Disjoint Sets:

Implementation:	Constructor	Connect	isConnected
ListOfSetsDS	$\Theta(n)$	$O(N)$	$O(N)$
QuickFindDS	$\Theta(n)$	$\Theta(N)$	$\Theta(1)$
QuickUnionDS	$\Theta(n)$	$O(N)$	$O(N)$
WeightedQuickUnionDS	$\Theta(n)$	$O(\log N)$	$O(\log N)$

WQU worst height is $\Theta(\log n)$ and best height is $\Theta(1)$.

Data Structures

Constructor always adds N elements

→ If nothing is connected, we must go through all elements

→ Constructor still adds N elements hence $\Theta(n)$ time

→ Connect requires going through array and manually changing each index } is connected is linear because we can just see if the numbers at index match

→ Assigned items a parent resulting in a tree like shape

→ If "tree" is spindly then we have to climb up the tree resulting in a worst case of $\Theta(N)$ time

→ When we connect, we can check the size of each set. We make the root of the heaviest set the combined root of both sets (AKA link up shorter tree below larger tree) worst case is if during all connect operation, if the trees being connected are the same (∴ $O(\log n)$ runtime).

WQU with Path Compression → tie all nodes seen by the root after every isConnected call and union call ∴ runtime is $O(M \log^* n)$

Binary Search Trees:

worst case when spindly, best case when bushy ($\Theta(n)$, $\Theta(\log n)$ respectively)

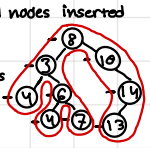
Binary Search tree: the left child must be smaller than the node and the right child must be greater than the node:

Min/Max-heap: For min-heap all descendants must be smaller than the root and vice versa. The Min-Max heap must always be a complete binary search tree while maintaining the above properties

Weighted QuickUnionDS: When connecting two elements, if they are tied arbitrarily break up the tie, else make the root of the node belonging to the "heavier" tree the root of the lighter nodes root.

2-3 tree: "overstuff" nodes until they are overfilled (2 or more elements) in that case, split the node (middle element is parent of other two nodes) Each can have a max of two elements and have a max of 3 children. 2-3 trees are always balanced

Left leaning Red black tree: Same as 2-3 tree but 2 nodes are split and connected by a red glue link. All nodes inserted automatically have a red glue link.



Tree traversal:

Pre-order: Visit, recurse left, recurse right (when we pass the left of a node

In-order: recurse left, Visit, recurse right (when we pass the bottom)

Post-order: recurse left, recurse right, visit (when we pass the right)

level order: visit level by level from left to right

Deletion:

→ **Deletion has no Children:** remove reference to that node

→ **Deletion has one child:** point Deletion keys parent to its child

→ **Deletion has two children:** swap Deletion key with either greatest key on left side or smallest key on right

B-Trees: "*just keep overstuffing the leaves*"

2-3-4 Trees: Nodes can have either 2, 3, or 4 children

2-3 Trees: Nodes can have either 2 or 3 children

left leaning-Red black Tree: Overstuffed Node are "split" are are connected with virtual nodes

2-3-4 and 2-3 tree insertion and deletion rules:

In a 2-3-4 tree or 2-3 tree, we want to keep stuffing nodes until it becomes overstuffed

Once it becomes overstuffed, we push the middle node up to its parent node and split the node at the middle node we then check if the parent node is overstuffed, we continue the same process as before and so on. Else we stop.

If the node has no parent then we split at the middle and the middle node becomes the new parent of the entire tree.

Left leaning Red black insertion and deletion rules:

Every time that we add a node, we always add a red virtual link to it.

If there is a right leaning 3-node, we have a left leaning violation so we must Rotate left the appropriate node to fix

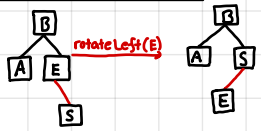
If there are two consecutive left links, we have an Incorrect 4 Node violation we must Rotate right the appropriate node to fix

If there are any nodes with two children, we have a temporary 4 Node then we must color flip the node to emulate the split operation

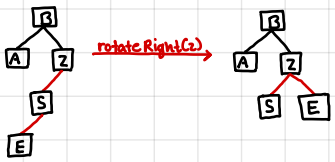
IT IS POSSIBLE THAT A ROTATION OR FLIP WILL CAUSE AN ADDITIONAL VIOLATION THAT NEEDS FIXING → CASCADING OPERATIONS

Left leaning Red black insertion and deletion examples:

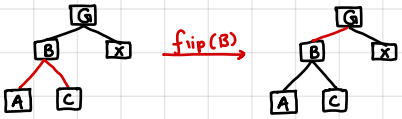
Left leaning violation:



