CMSC420 Advanced Data Structures

Contents

1 Lists **2**
- 1.1 Proof Amortized Cost 2

2 Trees **3**
- 2.1 Tree Representation 3
- 2.2 Extended Binary Trees 4
- 2.3 Threaded Binary Trees 4
- 2.4 Complete Binary Tree 4

3 Dictionaries **5**
- 3.1 Array Representation 5
- 3.2 Binary Search Tree Representation 5
- 3.3 Analysis of BST 6
- 3.4 Random Insertions and Deletions on BST 6

4 AVL Trees **7**

5 2-3 Trees, Red-Black Trees, AA Trees **11**
- 5.1 2-3 Trees 11
- 5.2 Red-Black Trees 12
- 5.3 AA Trees 13

6 Treaps and Skip Lists **17**
- 6.1 Treaps 17
- 6.2 Skip Lists 18

7 Splay Trees **19**

8 B-Trees **21**

9 Hashing **25**

10 Extended BST and Scapegoat Trees **28**
- 10.1 Extended BST 28
- 10.2 Scapegoat Tree 29

11 Geometric Data Structures **31**

12 Memory Management **41**

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)

Arrays and LinkedLists can be used to create:

- 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 into one aggregate structure (e.g. ArrayList)
- 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

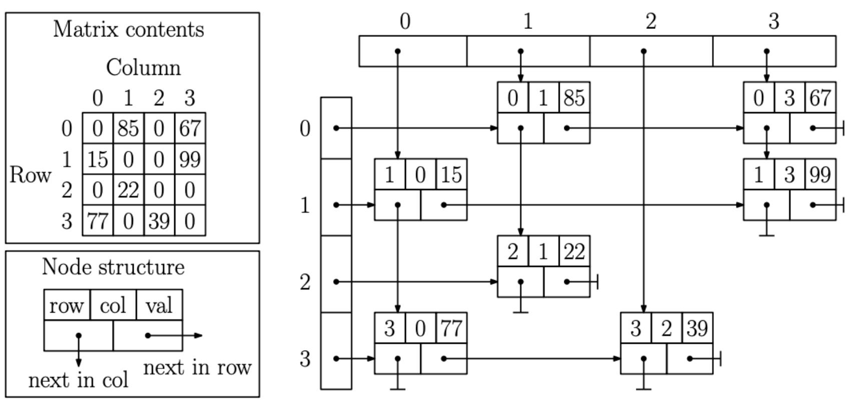


Fig. 2: Sparse matrix representation using a multilist structure.

1.1 Proof Amortized Cost

Theorem: When doubling array size for reallocation, amortized cost of list operation is $O(1)$.

Proof: Assume we are using a stack (can be proven for other ADT). Let n be the current size of the array. Each time we push an element, we store an additional 4 tokens into a bank. Once the array is full, we reallocate an array of size $2n$ which requires intializing the array and copying elements (this will take $2n$). Last time we reallocated was when we transitioned from an array of size $n/2$ to n so we must have performed at least $n - n/2 = n/2$ insertions. Now we have $4(n/2) = 2n$ from the extra tokens we stored.
If we had instead added an additional c spaces when we reallocated, there would be an issue when the table size becomes too large since reallocation would take a greater magnitude than the savings we collect.
If we had instead raised the size of the array by a power, this works but we end up wasting a lot of space.

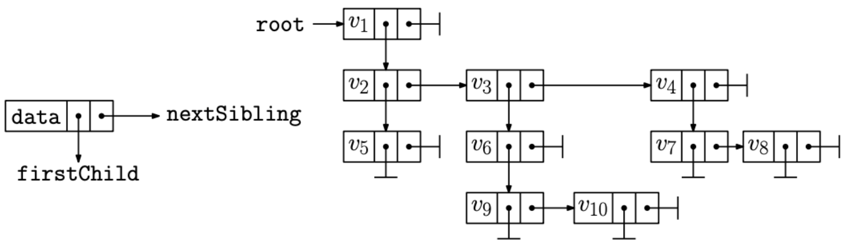
2 Trees

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

2.1 Tree Representation

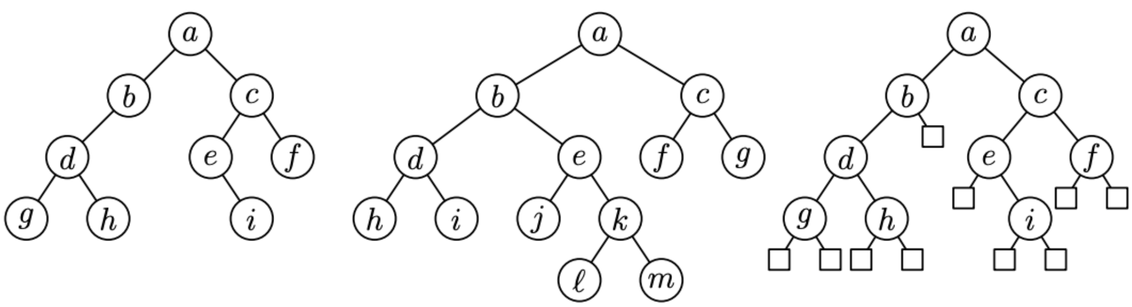
Binary representation of a rooted tree: nodes have a pointer to first child and another pointer to next sibling



(a) (b)

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 any binary tree a full binary tree by extending tree by adding external nodes to replace all empty subtrees



(a) (b) (c)

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

```java
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

2.2 Extended Binary Trees

If there are n internal nodes, there are $n + 1$ external nodes. Quasi proof by noticing that we will essentially replace every leaf with 1 internal and 2 external nodes.

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

Proof:

- 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
- $x(n) = x(n_L) + x(n_R) = (n_L + 1) + (n_R + 1) = (1 + n_L + n_R) + 1 = n + 1$ external nodes
- so $n + 1$ (external) + n (internal) = $2n + 1$
- Moreover, about 1/2 of nodes of extended Binary Tree are leaf nodes

2.3 Threaded Binary Trees

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

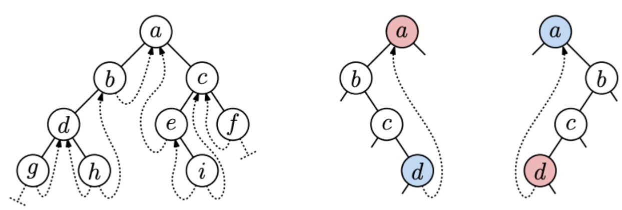


Fig. 6: A Threaded Tree.

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

To find the inorder successor of a node:

- 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 until we find the last node before the thread

2.4 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
```

3.1 Array Representation

- Unsorted Array has O(n) search and delete, O(n) because we need to check for duplicates
- Sorted Array has O(logn) search and O(n) insertion and deletion because when we modify the array we need to shift elements

3.2 Binary Search Tree Representation

Search: O(n) for degenerate tree, O(logn) for balanced tree
Can use extended BST to give info about how the target key is inbetween its inorder predecessor and inorder successor

```
//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;
}
```

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

- 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

```
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;
}
```

Delete:

- if target node is a leaf, remove it
- if target node only has 1 child, replace it with the child
- else replace target node with inorder successor (aka leftmost on right subtree). End the delete the replacement node

---

```
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 ;
}
```

3.3 Analysis of BST

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

Theorem: 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 $\sum_{i=2}^{n} \frac{1}{i} \approx \ln(n)$

3.4 Random Insertions and Deletions on BST

Because the replacement node will always be the inorder successor, if we repeatedly add and remove random elements from a BST, this biased choice will eventually bring the height of the tree to $O(\sqrt{n})$
This bias can be handled by randomly selecting the replacement node as the inorder successor or the inorder predecessor

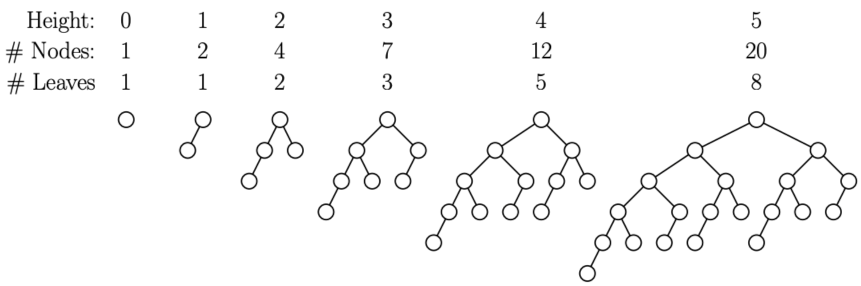
4 AVL Trees

AVL Balance Condition: For every node in tree, absolute difference between heights of left and right subtrees is at most 1. A node's balance factor is determined by

$$\text{balance(v)} = \text{height(v.right)} - \text{height(v.left)}$$

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. So 1 child will have height h-1 and the other will have height h-2
- N(0) = 1, N(1) = 2, N(h) = $1 + N(h_L) + N(h_R) = 1 + N(h-1) + N(h-2)$
- Now $N(h) = n \geq c\varphi^h \rightarrow h \leq log_\varphi n \rightarrow h = O(logn)$
- Also find method using AVL is O(logn)



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
}
```

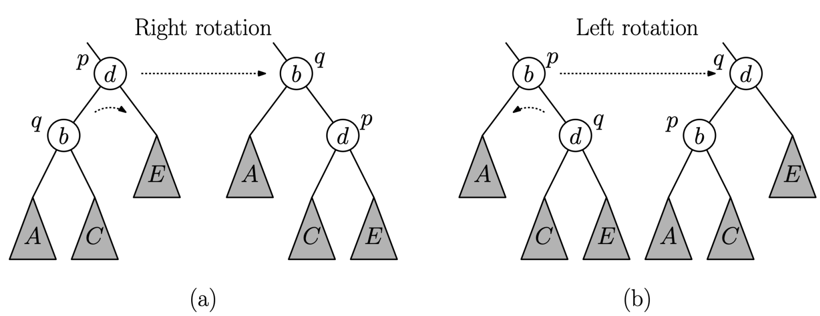


Fig. 3: (Single) Rotations. (Triangles denote subtrees, which may be null.)

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

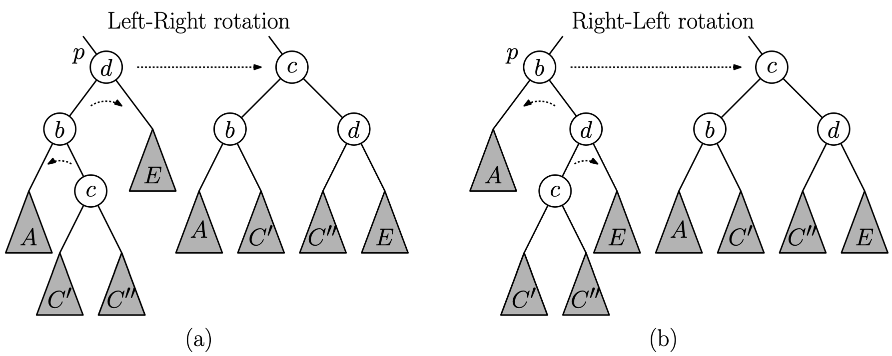


Fig. 4: Double rotations (`rotateLeftRight(p)` and `RotateRightLeft(p)`).

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 on unbalanced node
- Right-Right subtree too deep then rotate left on unbalanced node
- Left-Right subtree too deep then rotate left-right (left child of unbalanced node then unbalanced node)
- Right-Left subtree too deep then rotate right-left (right child of unbalanced node then unbalanced node)

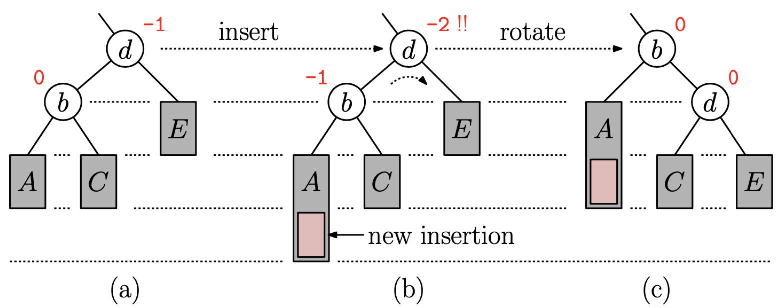


Fig. 23: Restoring balance after insertion through a single rotation.

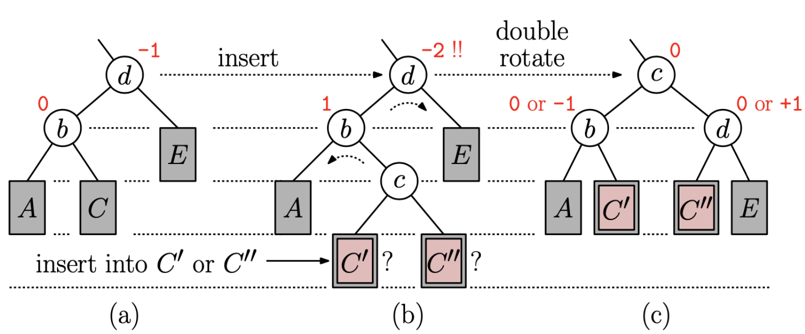


Fig. 24: Restoring balance after insertion through a double rotation.

```
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 (e.g. if one of the inner subtrees is too tall, we need to call a double rotation)

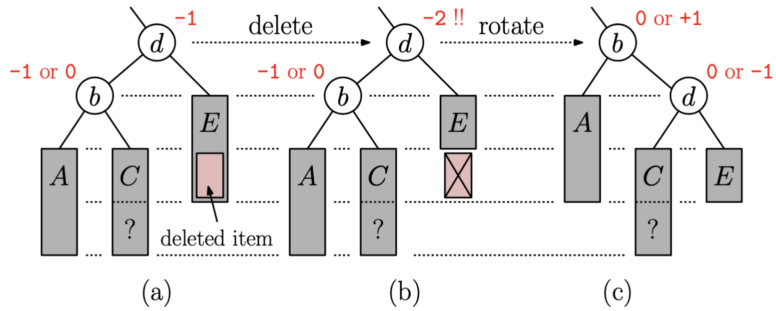


Fig. 25: Restoring balance after deletion with single rotation.

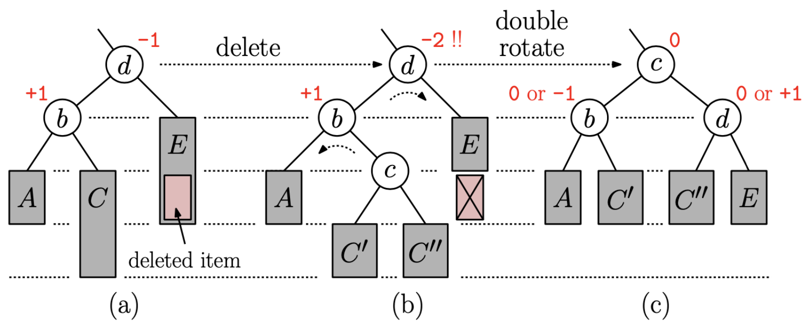


Fig. 26: Restoring balance after deletion with double rotation.

