Linked Lists

Big O:

- space O(n)
- time
  - search worst O(n), average O(n)
  - insert worst O(1), average O(1)
  - delete worst O(1), average O(1)

Advantages:

- Linked lists are a dynamic data structure, allocating the needed memory while the program is running
- Insertion and deletion node operations are easily implemented in a linked list
- Linear data structures such as stacks and queues are easily executed with a linked list
- They can reduce access time and may expand in real time without memory overhead

Disadvantages:

- They have a tendency to use more memory due to pointers requiring extra storage space
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access
- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards[1] and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer

Uses:

- Stack
- Queue

- Memory Allocation

Creating a Linked List:

```
class Node {
   Node next = null;
   int data;
   public Node(int d) { data = d; }
   void appendToTail(int d) {
      Node end = new Node(d);
      Node n = this;
      while (n.next != null) { n = n.next; }
      n.next = end;
   }
}
```

Deleting a node:

```
Node deleteNode(Node head, int d) {
   Node n = head;
   if (n.data == d) {
      return head.next; /* moved head */
   }
   while (n.next != null) {
      if (n.next.data == d) {
         n.next = n.next.next;
         return head; /* head didnt change */
      }
      n = n.next;
   }
}
```

Notes:

- Alternative to array to implement stack and queue
- Allows any length

Queues

Implementing a Queue:

```
class Queue {
   Node first, last;
   void enqueue(Object item) {
      if (!first){
         back = new Node(item);
         first = back;
      } else {
         back.next = new Node(item);
         back = back.next;
      }
   }
   Node dequeue(Node n) {
      if (front != null) {
         Object item = front.data;
         front = front.next;
         return item;
      }
      return null;
   }
}
```

Notes:

- First in First out

Stacks

Implementing a Stack:

```java
class Stack {
   Node top;
   Node pop() {
      if (top != null) {
      Object item = top.data;
      top = top.next;
      return item;
      }
      return null;
   }
   void push(Object item) {
      Node t = new Node(item);
      t.next = top;
      top = t;
   }
}
```

Notes:

- Last in First out

Hash Tables

Big O:

- space O(n)
- time
  - search worst O(n), average O(1)
  - insert worst O(n), average O(1)
  - delete worst O(n), average O(1)

Advantages:

- faster than other structures on large entries
- efficient when max entries known (dont have to resize)

Disadvantages:

- have to resize for more data

Uses:

- associative arrays (arrays index through arbitrary strings)
- database indexing
- caches
- sets (?)
- object rep (key is method or object, value is pointer to member or method)

Properties:

- keys have to be hash able (able to compute numeric value from it)
- entries in no particular order

Creating a Hash Table:

```java
public HashMap<Integer, Student> buildMap(Student[] students) {
   HashMap<Integer, Student> map = new HashMap<Integer, Student>();
   for (Student s : students) map.put(s.getId(), s);
   return map;
}
```

Notes:

- Alternative to array to implement stack and queue
- Allows any length
- can be made more efficient with better fit hash function

Dictionary/Map

Operations:

- Add(K key, V value) adds given key-value pair in the dictionary. With most implementations of this class in .NET, when adding a key that already exists, an exception is thrown.
- Get(K key) returns the value by the specified key. If there is no pair with this key, the method returns null or throws an exception depending on the specific dictionary implementation.
- Remove(key) removes the value, associated with the specified key and returns a Boolean value, indicating if the operation was successful.
- Contains(key) returns true if the dictionary has a pair with the selected key
- Count returns the number of elements (key value pairs) in the dictionary

Notes:

- Abstract Data Structure
- ”map” or ”associative array”
- maps keys to values
- hash table is one implementation

Tries

Associated trees

Advantages (over BST):

- no collisions
- no hash function

Disadvantages:

- slower than hash table
- some keys can be meaningless (floating point nos)

Applications:

- dictionary (autocomplete)

Operations:

- look-up
- insert

Notes:

- bitwise vs compressive implementation

Heap

Binary Heap Operations:

- findmax (O(1))
- insert (O(log n) (Binary Imp))
  - insert item into next place in the BT
  - Swap item up with parent until heap invariant is maintained
- remove-max (O(log n) (Binary Imp))
  - Remove the root (and store it to return)
  - Place the last element inserted at the root
  - Swap item down with child until heap invariant is maintained

Applications:

- heapsort
- graph algorithms
- order stats
- Priority Queue

Notes:

- tree-based structure that satisfies heap property (max heap parent greater than or equal to children)

sources:

- `http://interactivepython.org/runestone/static/pythonds/Trees/BinaryHeapImplementation.html`