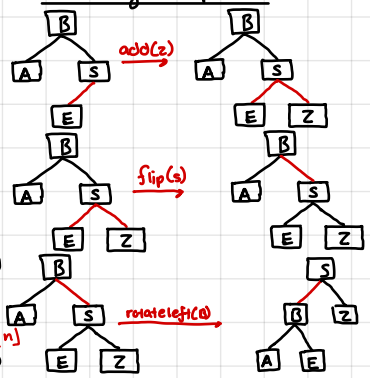
Incorrect 4 node violation:



Temporary 4 node violation:



Cascading example:



Rotateleft(node): Let x be the right child of node. Make node the new left child of x

Rotateright(node): Let x be the left child of node. Make node the new right child of x

Rotation allows for balancing of a BST in $O(n)$ move: ROTATION ALLOWS PERSERVES SEARCH ORDER

Fun Facts: Every path from the root to null has the same number of black links (because 2-3 trees have the same number of links)

LLRB has height $O(\log n)$
Contains is trivially $O(\log n)$
Insert is $O(\log n)$ as well as rotations

} Given 2-3 tree of height n, the maximum nodes that can be packed are $3^{n+1}-1$.
A LLRB's height won't be more than $2h+1$ where h is the height of the 2-3 tree.
Maximum height of LLRB is when the entire left side is stuffed and the minimum is $\lfloor \log_2 n \rfloor$

Asymptotics: $\Theta(f(N))$ → order of growth is equal to $f(N)$; $O(N)$ order of growth is less than or equal to $f(N)$; $\Omega(f(N))$ order of growth is greater than $f(N)$

Important series:
Sum of natural number: $1+2+3+\ldots+Q = Q(Q+1)/2 = \Theta(Q^2)$
Sum of first powers of 2: $1+2+4+8+\ldots+Q = 2Q-1 = \Theta(Q)$

} use the following formula

$$\sum_{i=0}^{\#\text{ of layers}} \frac{\text{work}}{\text{nodes}} * \frac{\#\text{ of nodes}}{\text{layer}}$$

arithmetic series converges to $\Theta(n^2)$
=> geometric series converges to $\Theta(n)$

Master's Theorem:

If $aT(\lceil n/b \rceil) + O(n^d)$ a = size layers grow by, n/b = how n changes per layer, $O(n^d)$ = work done per layer

$$T(n) \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Recursion:

Method: count the number of calls to the function and then multiply by work done per call.

Other method: use recurrence relation

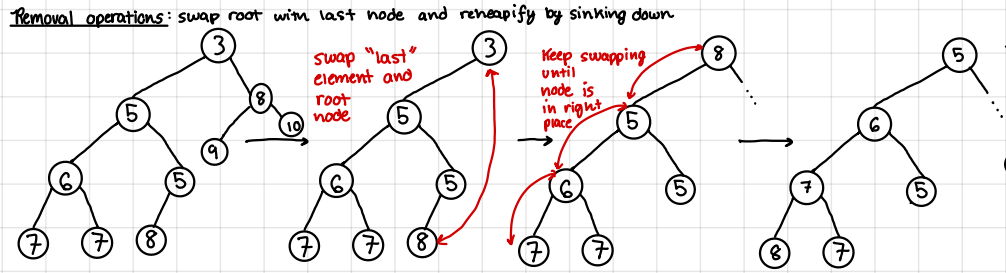
1. Draw the tree
2. Using the tree fill out the table with rows labeled level, size, number of nodes, cost per node, total cost (cost per node × # of nodes)
3. Compute total cost at i and use the summation $\sum_{i=0}^{}$ (total cost at i) which should give the runtime

Reminder: Θ is used when the tightest upper and lower bounds are the same otherwise Ω and O.

Big O is the upper bound, Omega is the lower bound, and theta is the "average" time

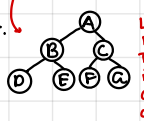
Min-Max heap insertion and deletion:

Removal operations: swap root with last node and reheapify by sinking down



Array representation of BST/MinMax heap:

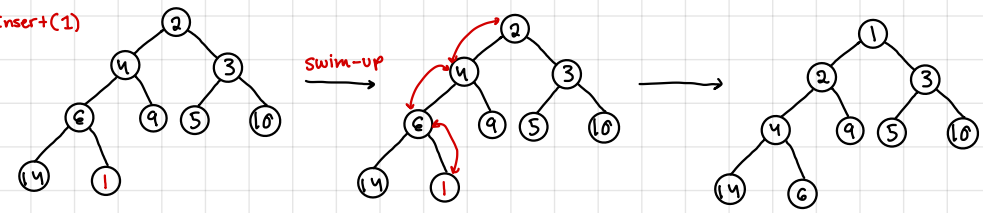
[-, A, B, C, D, E, F, G]



List out the array Level wise. The parent of an index is $\lfloor \frac{n}{2} \rfloor$, the left child is $2n$ and the right child is $2n+1$

Addition Operations: When we add we reheapify (e.g. swim up)

Insert(1)



