, start at highest level of head then scan linearly at level i until we are about to jump to a key value > x

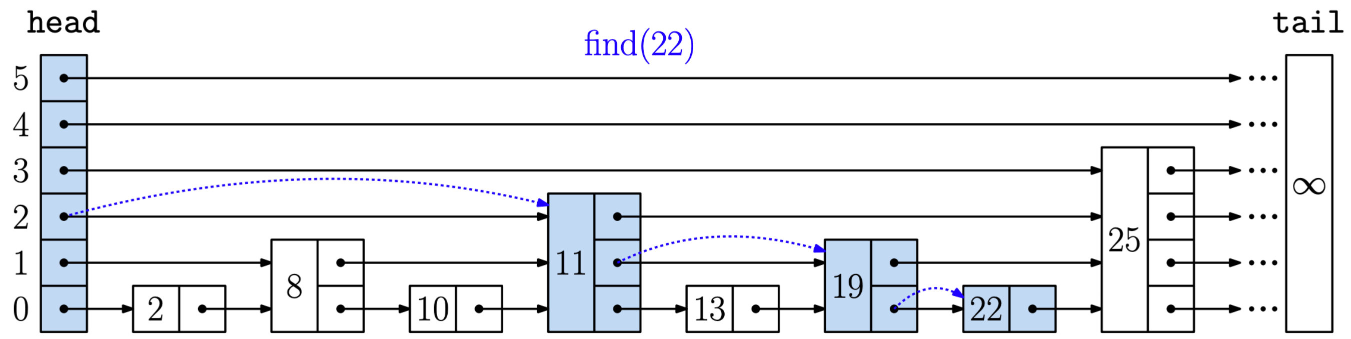


Fig. 5: Searching the ideal skip list.

We can randomize the number of nodes per level by flipping a coin and only stopping at a level if tails occurs

Now level k is expected to have about $\frac{n}{2^k}$ nodes meaning that the number of nodes at level $\lceil lg(n) \rceil$ is constant

Space Analysis: worst case every node has height log(n) so O(nlogn) total. Best each node has height 1 so O(n) total

Expected case: all n nodes contribute to level 0, n/2 contribute to level 1, n/4 to level 2, etc. so

$\sum_{i=0}^{h-1} \frac{n}{2^i} = n(2 - \frac{1}{2^h} \leq 2n = O(n)$

Search: for $0 \leq i \leq O(logn)$, let $E(i)$ represent the expected number of nodes visited in the skip list at the top i levels of the skip list

Look at the path going backwards so it will either go up or stay on the current level and go left

Whenever we arrive at some node of level i, the probability that it contributes to the next higher level is p and 1 - p to stay on same level. Counting the current node we just visited (+1) we have

$E(i) = 1 + pE(i-1) + (1-p)E(i) \rightarrow E(i) = \frac{1}{p} + E(i-1) = \frac{i}{p}$ and since i ≤ O(logn) then search is O(logn)

Insertion: search for x to find its immediate predecessors at each level then create node x and flip a coin until tails. Letting k denote the number of tosses made, height = min of k + 1 and height of list then link the k+1 lowest predecssors

Deletion: find the node and keep track of all predecessors at various level of list then unlink the target node at each level (like in standard linked list removal)

Implementation Notes: skip-list nodes have variable size, containing the key-value pair, variable-sized array of next pointers (p.next[i] points to the next node at level i). Also has 2 sentinel nodes (head and tail where tail.key is ∞ to stop search)

7 Splay Trees

Self adjusting tree taht dynamically adjusts its structure according to a dynamically changing set of access probabilities

nodes that are accessed more frequently are closer to the root

Binary Search Tree that uses rotations to maintain structure but doesn't need to store balance information

Whenever a deep node is accessed, the tree will restructure itself so tree is more balanced

$\Omega(n)$ worst operation but amortized O(logn)

T.splay(x): searches for key x in a tree T and reorganizes T while rotating x up to the root. If x not found, use preorder predecessor or successor.