Note that node d dropped one level so we need to call balance check on c. In this case deletion can end up calling a cascade of O(logn) rebalancing operations

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

5.1 2-3 Trees

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)
All leaves are on the same level

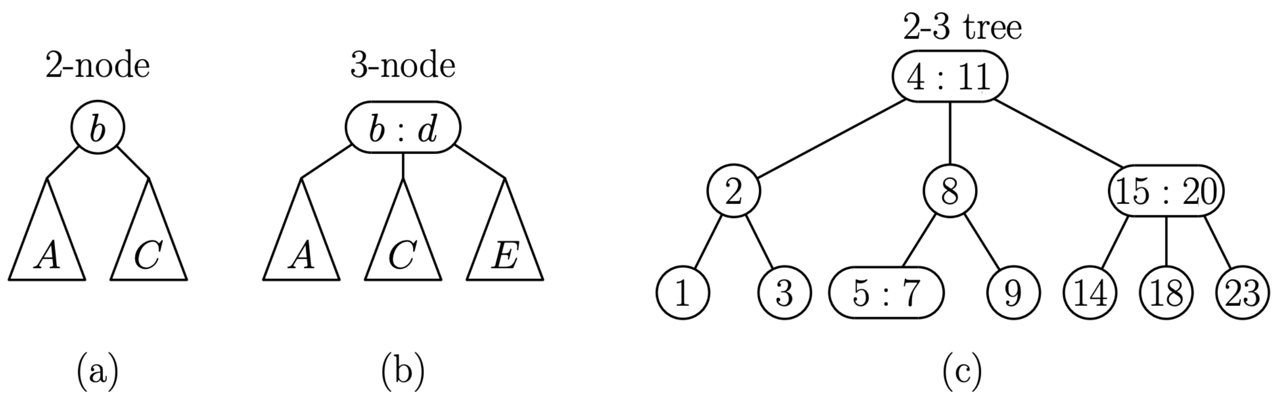


Fig. 1: (a) 2-node, (b) 3-node, and (c) a 2-3 tree.

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

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 and will continue to recurse up until it reaches the root

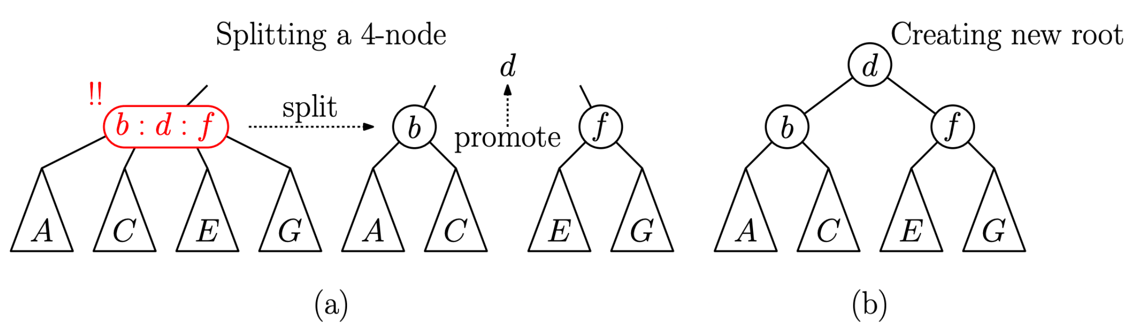


Fig. 2: 2-3 tree insertion: (a) splitting a 4-node into two 2-nodes and (b) creating a new root.

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

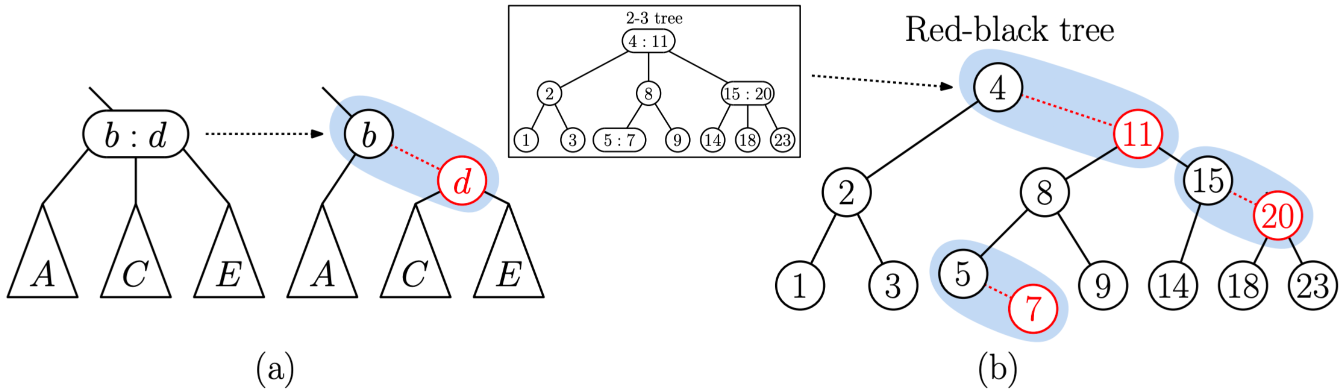


Fig. 6: Representing the 2-3 tree of Fig. 1 as an equivalent binary tree.

Created right subnode is 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

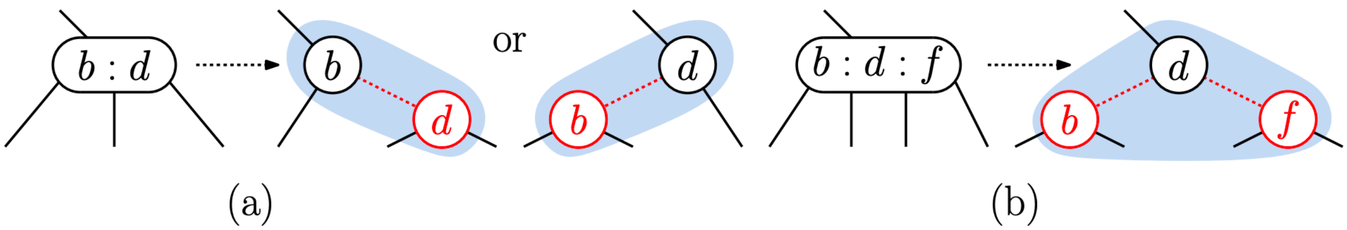


Fig. 7: Color combinations allowed by the red-black tree rules.

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

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

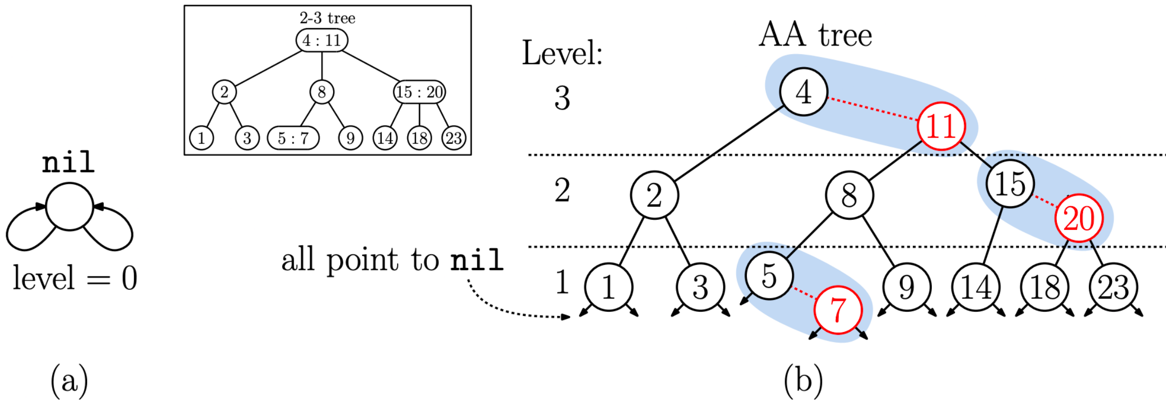


Fig. 8: AA trees: (a) the **nil** sentinel node, (b) the AA tree for the 2-3 tree of Fig. 1.

Find method works exactly the same as it does for BST

Insertion and Deletion require skew(p) and split(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

```
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;
}
```

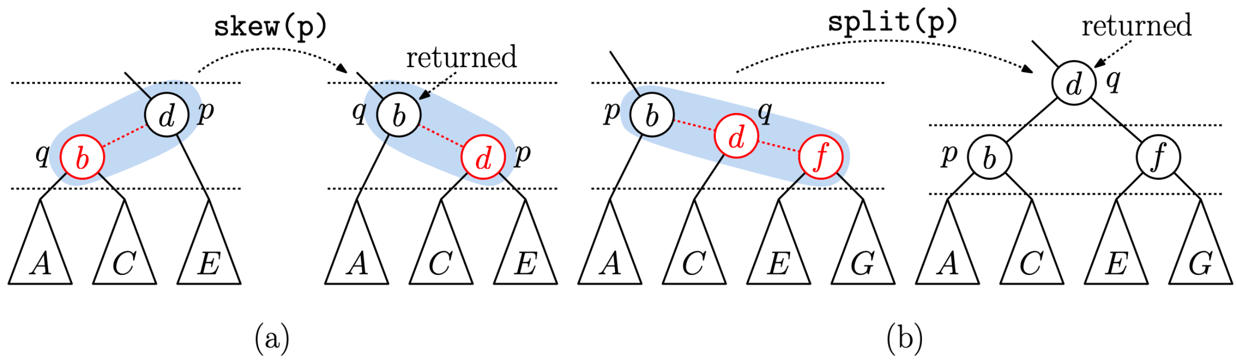


Fig. 9: AA restructuring opertions (a) skew and (b) split. (Afterwards q may be red or black.)

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

- 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

```
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)
}
```

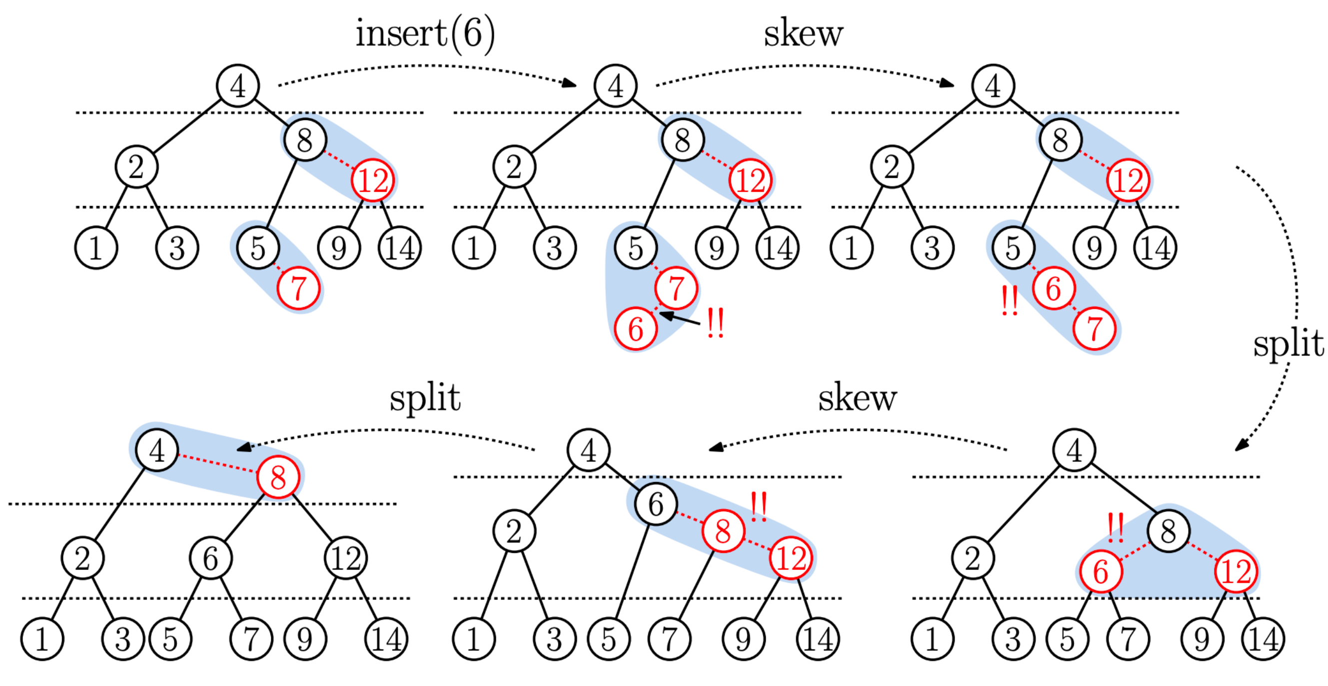


Fig. 11: Example of AA-tree insertion.

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

- 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). The example below shows how there might be a 2 level gap between a parent and a child, which could end up necessitating 3 skew operations which then require 2 splits to fix

```
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;
}
```

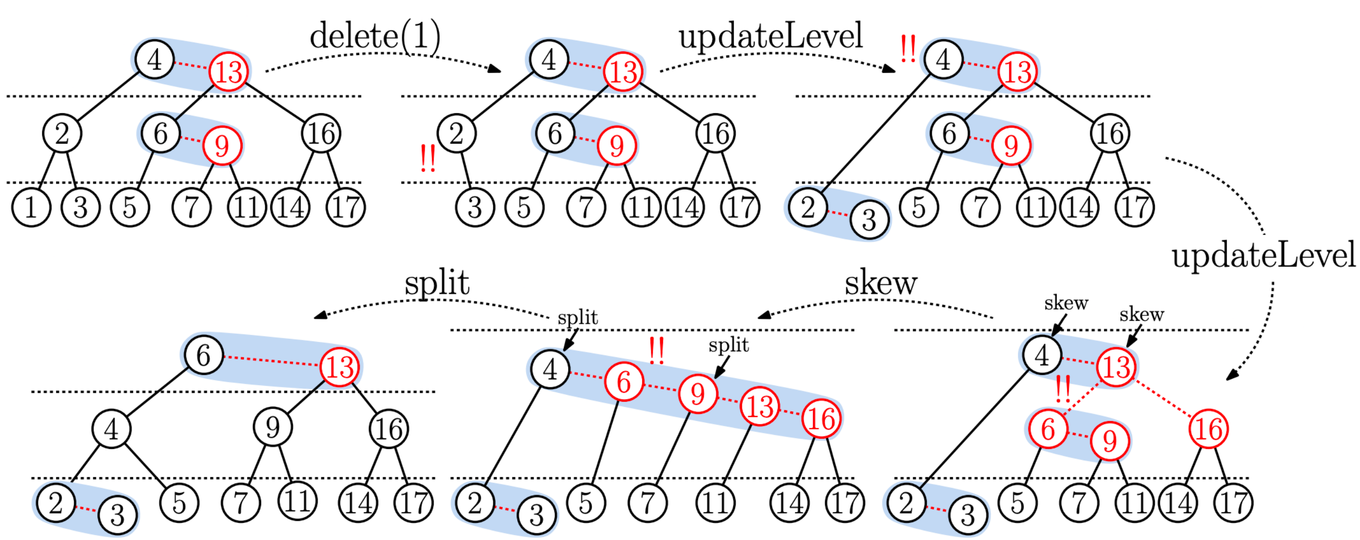


Fig. 12: Example of AA-tree deletion.

```
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(p);
  }
}
```

6 Treaps and Skip Lists

6.1 Treaps

Intuition is that if keys are inserted into BST in random order, then height will be $\approx O(\log(n))$

Insertion: Insert node based on key value then assign it a random priority (p.priority) and sort based on this priority by rotating the tree several times to balance it based on p.priority

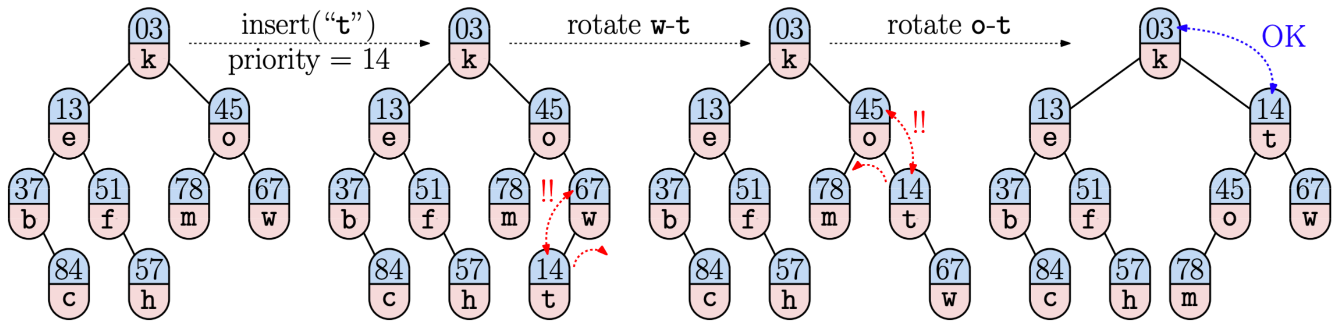


Fig. 2: Treap insertion.