Min-heap Operation	Runtime
create	$O(n \log n)$ (inserting 1 by 1) $\Theta(\log n)$ (bottom-up heapification)
insert	$O(\log n)$
Find-min	$\Theta(1)$
Remove-min	$O(\log n)$

```
for (int i=0; i<N; i*=3){
    System.out.println(i);
}
```
$\Theta(\log N)$

i	Work
1	1
3	1
9	1
⋮	⋮
3^k	1

$\log_3(3^k) = \log_3(N)$, $k = \log_3(N)$

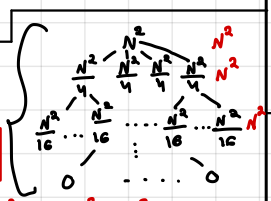
$\sum_{i=0}^{\log_3 N}(1) = \Theta(\log N)$ $\begin{cases} O(\log N) \\ \Omega(\log N) \end{cases}$

```
public static void p1(int N){
    for(int i=0; i<N; i+=1){
        for(int j=0; j<N; j+=2){
            // something linear
        }
    }
}
```
$\Theta(n^2)$

```
public static void p1(int N){
    for(int i=0; i<N; i+=1){ ← N
        for(int j=0; j<N; j*=2){ → log N
            // something linear
        }
    }
}
```
$\Theta(n \log n)$

```
public static void p1(int N){
    for(int i=0; i<N*N; i*=2){ → 1+2+4+8+...+N^2
        for(int j=0; j<i; j++){ → 1+2+4+8+...+N^2 → Θ(N^2)
            // something linear
        }
    }
}
```

```
public static void f4(int N){
    if(N==0){ return; }
    f4(N/2); f4(N/2); f4(N/2); f4(N/2);
    g(N); // runs in Θ(n^2) time
}
```



$\log_2 N$ levels

$N^2 + N^2 + \dots + N^2 = N^2 \log N$ ($\log N$ terms)

$\Rightarrow \Theta(N^2 \log N)$

```
public static void g1(int N){
    for(int i=0; i<N; i+=3){
        for(int j=0; j<i; j+=1){
            // something
        }
    }
}
```
$3+6+9+12+\dots+N \Rightarrow \Theta(N^2)$

```
public static void g2(int N){
    for(int i=0; i<N; i*=2){
        g2(i);
    }
}
```

$1+2+4+8+\dots+N/2 = \Theta(N)$

```
for(int i=0; i<N; i+=1){
    int numJ = Math.pow(2,i+1)-1
    for(int j=0; j<numJ; j+=1){
        // something
    }
}
```

i	work
1	1
2	4
3	8
4	16
5	32
6	64
⋮	⋮
N	2^N

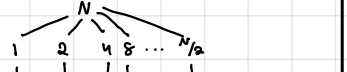
$\sum_{i=1}^{N} 2^N = 1+2+4+8+16+\dots+2^N \Rightarrow \Theta(2^N)$

```
for(int i=0; i<N; i++){
    for(int j=0; j<i; j++){
        // something
    }
}
```

i	work
1	1
2	2
3	3
4	4
5	5
⋮	⋮
N	N

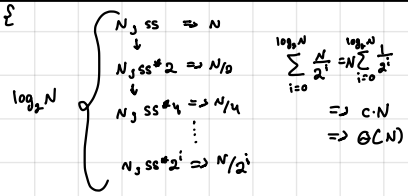
$1+2+3+4+\dots+N \Rightarrow \Theta(N^2)$

```
for(int i=0; i<N; i*=2){
    // recurse on i
}
```



$\Rightarrow \Theta(N/2) = \Theta(N)$

```
for(int i=0; i<N; i+=ss){
    // something constant
}
func(N, ss*2)
```



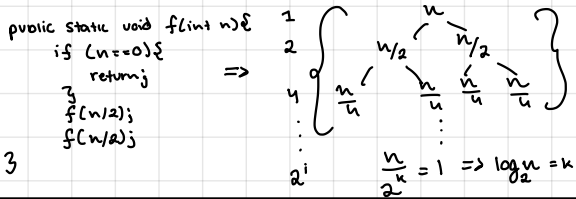
$\log_2 N$

$\sum_{i=0}^{\log_2 N} \frac{N}{2^i} = N \sum_{i=0}^{\log_2 N} \frac{1}{2^i}$

$\Rightarrow c \cdot N$

$\Rightarrow \Theta(N)$

```
public static void f(int n){
    if(n==0){
        return;
    }
    f(n/2);
    f(n/2);
}
```

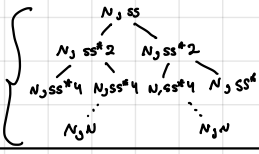


Asymptotics

$\log_2 N$ layers

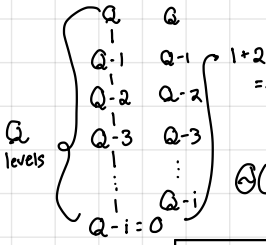
$\sum_{i=0}^{\log_2 N} 2^i = (1+2+4+\dots+N) \Rightarrow \Theta(N)$

```
for(int i=0; i<N; i+=ss){
    // something constant
}
func(N, ss*2)
func(N, ss*2)
```