simply rotating the target node up doesn't work because it can leave the tree skewed/unbalanced

instead take 2 nodes at a time and rotate both

For node p let parent be q and grandparent be r then

Zig-zig: if p and q are both right childnre or left children, apply rotation at r then q to bring p to the top

Zig-zag: if p and q are left-right or right-left children, apply rotation to q then r to bring p to top

Zig: if p is the child of the root, rotate root of T and make p the new root

if p is the root of T, we are done

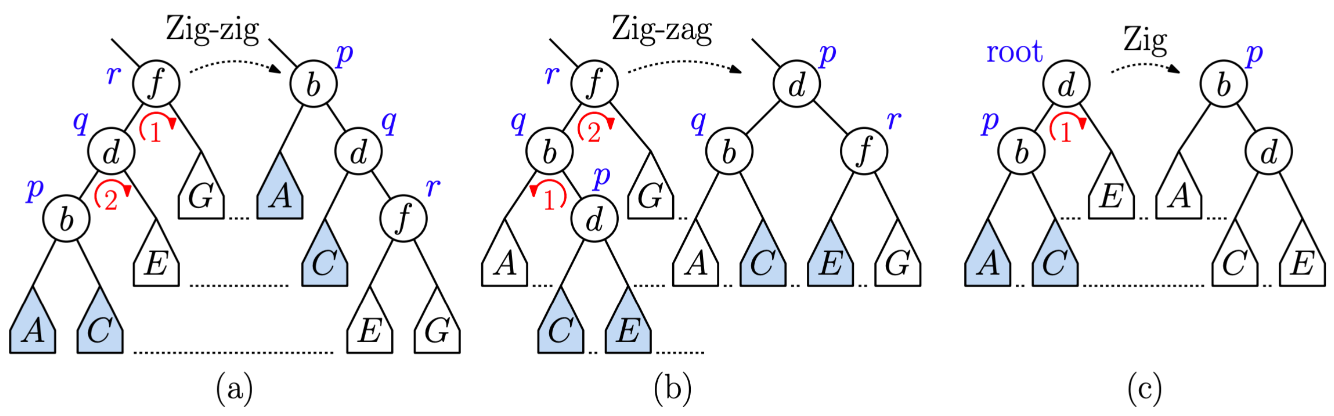


Fig. 3: Splaying cases: (a) Zig-Zig, (b) Zig-Zag and (c) Zig.

Everytime zig-zig or zig-zag is called, the subtree is raised up 1 level so for a long path, these rotations will reduce its height by 1/2

Find: invoke T.splay(x) which transports x to the root. If root.val != x then throw an error

Insert(x, v): invoke T.splay(x). If root.key = x then let current root = y. Either

y < x then all keys in R subtree > x so create a new root (x,v) and add y to L and add R to new root

y > x then all keys in L < x so create a new root (x,v) and add x to R and add L to new root