Deletion: 3 cases

- node is leaf just remove it
- node has 1 child then replace node with child
- node has 2 children then set its priority to ∞ and apply rotations to sift down to leaf and remove

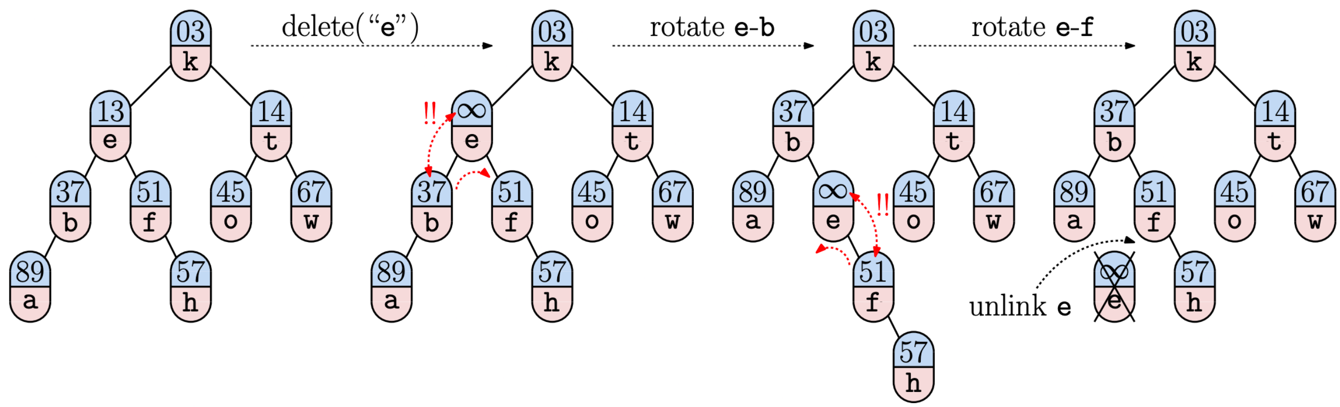


Fig. 3: Treap deletion.

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
- and tail nodes are always lifted and tail has the key value ∞
- proceess will repeat $\lceil lg(n) \rceil$ times

Search: start at highest level of head then scan linearly at level i until we are about to jump to a key value > x then step down one level

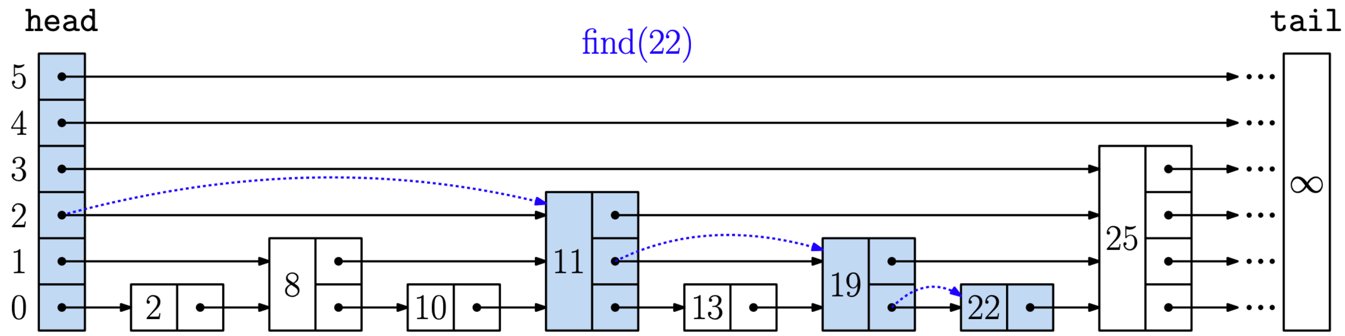


Fig. 5: Searching the ideal skip list.

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 Space: 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 Expected Runtime: 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 1/2 and 1 - 1/2 to stay on same level. Counting the current node we just visited (+1) we have

$E(i) = 1 + \frac{1}{2}E(i-1) + \frac{1}{2}E(i) \rightarrow E(i) = 2 + E(i-1) = 2i$ by recurrence analysis and since i $\leq$ 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 that 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

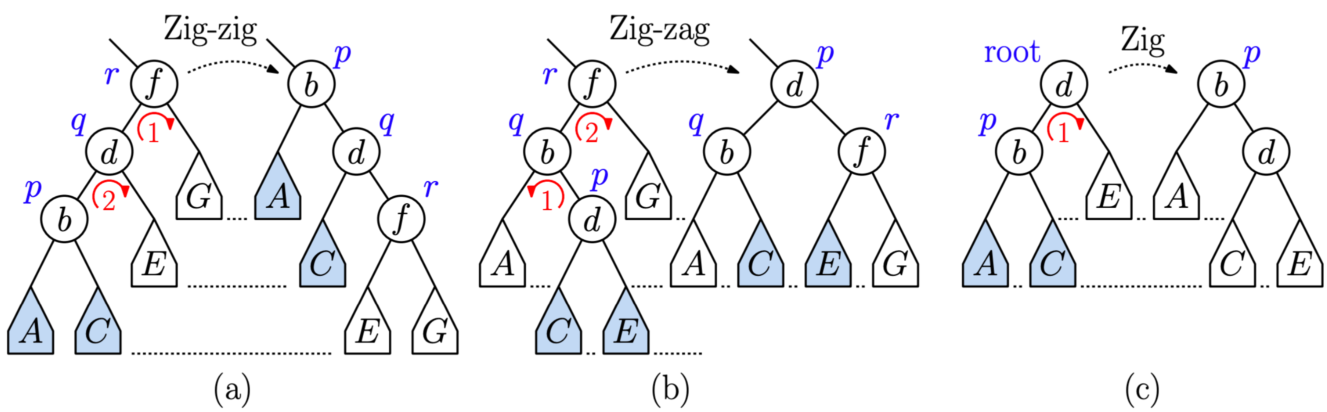


Fig. 3: Splaying cases: (a) Zig-Zig, (b) Zig-Zag and (c) Zig.

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). 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 y to R and add L to new root

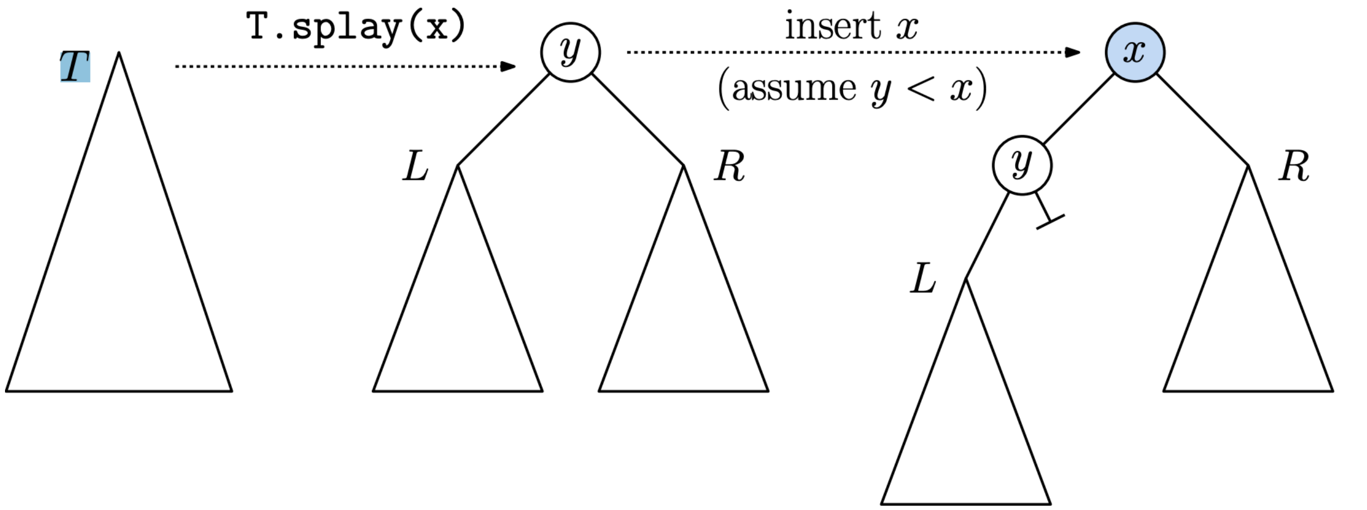


Fig. 5: Splay-tree insertion of x.

Delete: invoke T.splay(x) then if root != x throw error. Else

- if L is empty return R
- 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

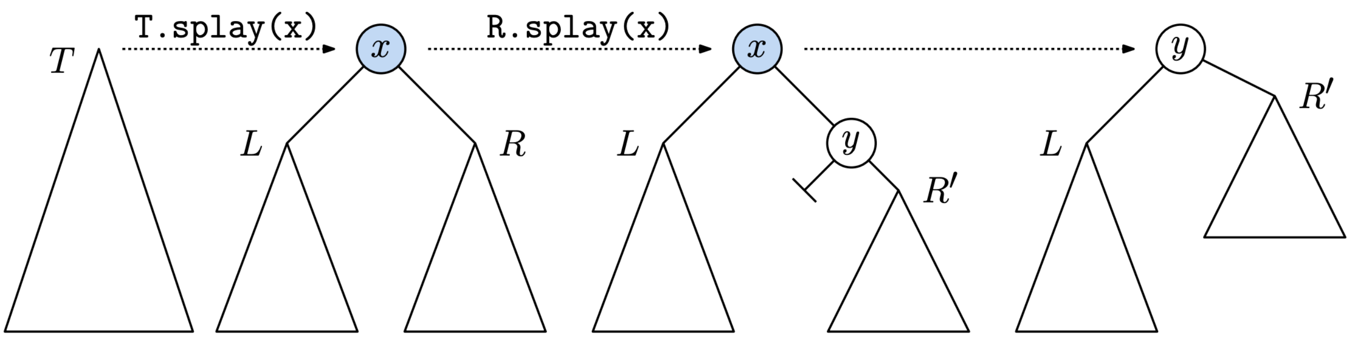


Fig. 6: Splay-tree deletion of x.

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 $\geq$ 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 $\geq$ 2 children that have $\geq$ 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 $\lceil m/2 \rceil$ 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

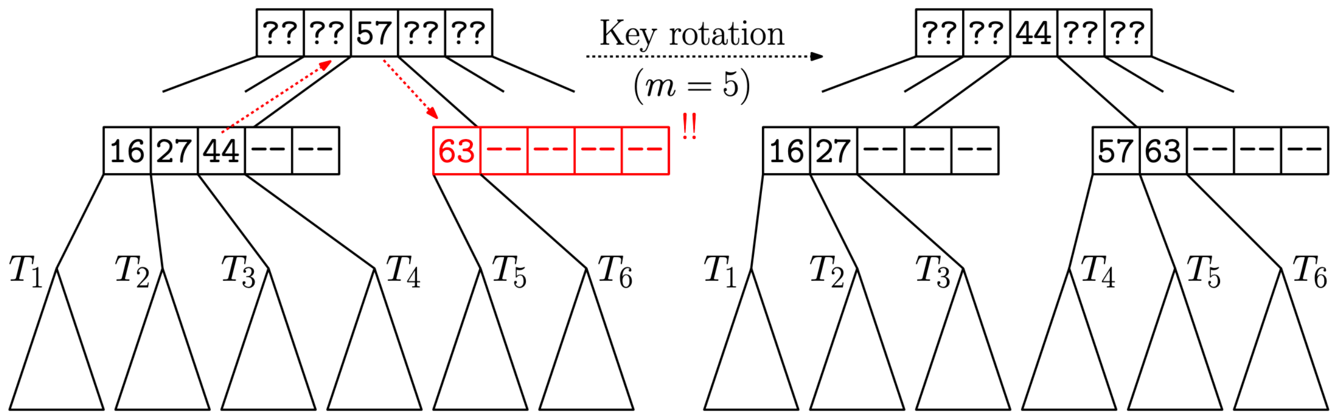


Fig. 3: Key rotation for a B-tree of order $m = 5$.

Node Splitting: node has 1 too many children (m + 1 children and m keys) 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 $(m' - 1) + (m'' - 1) = m - 1$, 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

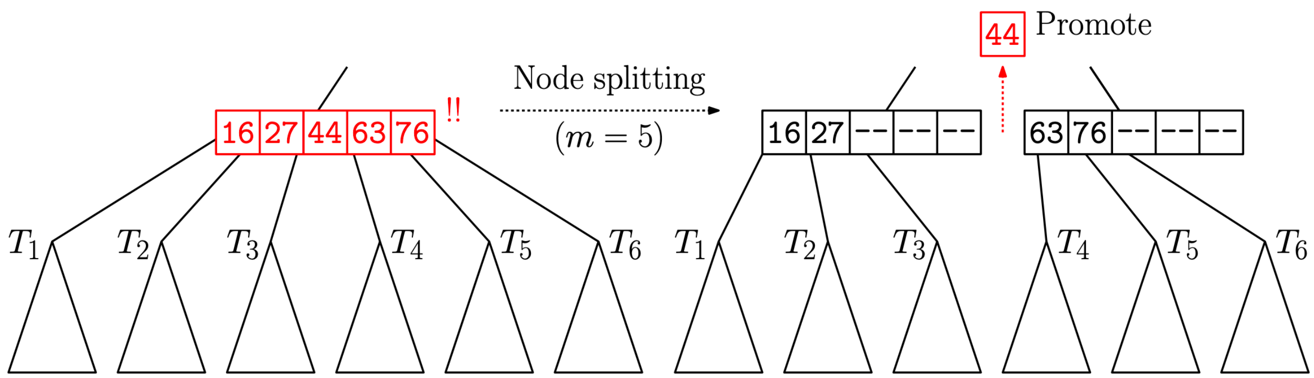


Fig. 4: Node splitting for a B-tree of order $m = 5$.

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 ($\lceil m/2 \rceil - 1$ nodes and one less keys) 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.
Since the parent lost a key and a node, recurse up to parent

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 $\frac{m+1}{2} \leq m \leq m$

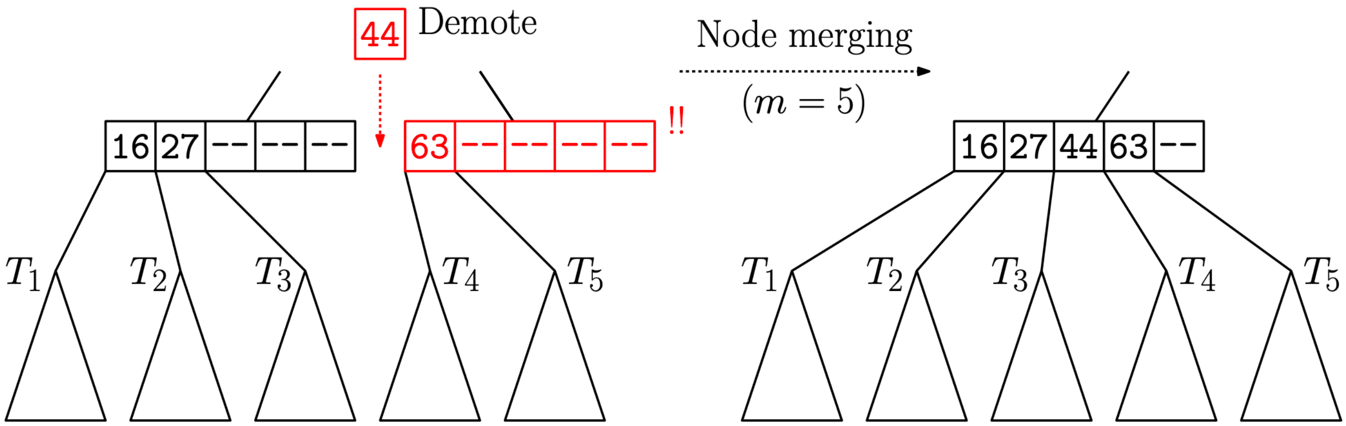


Fig. 5: Node merging for a B-tree of order $m = 5$.