$\sum_{i=0}^{\log N} N = N \sum_{i=0}^{\log N} 1 \Rightarrow N(\log N) \Rightarrow \Theta(N \log N)$

```
public void func(int N){
    int Q = 0;
    for(int i=1; i<N; i*=3){
        Q += 1;
    }
    disco(Q);
}
public void disco(int Q){
    if(Q==0){
        return;
    }
    for(int i=0; i<Q; i+=2){
        // something constant
    }
    disco(Q-1);
}
```



$1+2+3+\dots+(Q-3)+(Q-2)+(Q-1)+Q \Rightarrow \Theta(Q^2)$

$Q = \log_3 N$

$\Theta(Q^2) = \Theta(\log^2 N)$

```
if(N%2==0){
    return;
}
for(int i=1; i<N; i*=2){
    if(n(i)){
        n(i);
    }
}
```
best case $n/2 \Rightarrow \Theta(1)$

worst case $\neg(n/2) \Rightarrow$

i	work
1	1
2	2
4	4
8	8
16	16
⋮	⋮
N	N

$1+2+4+8+16+\dots+N \Rightarrow \Theta(N)$ worst case

```
if(n(N)){
    return;
}
for(int i=1; i<N; i*=2){
    for(int j=N-i; j<N; j++){
        int a = 2;
    }
}
if(N%2!=0){ return; }
if(N<=10){ return; }
```
best case n(n) evaluates to true $\Rightarrow \Theta(n)$

worst case $\Rightarrow$

i	work
1	1
2	2
4	4
8	8
⋮	⋮
N	N

$1+2+4+8+\dots+N \Rightarrow \Theta(N)$ worst case

Geometric resizing has a runtime of $O(N)$.
Arithmetic resizing has a runtime of $O(N^2)$.
Remember that better running algorithm may run worse than worser algorithms for smaller value of n

Remember that the TreeMap uses .compareTo method of the object in order to sort its keys and remember the HashSet/HashMap uses .hashcode and .equals. Wrong changes to any of these method (such as making the method rely on deterministic numbers) will destroy the data structure and lead to corrupted storage. HashSet/HashMap are not secure against attacks as they are vuenrable to collision based attacks.

Dynamic Method Selection:

	Static variable	static method	Instance var	Instance method
Type used by compiler	static type	static type	static type	static type
Type used by executer	static type	static type	static type	dynamic type

Instance var look up: start from static type class; If not there compile error
Instance method look up: start from dynamic type class. If not there go to superclass until object class is reached
Variables aren't polymorphic, Only instance methods are.

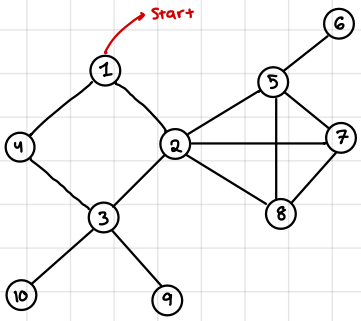
1. Linked List	2. ArrayList	3. Binary Search Tree	4. Balanced trees (2-3 tree and LLRB)	5. HashTable (2-3 Tree and LLRB) (amortized $\Theta(1)$ time)
get() = $O(N)$	get() = $O(1)$	search() = Average: $O(\log N)$ Worst: $O(N)$	get() = $\Theta(\log N)$	get() = Average: $O(1)$ Worst: $O(N)$
add() = $\Theta(1)$	add() = $\Theta(N)$	insert() = Average: $O(\log N)$ Worst: $O(N)$	insert() = $\Theta(\log N)$	insert() = Average: $O(1)$ Worst: $O(N)$
remove() = $\Theta(N)$	remove = $\Theta(N)$	delete() = Average: $O(\log N)$ Worst: $O(N)$	remove() = $\Theta(\log N)$	remove() = Average: $O(1)$ Worst: $O(N)$
space = $O(N)$	space = $O(N)$	space = $O(N)$	space() = $O(N)$	space() = $O(N)$

6. Stack
push() = $\Theta(1)$
pop() = $\Theta(1)$
peek() = $\Theta(1)$
space() = $O(N)$

7. Queue
add() = $\Theta(1)$
remove() = $\Theta(1)$
peek() = $\Theta(1)$

8. Heap
search() = $\Theta(1)$
insert() = Average: $O(1)$ Worst: $O(\log N)$
delete() = $O(\log N)$
peek() = $O(1)$

Traversing Graphs:



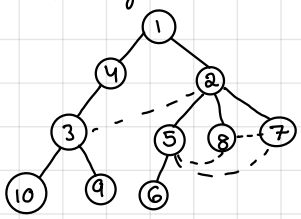
Visit all adjacent vertices and then traverse through vertex