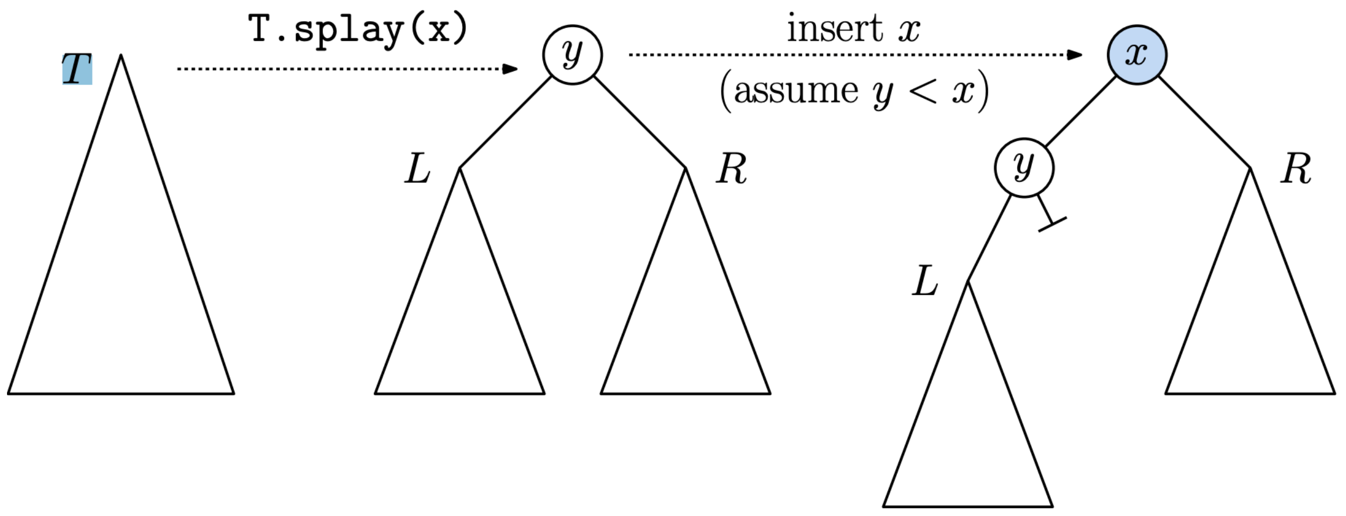


Fig. 5: Splay-tree insertion of x.

Delete: invoke T.splay(x) then if root != x throw error. Else if L is empty return R or if R is empty return L. Otherwise let R' = R.splay(x). This will find the inorder successor y. Since y will have no left subtree (all values in R are > x), make y the new root and link L

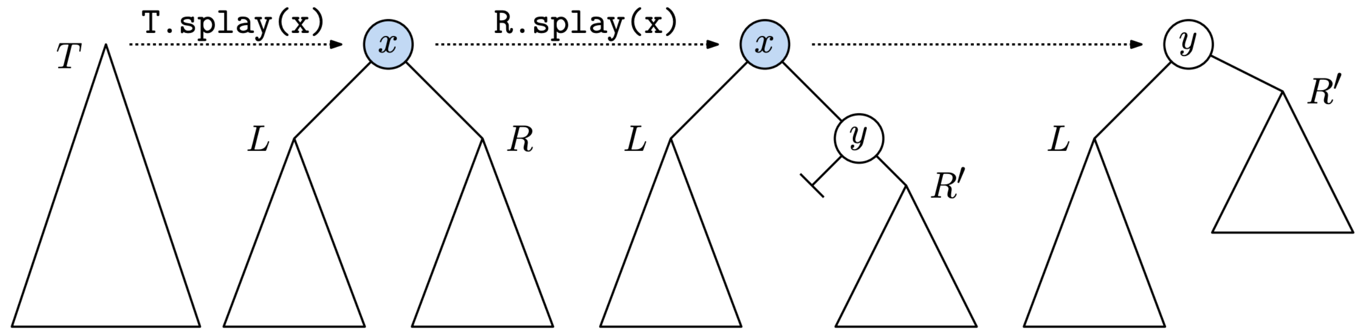


Fig. 6: Splay-tree deletion of x.

8 B-Trees

J-ary multiway search trees where each node stores reference to j subtrees $T_1, T_2, ..., T_j$ and has j-1 keys $a_1 < a_2 < a_{j-1}$ such that each T_i subtree stores nodes whos keys are $> a_{i-1}$ and $< a_i$
Achieves balance by constraining width of each node
For any int m ≥ 3, B-tree of order m is a multiway search tree if   root is leaf or has $2 \leq x \leq m$ children
  each node except root has between $\left\lceil \frac{m}{2} \right\rceil$ and m children which can be null
    node with j children has j-1 keys
  All leaves are on the same level of the tree
Height Analysis: as B-Trees grow wider, the height decreases
B-Tree of order m with n keys has height of at most $(lgn)/\gamma$ where γ = lg(m/2)
Proof: assume m is even and let N(h) = number of nodes in skinniest possible order-m B tree of height h
  Root has ≥ 2 childnre that have ≥ m/2 children
    therefore 2 nodes at depth 1
    $2(m/2)$ nodes at depth 2
    $2(m/2)^2$ nodes at depth 3
    $2(m/2)^{k-1}$ nodes at depth k
  So $N(h) = \sum_{i=1}^{h} 2(\frac{m}{2})^{i-1}$
  let c = m/2
  $N(h) = \frac{2(c^h-1)}{(c-1)} \approx \frac{2c^h}{c} = 2c^{h-1} = 2(\frac{m}{2})^{h-1}$
  Each node has $\geq \frac{m}{2} - 1$ keys $\approx \frac{m}{2}$
  $n \geq N(h) \geq 2(\frac{m}{2})^h \rightarrow h \leq \frac{lgn}{lg\frac{m}{2}}$
Node Structure: since B-Tree nodes can hold a variable number of items, every node is allocated max possible size

```
final int M = m;  \\order of B-tree
class BTreeNode {
  int nChildren;
  BTreeNode child[M];
  Key key[M-1];
  Value value[M-1];
}
```

