CSCI 104

Rafael Ferreira da Silva
rafsilva@isi.edu

Slides adapted from: Mark Redekopp

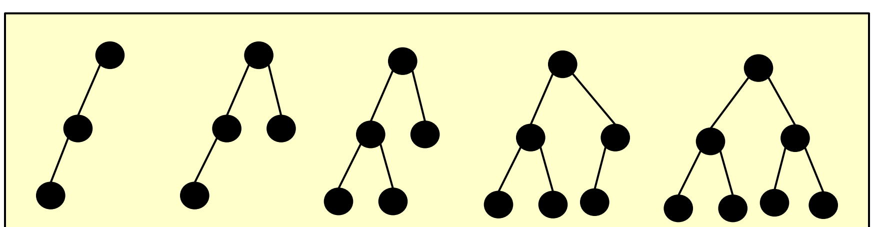
It is strongly recommended to be aware of the runtime (or expected runtime) of *insert/remove/search* for mostly data structures and algorithms

FINAL REVIEW

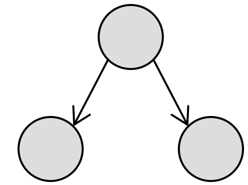
HEAPS

Binary Tree Review

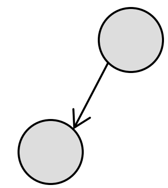
- Full binary tree: Binary tree, T, where
  - If height h>0 and both subtrees are full binary trees of height, h-1
  - If height h==0, then it is full by definition
  - (Tree where all leaves are at level h and all other nodes have 2 children)
- Complete binary tree
  - Tree where levels 0 to h-1 are full and level h is filled from left to right
- Balanced binary tree
  - Tree where subtrees from any node differ in height by at most 1



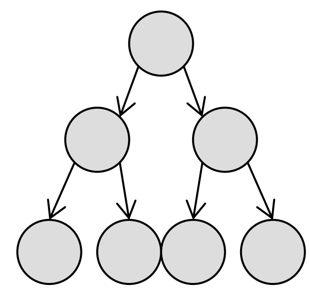
DAPS, 6th Ed. Figure 15-8



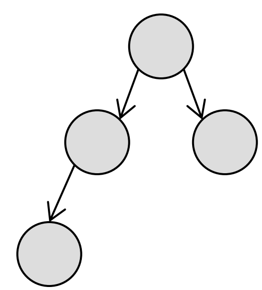
Full



Complete, but not full



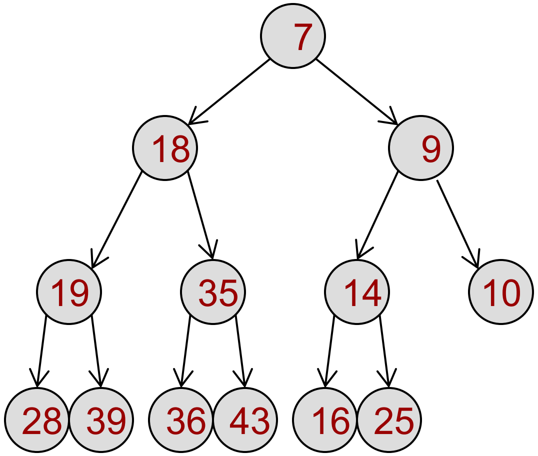
Full



Complete

Heap Data Structure

- Provides an efficient implementation for a priority queue
- Can think of heap as a ***complete*** binary tree that maintains the **heap property**:
  - Heap Property: Every parent is less-than (if min-heap) or greater-than (if max-heap) ***both*** children
  - But no ordering property between children
- Minimum/Maximum value is always the top element



Min-Heap

Heap Operations

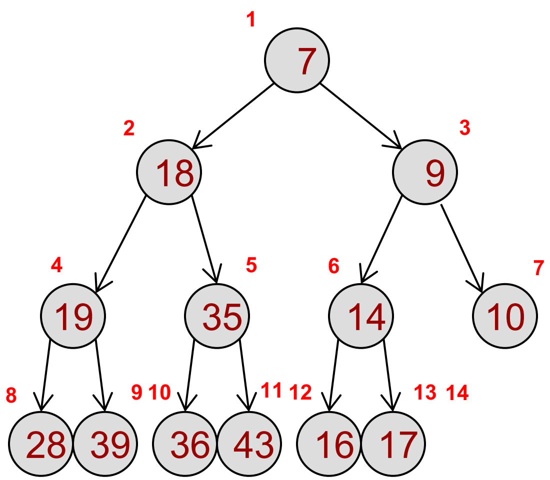
- Push: Add a new item to the heap and modify heap as necessary
- Pop: Remove **min/max** item and modify heap as necessary
- Top: Returns **min/max**
- Since heaps are complete binary trees we can use an array/vector as the container

```
template <typename T>
class MinHeap
{
public:
    MinHeap(int init_capacity);
    ~MinHeap()
    void push(const T& item);
    T& top();
    void pop();
    int size() const;
    bool empty() const;

private:
    void heapify(int idx);
    vector<T> items_;
}
```

Array/Vector Storage for Heap

- Recall: **Full binary tree** (i.e. only the lowest-level contains empty locations and items added left to right) can be modeled as an **array** (let's say it starts at index 1) where:
  - Parent(i) = i/2
  - Left_child(p) = 2*p
  - Right_child(p) = 2*p + 1



0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	9	19	35	14	10	28	39	36	43	16	17

parent(5) = 5/2 = 2

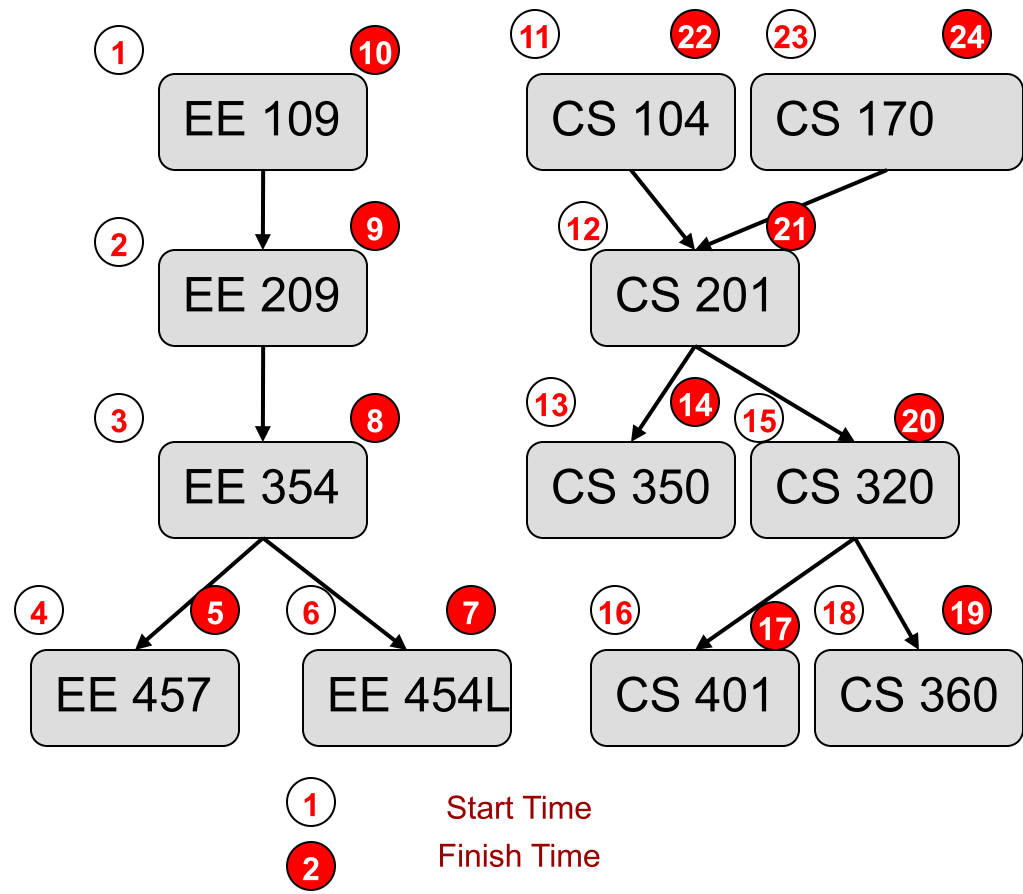
Left_child(5) = 2*5 = 10

Right_child(5) = 2*5+1 = 11

DEPTH FIRST SEARCH

Depth First Search

- Explores ALL children before completing a parent
  - Note: BFS completes a parent before ANY children
- For DFS let us assign:
  - A start time when the node is first found
  - A finish time when a node is completed
- If we look at our nodes in reverse order of finish time (i.e. last one to finish back to first one to finish) we arrive at a…
  - Topological ordering!!!



Reverse Finish Time Order

CS 170, CS 104, CS 201, CS 320, CS 360, CS 477, CS 350, EE 109, EE 209L, EE 354, EE 454L, EE 457

DFS Algorithm

- Visit a node
  - Mark as visited (started)
  - For each visited neighbor, visit it and perform DFS on all of their children
  - Only then, mark as finished
- DFS is recursive!!
- If cycles in the graph, ensure we don't get caught visiting neighbors endlessly
  - Color them as we go
  - White = unvisited,
  - Gray = visited but not finished
  - Black = finished

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

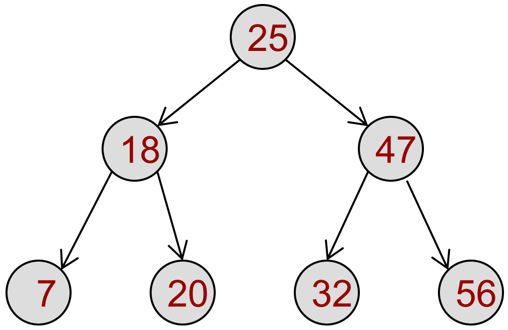
```
DFS-Visit (G, u)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color == WHITE then
       DFS-Visit (G, v)
  u.color = BLACK
  finish_list.append(u)
```

Properties, Insertion and Removal

BINARY SEARCH TREES

Binary Search Tree

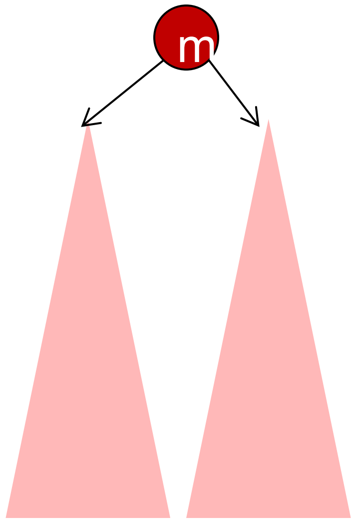
- Binary search tree = binary tree where all nodes meet the property that:
  - All values of nodes in left subtree are less-than or equal than the parent's value
  - All values of nodes in right subtree are greater-than or equal than the parent's value



If we wanted to print the values in sorted order would you use an pre-order, in-order, or post-order traversal?