Breadth First Search: 1, 4, 2, 3, 5, 7, 10, 9, 6

Queue

1	4	2	3	5	7	8	10	9	6

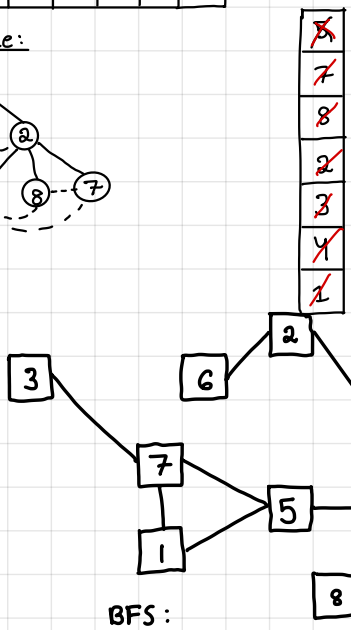
BFS spanning tree:



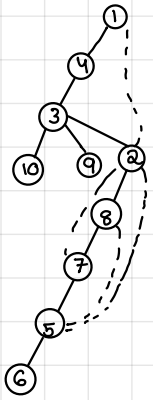
Algorithm: Start at node → Suspend first node in stack and traverse through child → If child does not have children look in stack and traverse next child of node on top

DFS Preorder: 1, 4, 3, 10, 9, 2, 8, 7, 5, 6

Stack:



DFS Spanning tree:



Heights, nodes, and everything else about trees:

Min heap:

- Largest item is always at the lower level on the left or right hand side.
- The minimum item is always the root node.
- The median item can occur at a node if the node is not a root node and if the number of ancestors of the node is less than or equal to the number of elements above the median.
- After inserting a node, either the inserted node is the new minimum or the previous minimum is still the min-imum.

BFS Search:

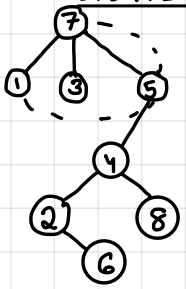
7 1 3 5 4 8 2 6

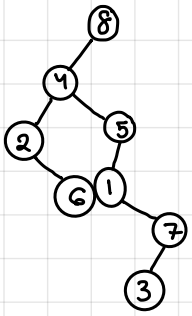
BFS:

Q:

7	1	3	5	4	2	8	6

DFS PRE-ORDER: 8 4 2 6 5 1 7 3





BST tree Properties:

- BST for Sets: $\Theta(\log n)$ if balanced; Hibbard deletion
- BST: Random n insertions averages out to $\Theta(\log N)$ for depth and runtime
- ALL left nodes are smaller; ALL right nodes are greater

B-Tree (2-3 Tree): Always balanced, slow and complicated

- Balanced, $\Theta(\log N)$ for contains() and add()
- Inv: All leaves must be the same distance from the source
- Inv: A non-leaf node with K items has exactly K+1 children
- height always between $\sim \log_2 N$ and $\log_{L+1} N$ where L is the node limit
- A 2-3 tree with a height of h can have a maximum of $3^{h+1}-1$ elements and therefore $3^{h+1}-1/2$ nodes
- Contains worst case: number of nodes to inspect is height+1
- Worst case number of items to inspect per node: L
- Overall runtime is $O(HL) \rightarrow O(L \log N) \rightarrow O(\log N)$

- Smallest possible height is when all nodes have L items ($\log_{L+1}(N)$)
- Largest possible height is all non-leaf items have 1 item ($\log_2(N)$)
- Non-leaf node with K items must have exactly K+1 children

Stack:

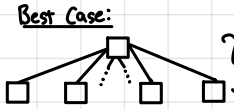
Left-Leaning Red Black tree:

- No node has 2 red links.
- Every path from root to leaf has same number of black links (because 2-3 trees have same number of links to every leaf).
- Maximum height of a LLRB is never more than $\sim 2\times$ the height of its 2-3 tree
- Worst case height of a LLRB is if there exists a path in the corresponding 2-3 tree such that all nodes on that path have 2 items in them

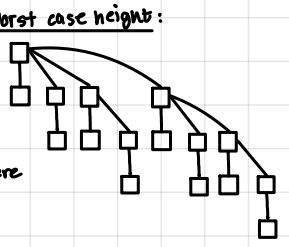
Weighted Quick Union:

- Best case height is 1 and worst case height $L \log_2 N \rfloor$

Best Case:



Worst case height:



HashMap:

