# Sorting

## Merge Sort

- $T(N) = 2T(\frac{N}{2}) + O(N)$
- $T(N) = Nlog(N)$

## Quick Sort

```
if (N>1)
{
  pick a pivot
  partition around the pivot
  sort left of pivot
  sort right of pivot
}
```

- use one of the array values as the partition
- partition with median is best
    - cannot find median with small time ~ can find median with first couple items(3)
- Best Case
    - Every pivot splits the array perfectly in half
    - $Nlog(N)$
- Worst Case
    - Every pivot doesn't split the array at all
    - $O(N^2)$

## Improvements (Qs)

- if sections are 9 or less, don't sort
    - one last pass using insertion sort to sort smaller sections
    - much faster as items are close to where they should be
- If recursion pass hits a limit, $2log(n)$, switch to merge sort for that section
    - Introsort - worst case $Nlog(N)$

  - most libraries use introsort

# Trees

- A tree is a collection of nodes
- nodes are connected by edges
- paths are edges from one node to each other
- A tree is defined by a **root node**
  - there is a unique path from a root node to each child node
  - No cycles
- **leaf node** has no children
  - other nodes are **non-leaf** or **interior**
- **depth** of a node is how many steps it takes to reach that node from the root
- **height of the tree** is the depth of the deepest node

Example code for node:

```
struct Node
{
    string name;
    vector<Node*> children;
}
```

# Algorithms

Given a tree, find the number of nodes

```
int countNodes(const Node* p)
{
    if(p == nullptr)
        return 0;
    int total = 1; // starts at one to count itself
    for(int k = 0; k < p-> children.size(); k++)
        total += countNodes(p->children[k])
    return total;
}
```

Print out the tree hierarchy

```
void printTree(const Node* p, int depth)
{
    if(p!= nullptr)
    {
        cout << string(2*depth, ' ');
        cout << p->name << endl;
        for(int k = 0; k < p-> children.size(); k++)
            printTree(p->children[k], depth + 1);
    }
}
```

- **pre-order** traversal of the tree
  - the node is processed before the subtrees
- count tree is **post-order** traversal
  - node is processed after the subtrees

# Binary Trees

A binary tree is empty, or a node with a left binary tree and a right binary tree

- distinguishes between a left child and right child

Every Tree can be represented as a binary tree

- All siblings are connected by right pointers

- All children are connected by left pointers
- pointers can be renamed oldestChild and nextYoungestSibling

A **binary search tree (BST)** is empty, or a node with a left binary tree and a right binary tree such that:

- the value at every node in the left subtree is <= the value at this node
- the value at every node in the right subtree is >= the value at this node

# Insertion and Deletion on Binary Trees

Advantage over binary search on vector -> easy and cheap insertion and deletion

**Insertion:**

- follow tree until nullptr, insert element there

**Deletion:**

Three cases

1. Leaf:
   Easy, simply delete the node and reset the pointers
2. Node with 1 branch:
   Store branch, delete node, then replace node with stored branch
3. Node with 2 branches:
   - Need to find child node to be "promoted", replacing deleted node
   - Either the largest child in the left branch or the smallest child in the right branch
   - Then, delete the node to be promoted, using the two algorithms above, and replace the original deleted node with the promoted node

With a reasonably balanced binary search tree, Insertion, deletion, and lookup are all $O(logN)$

When selecting the node to be promoted, if one side is selected a lot, the tree may become unbalanced

- usually alternating choices works
  - if there is periodicity in the data, could just select randomly

## Printing a BST

```
void printTree(const Node* p)
{
    if(p != nullptr)
    {
        printTree(p->left);
        cout << p->name << endl;
        printTree(p->right);
    }
}
```

**Inorder Traversal:** process left before, right after

## 2-3 Tree

- Nodes can have 2-3 children
- Nodes with 3 children have 2 values
- Left child is less than both
- Middle child is in between both
- Right child is greater than both*