Successors & Predecessors

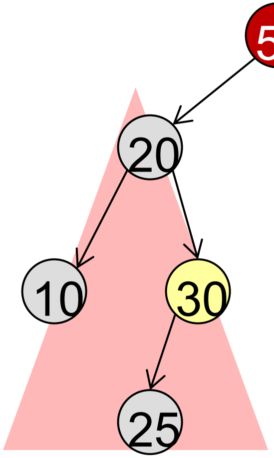
- Let's take a quick tangent that will help us understand how to do **BST Removal**
- Given a node in a BST
  - Its predecessor is defined as the next smallest value in the tree
  - Its successor is defined as the next biggest value in the tree
- Where would you expect to find a node's successor?
- Where would find a node's predecessor?



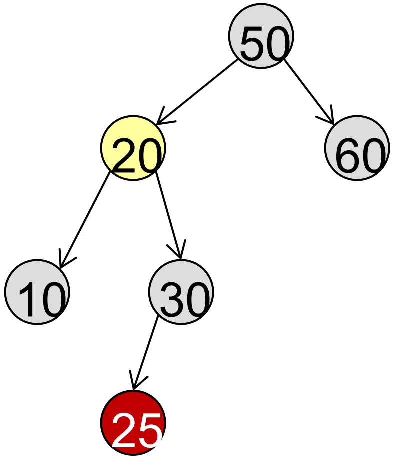
Predecessors

- If left child exists, predecessor is the right most node of the left subtree
- Else walk up the ancestor chain until you traverse the first right child pointer (find the first node who is a right child of his parent…that parent is the predecessor)
  – If you get to the root w/o finding a node who is a right child, there is no predecessor

Pred(50) = 30



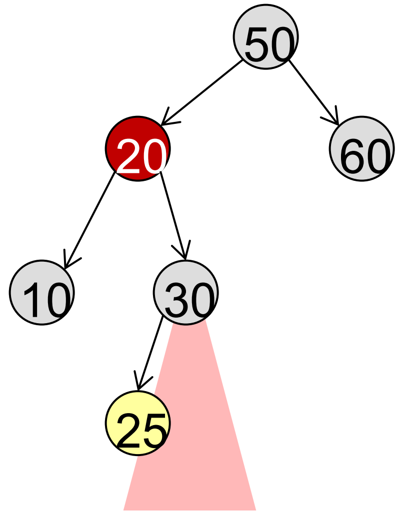
Pred(25)=20



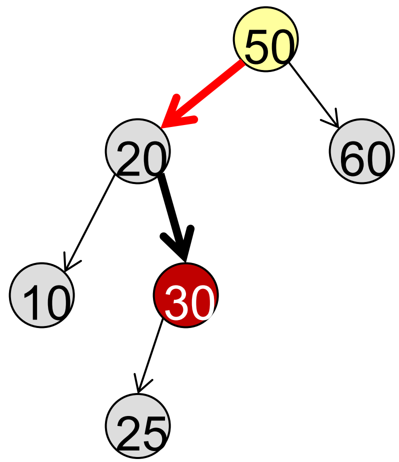
Successors

- If right child exists, successor is the left most node of the right subtree
- Else walk up the ancestor chain until you traverse the first left child pointer (find the first node who is a left child of his parent…that parent is the successor)
  - If you get to the root w/o finding a node who is a left child, there is no successor

Succ(20) = 25

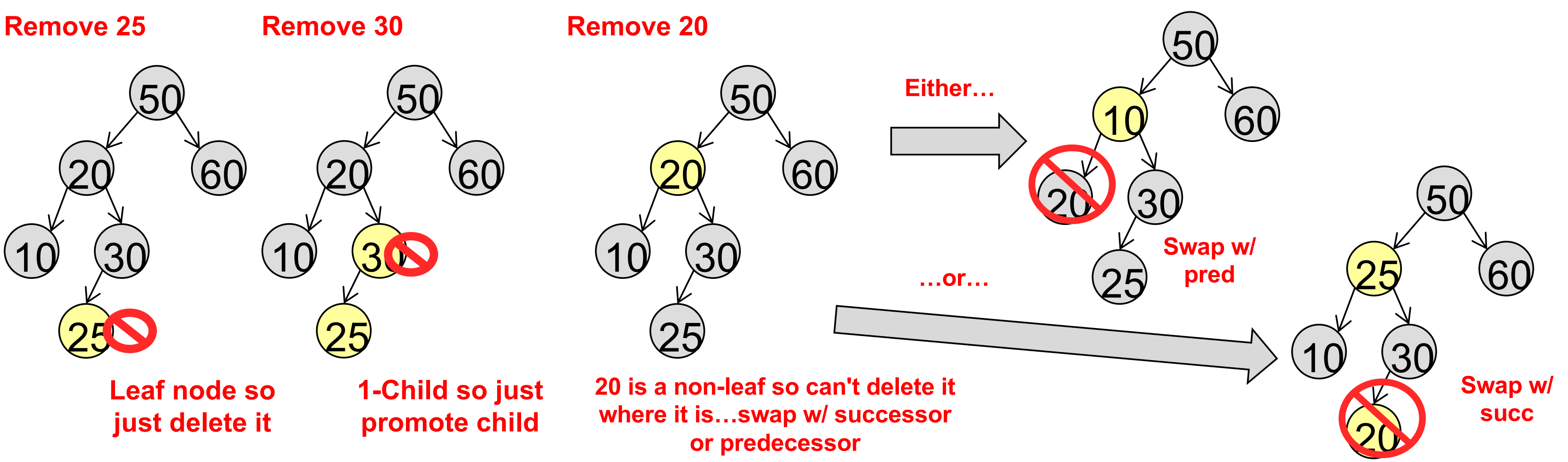


Succ(30)=50



BST Removal

- To remove a value from a BST…
  - First find the value to remove by walking the tree
  - If the value is in a leaf node, simply remove that leaf node
  - If the value is in a non-leaf node, swap the value with its in-order successor or predecessor and then remove the value
    - A non-leaf node's successor or predecessor is guaranteed to be a leaf node (which we can remove) or have 1 child which can be promoted
    - We can maintain the BST properties by putting a value's successor or predecessor in its place

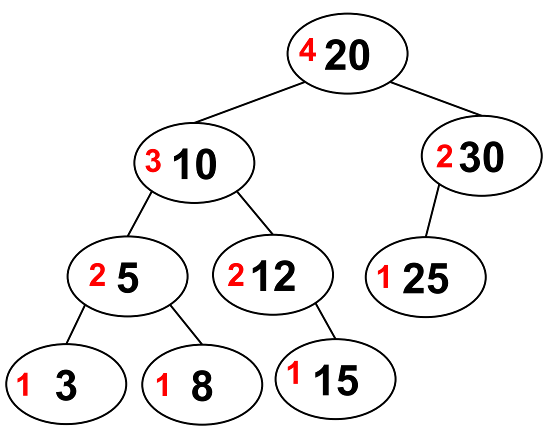


Self-balancing tree proposed by Adelson-Velsky and Landis

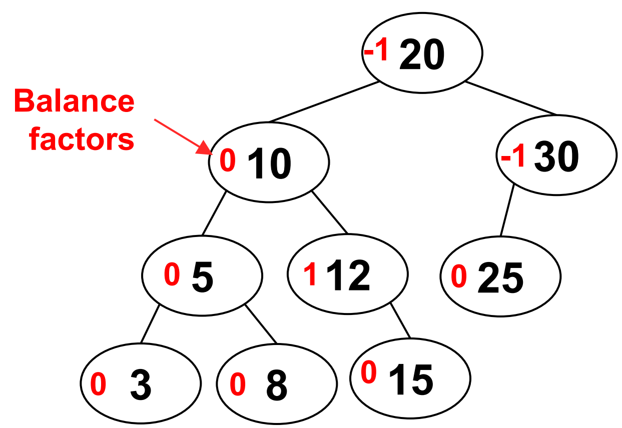
AVL TREES

AVL Trees

- A binary search tree where the **height difference** between left and right subtrees of a node is **at most 1**
  - Binary Search Tree (BST): Left subtree keys are less than the root and right subtree keys are greater
- Two implementations:
  - Height: Just store the height of the tree rooted at that node
  - Balance: Define b(n) as the balance of a node = (Right – Left) Subtree Height
    - **Legal values are -1, 0, 1**
    - Balances require at most 2-bits if we are trying to save memory.
    - **Let's use balance for this lecture.**