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.

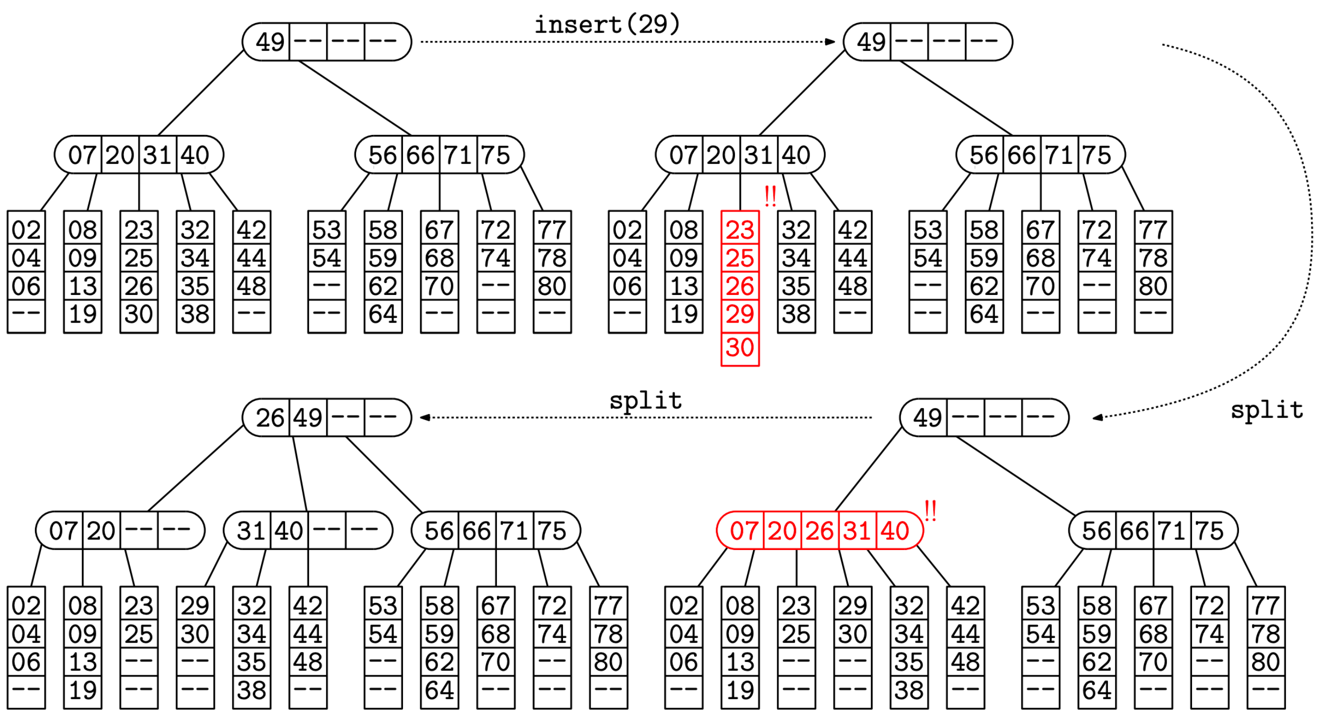


Fig. 6: Insertion of key 29 ($m = 5$).

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

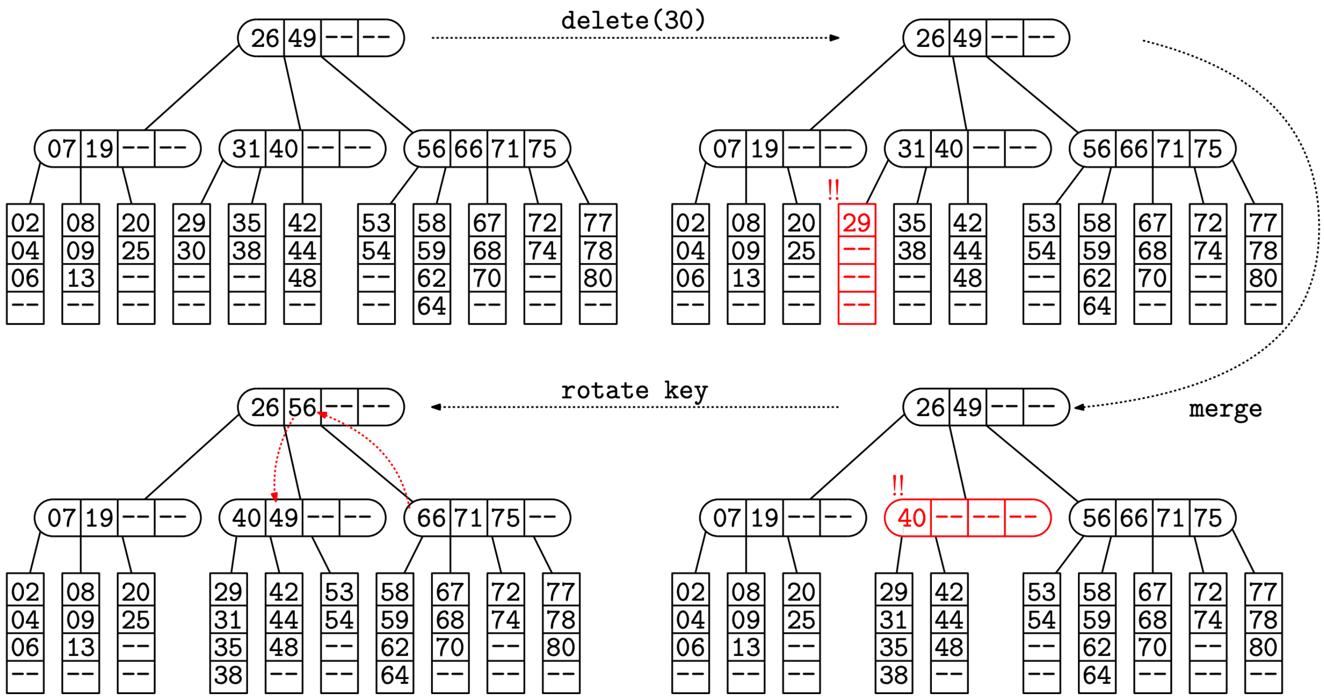


Fig. 8: Deletion of key 30 ($m = 5$).

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}

9 Hashing

Supports O(1) dictionary operations but cannot perform search operations like range queries (finding keys x such that $x_1 \leq x \leq x_2$) or nearest-neighbor queries (find key closest to a given key x)
Given a table of size m ¿ n and a hash function h(x), we use h(x) to map a key to a random index $[0...m-1]$
Possible issue of collisions where 2 keys land in the same index after hashing

Good hash function:
- Efficiently computable
- Produces few collisions
  - function on every bit of key
  - scatters naturally occuring clusters of keys

Types of hasing:
- Division Hashing: $h(x) = xmodm$
  - Fails 2nd rule (issue with clusters)
- Multiplicative Hashing: $h(x) = (ax)modm$
  - Where a is a large prime number
- Linear Hashing: $h(x) = (ax + b)modm$
  - Enhances Multiplicative hashing with added constant
- Polynomial Hashing: $h(x_0, ..., x_n) = (\sum_{i=0}^{k-1} c_i p^i)modm$
  - Useful for keys that have a sequence of objects (strings or coordinates)
  - Can use Horner's Rule to make summation faster $c_0 + c_1p + c_2p^2 + c_3p^3 = ((c_3p + c_2)p + c_1)p + c_0$

Universal Hashing: hash function is selected from large class of functions so probability of collision between 2 fixed keys is about 1/m
Consider a large prime p and two random integers $a \in \{1, 2, ..., p-1\}$ and $b \in \{0, 1, ..., p-1\}$
Use a linear has function $h_{a,b}(x) = ((ax + b)modpmodm$. As a and b vary, they will define a family of functions.
Let H_p denote the class of hash functions that arise from all possible combos of a and b. If we consider any two integers x and y such that $0 \leq y < x < p$ then the probability $h_{a,b}(y) = h_{a,b}(x)$ is 1/m

Handling Collisions Separate Chaining: have each index be a linked list and store collisions by adding to these linked lists.
- We define the load factor $\lambda = n/m$ and expect each list to have about λ elements
- If we are successful in finding the desired element, it'll take about $1 + \frac{\lambda}{2}$ (about halfway). Otherwise failure will take $1 + \lambda$. Additional 1 is for null checks
- Insertion and deletion will take about constant time so all dictionary operations will take $O(1 + \lambda)$
- Drawback of we need to use additional storage to store pointers to linked lists

Controlling Load Factor and Rehashing: we want to maintain a few invariants

$$0 < \lambda_{min} < \lambda_{max} < 1 \quad \lambda_{min} \leq \lambda \leq \lambda_{max} \quad n \leq \lambda_{max}m \quad m \leq n/\lambda_{min}$$

We don't want too large of a table or too small of a table so the optimal load factor is $\lambda_0 = (\lambda_{max} + \lambda_{min})/2$
If load factor is too big ($n > lambda_{max}m$) or too small ($n < \lambda_{min}m$) then we rehash with a larger table
- Allocate a table of size $m' = \lceil n/\lambda_0 \rceil$
- generate new hash function h' using new table size
- Insert every entry from old table to new table using new hash function
- remove old table
- New load factor $(n/m') \approx \lambda_0$ so we have restored optimal load factor

Amortized cost of rehashing is still good since we only rehash every so often

Open Addressing: To know which table entries have values and which are empty we store a special value *empty*. Now whenever we insert an element and its hashed index is already occupied we probe around nearby entries until we find an empty slot. The secondary search involves a function f so now the probe sequence is

$$(h(x) + f(1))modm, (h(x) + f(2))modm, ...$$

Linear Probing: probe function is $f(i) = i$ and we search sequential locations until we find an empty slot

- good for low load ($< 75\%$)factor. As load factor approaches 1, becomes very bad
- Issue with secondary clustering (when keys has to different locations but collision-resolution results in new collisions)
- Successful search expected cost: $(\frac{1}{2}(1 + \frac{1}{1-\lambda}))$ Unsuccessful serach expected cost: $(\frac{1}{2}(1 + (\frac{1}{1-\lambda})^2))$

Quadratic Probing: Avoids secondary clustering by using a nonlinear probing function, scattering subsequent probes. Example Code:

```
Value find(Key x) {
  int c = h(x);
  int i = 0;
  while ((table[c].key != empty) && (table[c].key != x)){
    c += 2*(++i) - 1;
    c = c \% m
  }
  return table[c].value;
}
```

Quadratic Probing has a potential issue of skipping potential slots due to growth factor. However if m is prime, we can guarantee that $\lceil m/2 \rceil$ probe sequences are distinct. Proof:
Contradiction: assume $0 \leq i < j \leq \lceil m/2 \rceil$ then
$h(x) + i^2 \equiv h(y) + j^2 \iff i^2 \equiv j^2 \equiv i^2 - j^2 \equiv 0 \iff (i-j)(i+j) \equiv 0 mod m$ but its impossible since m is prime
Other cool properties:

- if $m = 4k + 3$ and is prime, then quadratic probe will work for all table entries before repeating
- if m is a power of 2 and the increment factor is $\frac{1}{2}(i^2 + i)$ then we can probe every table entry before repeating

Double Hashing: use a hash function to figure out the probe sequence $f(i) = i * g(x)$. Now

$$h(x) + g(x), h(x) + 2g(x), h(x) + 3g(x), ...$$

To ensure that there are no cycles, m and $g(x)$ must be relatively prime

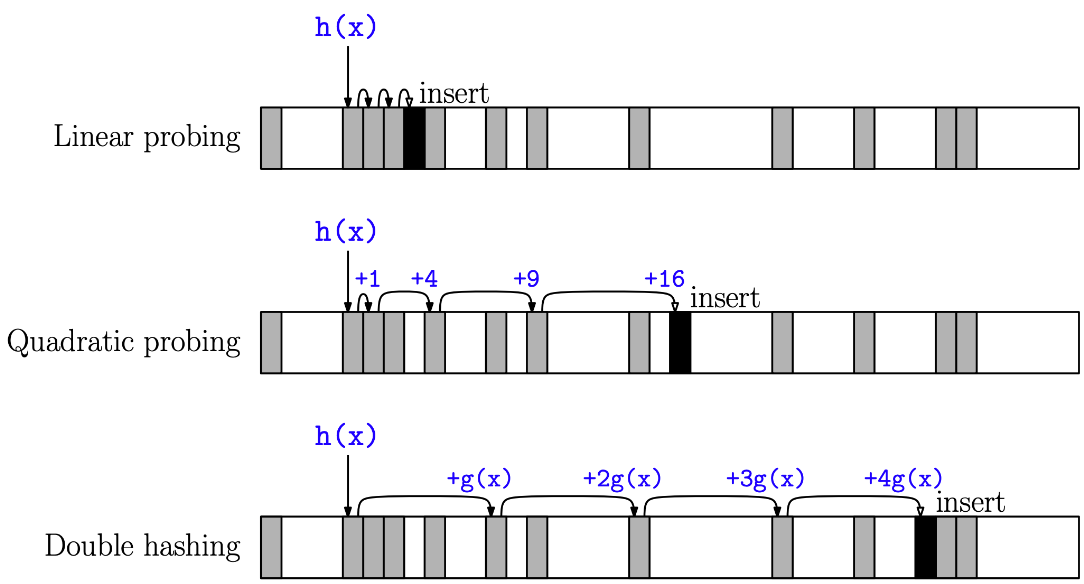


Fig. 4: Various open-addressing systems. (Shaded squares are occupied and the black square indicates where the key is inserted.)

Successful search expected cost: $(\frac{1}{\lambda}\ln(\frac{1}{1-\lambda}))$ Unsuccessful serach expected cost: $(\frac{1}{1-\lambda})$

Deletition can be tricky since if we delete a node in the probe sequence, we cannot find the latter elements in that probe sequence. To resolve this, we create a special value called *deleted* meaning that that slot is available for insertion but search method can continue searching the probe sequence until it finds the target element or reaches an empty cell

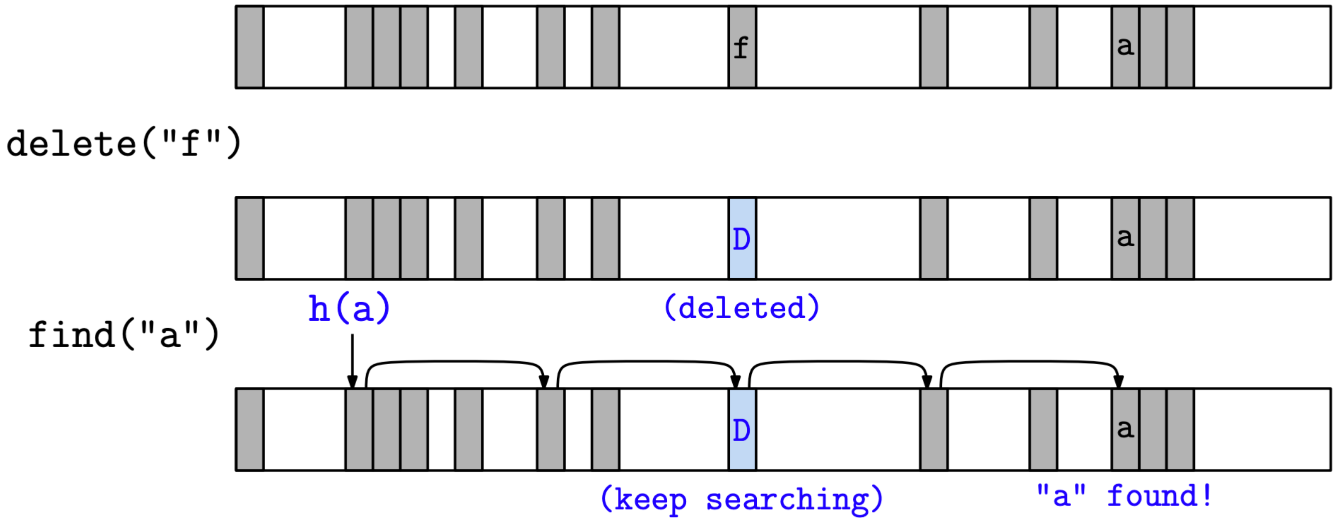


Fig. 6: Deleting in open-addressing by using special *empty* entry.