Common Binary Search Trees

Big O:

- space O(n)
- time
  - search worst O(n), average O(log(n))
  - insert worst O(n), average O(log(n))
  - delete worst O(n), average O(log(n))

BST Node:

- Data
- Left Child
- Right Child

Comparison:

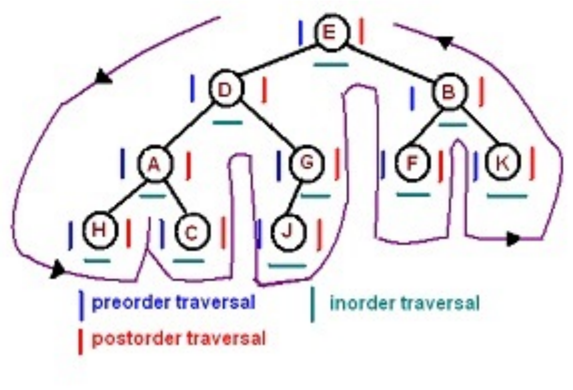
- advantages
  - related sorting algorithms
  - search algorithms
  - inorder traversal
- disadvantages
  - shape depends on insertions
  - keys has to be compared when inserting or searching
  - height grows n, which grows much faster than log n
- uses
  - sets, multisets, associative arrays, priority queue

Operations:

- Look-Up

  - compare to current node: go left if less, go right if greater
- Insert
  - Same as look-up but replace node where it should be
- Delete
  - no children: just delete
  - 1 child: remove node and replace it with the child
  - 2 children: find in-order successor or predecessor R to the current node N, switch it with N then call delete on the respective child with N)

Traversals:



- In-Order
  - Traverse left node, current node, then right
  - everything in order
- Pre-Order
  - Traverse current node, left node, then right
  - while duplicating nodes and edges can make duplicate binary tree
- Post-Order
  - Traverse left node, right node, then current
  - while deleting and freeing can delete and free an entire tree