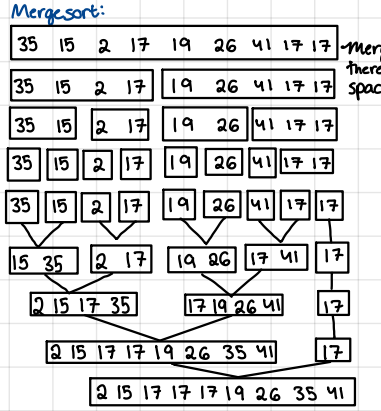
- If we change the .equals method of an object then we must change the .hashcode() function
- Valid hash function return the same hashcode for the same function no matter what
- For negative hashcodes, start at the last index and countdown all way to -1, where ever it stops is where you put the object.
- When we resize, we rehash all of the previous buckets into the new buckets
- In the best case with a good hash function, inserting n elements will take n time
- With a really bad hashfunction it will take $O(n^2)$ time to insert n items because we would have to scan through a bucket of n items to see if it contains the item and then add the item
- When thinking about putting, think about how many elements you are putting and how much time it takes to put one element

Shortest paths: BFS returns path with fewest edges but not necessarily the shortest path
DFS is worse for spindly tree since call stack gets very deep (needs $\Theta(V)$ memory)
BFS is horrible for very bushy graphs - Queue get very very large
Dijkstra's algorithm: Find the shortest possible path from the starting node to all other nodes
A* is like Dijkstra's algorithm, however, Each node is assigned a heuristic — all consistent heuristics are admissible (:= N) — admissible means that the heuristic never overestimates
In order for the A* heuristic to be valid, Each heuristic must be ≤ the true distance (Admissible) and for each neighbor of w, the estimated cost of reaching the goal from w is no greater than the step cost of getting to N plus the estimated cost of reaching the goal from N (consistent)
Time of A* is dependant on the heuristic
Prims - Start from the node and connect the neighbor whose distance is the smallest from the tree (Dijkstra's is from the starting node)
Kruskals - Connect vertices with the smallest edge weight between them. If the two nodes are already connected to the WQU, ignore the edge and proceed to the next smallest weight

Sorts:

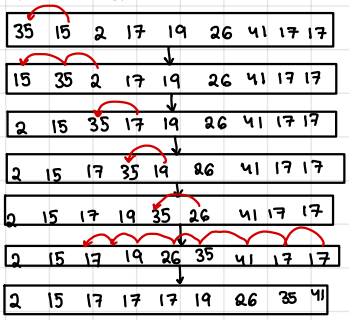
Selection Sort: Find the smallest item, swap this item to the front and fix in place, Repeat until all items are fixed → $\Theta(N^2)$ regardless if sorted or not $\Theta(1)$ space
Inplace Heapsort: Treat the input array as a heap, keep sinking nodes until the array is heapified, swap front to the back of the unsorted array
array of all duplicates yields linear runtime and heap operations are iteratively implemented rather than recursively → $\Theta(N)$ best case $\Theta(N \log N)$ worst case $\Theta(1)$ space

Mergesort:



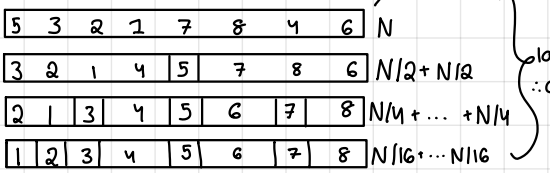
Merging takes N time which we do log N times therefore merging takes $\Theta(N \log N)$ time and $\Theta(N)$ space complexity of $\Theta(N)$

Insertion Sort:



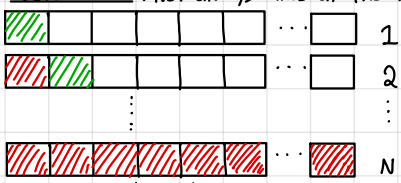
$\Omega(N)$ when array is already/almost sorted
$O(N^2)$ when array is completly unsorted array
Keep swapping elements until element on the right is greater than the element on the left
Runtime is $\Theta(N+K)$ where K is number of inversions
An almost sorted array is an array whose number of inversions is less than cN for some c.
Insertion sort is excellent on these arrays.

Quicksort: Best case: Pivot always lands in the middle



Tony Hoare partition algorithm is faster than 3-scan algo

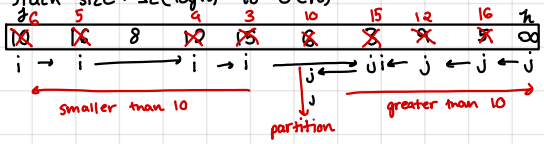
Worst case: Pivot always lands at the beginning of Array



/// = pivot
/// = sorted

Worst case: $\Theta(N^2)$
Best case: $\Theta(N \log N)$
Average case: $\Theta(N \log N)$
QUICKSORT IS HORRIBLE FOR ALMOST SORTED LIST

Select middle element as pivot or select random element as pivot to "Remove" worst case
Stack size: $\Omega(\log n)$ to $O(n)$




```
partition(f, h)
  pivot = A[f]
  while(i<j){
    do{
      i++j
    } while(A[i] ≤ pivot);
    do{
      j--j
    } while(A[j] > pivot);
    if(i<j){
      swap(A[i], A[j])
    }
  }
  return j;
```