However, this solution makes can make the search path extremely long (even if the load factor is low). Another possible solution is to bring up the latter elements of the probe sequence up after deleting an element, but that makes deletion take longer.

10 Extended BST and Scapegoat Trees

10.1 Extended BST

Key are in external nodes and internal nodes have splitters with the property that all external nodes $x \leq s$ on left and $x > s$ on right

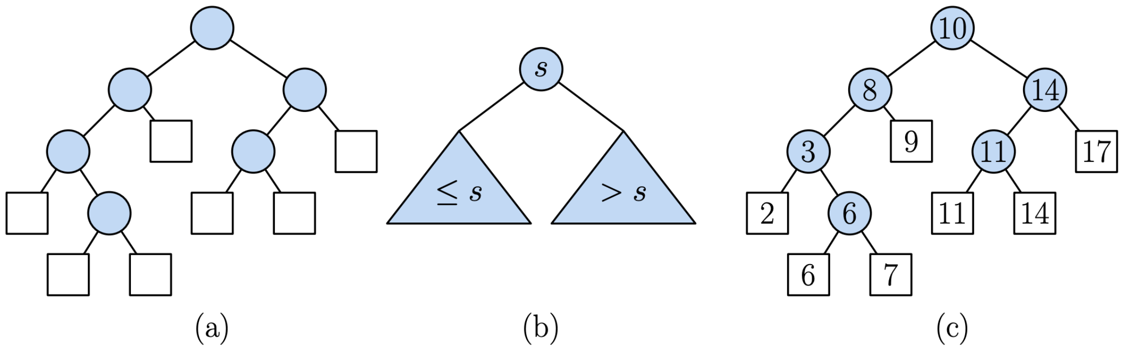


Fig. 1: (a) Extended binary tree, (b) extended binary search tree structure, and (c) extended binary search tree containing the keys $\{2, 6, 7, 9, 11, 14, 17\}$.

Find(Key x, Node P): initial call find(x, root) then

- if $x \leq p.key$ recurse on p.left
- else recurse of p.right
- once at external node u, test if $x = u.key$

Insert(Key x, Value v, Node p): returns reference to root of updated subtree where x was inserted. Initial call insert(x, v, root) then

- if $x \leq p.key$ then recurse on p.left
- else recurse on p.right
- if empty tree craete external node with x and return it
- otherwise create an external node with x and an internal to split x and p.key with the value min(x, p.key)

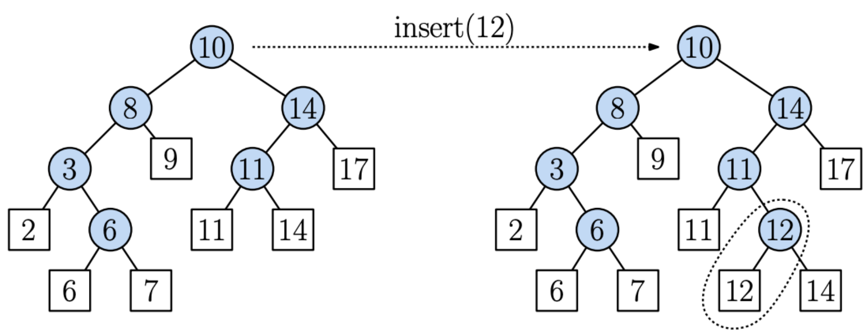


Fig. 4: Inserting a key 12 into an extended binary search tree. A search for 12 leads to the external node 14. Two nodes are created, one external node containing 12 and one internal node containing the minimum of 12 and 14.

Delete(Key x, Node p): returns the root of the updated subtree from which x is deleted. Initial call delete(x, root) then

- traverse through left or right subtree accordingly
- if x is root, delete the external node
- otherwise delete the external node and its parent and return a reference to the other child of parent

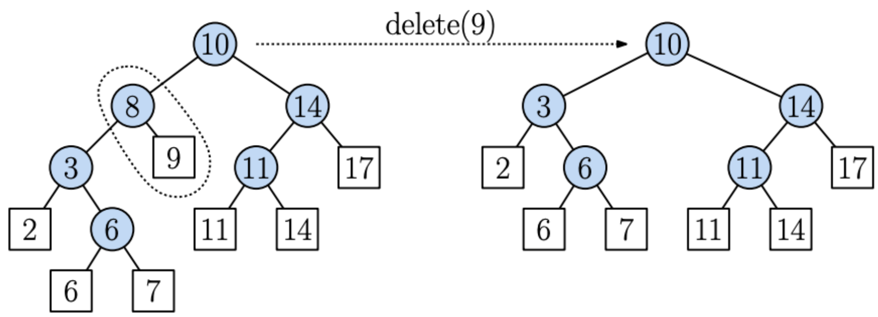


Fig. 5: Deleting a key 9 from an extended binary search tree. After finding the external node containing 9, we remove it and its parent, and link the other child of the parent into the grandparent.

Expected height is O(logn) so dictionary operations are expected to take O(logn)

10.2 Scapegoat Tree

Do not rely on rotations. Instead rebuild subtrees with poor balance. Nodes only contain key, value, left, and right and height will always be O(logn)

Scapegoat Tree nodes have to store 2 values: n (number of keys in the tree) and m (upper bound size of tree)

- Whenever we insert a key, we increment both n and m
- Whenever we delete a key, we only decrement n and do not change m
- This means that $m \leq n$
- Whenever $m > 2n$ that means number of deletes exceeds number of nodes in subtree so we rebuild it

Rebuild: Perform inorder traversal over all nodes in the scapegoat subtree rooted at p with k nodes and stored result in an array. Then recursively extract mins from the array. This will take O(k)

```
BinaryNode buildSubtree(Key[] A, int i, int k) {
  if (k == 0) return null
  else {
    int m = ceiling(k/2);
    BinaryNode p = new BinaryNode(A[i+m]);
    p.left = buildSubtree(A, i, m);
    p.right = buildSubtree(A, i+m+1, k-m-1);
    return p;
  }
}
```

Find(Key x): Exactly the same as BST find. Height of tree will never exceed $\log_{3/2} n$ so run time is guaranteed to be O(logn)

Deletion(Key x): Standard BST delete but when $m > 2n$ rebuild entire tree and set $m = n$

Insertion: Standard BST insert but need to

- monitor the depth of the inserted node. If depth $> \log_{3/2} m$ that means one of the nodes in its search path is unbalanced
- Traverse up through search path and if at any point $\frac{size(u.child)}{size(u)} > \frac{2}{3}$ then we rebuild u
- Intuition is that this subtree has twice as many nodes as siblings so we should trigger a rebuild

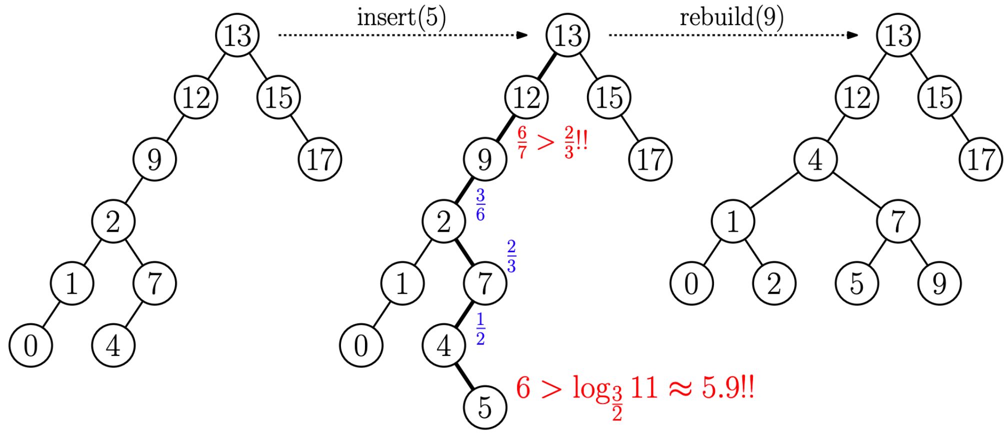


Fig. 6: Inserting a key into a scapegoat tree, which triggers a rebuilding event. The node containing 9 is the first scapegoat candidate encountered while backtracking up the search path and is rebuilt.

Proof that if the depth of a node p is $> log_{3/2}n$ then p has an ancestor (possibly p itself) that is a scapegoat candidate

- Proof by contradiction: assume that all nodes in the path have $size(u.child) \leq \frac{2}{3}size(u)$
- since the root has n nodes, we have $size(p) \geq (\frac{2}{3}^k)n$ for the inserted node p
- since size(p) $\geq$ 1 we have $1 \geq (\frac{2}{3}^k)n \rightarrow k \leq log_{3/2}n$ which is a contradiction since p's depth $> log_{3/2}n$

Calculating size(u): we need to find the size of both of u's children (u' and u") so now size(u) = 1 + u' + u". However, cost of counting process can be accounted for in the rebuilding process (we have to count these nodes anyways)
Alternate idea of counting: store size value in nodes so that

$$\text{size(u)} = (\text{u} == \text{null} \ ? \ 0 : \text{size(u.left)} + \text{size(u.right)});$$

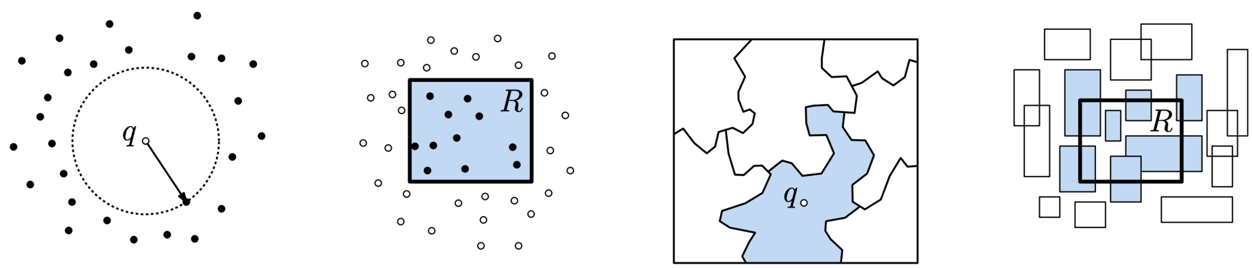
Height can also be updated with this idea with

$$\text{height(u)} = (\text{u} == \text{null} \ ? \ 0 : 1 + \max(\text{height(u.left)}, \text{height(u.right)}))$$

11 Geometric Data Structures

Goal is to store large datasets of geometric objects(points, lines, shapes) in order to answer queries on them. These queries don't focus on exact matches and instead focus on "close to", "contained with", or "overlapping with"

Nearest Neighbor Search: store set of points so giveen a query point q it is possible to find the closest point of the set to q
Range Searching: store set of points so that given a query region R (e.g. circle) it is possible to count all points of set that are in R
Point Location: store subdivision of space into disjoint regions so that given a query point q, it is possible to determine which region has q efficiently
Intersection Searching: store collection of geometric objects so that given a query of objects R of the same type, report all objects in set that intersect with R
Ray shooting: store collection of objects so that given an query ray, ran determine whether ray hits any object in set, and which it hits first



(a) (b) (c) (d)

Fig. 1: Common geometric queries: (a) nearest-neighbor searching, (b) range searching, (c) point location, (d) intersection searching.

Point Representation: Can represent as 2d array with n points of d dimension

$$\text{float[][] p = new float[n][d]}$$

Can also represent using a Point class

```
public class Point {
  private float[] coord; // coordinate storage
  public Point(int dim) { ... } // constructor

  public int getDim {return coord.length} //dimension of point
  public float get(int i) {return coord[i]}

  public boolean equals(Point p) { ... }
  public float distanceTo(Point p) { ... }
}
```

Point quadtree: Generalizing for a 2d coordinate plane, each node has 4 children (NW, NE, SW, SE). Traverse through tree based on coordinate x and y comparison. This creates rectangle regions called cells that can be recursively split. Can be extended to generalize points with d dimensions using 2^d child nodes but that takes up a lot of space

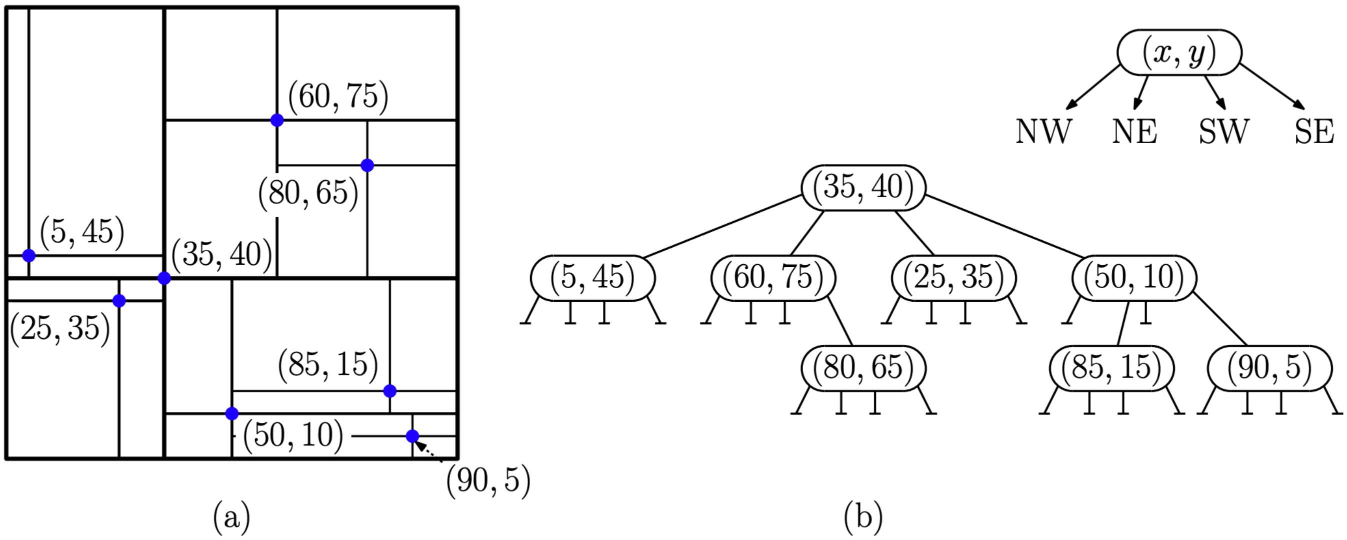


Fig. 2: Point quadtree.