Searching: When arriving at an interval node, search through keys
  if x is found then return the corresponding value
  Else determine index i such that $a_{i-1} < x < a_i$ note that $(a_0 = -\infty, a_j = \infty)$
  Then recurse into subtree T_i
Insertion and Deletion require some restructuring methods (rotation, splitting, and merging)
Rotation: Node can have between $\left\lceil m/2 \right\rceil$ and m children, and one less keys. Insertion and deletion might make a node have too many or too few nodes so we fix this imbalance by moving a child into or from one of its siblings, assuming the sibling isn't full

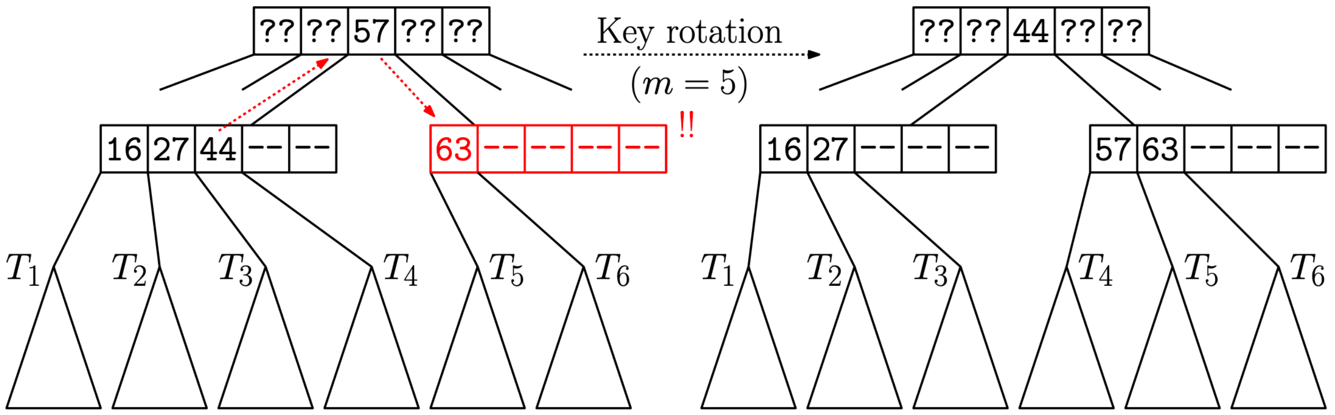


Fig. 3: Key rotation for a B-tree of order $m = 5$.

Node Splitting: node has 1 too many children and key rotation is not available so split node into 2 nodes, one with $m' = \lceil m/2 \rceil$ children and the other with $m'' = m + 1 - \lceil m/2 \rceil$ children
Since $(m' - 1) + (m'' - 1) = m - 2$, we have one extra key that is doesn't fit into L and R subchildren so it is promoted to parent and then handled up there