A sort is said to be stable if order of equivalent items is preserved (e.g. equivalent items don't cross over when being stably sorted)
Insertion sort is stable because equivalent items never move past their equivalent brethren
Quicksort can be stable depending on the partitioning strategy

Counting Sort:

S	Lauren
H	Delbert
O	Glaser
C	Edith
S	Js
O	Sandra
M	Swimp
H	James
C	Lee
H	Dave
C	Bearman
O	Lisa

Alphabet: {C, S, H, O}

0	C	3
1	S	2
2	H	4
3	O	3

counts

0	C	~~0~~ ~~2~~ 3
1	S	~~3~~ ~~4~~ 5
2	H	~~5~~ ~~6~~ ~~7~~ ~~8~~ 9
3	O	~~9~~ ~~10~~ ~~11~~ 12

Starting Points

C	Edith
C	Lee
C	Bearman
S	Lauren
S	Js
H	Delbert
H	Swimp
H	James
H	Dave
O	Glaser
O	Sandra
O	Lisa

Algorithm:
1. Go through original list counting the number each alphabet appears
2. Find the starting positions
3. Iterate through list again incrementing the list of starting points. The algorithm stop when the counter of the last alphabet reaches the length of the array

Total Runtime on N keys with alphabet of size R is $\Theta(N+R)$
- Creating and filling our count related array : $\Theta(R)$
Counting each item and copying into new array $\Theta(N)$
Counting sort is stable

Primitive values do not need stability because primitives cannot be distinguished from one another if they are both equal by .equal. However, the same is not applicable for objects since two different objects at different addresses can still be equal therefore stability may be desireable when sorting object hence java uses quicksort for primatives and mergesort for objects

Sorting Runtimes and Stability:

* Assumes constant compareTo time

Sorting Algo	worst	Average	Best	space	Stable
* insertion sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$	✓
selection sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$	×
* merge sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(n)$	✓
* heap sort	$\Omega(N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$	×
* Quick sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(\log(N))$	×
MSD sort	$\Omega(N+R)$	$\Theta(WN+WR)$	$O(WN+WR)$	$O(N+R)$	✓
LSD sort	$\Omega(WN+WR)$	$\Theta(WN+WR)$	$O(WN+WR)$	$O(N+R)$	✓
Counting sort	$\Omega(N+R)$	$\Theta(N+R)$	$O(N+R)$	$O(N+R)$	✓

→ Only when Tony Hoare is used, not when 3-scan is used

LSD sort is much faster than mergesort when there are a large number of strings and all strings are similar.
when n is large the asymptotic benaviour of LSD sort beats mergeSort and when strings are highly similar mergesort takes linear time to compare strings.

We use LSD over counting sort because although the runtime is faster, counting sort can take a huge amount of memory. for large N, counting sort is faster
counting sort is stable as opposed to quicksort
Counting sort does not need comparisons which might be useful when elements are not comparable.

Algorithm Implementations:

Invariants:
edgeTo[v] is the best known predecessor of v
distTo[v] is the best known distance from source to v
PQ contains all unvisited vertices in order of distTo

Dijkstra's:

```
PQ.add(Source, 0)
For all other vertices v, PQ.add(v, infinity) → adding V times totaling to VlogV
While PQ is not empty
    p = PQ.removeSmallest();        → remove smallest V times totaling VlogV
    if distTo[p] + w < distTo[p]:
        distTo[p] = distTo[p] + w
        edgeTo[q] = p
        PQ.changePriority(q, distTo[q]) → Change priority of E edges totaling ∴ Runtime ⇒ O(VlogV + VlogV + ElogV)
                                          ElogV time
```

Prims:

```
public PrimMST(EdgeWeightedGraph G){
    edgeTo = new Edge[G.V()];
    distTo = new double[G.V()];
    marked = new boolean[G.V()];
    fringe = new SpecialPQ<double>(G.V()); → initialize PQ of V item ∴
    distTo[s] = 0.0;
    setDistancesTo∞ExceptS(s) → Must be a special PQ like Dijkstras → fill fringe with V items taking VlogV time
    while (!Fringe.isEmpty){
        int v = fringe.delMin(); → Vertex is closest so add to MST → delete V items from the fringe
                                   taking VlogV time
        for (Egde e : G.adj(V)){
            int w = e.other(v);
            if (marked[w]){continue;} → Already in MST, so go to next edge
            if(e.weight() < distTo[w]){ → Better path to a particular vertex found
                distTo[w] = e.weight();   so update current best known for that
                edgeT[w] = e;             vertex
                pq.decreasePriority(w, distTo[w]); → decrease priority E times taking ElogV time
            }
        }
    }
}
```