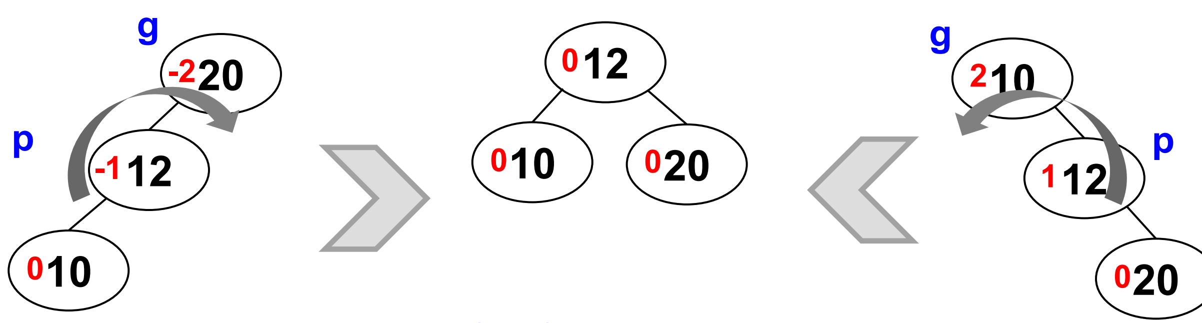
AVL Tree storing Heights



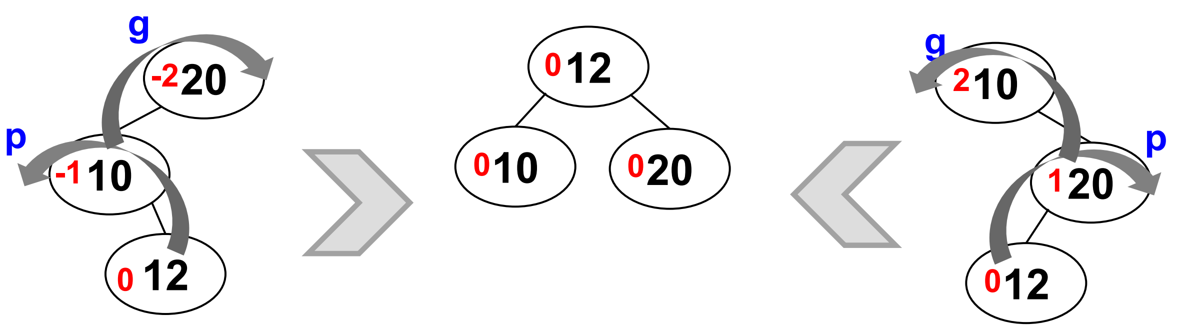
AVL Tree storing balances

To Zig or Zag

- The rotation(s) required to balance a tree is/are dependent on the grandparent, parent, child relationships
- We can refer to these as the zig-zig case and zig-zag case
- Zig-zig requires 1 rotation
- Zig-zag requires 2 rotations (first converts to zig-zig)



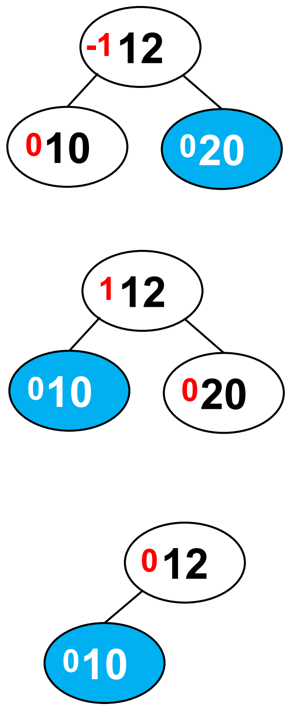
Left-left or Right-right
(a.k.a. Zig-zig)
[One left/right rotation of g]



Left-right or Right-left
(a.k.a. Zig-zag)
[Rotate p then g]

Insert(n)

- If empty tree => set as root, b(n) = 0, done!
- Insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p, n)



Insert-fix(p, n)

- Precondition: p and n are balanced: {+1,0,-1}
- Postcondition: g, p, and n are balanced: {+1,0,-1}
- If p is null or parent(p) is null, return
- Let g = parent(p)
- Assume p is left child of g [For right child swap left/right, +/-]
  - g.balance += -1
  - if g.balance == 0, return
  - if g.balance == -1, insertFix(g, p)
  - If g.balance == -2
    - If zig-zig then rotateRight(g); p.balance = g.balance = 0
    - If zig-zag then rotateLeft(p); rotateRight(g);
      - if n.balance == -1 then p.balance = 0; g.balance(+1); n.balance = 0;
      - if n.balance == 0 then p.balance = 0; g.balance(0); n.balance = 0;
      - if n.balance == +1 then p.balance = -1; g.balance(0); n.balance = 0;

Note: If you perform a rotation, you will NOT need to recurse. You are done!

Remove Operation

- Remove operations may also require rebalancing via rotations
- The key idea is to update the balance of the nodes on the ancestor pathway
- If an ancestor gets out of balance then perform rotations to rebalance
  - Unlike insert, performing rotations does not mean you are done, but need to continue
- There are slightly more cases to worry about but not too many more

Remove

- Let n = node to remove (perform BST find) and p = parent(n)
- If n has 2 children, swap positions with in-order successor and perform the next step
  - Now n has 0 or 1 child guaranteed
- If n is not in the root position determine its relationship with its parent
  - If n is a left child, let diff = +1
  - if n is a right child, let diff = -1
- Delete n and update tree, including the root if necessary
- removeFix(p, diff);

RemoveFix(n, diff)

- If n is null, return
- Let ndiff = +1 if n is a left child and -1 otherwise
- Let p = parent(n). Use this value of p when you recurse.
- If balance of n would be -2 (i.e. balance(n) + diff == -2)
  - [Perform the check for the mirror case where balance(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - If balance(c) == -1 or 0 (zig-zig case)
    - rotateRight(n)
    - if balance(c) == -1 then balance(n) = balance(c) = 0, removeFix(p, ndiff)
    - if balance(c) == 0 then balance(n) = -1, balance(c) = +1, done!
  - else if balance(c) == 1 (zig-zag case)
    - rotateLeft(c) then rotateRight(n)
    - Let g = right(c)
    - If balance(g) == +1 then balance(n) = 0, balance(c) = -1, balance(g) = 0
    - If balance(g) == 0 then balance(n) = balance(c) = 0, balance(g) = 0
    - If balance(g) == -1 then balance(n) = +1, balance(c) = 0, balance(g) = 0
    - removeFix(parent(p), ndiff);
- else if balance(n) == 0 then balance(n) += diff, done!
- else balance(n) = 0, removeFix(p, ndiff)

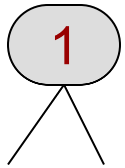
An example of B-Trees

2-3-4 TREES

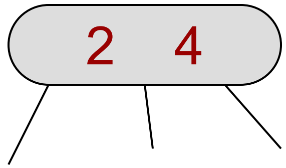
Definition

- 2-3-4 trees are very much like 2-3 trees but form the basis of a balanced, **binary** tree representation called Red-Black (RB) trees which are commonly used [used in C++ STL map & set]
  - We study them mainly to ease understanding of RB trees
- 2-3-4 Tree is a tree where
  - Non-leaf nodes have 1 value & 2 children or 2 values & 3 children or 3 values & 4 children
  - All leaves are at the same level
- Like 2-3 trees, 2-3-4 trees are always full and thus have an upper bound on their height of $\log_2(n)$

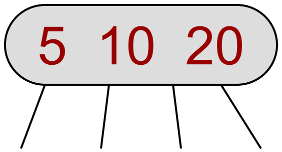
a 2 Node



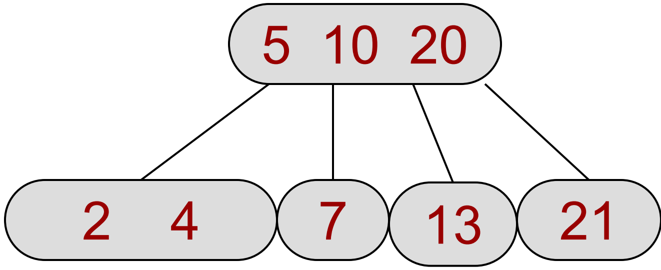
a 3 Node



a 4 Node

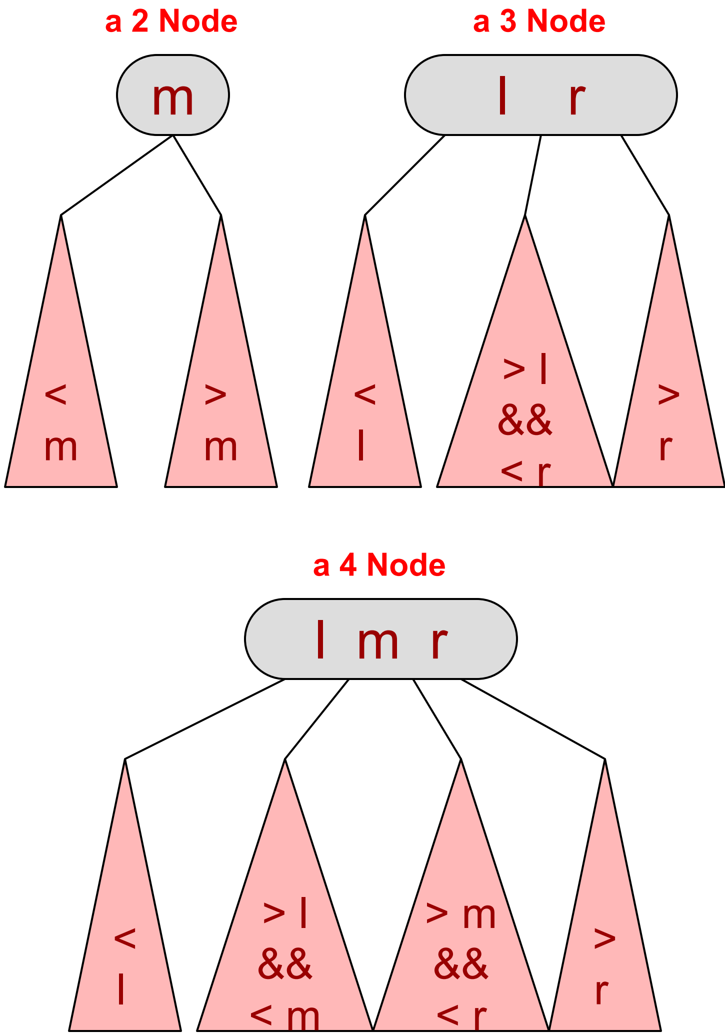


a valid 2-3-4 tree



2-3-4 Search Trees

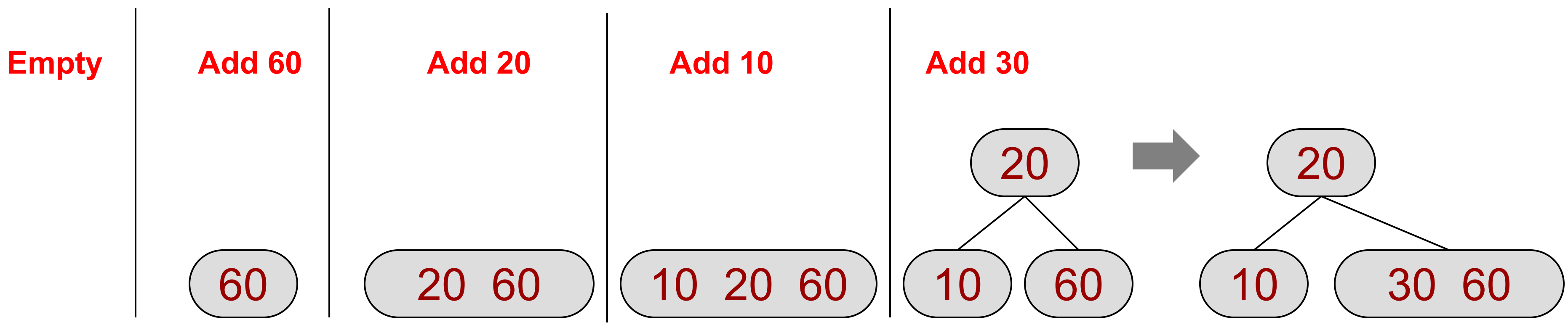
- Similar properties as a 2-3 Search Tree
- 4 Node:
  - Left subtree nodes are < *l*
  - Middle-left subtree > *l* and < *m*
  - Middle-right subtree > *m* and < *r*
  - Right subtree nodes are > *r*