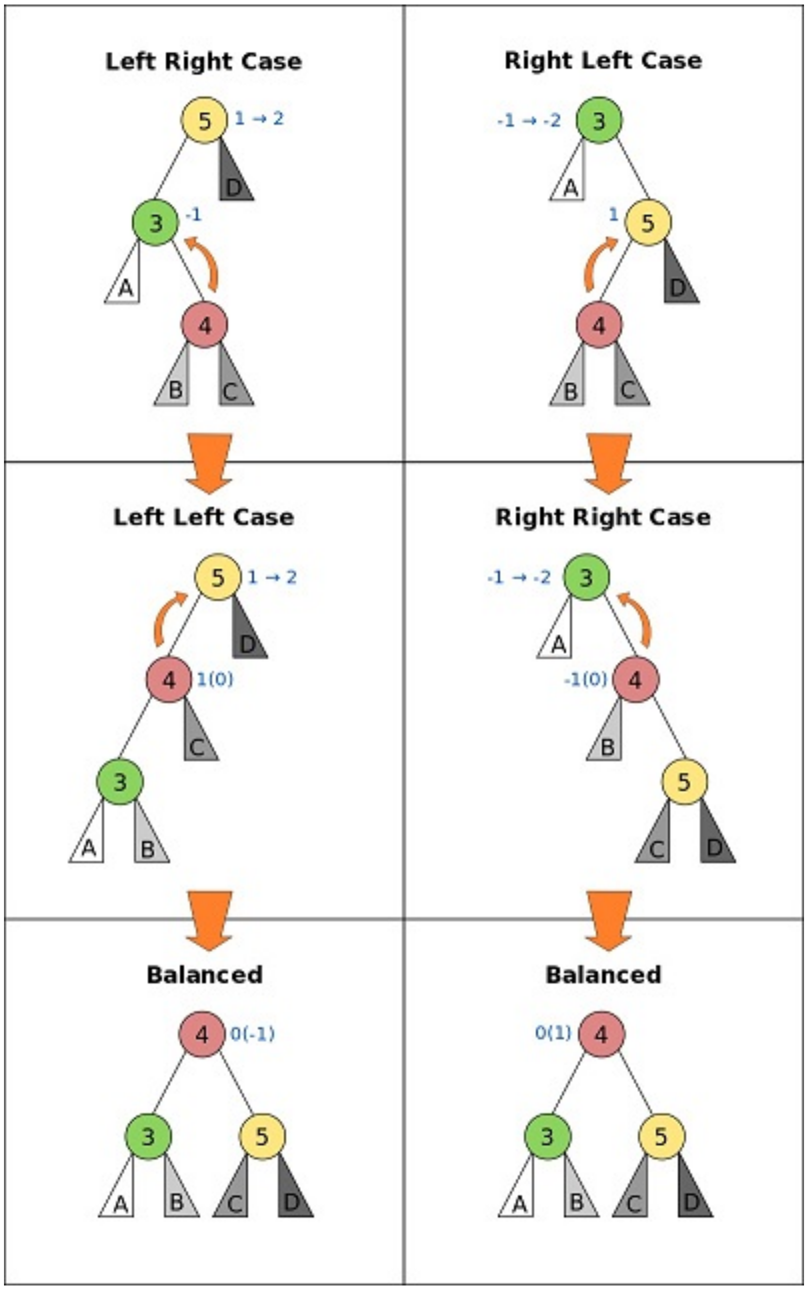
---

AVL:

- General
  - rebalance when insertion so that height never exceeds O(log n)

  - shape of tree changes during insertion and deletion
  - the height of an AVL tree is at most 1.44log(N)
  - AVL tree may need O(log(N)) operations to rebalance the tree

- searching-BST
- insertion - check balance factor for all ancestors if | balance factor | > 1 then rebalance
  - balance factor = height(left sub tree) - height(right sub tree)
  - rebalance starts from the inserted node up (from bottom up)
- deletion
  - Let node X be the node with the value we need to delete, and let node Y be a node in the tree we need to find to take node X's place, and let node Z be the actual node we take out of the tree
  - Steps to consider when deleting a node in an AVL tree are the following:
    - If node X is a leaf or has only one child, skip to step 5 with Z:=X.
    - Otherwise, determine node Y by finding the largest[citation needed] node in node X's left subtree (the in
    - order predecessor of X it does not have a right child) or the smallest in its right subtree (the in
    - order successor of X it does not have a left child).
    - Exchange all the child and parent links of node X with those of node Y. In this step, the in
    - order sequence between nodes X and Y is temporarily disturbed, but the tree structure doesn't change.
    - Choose node Z to be all the child and parent links of old node Y = those of new node X.
    - If node Z has a subtree (which then is a leaf), attach it to Z's parent.
    - If node Z was the root (its parent is null), update root.
    - Delete node Z.
    - Retrace the path back up the tree (starting with node Z's parent) to the root, adjusting the balance factors as needed.



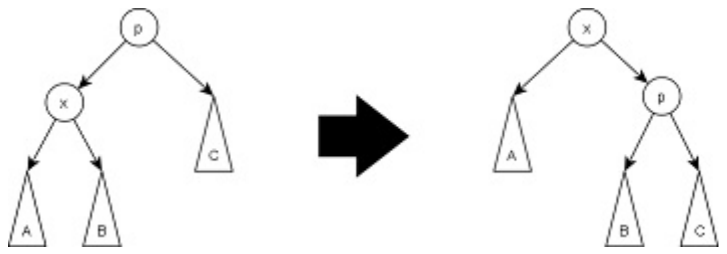
Red/Black:

- General
  - the maximum height of a red-black tree, ∼ 2log(N)
  - red-black tree needs O(1) operations to rebalance the tree
  - each node has an extra bit for color red or black
- In addition to the requirements imposed on a binary search tree the following must be satisfied by a redblack tree
  - A node is either red or black

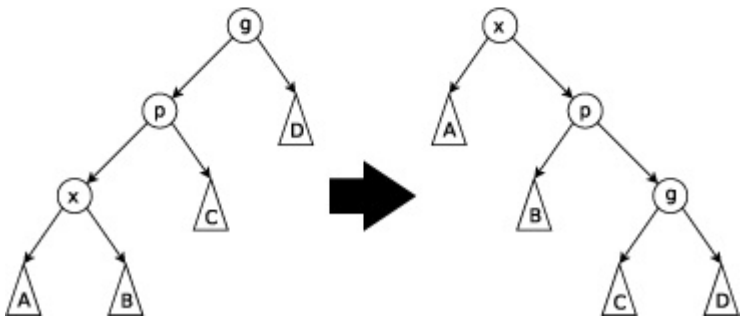
  - The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis
  - All leaves (NIL) are black
  - If a node is red, then both its children are black
  - Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes. The uniform number of black nodes in the paths from root to leaves is called the black height of the redblack tree