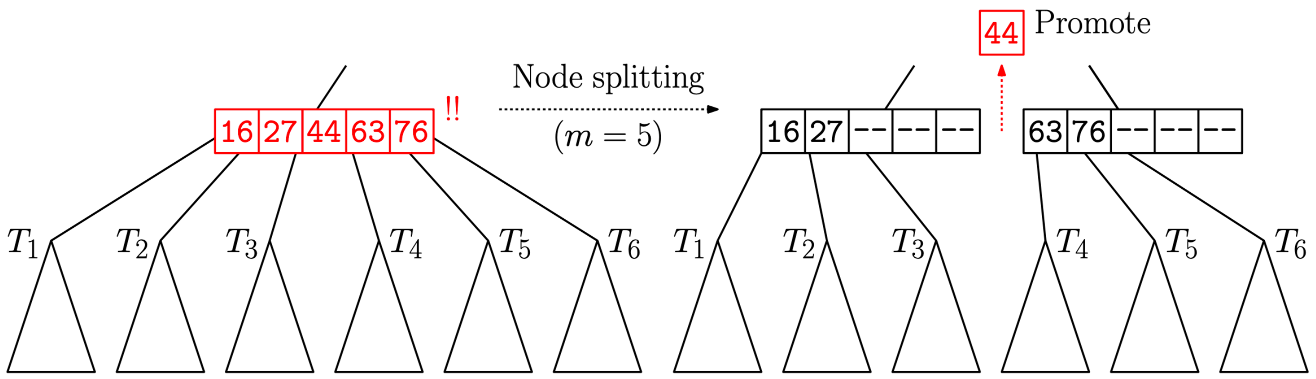


Fig. 4: Node splitting for a B-tree of order $m = 5$.

Proof for Node Splitting:
If m is even then $\frac{m}{2} \leq m + 1 - \frac{m}{2} = \frac{m}{2} + 1 \leq m$
If m is odd then $\frac{m+1}{2} \leq m + 1 - \frac{m+1}{2} = \frac{m+1}{2} \leq m$
Node Merging: Node might have 1 too few children after deletion. If key rotation isn't available, then we know that the sibling must have the minimum number of children ($\lceil m/2 \rceil$). Now merge node with the sibling into a node with $m' = (\lceil m/2 \rceil - 1) + \lceil m/2 \rceil = 2\lceil m/2 \rceil - 1$ children
Note that $\lceil m/2 \rceil - 2 + \lceil m/2 \rceil = m - 2$ which is one too few so we demote the appropriate key from the parent's node to get desired number of keys

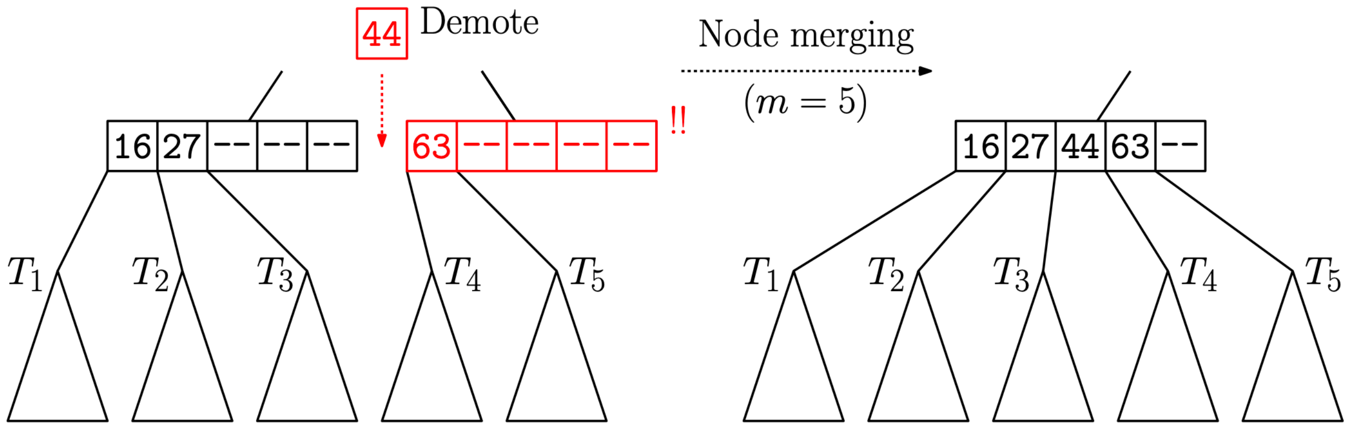


Fig. 5: Node merging for a B-tree of order $m = 5$.