2-3-4 Insertion Algorithm

- Key: Rather than search down the tree and then possibly promote and break up 4-nodes on the way back up, split 4 nodes on the way down
- To insert a value,
  - 1. If node is a 4-node
    - Split the 3 values into a left 2-node, a right 2-node, and promote the middle element to the parent of the node (which definitely has room) attaching children appropriately
    - Continue on to next node in search order
  - 2a. If node is a leaf, insert the value
  - 2b. Else continue on to the next node in search tree order
- Insert 60, 20, 10, 30, 25, 50, 80

Key: 4-nodes get split as you walk down thus, a leaf will always have room for a value



"Balanced" Binary Search Trees

RED BLACK TREES

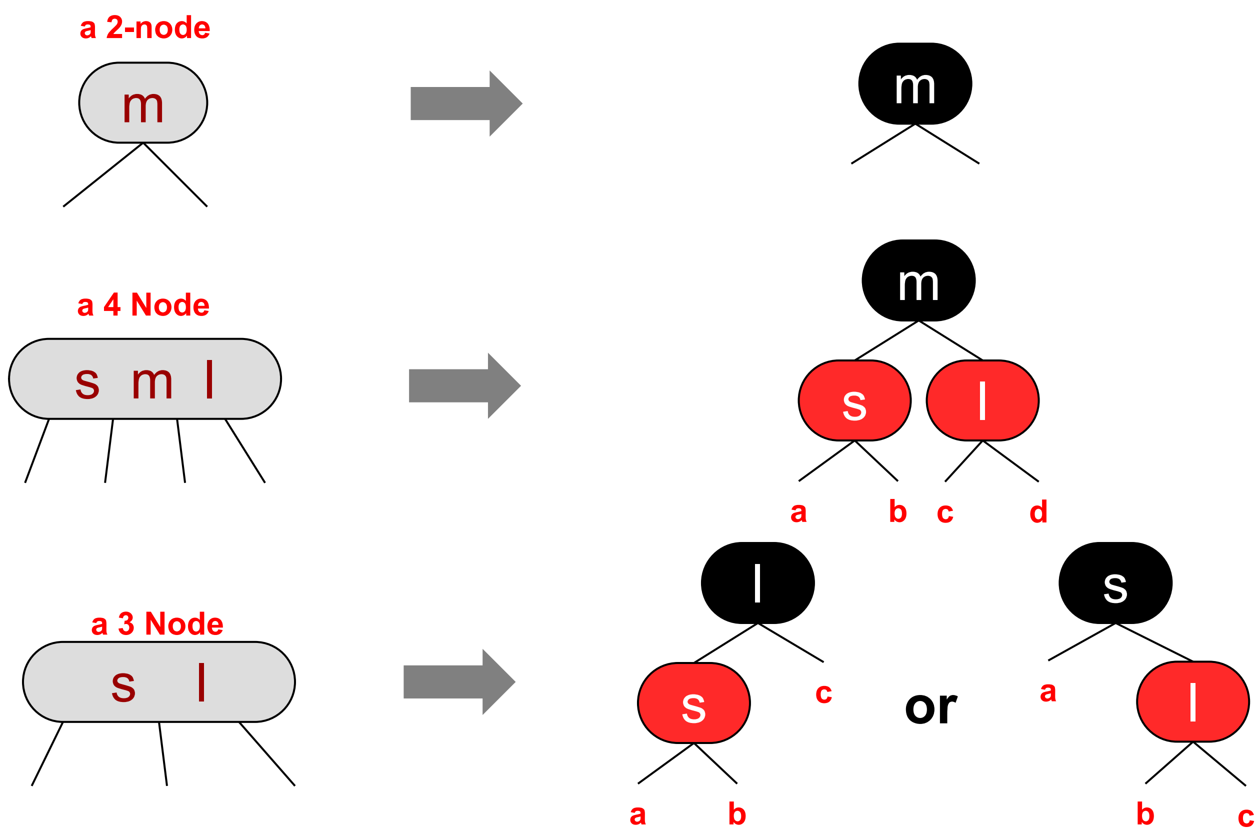
Red Black Trees

- A red-black tree is a binary search tree
  - Only 2 nodes (no 3- or 4-nodes)
  - Can be built from a 2-3-4 tree directly by converting each 3- and 4- nodes to multiple 2-nodes
- All 2-nodes means no wasted storage overheads
- Yields a "balanced" BST
- "Balanced" means that the height of an RB-Tree is at MOST **twice** the height of a 2-3-4 tree
  - Recall, height of 2-3-4 tree had an upper bound of $\log_2(n)$
  - Thus height or an RB-Tree is bounded by $2*\log_2 n$ which is still $O(\log_2(n))$

Red Black and 2-3-4 Tree Correspondence

- Every 2-, 3-, and 4-node can be converted to…
  - At least 1 black node and 1 or 2 red children of the black node
  - Red nodes are always ones that would join with their parent to become a 3- or 4-node in a 2-3-4 tree

S = Small
M = Median
L = Large

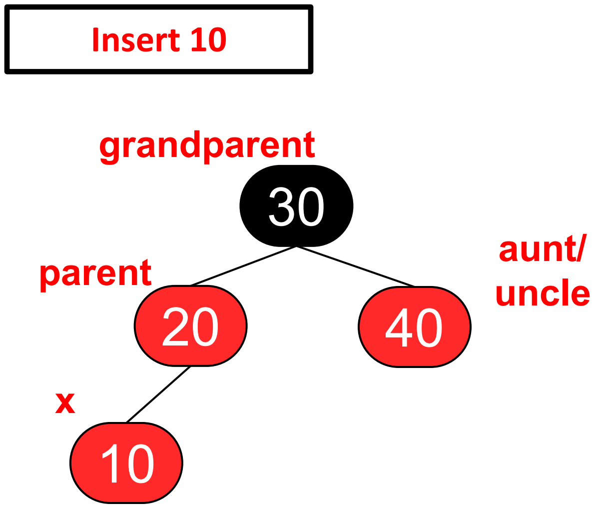


Red-Black Tree Properties

- Valid RB-Trees maintain the invariants that…
- 1. No path from root to leaf has two consecutive red nodes (i.e. a parent and its child cannot both be red)
  - Since red nodes are just the extra values of a 3- or 4-node from 2-3-4 trees you can't have 2 consecutive red nodes
- 2. Every path from leaf to root has the same number of black nodes
  - Recall, 2-3-4 trees are full (same height from leaf to root for all paths)
  - Also remember each 2, 3-, or 4- nodes turns into a black node ***plus*** 0, 1, or 2 red node children
- 3. At the end of an operation the root should always be black
- 4. We can imagine leaf nodes as having 2 non-existent (NULL) black children if it helps

Red-Black Insertion

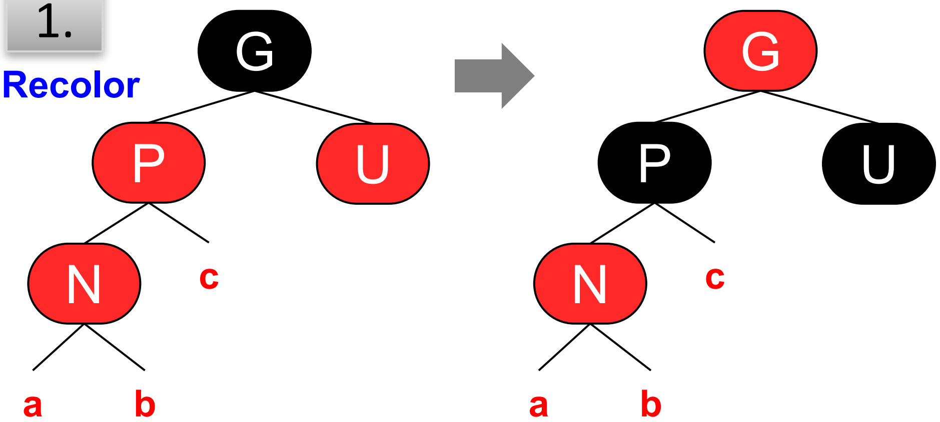
- Insertion Algorithm:
  - 1. Insert node into normal BST location (at a leaf location) and color it RED
  - 2a. If the node's parent is black (i.e. the leaf used to be a 2-node) then DONE (i.e. you now have what was a 3- or 4-node)
  - 2b. Else perform fixTree transformations then repeat step 2 on the parent or grandparent (whoever is red)
- fixTree involves either
  - recoloring or
  - 1 or 2 rotations and recoloring
- Which case of fixTree you perform depends on the color of the new node's "aunt/uncle"



fixTree Cases

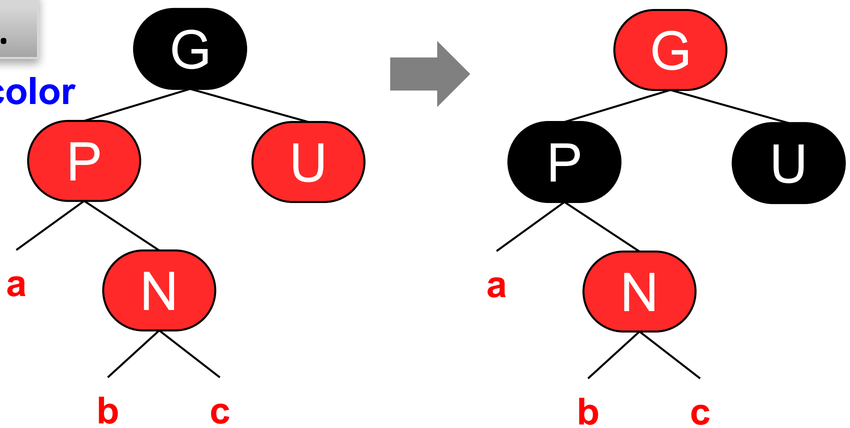
1.