# Sets

- no duplicates, lookup by value

```
#include <set>

set<int> s;
s.insert(10);
s.insert(30);
s.insert(10);
s.insert(5);
s.find(30);
for(set<int>::iterator p = s.begin(); p != end(); p++)
    cout << *p << endl; //writes 5 10 30
```

- set needs a less than operator
- compare with itself returns false
- considered duplicates if neither one is less than the other

# Hash Tables

Data type for quick lookups

## Structure

Array of linked lists

**Ex** : Student ID numbers

### Insertion

- Student ID inserted at index (a "bucket") with same number as first n digits of ID number
- a **collision** occurs when more than 1 item ends up in the same bucket
  - Want to keep collisions to a minimum

### Note

- Keys sometimes have patterns, elements end up in a lot of the same buckets
  - ID numbers have a check digit
- **load factor** average number of items in each bucket: $\frac{number of items}{number of buckets}$
  - tend to choose around 0.7
- if key is not int, can convert key into an int using a **hash function**
  - Use large prime as # of buckets to hash, avoids collisions

#### String Hash Function

Typical Format

```
unsigned int h = 2166136261u //use unsigned int for u
for(char c: string s)
{
    h += c;
    h *= 16777619
}
return h;
```

Usage

```
#include <functional>
using namespace std;

string s = "hello";

unsigned int x = std::hash<string>()(s) % numberOfBuckets; //overloaded function call ope
```

# Hash Functions

- produce uniformly distributed values
- cheap
- deterministic

# Time Complexity

For a hash table with a constant number of buckets, operations are O(N)

- still very good for small n, better than logN

To make it faster, assume maximum load factor

- when maximum load factor exceeded, rehash the table by doubling the number of buckets
  - lookups become constant time in general
  - rehashing takes time, but doesn't occur much

# Incremental Rehashing

Rehash a constant number of items from old table every time a new item is inserted

- lookup needs to look in two tables, but still constant
- no extremely expensive rehashing step
    - upper bound for time complexity

# Maps (STL)

Implemented using BST

```
#include <map>
maps<string, double> ious; // key type requires < operator
while(cin >> name >> amt)
  ious[name] += amt;
for(map<string, double>::iterator p = ious.begin(); p != ious.end(); p++) //always visits
  cout << p->first << endl;

// ious
// ===
// fred ==> 13
// ethel ==> 5
```

# Unordered Sets

Implemented using hash table

```
#include <unordered_set>

unordered_set<int> s;
s.insert(10);
s.insert(30);
s.insert(10);
cout << s.size(); //2
if (s.find(20 == s.end()))
  cout << "20 is not in the set";
for(unordered_set<int>::iterator p = s.begin(); != s.end(); p++)
  cout << *p << endl;
s.erase(30);
```

# Heaps

Every item has a priority, highest priority is returned first

A *complete binary tree* is a binary tree that is completely filled at every level, except possibly the deepest level, which is filled from left to right.

A (max) heap is a complete binary tree in which the value at every node is greater than values of all the nodes in its subtrees

A min heap is a complete binary tree in which the value at every node is less than values of all the nodes in its subtrees

## Deletion

- remove root
- remove last item added and set as root (to maintain complete binary tree)
- reform the heap quality
    - "trickle" the new root down
- O(logN), better than binary search tree

# Insertion

- add element in proper place for complete binary tree
- "bubble" element up to proper place
    - compare with parents, and if bigger, swap
- O(logN) also better than binary search tree

# Array Representation

$parent(i) = \lfloor \frac{i-1}{2} \rfloor$
$children(j) = 2j + 1, 2j + 2$

# Heapsort

Sorting by putting elements in array, then taking out, as it comes out in order of priority

$$O(N!) = O(NlogN)$$

- turn array into heap
    - starting with leaf nodes, incrementally make the smaller sub-trees heaps
    - $O(N)$
- repeatedly remove items from the heap
    - take first item and swap with last (equivalent to deletion of first item)
    - size of heap goes down 1
    - repeat