Lemma: For all $m \geq 2$, $\lceil m/2 \rceil \leq 2\lceil m/2 \rceil - 1 \leq m$
If m is even then $\frac{m}{2} \leq m - 1 \leq m$
if m is odd then $\frac{m+1}{2} \leq m \leq m$

Insertion: creating nodes is an expensive operation so try to rotate whenever possible. Search for key x and if
found thrown an exception
leaf is not at full capacity (fewer than m-1 keys) then we insert key and done
may involve sliding around but can ignore the cost since m is constant
otherwise node overflows and check if either sibling is less than full. If so then perform a rotation else perform node split.

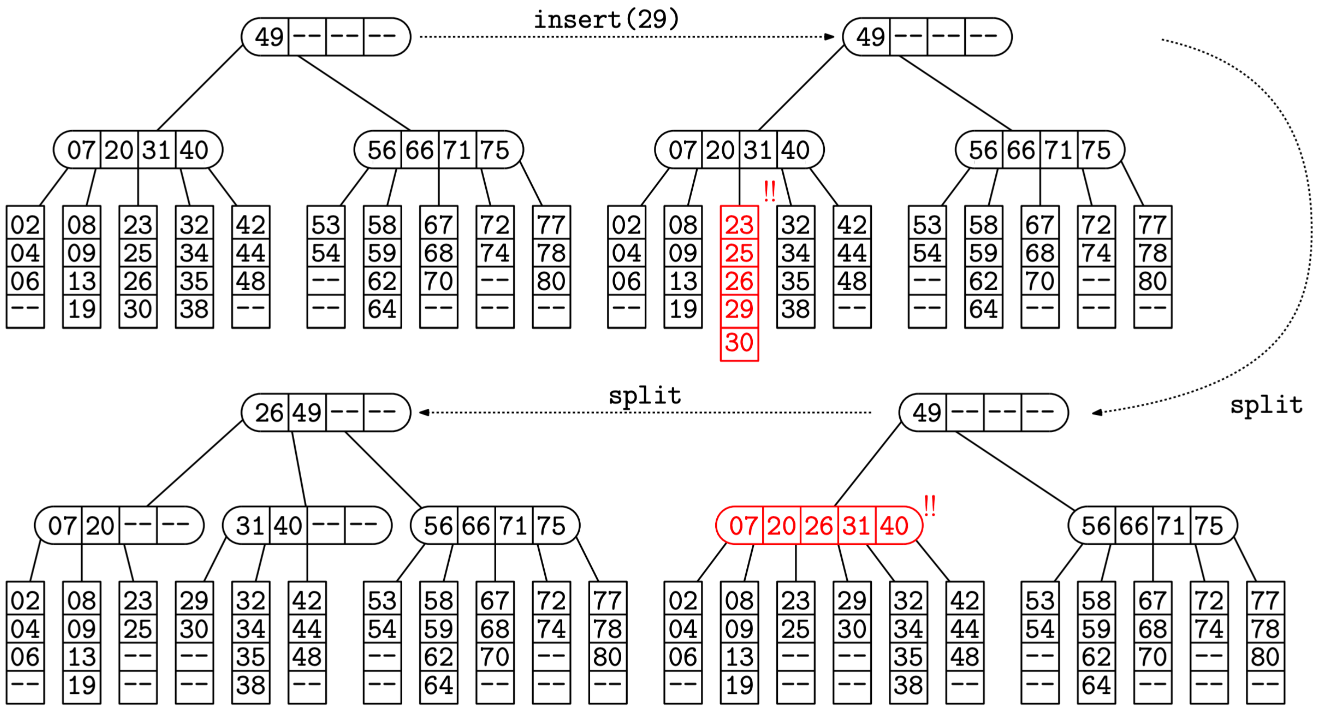


Fig. 6: Insertion of key 29 ($m = 5$).