Recolor



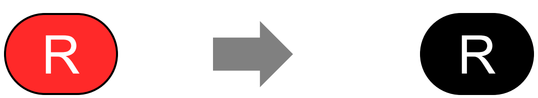
2.

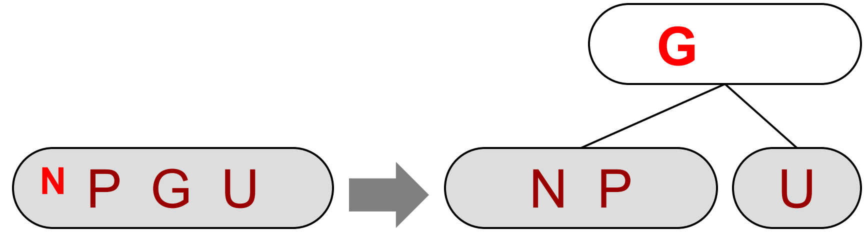
Recolor

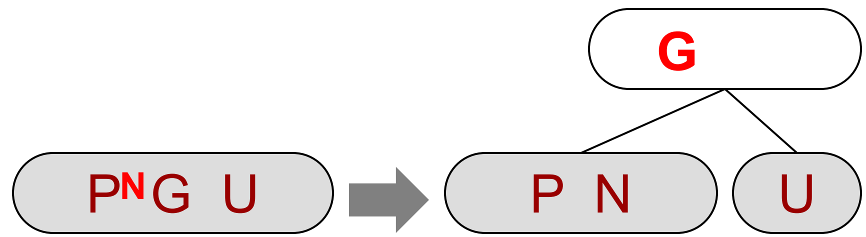


3.

Recolor Root





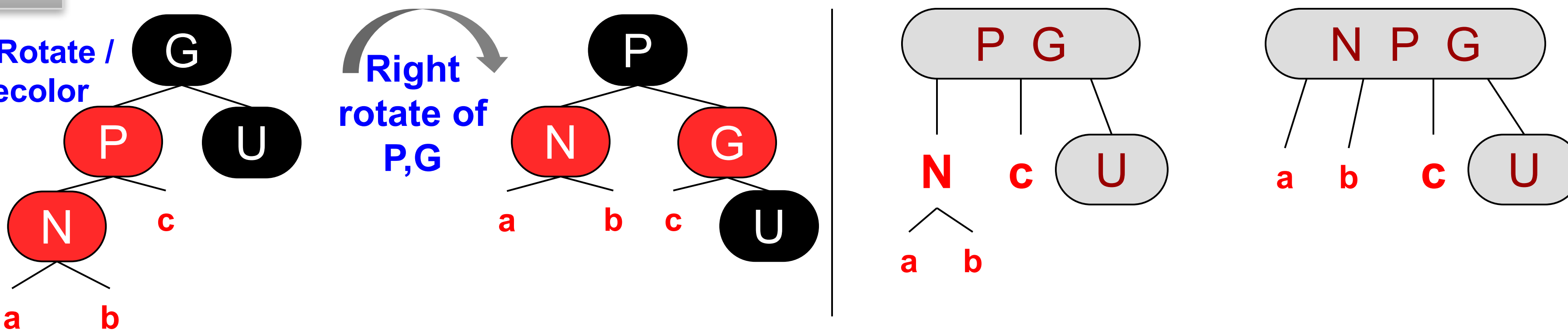


Note: For insertion/removal algorithm we consider non-existent leaf nodes as black nodes

fixTree Cases

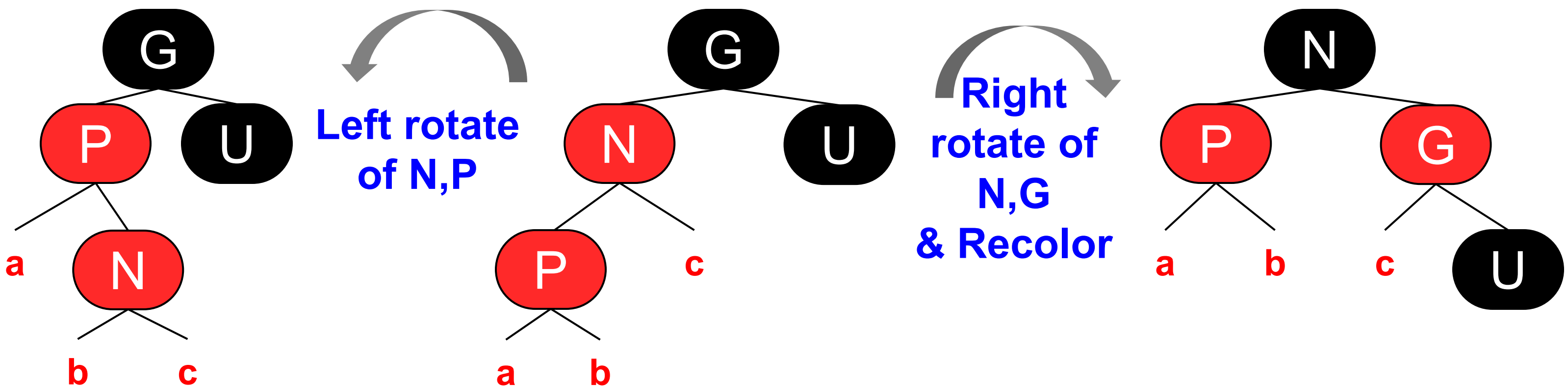
4.

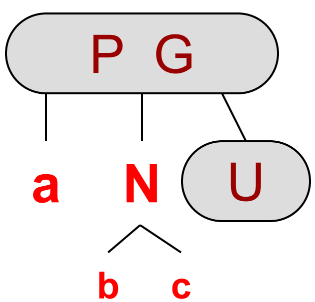
1 Rotate / Recolor

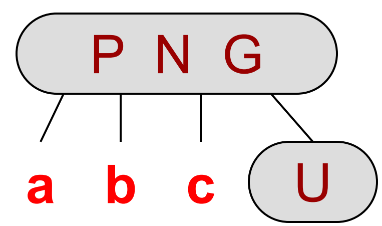


5.

2 Rotates / Recolor





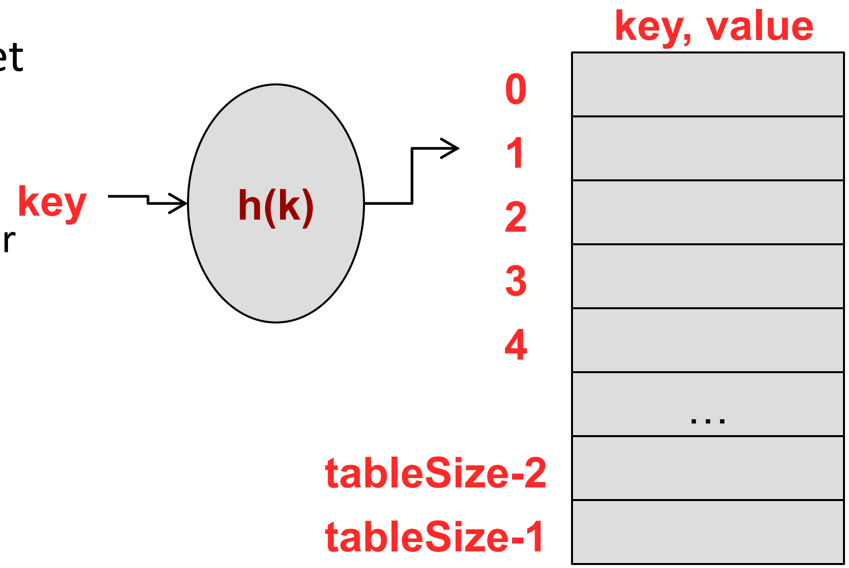


www.cse.ohio-state.edu/~gurari/course/cis680Ch11.html

HASH TABLES

Hash Tables

- A hash table is an array that stores key,value pairs
  - Usually smaller than the size of possible set of keys, |S|
    - USC ID's = 10^{10} options
    - Pick a hash table of some size much smaller (how many students do we have at any particular time)
- The table is coupled with a function, *h(k)*, that maps keys to an integer in the range [0..tableSize-1] (i.e. [0 to m-1])
- What are the considerations…
  - How big should the table be?
  - How to select a hash function?
  - What if two keys map to the same array location? (i.e. h(k1) == h(k2))
    - Known as a collision



Define
m = tableSize
n = # of used entries

Hash Functions First Look

- Define **N** = # of entries stored, **M** = Table/Array Size
- A hash function must be able to
  - convert the key data type to an integer
  - That integer must be in the range [0 to **M-1**]
    - Keeping h(k) in the range of the tableSize (**M**)
    - Fairly easy method: Use modulo arithmetic (i.e. h(k) % **M**)
- Usually converting key data type to an integer is a user-provided function
  - Akin to the operator<() needed to use a data type as a key for the C++ map
- Example: Strings
  - Use ASCII codes for each character and add them or group them
  - "hello" => 'h' = 104, 'e'=101, 'l' = 108, 'l' = 108, 'o' = 111 =
  - Example function: h("hello") = 104 + 101 + 108 + 108 + 111 = 532 **% M**