Point kd-trees: represent multi-dimensional points with using a binary tree. Whenever a new point is inserted into a node, we split the cell by a splitting line (0 to d-1). Cutting dimension is chosen by alternating among possible axes at each new level of the tree.

```
class KDNode {
  Point point;   // splitting point
  int cutDim;    // cutting dimension
  KDNode left;
  KDNode right;

  KDNode(Point point, int cutDim) {
    this.point = point;
    this.cutDim = cutDim;
    left = right = null;
  }

  boolean inLeftSubTree(Point x) { // is in left subtree
    return x[cutDim] < point[cutDim]
  }
}
```

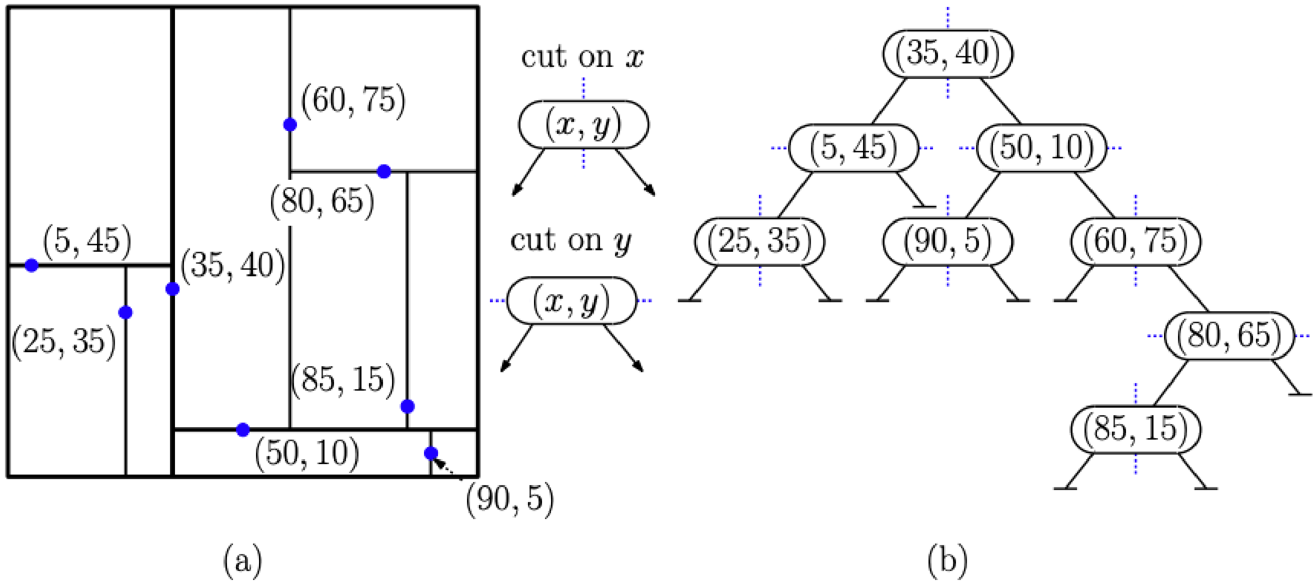


Fig. 3: Point kd-tree decomposition.

Insertion into kd-tree: Same as BST. When we create a node containing the point, we assign its cutting dimension. Initial call root = insert(x, root, 0).

```
KDNode insert(Point x, KDNode p, int cutDim) {
  if (p == null) p = new KDNode(x, cutDim);
  else if (p.point.equals(x)) throw Exception("duplicate point");
  else if (p.inLeftSubtree(x)) p.left = insert(x, p.left, (p.getDim + 1) \% x.getDim());
  else p.right = insert(x, p.right, (p.cutDim + 1) \% x.getDim());
  return p;
}
```

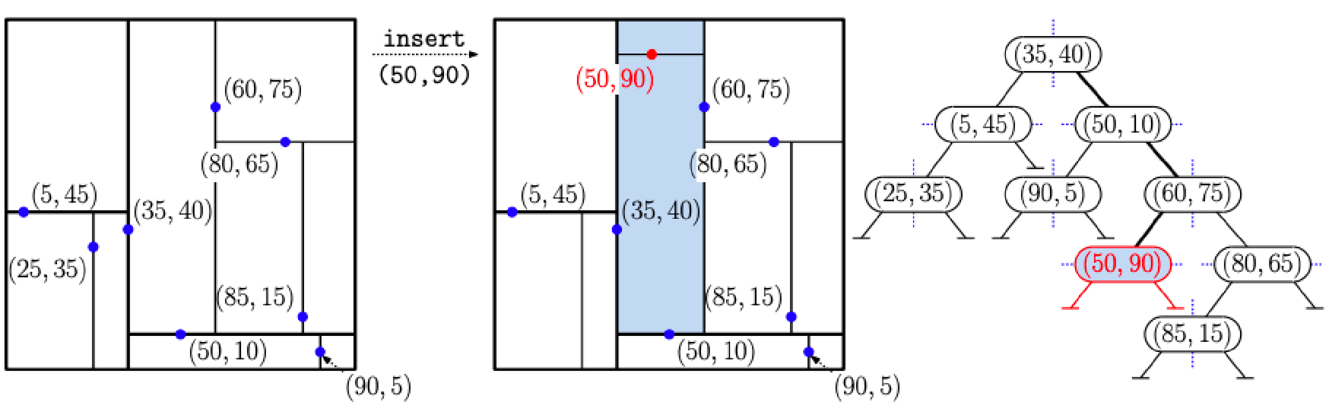


Fig. 4: Insertion into the point kd-tree of Fig. 3.

Deletion in kd-tree: Issue with finding a replacement point since each adjacent level of tree has different cutDim value. Need to use an auxiliary meethod findMin()
findMin(p, cutDim): if the node we are testing has the same cutDim value, then we recurse on left side. Otherwise we have to use another auxiliary method minAlongDim() which returns whicever point p1 or p2 is the smallest among coordinate cuttDim.

```
Point findMin(KDNode p, int i) {
  if (p == null) return null;
  if (p.cutDim == i) {
    if (p.left == null) return p.point;
    else return findMin(p.left, i);
  } else {
    Point q = minAlongDim(p.point, findMin(p.left, i), i);
    return minAlongDim(q, findMin(p.right, i), i);
  }
}

Point minAlongDim(Point p1, Point p2, int i) {
  if (p2 == null || p1[i] <= p2[i]) return p1;
  else return p2;
}
```

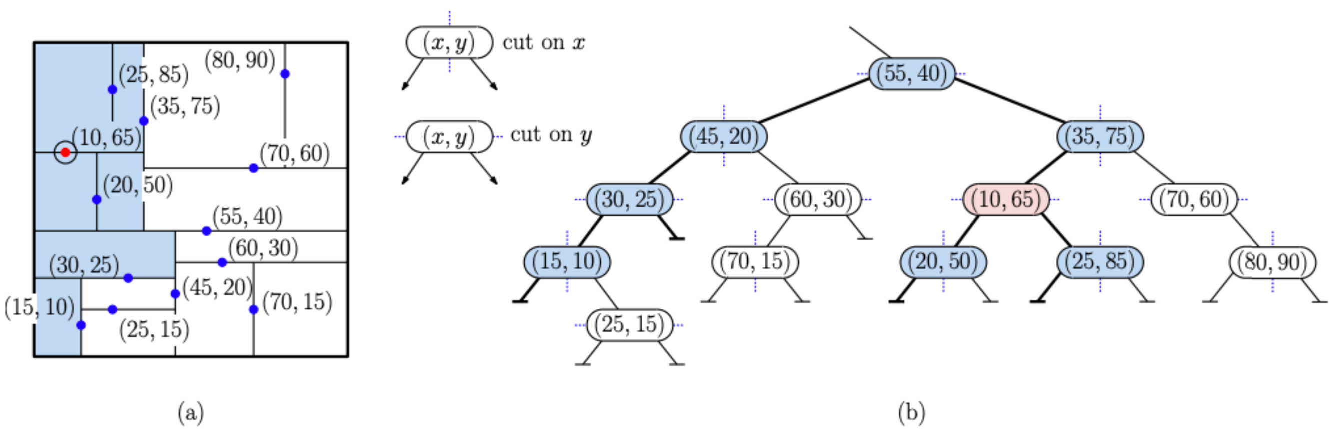


Fig. 5: Example of **findMin** when $i = 0$ (the x-coordinate) on the subtree rooted at $(55, 40)$. The function returns $(10, 65)$.

Another issue of deletion for kd-tree is that if the deleted node has a single child, we can't just bring that child subtree up since it has a different cutDim parity. Solution is to use findMin() to get the smallest x-coordinate or smallest y-coordinate in the right subtree. If the right subtree is empty then we find the min on the left subtree and and move the left subtree to the right subtree, setting the left-child pointer to null. Issue with finding the max on the left subtree is that if another point has the same coordinate value, it messes up the invariant (left subtree has points $<$ and right subtree has points $\geq$)

```
KDNode delete(Point x, KDNode p) {
  if (p == null) throw Eception("point DNE");
  else if (p.point.equals(x)) {
    if (p.right != null) {
      p.point = findMin(p.right, p.cutDim);
      p.right = delete(p.point, p.right)
    } else if (p.left != null) {
      p.point = findMin(p.left, p.cutDim);
      p.right = delete(p.point, p.left);
      p.left = null;
    } else if (p.inLeftSubTree(x)) p.left = delete(x, p.left);
    else p.right = delete(x, p.right);
    return p;
  }
}
```

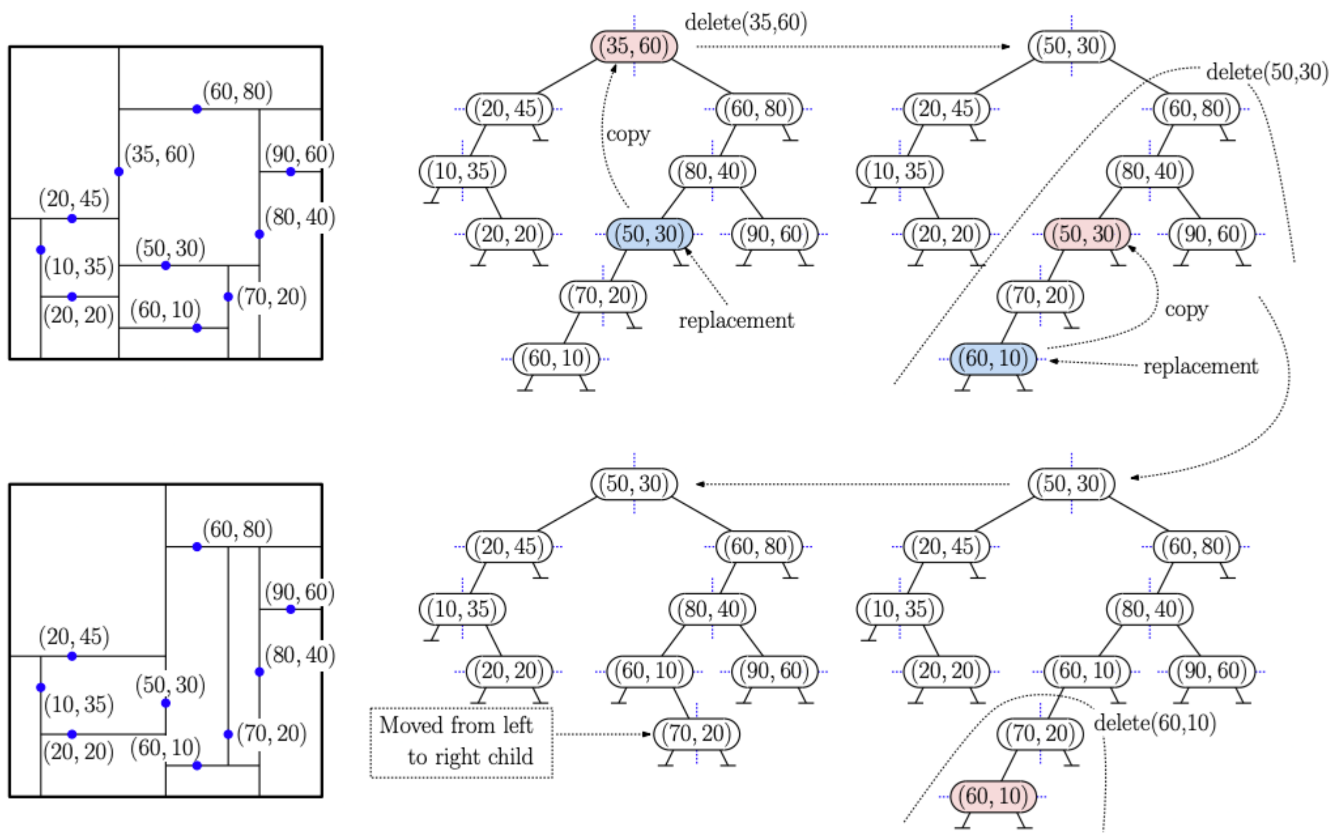


Fig. 6: Deletion from a kd-tree.

Analysis: Storage O(n). We treat the number of dimensions d as a constant so O(dn) = O(n)
Height is O(logn). Since we chose replacement nodes in a biased way height can mutate to be $O(\sqrt{n})$ We can build a kd-tree by choosing the splitting point to be the median according to x-coordinates. Then we split each partition using their medians according to y-coordinates. Keep recursing until done and height will be O(logn). Building takes O(nlogn)

Range Queries: find how many points lie in a given region R. Use a kd-tree to store points. We will need a class for storing multi-dimensional rectangles which consists 2 points (low and high). A point q lies within the rectangle if $low[i] \leq q[i] \leq high[i]$.
Notable methods:
boolean contains(Point q): returns true iff point q is contained within this rectangle
boolean contains(Rectangle c): returns true iff this rectangle contains rectangle c. Have to test containment on all intervals defining each of the rectangles' sides:

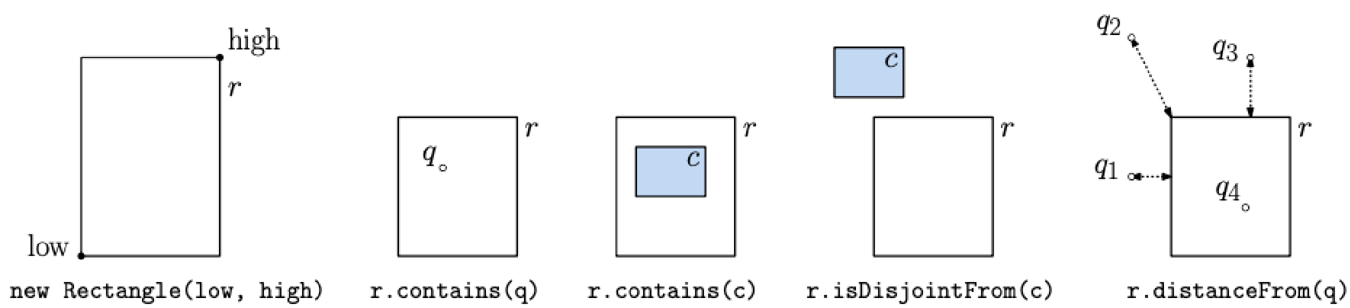
$$[c.low[i], c.high[i]] \in [low[i], high[i]], \text{ for all } 0 \leq i \leq d-1$$

boolean isDisjointFrom(Rectangle c): returns true iff rectangle c is disjoint from this rectangle. Have to test whether any of the defining intervals are disjoint

$$r.high[i] < c.low[i] or r.low[i] > c.high[i], \text{ for any } 0 \leq i \leq d-1$$

float distanceTo(Point q): returns minimum Euclidean distance from q to any point on rectangle. Computed by computing distance from coordinate q[i] to rectangle's ith defining interval

$$\sqrt{\sum_{i=0}^{d-1} (distance(q[i], [low[i], high[i]]))^2}$$



Rectangle leftPart(int cd, Point s): Given a rectangle r and a splitting point s, we want to cut the rectangle into two sub-rectangles by a line that passes through the splitting point. leftPart returns a rectangle whose low point is the same as r.low and whose high point is the same as r.high, except that the cd-th coordinate is set to s[cd].