Deletion: Search for the node to be deleted. Need to find replacement so take largest key in left child or smallest key in right child and move this key up to fill the hole

if left node has $\geq \lceil m/2 \rceil - 1$ keys we are done
else node will underflow so key rotate if possible else use node merge and recurse in parent

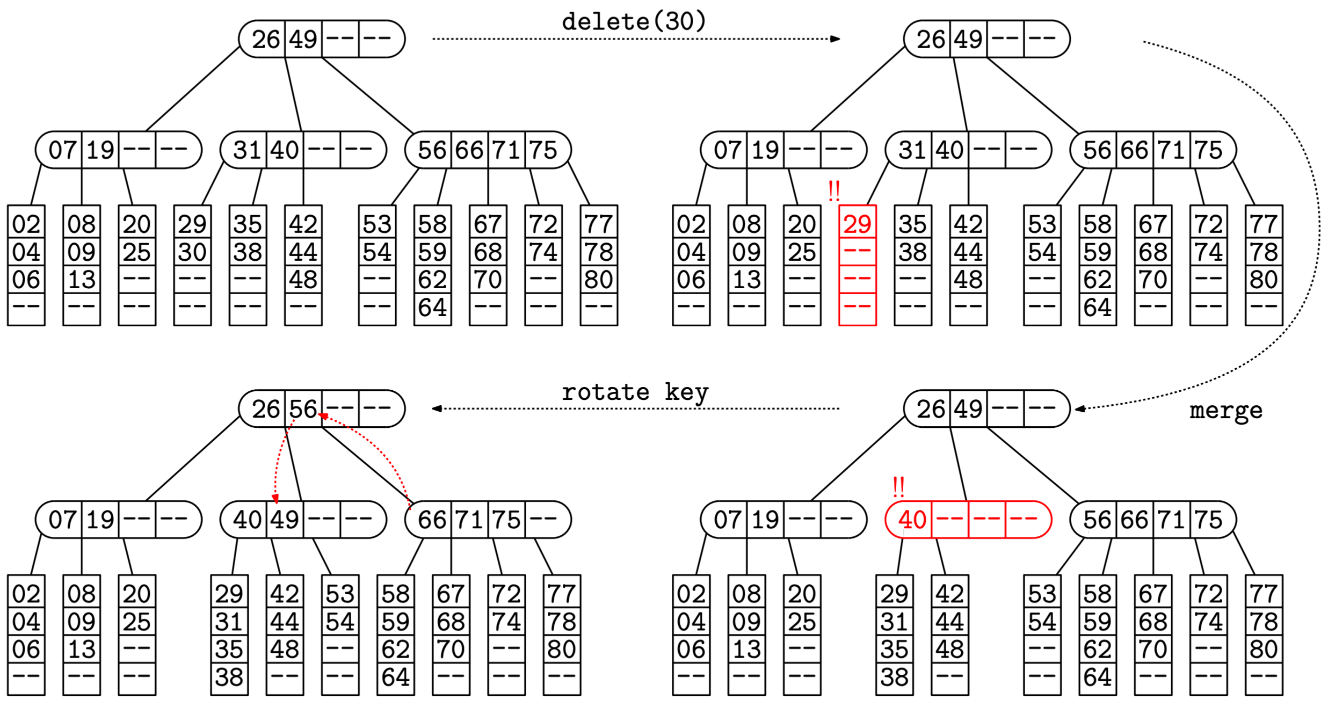


Fig. 8: Deletion of key 30 ($m = 5$).

B+ Trees: internal nodes only store keys (not values)
Keys are used solely for locating leaf node containing actual data so it's not necessary that every key in internal node to correspond to a key-value pair
Each leaf node has a next-leaf pointer, which pointers to the next leaf in sorted order
Storing only in internal nodes save space and allows increased tree fan out $\rightarrow$ lowers height ofr tree
Internal nodes are an index to locate actual data which resides at the leaf level
Now internal nodes with keys $a_1, ..., a_{j-1}$, subtree T_j has keys x such that $a_{i-1} < x \leq a_i$
Next leaf enables efficient range reporting quries where we can list keys in range $[x_{min}, x_{max}]$
so we now find the leaf node x_{min} and follow next leaf links until we reach x_{max}