Hash Function Desirables

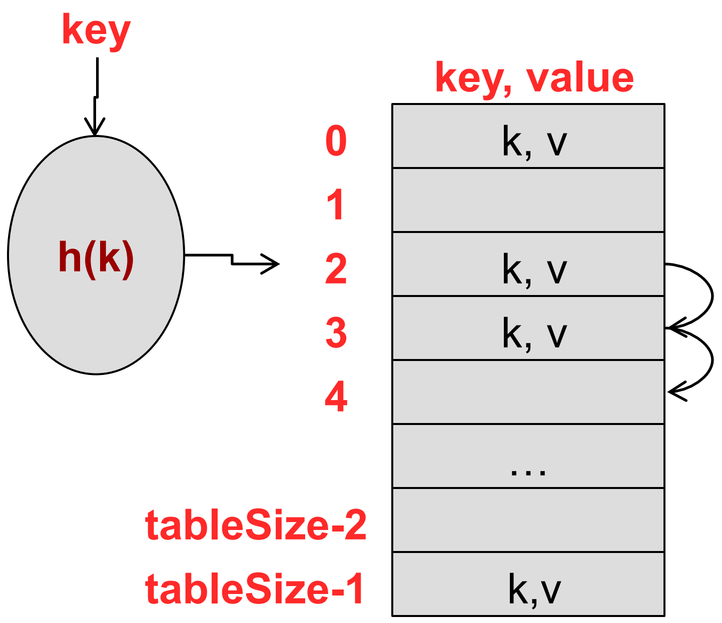
- A "perfect hash function" should map each given key to a unique location in the table
  - Perfect hash functions are not practically attainable
- A "good" hash function
  - Is easy and fast to compute
  - Scatters data uniformly throughout the hash table
    - $P(\ h(k) = x\ ) = 1/\mathbf{M}$

Resolving Collisions

- Example:
  - A hash table where keys are phone numbers: (XXX) YYY-ZZZZ
  - Obviously we can't have a table with 10^{10} entries
  - Should we define h(k) as the upper 3 or 4 digits: XXX or XXXY
    - Meaning a table of 1000 or 10,000 entries
  - Define h(k) as the lowest 4-digits of the phone number: ZZZZ
    - Meaning a table with 10,000 entries: 0000-9999
  - Now 213-740-4321 and 323-681-4321 both map to location 4321 in the table
- Collisions are hard to avoid so we have to find a way to deal with them
- Methods
  - Open addressing (probing)
    - Linear, quadratic, double-hashing
  - Buckets/Chaining (Closed Addressing)

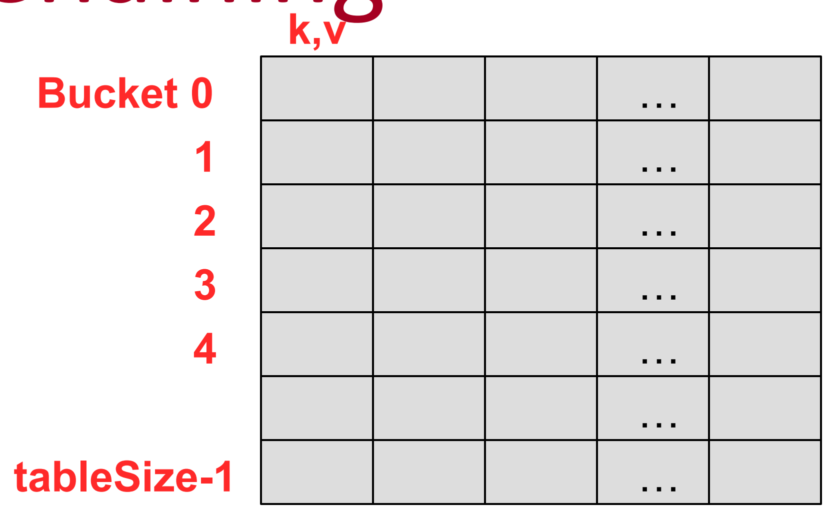
Open Addressing

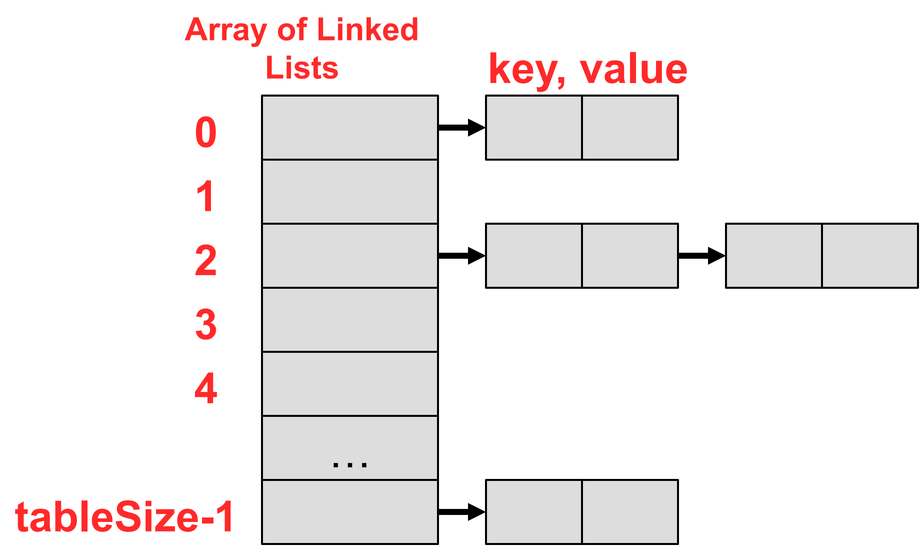
- Open addressing means an item with key, k, may not be located at h(k)
- Assume, location 2 is occupied with another item
- If a new item hashes to location 2, we need to find another location to store it
- Linear Probing
  - Just move on to location h(k)+1, h(k)+2, h(k)+3,…
- Quadratic Probing
  - Check location $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$, …



Buckets/Chaining

- Rather than searching for a free entry, make each entry in the table an ARRAY (bucket) or LINKED LIST (chain) of items/entries
- Buckets
  - How big should you make each array?
  - Too much wasted space
- Chaining
  - Each entry is a linked List





Hash Tables

- Suboperations
  - Compute h(k) should be O(1)
  - Array access of table[h(k)] = O(1)
- In a hash table, what is the expected efficiency of each operation
  - Find = O(1)
  - Insert = O(1)
  - Remove = O(1)

Summary

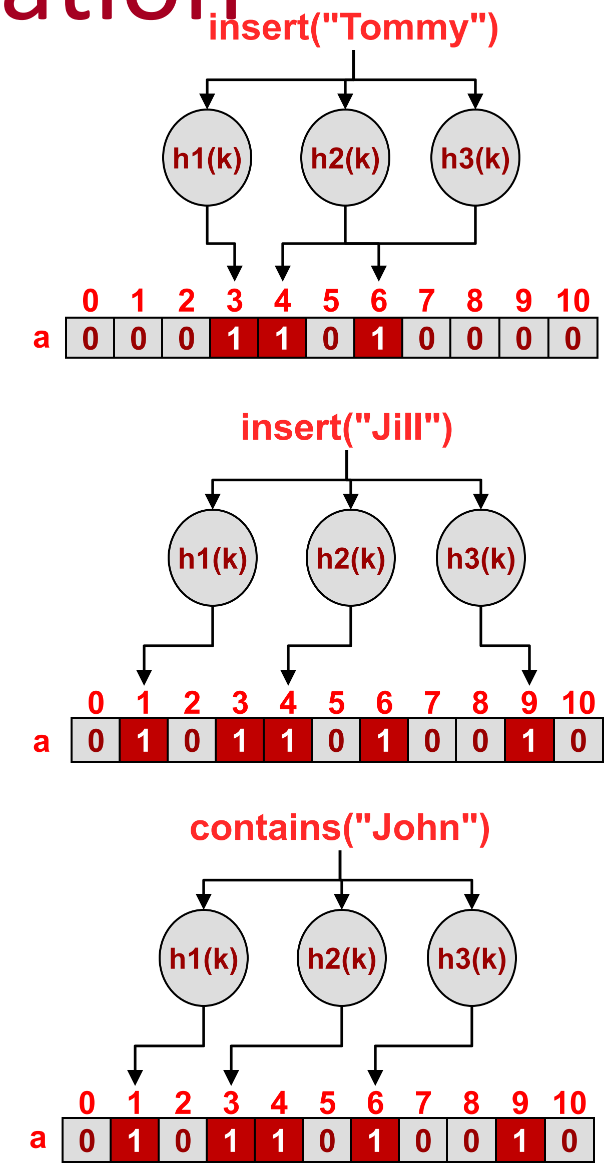
- Hash tables are LARGE arrays with a function that attempts to compute an index from the key
- In the general case, **chaining** is the best collision resolution approach
- The functions should spread the possible keys evenly over the table

An imperfect set…

BLOOM FILTERS

Bloom Filter Explanation

- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \dots h_s(k)\}$
- Insert()
  - Apply each $h_i(k)$ to the key
  - Set $a[h_i(k)]$ = True
- Contains()
  - Apply each $h_i(k)$ to the key
  - Return True if ***all*** $a[h_i(k)]$ = True
  - Return False otherwise
  - In other words, answer is "Maybe" or "No"
    - May produce "false positives"
    - May NOT produce "false negatives"
- We will ignore removal for now