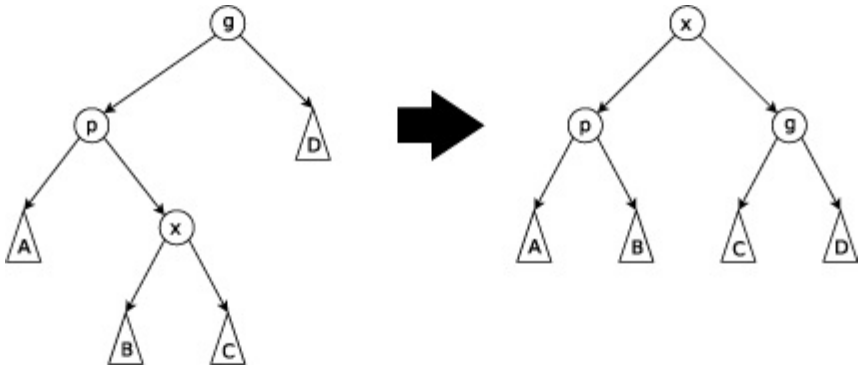
- searching-BST
- insertion O(log n)
  - Insert as BST
  - Fix any red-black vialations starting with inserted node continuing up the path
- deletion O(log n)
  - Delete as BST
  - Restore red-black properties

Splay:

- General
  - rebalance when look-up so frequently accessed elements move up
  - rebalances on look-up
  - shapes is not constrained and depends on look-ups
- Advantages
  - ave case is efficient as others
  - small memory
  - works with duplicate keys
- Disadvantages-height can be linear
- splaying - x=accessed node, p=parent of x, g parent of p
  - when p is the root tree is rotated on edge between p and x
  - 

  - p is not the root and x and p are either both right children or are both left children

- 

- p is not the root and x is a right child and p is a left child or vice versa
- 

Comparisons:

- AVL trees
  - AVL trees gurantee fast lookup (O(log n)) on each operation
  - AVL more useful in multithreaded environ with lots of look-ups because can be done in parallel (splay not in parallel)
  - Real time (and more) look-ups AVL is better
- Red Black
  - red black better with more inserts and deletes
- Splay trees
  - Splay trees gurantee any sequence of n operations take at most O(nlog n)
  - Splay faster on average on more look-ups
  - Splay better for splitting and merging efficiently
  - Splay more memory efficient (no need to store balance info)
  - Splay more effective when you only access a small subset
  - Splay rotation logic easier therefore easier to implement

Notes:

BST Notes

- Not all binary trees are BST
- Balancing
- Height: longest path from root to leaf
- Depth: The depth of a node is the number of edges from the node to the tree's root node

---

sources:

- `https://en.wikipedia.org/wiki/AVL_tree`
- `https://en.wikipedia.org/wiki/Red%E2%80%93black_tree`
- `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`
- `https://www.topcoder.com/community/data-science/data-science-tutorials/an-introduction-to`
- `https://en.wikipedia.org/wiki/Splay_tree`
- `https://attractivechaos.wordpress.com/2008/10/02/comparison-of-binary-search-trees/`

Treap

Big O:

- space O(n)
- time
  - search worst O(n), average O(log(n))
  - insert worst O(n), average O(log(n))
  - delete worst O(n), average O(log(n))

Advantages:

- Treap is same shape regardless of history
  - security: cant tell history
  - efficient sub tree sharing
  - useful for sets

Notes:

- heap invariant (children less or equal to parent)?
- formed by inserting nodes highest priority first into a BST without rebalancing
- each node has priority (heap) and key (BST)

Graphs

Notation:

G graph

V Vertices

E edges

Representation:

- Adjacency List
  - Storage O(|V| + |E|)
  - Add vertex O(1)
  - Remove vertex O(|V| + |E|)
  - Add edge O(1)
  - Remove edge O(|E|)
  - Query O(|V|)
- Adjacency Matrix
  - Storage O($| V |^2$)
  - Add vertex O($| V |^2$)
  - Remove vertex O($| V |^2$)
  - Add edge O(1)
  - Remove edge O(1)
  - Query O(1)

Searches:

- DFS
  - stack
  - DFS Numbering (entrance(d)/exit (f))
    - d value of a vertex u is lesser than the d value of all the descendants of u.
    - f value of a vertex u is higher than the f value of all the descendants of u.
  - visited set
  - cycles?
  - DFS tree?

        * forward edge (node to descendant)
        * back (node to ancestor)
        * cross (neither forward nor back)

- BSF
    - queue

Notes:

- undirected vs directed

General

Cycle Detection:

- Tortoise and hare
  - tortoise pointer moves 1 at each step
  - hare moves 2 at each step
  - if they ever meet their is a cycle
  - you can also use this algorithm (with more steps) to find start of cycle and the length of the cycle
- Applicable data structures
  - Linked List
  - Graph

General Common Issues:

- Space vs Indexing speed vs Sorting speed

sources:

- `https://en.wikipedia.org/wiki/Cycle_detection`