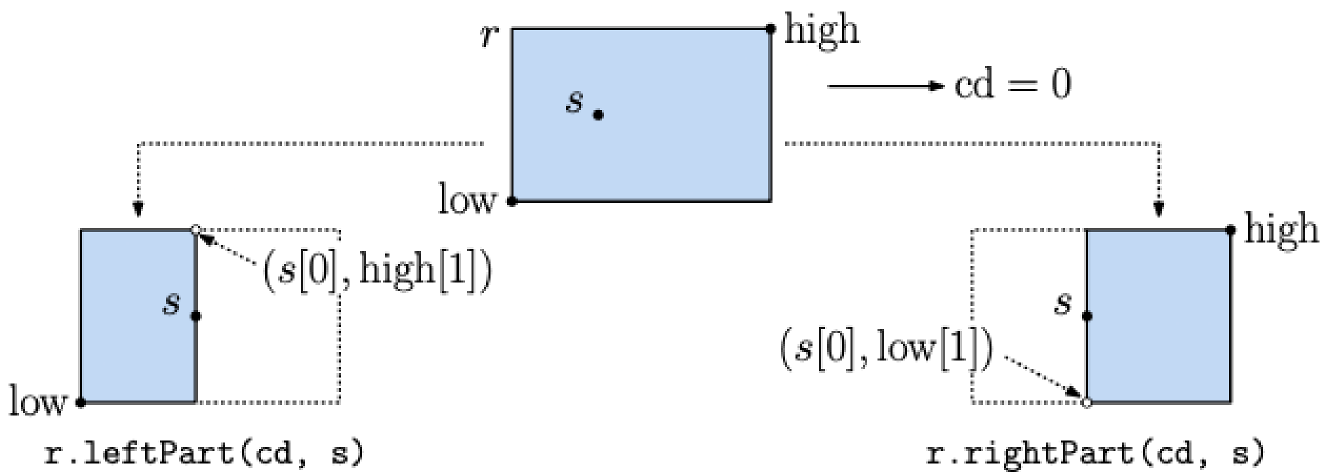


Fig. 3: The functions `leftPart` and `rightPart`.

Answering Range Query: assume that each node p has a member p.size indicating the number of points lying within the tree. This can be easily updated as points are inserted and deleted.
The rangeCount(r, p, cell) function operates recursively, returning a count of the number of points that lie in range r. Initial call is rangeCount(r, root, boundingBox) where boundingBox bounds the entire kd-tree. If the

- if we fall out of tree there is nothing to count
- if the current node's cell is disjoint from the query range, return 0
- if query range completely contains cell, return p.size
- if the range partially overlaps the cell, apply function recursively to each of our two children

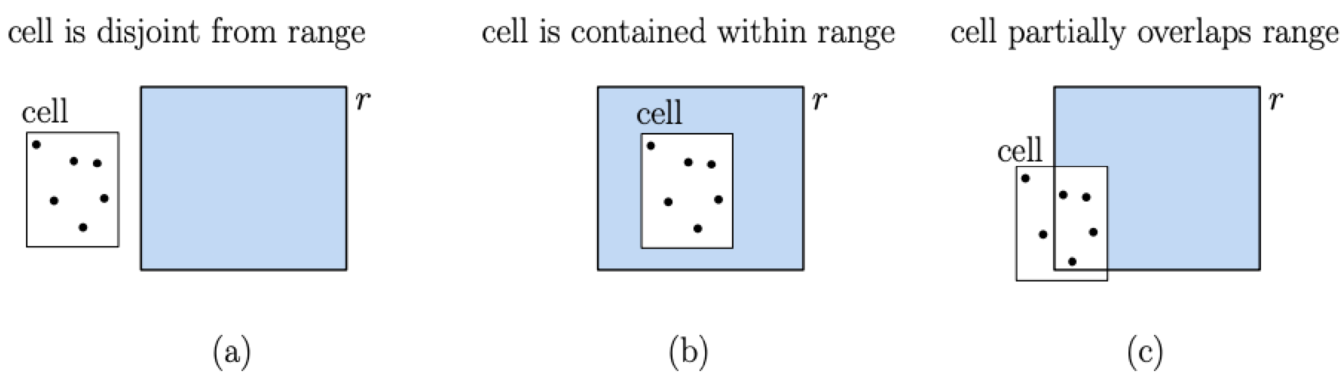


Fig. 4: Cases arising in orthogonal range searching.

```java
int rangeCount(Rectangle r, KDNode p, Rectangle cell) {
  if (p == null) return 0;
  else if (r.isDisjointFrom(cell)) return 0;
  else if (r.contains(cell)) return p.size;
  else {
    int count = 0;
    if (r.contains(p.point)) count++;
    count += rangeCount(r, p.left, cell.leftPart(p.cutDim, p.point));
    count += rangeCount(r, p.right, cell.rightPart(p.cutDim, p.point));
    return count;
  }
}
```

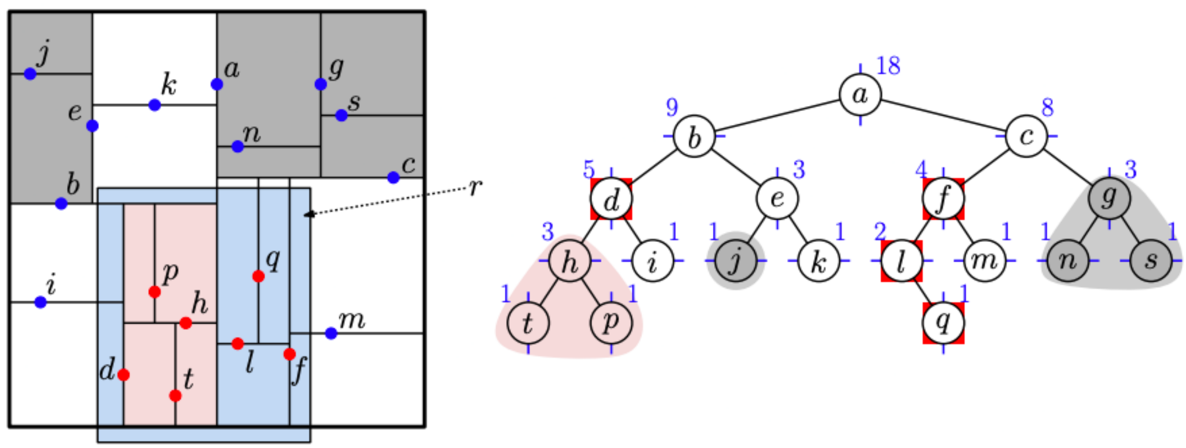


Fig. 5: Range search in kd-trees. The subtree rooted at h is counted entirely. The subtrees rooted at k and g are excluded entirely. The other points are checked individually.

Range Queries can be answered in $O(\sqrt{n})$ time

A node is processed (both children visited) iff the call overlaps the range without being contained within the range (stabbed by the query). To bound total number of nodes processed in search, we count the total number of nodes whose cells are stabbed by the query rectangle.

Given any balanced kd-tree with n points, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

Since the tree is balanced, its height is O(logn) ≈ lgn

Consider only stabbing using vertical or horizontal lines instead of rectangles. When processing a node, the line will stab one of the two children. This means when alternating between x and y, at most two of the possible four grandchildren of each node is stabbed.

Since we have an exponentially increasing number, total sum is dominated by the last term

$$2^{h/2} \approx 2^{(lgn)/2} = \sqrt{n}$$

For the rectangle case, there are 4 sides to the rectangle so $O(4\sqrt{n})$

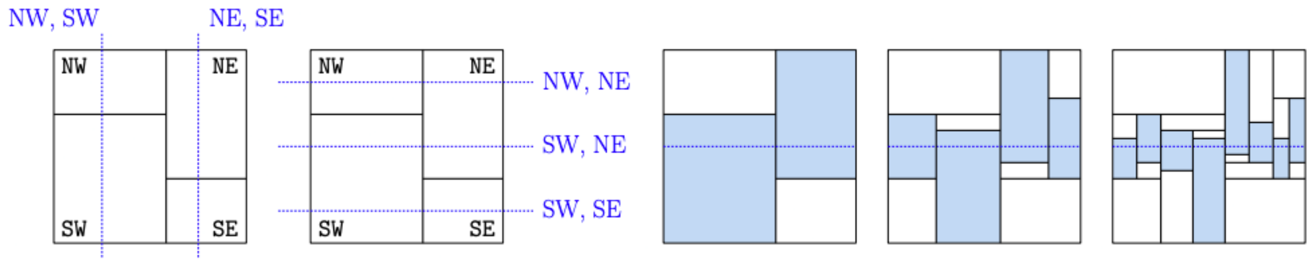


Fig. 6: An axis-parallel line in 2D can stab at most two out of four cells in two levels of the kd-tree decomposition. In general, it stabs 2^i cells at level $2i$.

Nearest-Neighbor Queries: given a set of points P storedi n a kd-tree and a query point q, return the point of P closest to q. Assume that distances are measured using Euclidea distances.

$$dist(p, q) = \sqrt{(p_1 - q_1)^2 + ... + (p_d - q_d)^2}$$

May need to visit every leaf that overlaps with the range. However, number of nodes overlapping the range is usually much smaller than the total number of nodes in the tree.

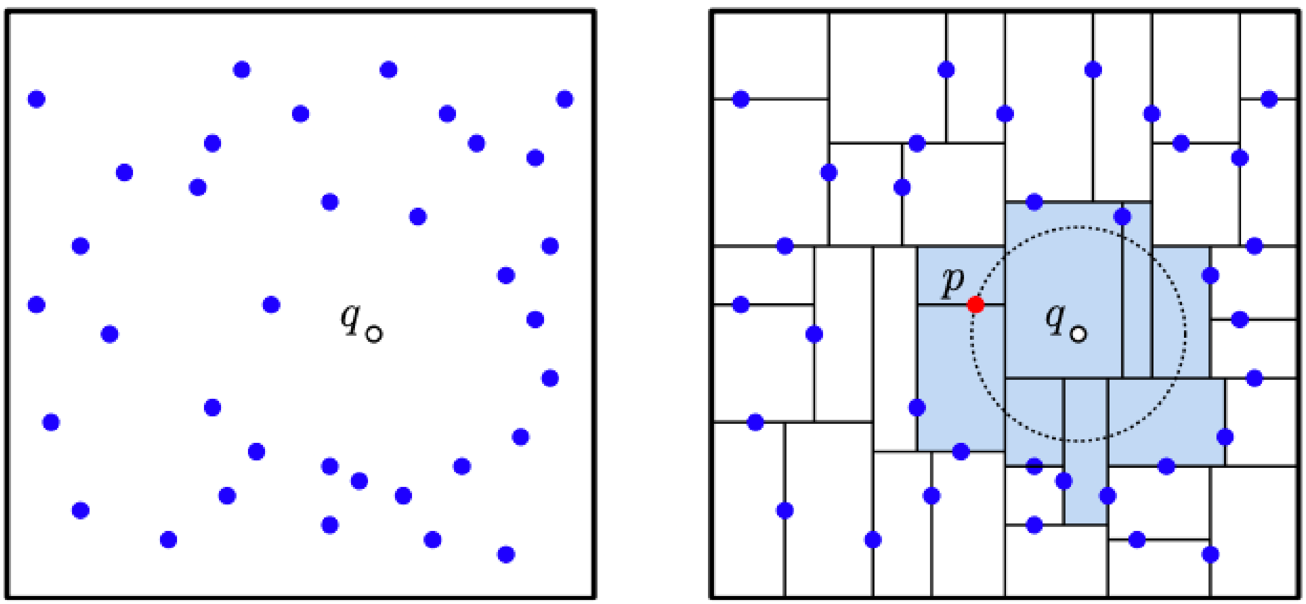


Fig. 7: Nearest-neighbor searching using a kd-tree.

Another approach would be to find the leaf node of the kd-tree that contains q and then search this and its neighboring cells in the kd-tree. However, there might be an issue that the nearest neighbor may actually be very far away. Three things to consider for range and nearest neighbor processing:

- Partial results: store the intermediate results of query and update results as query proceeds
- Traversal order: visit the subtree first that is more likely to be relevant
- Pruning: Don't visit any subtree that is irrelevant

How Nearest-neighbor Code works:

- If p is null so just return bestDist
- Compute the distance from p.point to q, and update bestDist as necessary
- either recurse of leftPart or rightPart accordingly

```
float nearNeighbor(Point q, KDNode p, Rectangle cell, float bestDist) {
  if (p != null) {
    float thisDist = q.distanceTo(p.point);
    bestDist = Math.min(this.Dist, bestDist);

    int cd = p.cutDim;
    Rectangle leftCell = cell.leftPart(cd, p.point);
    Rectangle rightCell = cell.rightPart(cd, p.point);

    if (q[cd] < p.point[cd]) {
      bestDist = nearNeighbor(q, p.left, leftCell, bestDist);
      if (rightCell.distanceTo(q) < bestDist) bestDist = nearNeighbor(q, p.right, rightCell, bestDist);
    } else {
      bestDist = nearNeighbor(q, p.right, rightCell, bestDist);
      if (leftCell.distanceTo(q) < bestDist) bestDist = nearNeighbor(q, p.left, leftCell, bestDist);
    }
  }
  return bestDist;
}
```

Worst case is we need to visit every node but this is typically not the case. Usually $O(2^d + logn)$ where d is the dimension of the space. Intuition is that we need to visit some set of nodes in the neighborhood and require logn to descend tree

12 Memory Management

2 principles to dynamic memory allocation.

- Stack where arguments and local variables are pushed onto when a procedure is called. These are popped when a procedure returns.
- Heap where objects are allocated and deallocated, fragmenting the heap into pieces.

Issues with memory allocation:

- Explicit deallocation (languages like C) are a burden for programmers. Blocks of memory may be leaked (inaccessible and has not be deallocated). Another issue of aliasing where two pointers refer to the same block of memory leading to the issue of when one pointer is released, the other still references that chunk of memory
- Implicit deallocation (Languages like Java) the system determines which objects are no longer accessible using garbage collection but put a burden on the memory allocation system

Explicit Allocation/Deallocation: one case where explicit deallocation is easy to handle: when all objects being allocated are of the same size (array). Very easy to allocate a contiguous space in heap and blocks are linked together. However for objects of varying sizes, we can run into an issue of external fragmentation, essentially wasting space
Explicit allocation works in that when allocating a block, we search through the list of available blocks of memory that is large enough. There are two strategies to this:

- First-fit: search available blocks sequentially until find a large enough space
- Best-fit: search all available blocks and insert into smallest block large enough

Best-fit usually performs worse than first-fit (first-fit is faster to execute and best-fit tends to produce a large amount of fragmentation by selecting blocks taht are barely larger than the request size, leaving tiny residual blocks). One possible solution to reduce fragmentation is just make the requested block take up all the available space, keeping the list of available blocks free of tiny fragments and speeding up search time. However this creates internal fragmentation. When deallocating a block, we should merge available space to create large blocks of free space (this is called merging) but there's an issue of figuring out how to efficiently merge.

For each allocated blocks we record its size, inUse, and prevInUse
For each available block we store these blocks in a doubly-linked circular list (avail) and has size, inUse, prevInUse, prev , next, and size2.

Allocation: to allocate a block, search through linked list of available blocks until find one of sufficient size. If the size pretty much fits, remove the block from the list of available blocks, updating linkage and prevInUse as necessary. If desired allocation is much smaller than the block chosen, split the block into two smaller blocks and relink as necessary.