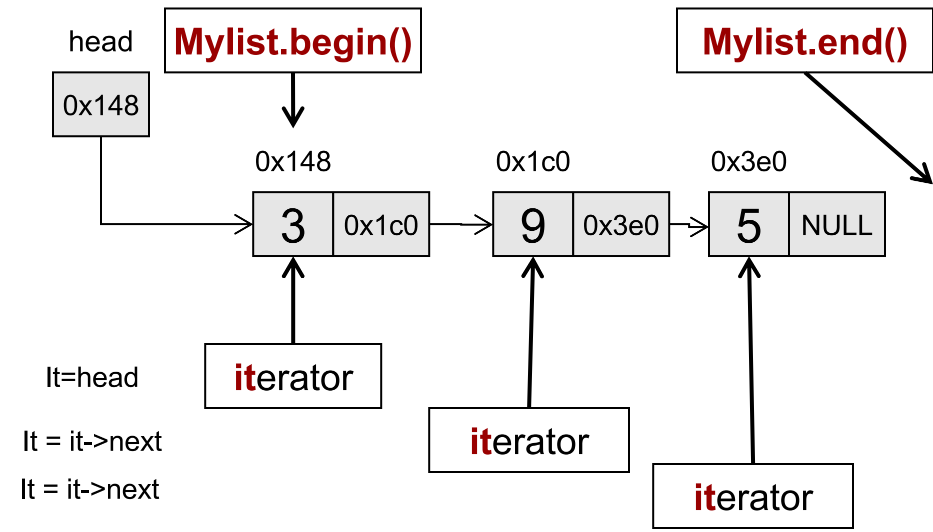
Sizing Analysis

- Can also use this analysis to answer or a more "useful" question...
- ...To achieve a desired probability of false positive, what should the table size be to accommodate n entries?
  - Example: I want a probability of p=1/1000 for false positives when I store n=100 elements
  - Solve $2^{-m*\ln(2)/n} < p$
    - Flip to $2^{m*\ln(2)/n} \geq 1/p$
    - Take log of both sides and solve for m
    - $m \geq [n*\ln(1/p)] / \ln(2)^2 \approx 2n*\ln(1/p)$ because $\ln(2)^2 = 0.48 \approx ½$
  - So for p=.001 we would need a table of m=14*n since ln(1000) ≈ 7
    - For 100 entries, we'd need 1400 bits in our Bloom filter
  - For p = .01 (1% false positives) need m=9.2*n (9.2 bits per key)
  - Recall: Optimal # of hash functions, j = ln(2) / α
    - So for p=.01 and α = 1/(9.2) would yield j ≈ 7 hash functions

ITERATORS

Building Our First Iterator

- Let's add an iterator to our Linked List class
  - Will be an object/class that holds some data that allows us to get an item in our list and move to the next item
  - How do you iterate over a linked list normally:
    - **Item<T>* temp = head;**
    - **While(temp) temp = temp->next;**
  - So my iterator object really just needs to model (contain) that 'temp' pointer
- Iterator needs following operators:
  - *
  - ->
  - ++
  - == / !=
  - < ??




```
template <typename T>
struct Item {
  T val;
  Item<T>* next;
};

template <typename T>
class LList {
public:
   LList();   // Constructor
   ~LList();  // Destructor

 private:
   Item<T>* head_;
};
```

Friends and Private Constructors

- Let's only have the iterator class grant access to its "trusted" friend: Llist
- Now LList<T> can access iterators private data and member functions
- And we can add a private constructor that only 'iterator' and 'LList<T>' can use
  - This prevents outsiders from creating iterators that point to what they choose
- Now begin() and end can create iterators via the private constructor & return them

```
template<typename T>
class LList
{ public:
  LList() { head_ = NULL; }

 class iterator {
  private:
    Item<T>* curr_;
    iterator(Item<T>* init) : curr_(init) {}
  public:
    friend class LList<T>;
    iterator(Item<T>* init);
    iterator& operator++() ;
    iterator operator++(int);
    T& operator*();
    T* operator->();
    bool operator!=(const iterator & other);
    bool operator==(const iterator & other);
  };
 iterator begin()  { iterator it(head_);
                     return it;    }
 iterator end()    { iterator it(NULL);
                     return it;    }
 private:
  Item<T>* head_;
  int size_;
};
```

Kinds of Iterators

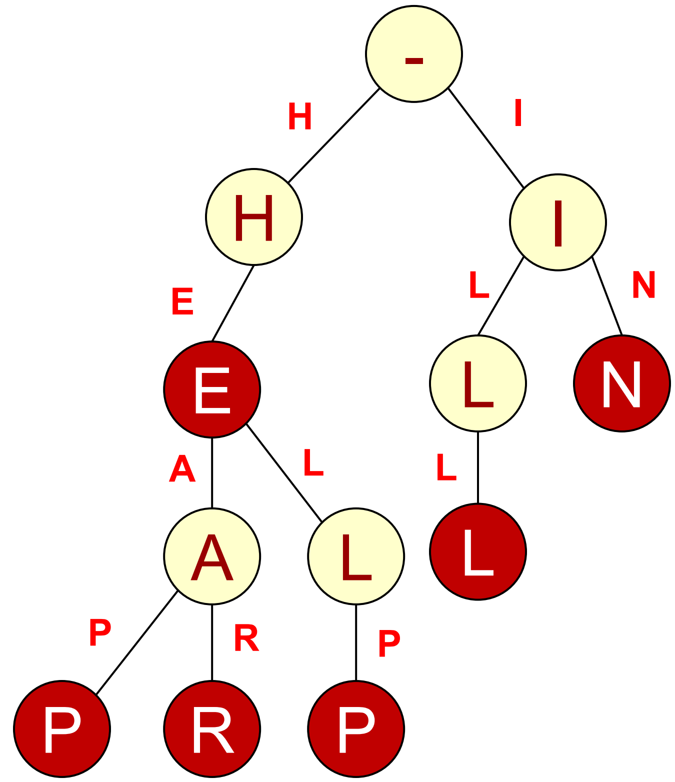
- This leads us to categorize iterators based on their capabilities (of the underlying data organization)
- Access type
  - Input iterators: Can only READ the value be pointed to
  - Output iterators: Can only WRITE the value be pointed to
- Movement/direction capabilities
  - Forward Iterator: Can only increment (go forward)
    - ++it
  - Bidirectional Iterators: Can go in either direction
    - ++it or --it
  - Random Access Iterators: Can jump beyond just next or previous
    - it + 4 or it – 2
- Which movement/direction capabilities can our LList<T>::iterator naturally support

Prefix Trees

TRIES

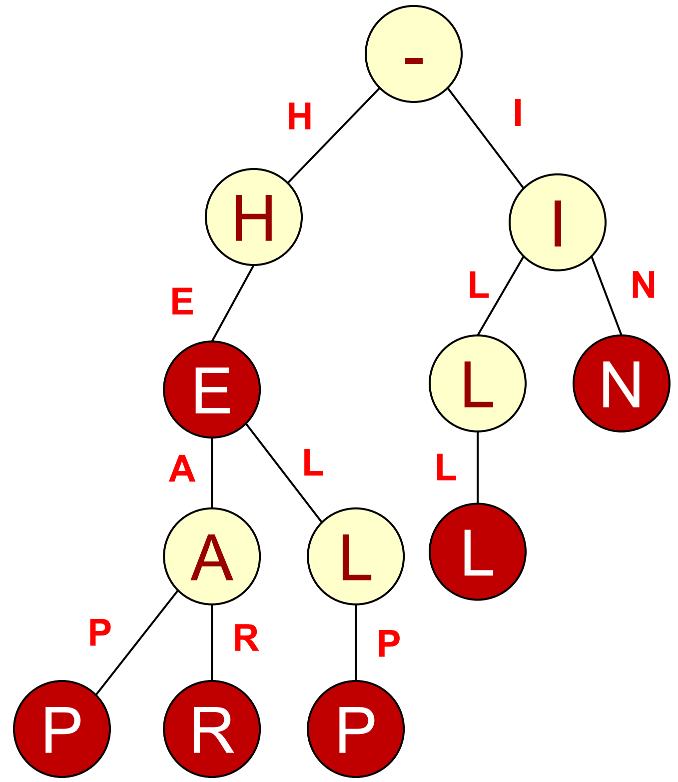
Tries

- Assuming unique keys, can we still achieve O(m) search but not have collisions?
  - O(m) means the time to compare is *independent* of how many keys (i.e. n) are being stored and only depends on the length of the key
- Trie(s) (often pronounced "try" or "tries") allow O(m) retrieval
  - Sometimes referred to as a radix tree or prefix tree
- Consider a trie for the keys
  - "HE", "HEAP", "HEAR", "HELP", "ILL", "IN"



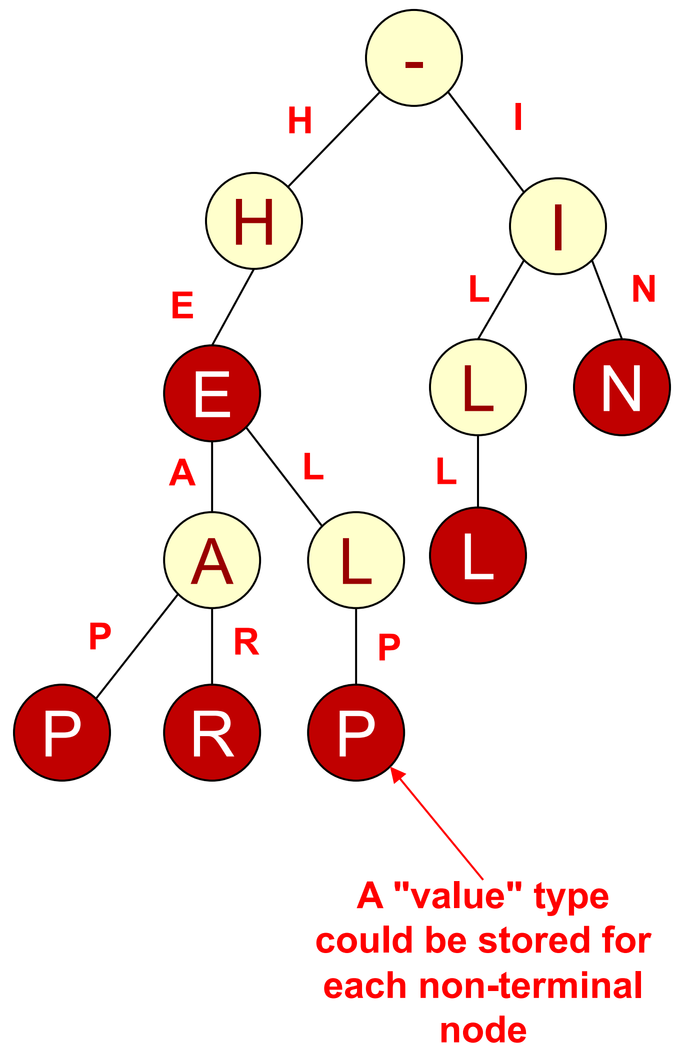
Tries

- Rather than each node storing a full key value, each node represents a prefix of the key
- Highlighted nodes indicate terminal locations
  - For a map we could store the associated value of the key at that terminal location
- Notice we "share" paths for keys that have a common prefix
- To search for a key, start at the root consuming one unit (bit, char, etc.) of the key at a time
  - If you end at a terminal node, SUCCESS
  - If you end at a non-terminal node, FAILURE



Tries

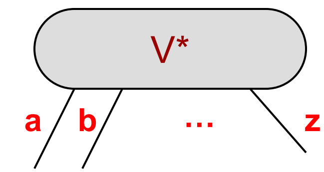
- To search for a key, start at the root consuming one unit (bit, char, etc.) of the key at a time
  - If you end at a terminal node, SUCCESS
  - If you end at a non-terminal node, FAILURE
- Examples:
  - Search for "He"
  - Search for "Help"
  - Search for "Head"
- Search takes O(m) where m = length of key
  - Notice this is the same as a hash table



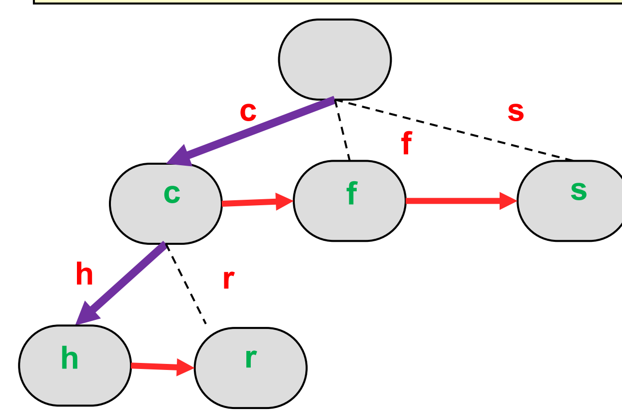
Structure of Trie Nodes

- What do we need to store in each node?
- Depends on how "dense" or "sparse" the tree is?
- Dense (most characters used) or small size of alphabet of possible key characters
  - Array of child pointers
  - One for each possible character in the alphabet
- Sparse
  - (Linked) List of children
  - Node needs to store ______

```
template < class V >
struct TrieNode{
  V* value; // NULL if non-terminal
  TrieNode<V>* children[26];
};
```




```
template < class V >
struct TrieNode{
  char key;
  V* value;
  TrieNode<V>* next;
  TrieNode<V>* children;
};
```



Search

- Search consumes one character at a time until
  - The end of the search key
    - If value pointer exists, then the key is present in the map
  - Or no child pointer exists in the TrieNode
- Insert
  - Search until key is consumed but trie path already exists
    - Set v pointer to value
  - Search until trie path is NULL, extend path adding new TrieNodes and then add value at terminal

```
V* search(char* k, TrieNode<V>* node)
{
  while(*k != '\0' && node != NULL){
    node = node->children[*k - 'a'];
    k++;
  }
  if(node){
    return node->v;
  }
}
```

```
void insert(char* k, Value& v)
{
  TrieNode<V>* node = root;
  while(*k != '\0' && node != NULL){
    node = node->children[*k - 'a'];  k++;
  }
  if(node){
    node->v = new Value(v);
  }
  else {
    // create new nodes in trie
    // to extend path
    // updating root if trie is empty
  }
}
```

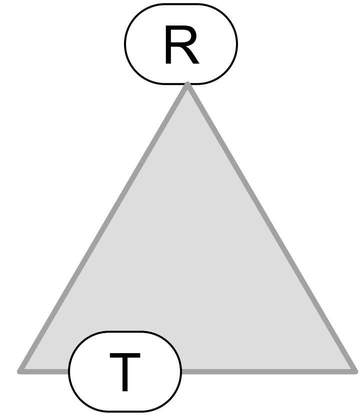
SPLAY TREES

Splay Tree Intro

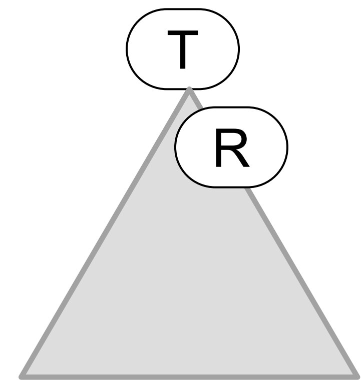
- Another map/set implementation (storing keys or key/value pairs)
  - Insert, Remove, Find
- Recall…To do **m** inserts/finds/removes on an RBTree w/ **n** elements would cost?
  - $O(m*\log(n))$
- Splay trees have a worst case find, insert, delete time of…
  - $O(n)$
- However, they guarantee that if you do **m** operations on a splay tree with **n** elements that the total ("amortized"…uh-oh) time is
  - $O(m*\log(n))$
- They have a further benefit that recently accessed elements will be near the top of the tree
  - In fact, the most recently accessed item is always at the top of the tree

Splay Operation

- Splay means "spread"
- As you search for an item or after you insert an item we will perform a series of splay operations
- These operations will cause the desired node to always end up at the top of the tree
  - A desirable side-effect is that accessing a key multiple times within a short time window will yield fast searches because it will be near the top
  - See next slide on principle of locality



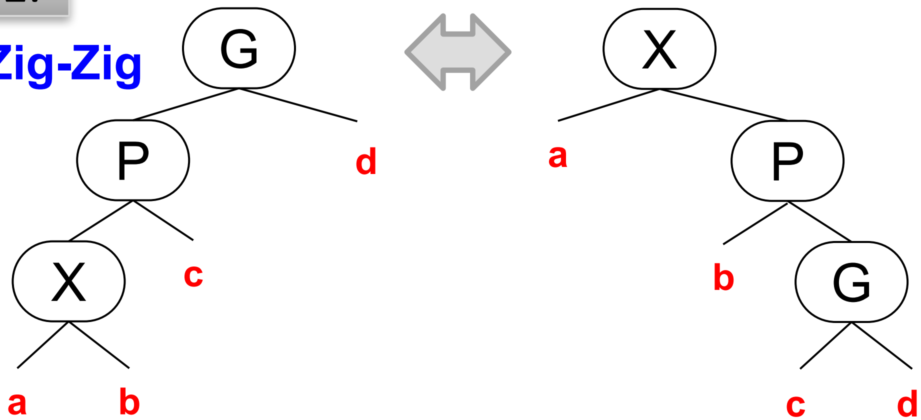
If we search for or insert T…



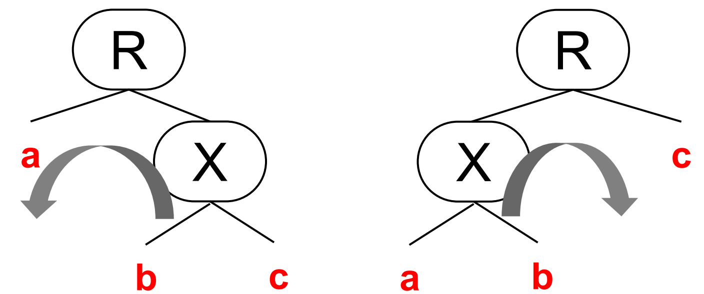
…T will end up as the root node with the old root in the top level or two

Splay Cases

1. **Zig-Zig**

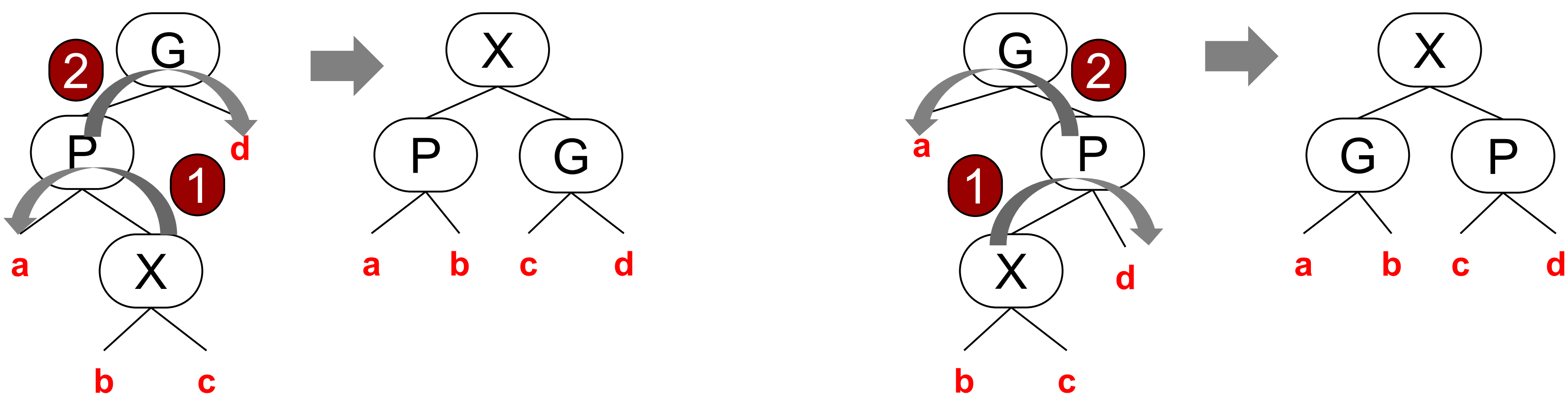


3. **Root/Zig Case (Single Rotation)**



Left rotate of X,R **Right rotate of X,R**

2. **Zig-Zag**



Splay Tree Supported Operations

- Insert(x)
  - Normal BST insert, then splay x
- Find(x)
  - Attempt normal BST find(x) and splay last node visited
    - If x is in the tree, then we splay x
    - If x is not in the tree we splay the leaf node where our search ended
- Remove(x)
  - Find(x) splaying it to the top, then overwrite its value with is successor/predecessor, deleting the successor/predecessor node

Summary

- Splay trees don't enforce balance but are self-adjusting to attempt yield a balanced tree
- Splay trees provide efficient amortized time operations
  - A single operation may take O(n)
  - m operations on tree with n elements => O(m(log n))
- Uses rotations to attempt balance
- Provides fast access to recently used keys

Online Tools for Trees

- http://www.cs.usfca.edu/~galles/visualization/AVLtree.html
- http://www.cs.usfca.edu/~galles/visualization/BTree.html
- http://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- http://www.cs.usfca.edu/~galles/visualization/SplayTree.html