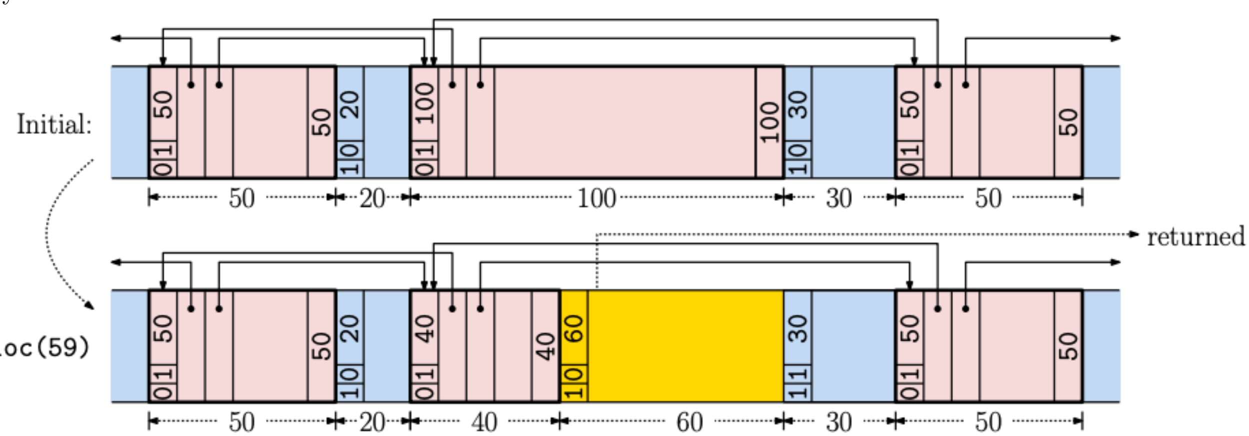


Fig. 3: An example of block allocation.

```
(void *) alloc(int b) {
  b += 1;      // extra space for system overhead
  p = search available space list for block of size at least b;
  if (p == null) throw Exception("Too small");
  if (p.size - b < TOO\_SMALL) {
    avail.unlink(p);
    q = p;
  } else {
    p.size -=b;
    *(p + p.size - 1) = p.size; // set new block's size2 field
    q = p + p.size;
    q.size = b;
    q.prevInUse = 0;
  }
  q.inUse = 1;
  (q + q.size).prevInUse = 1;
  return q + 1;  //offset link to avoid header
}
```

Deallocation: to deallocate we check if the next block or preceding blocks are avaliable. For the next block we can find it's first word and check its inUse field. For the preceding block we just use the prevInUse field.
If the prev block isn't in use, we use the size value stored in the last word to find the header values appropriately. If either of these blocks is available, merge the two blocks and update header value appropriately. If both are available, we need to delete one of these blocks from available space list. If both are in-use, we link current block to list of available blocks.
There are four cases to consider for deallocation

- If the following block q is available, merge with p. Update p's size and move q's record in the available space list to p using move(p,q), copying q's previous and next fields to p and appropriately update the entries in the available space list that point to q. Otherwise add p to available space list
- If preceding block isn't in use, merge this block with p and unlink p from the available space list

---

```
void dealloc(void* p) {
  p--;     // back up to the header;
  q = p + p.size; // pointer to following block
  if (!q.inUse) {
    p.size += q.size;
    avail.move(q, p);
  } else avail.insert(p)

  p.inUsee = 0;
  *(p + p.size = 1) = p.size;

  if (!p.prevInUse) {
    q = p - *(p-1);
    q.size += p.size;
    *(q + q.size - 1) = q.size;
    avail.unlink(p);
    (q + q.size).prevInUse = 0;
  }
}
```

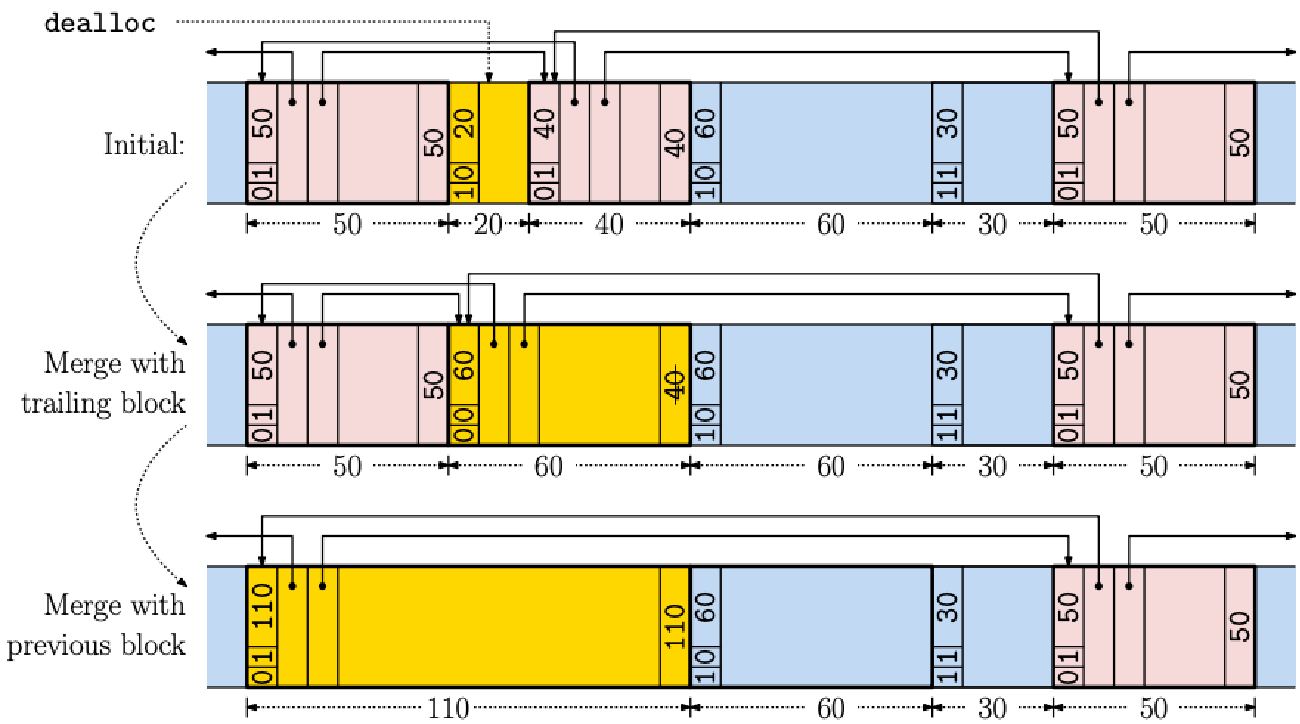


Fig. 4: An example of block deallocation and merging.

Analysis: typically methods achieve utilization of around 2/3 storage before failing to satisfy a request so in general allocate a heap of at last 10 times size of largest block to be allocated

Buddy System: Long sequences of allocations and deallocations of objects of varius sizes tends to result in highly fragmented space. We can minimize this by limiting the possible sizes of blocks and their positions using well-structured allocation. Because block sizes are limited, internal fragmentation becomes an issue.
Start with a block of memory whose size is a power of 2 and then hierarchically subdivide each block into blocks of equal sizes.

- Sizes of all blocks are powers of 2. A request is artificially rounded up to the next power of 2
- Blocks of size 2^k start at memory addresses that are multiples of 2^k

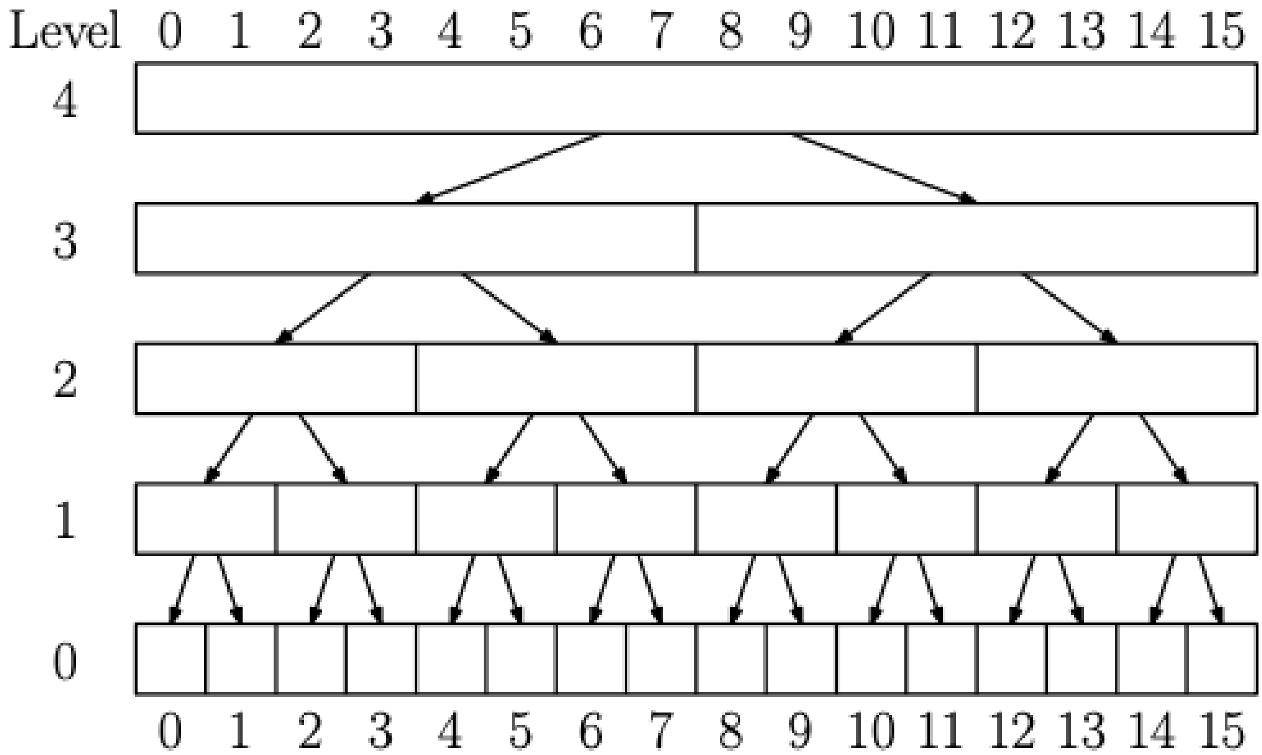


Fig. 5: Buddy system block structure.

For every block, there is exactly 1 other block with which this block can be merged (called the buddy)

$$buddy_k(x) = \begin{cases} x + 2^k & if 2^{k+1} divides x \\ x - 2^k & Otherwise \end{cases}$$

Maintain an array of linked lists, one for the available block list for each size group
Allocation will work in that we will allocate a block of size 2^k. If size doesn't fit then we use the smallest available block of the largest size, remove it from the available space list, and recursively split it into subblocks until we get the desired size.

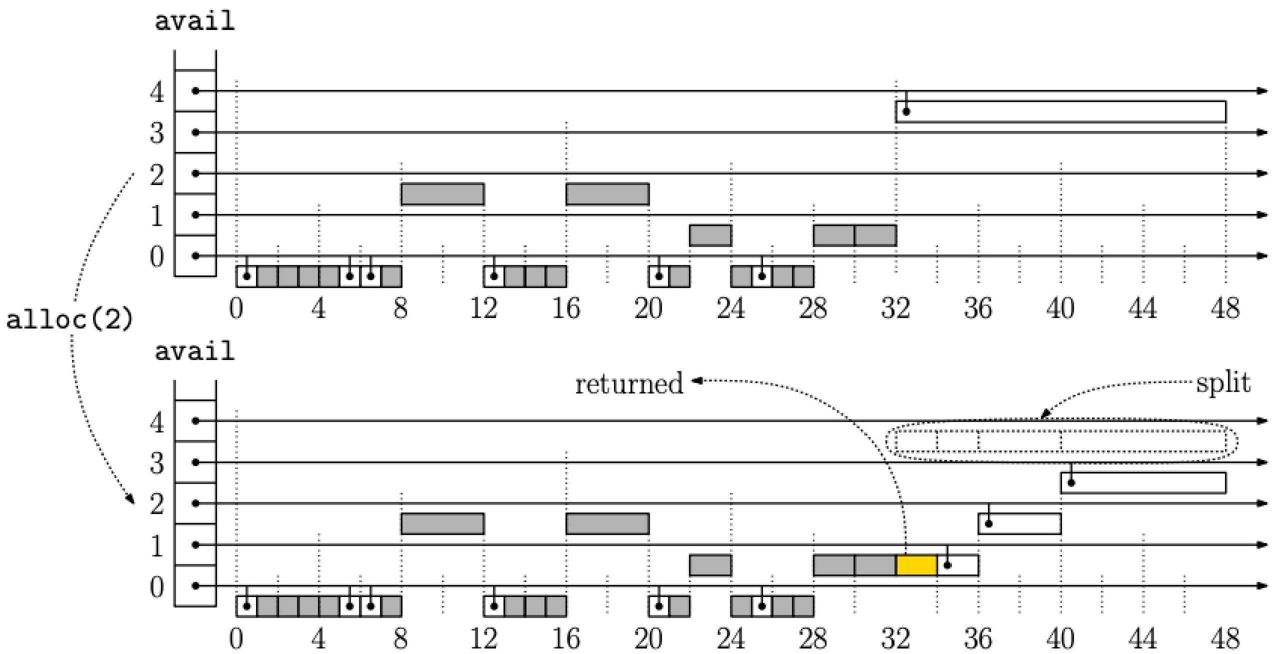


Fig. 7: Example of allocating a block in the buddy system.

Deallocation: we first mark this block as being available then we check to see if its buddy is available and merge if so. Recursively continue until we find that the buddy is allocated

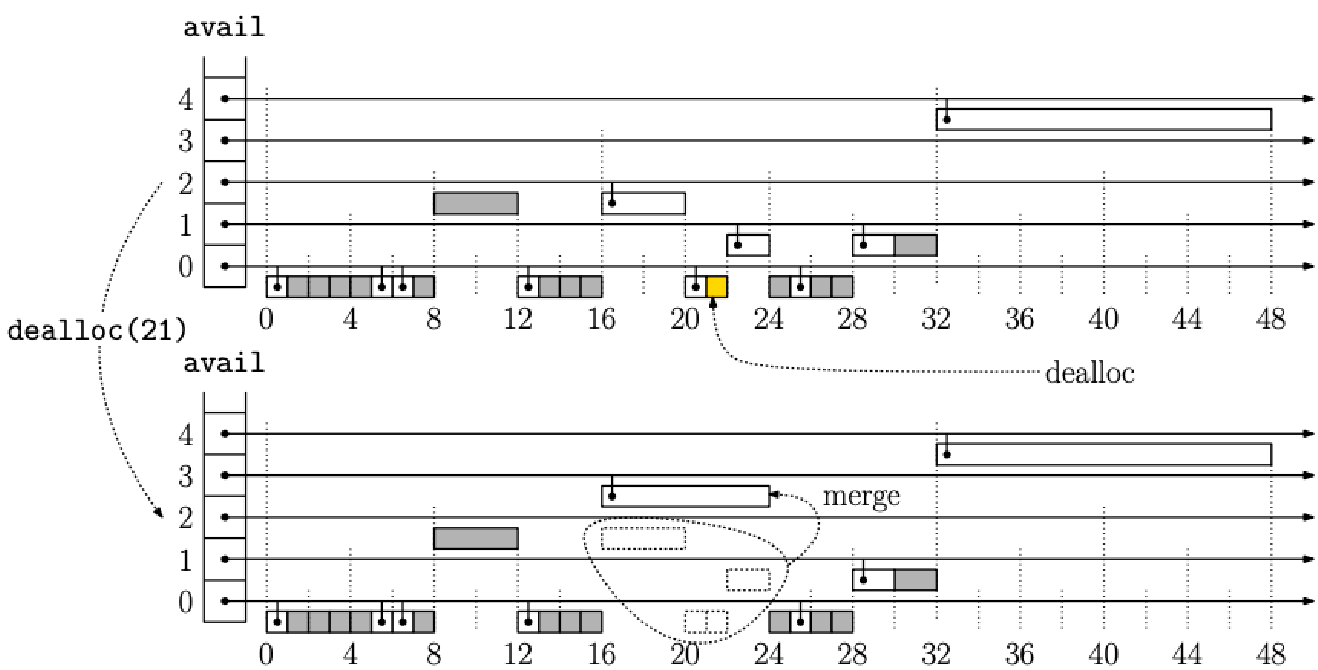


Fig. 8: Example of deallocating a block in the buddy system.

Can use other number systems for buddy system (e.g. Fibonacci numbers)