∴ Total time complexity is O(VlogV + VlogV + ElogV) time

Kruskals → instance variable: private List<Edge> mst = new ArrayList<Edge>();

```
public KruskalMST(EdgeWeightedGraph G){
    MinPQ<Edge> pq = new MinPQ<Egde>(G);
    for (Edge e : G.edges()){
        pq.insert(e);   ⟶ O(ElogV) total
    }
    WeightedQuickUnionPC uf = new WeightedQuickUnionPC(G.V()); → O(log*V) total
    While(!pq.isEmpty() && mst.size() < G.V()-1){
        Edge e = pq.delMin(); → O(ElogE) total
        int v = e.from();
        int w = e.to();
        if(!uf.connected(v,w)){ → O(Elog*V) total
            uf.union(v,w);      → O(Elog*V) total
            mst.add(e);         → O(E) total
        }
    }
}
```

∴ Kruskals algorithm will run in O(ElogE) is edges are unsorted and O(Elog*V) if sorted

```
private void dfs(Graph G, int v){
    marked[v] = true;
    for (int w : G.adj(v)){
        if(!marked[w]){
            edgeTo[w] = v;
            dfs(G,w);
        }
    }
}
```

O(V+E)
- Each vertex is visited at most O(V)
- Each edge is considered at most twice O(E)

```
private void bfs(Graph G, int s){
    Queue<Integer> Fringe = new Queue<Integer>();
    fringe.enqueue(s);
    marked[s] = true;
    while(!fringe.isEmpty()){
        int v = fringe.deque();
        for (int w : G.adj(V)){
            if(!marked[w]){
                fringe.enqueue(w);
                marked[w] = true;
                edgeTo[w] = v;
            }
        }
    }
}
```

O(V+E) as well O(V²) in adj. matrix because we have to create V iterators which takes Θ(V) time to iterate over

Recursive Depth First Search Algorithms:

```
Preorder:
void preorderTraverse(Graph G, Node v){
    if v is unmarked:
        mark(v)
        visit(v)
        for Edge(v,w) in G:
            traverse(G,w)
```

```
Postorder:
void PostorderTraverse(Graph G, Node v){
    if v is unmarked:
        mark(v)
        for Edge(v,w) in G:
            traverse(G,w)
        visit(v)
```

DFS - PostOrder traversal:

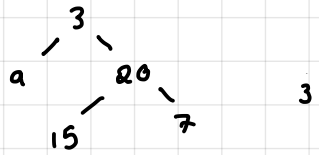
DFS-Postorder is different from pre-order is different in the fact that we note down elements when they hit a dead-end (eg no other new vertices to explore).

Sorting Algorithms Implementation:

```
public static void selectionSort(Comparable[] a){
    int n = a.length;
    for(int i=0; i<n; i++){
        int min = i;
        for(int j=i+1; j<n; j++){
            if(less(a[j], a[min])) min = j;
        }
        exchange(a, i, min);
    }
}
public static void insertionSort(Comparable[] a){
    int n = a.length;
    for(int i=1; i<n; i++){
        for(int j=i; j>0 && less(a[j], a[j-1]); j--){
            exchange(a, j, j-1);
        }
    }
}
public static double[] mergeSort(double[] input){
    int N = input.length;
    if (N<=1) return input;
    double[] a = new double[N/2];
    double[] b = new double[N-N/2];
    for(int i=0; i<a.length; i++)
        a[i] = input[i];
    for(int i=0; i<b.length; i++)
        b[i] = input[i+N/2];
    return merge(mergesort(a), mergesort(b));
}
```

Recursive MSD

```
public static List<T> msd(List<T> items, int index){
    if(index >= items.get(0).length() || items.size() <=1){
        return items;
    }
    int start = 0;
    List<T> answer = new ArrayList<>();
    stableSort(items, index);
    for(int end=1; end<=items.size(); end++){
        if(end==items.size() || items.get(start).charAt(index) !=
                items.get(end).charAt(index)){
            List<T> sublist = items.subList(start, end);
            answer.addAll(msd(sublist, index+1));
            start = end;
        }
    }
    return answer;
}
```



Dangers and Benefits of all Graph Traversal and Sorting Algorithms:

DFS - Stack Overflow due to opening to many recursive calls on the stack. / Good for pathfinding from one node to a node deep in a tree

BFS - Memory Overflow due to overflowing the Queue because of a bushy graph / Good for finding min path from starting node to all other nodes in the graph

Dijkstra's - Good for finding all shortest paths from a node to all other nodes

A* - Good for finding shortest path between two points / Horrible when heuristic is bad however bad heuristic does not mean that shortest path is incorrect / In place ∴ less mem. O(log n)

Prim / Kruskal - Good for calculating minimum spanning trees using the cut property

Divide and Conquer Algorithms:
- Mergesort good for ≥O(N) inversion, mergesort better with linked lists, Quicksort better with arrays
- Quicksort - Really, Really good when number of inversions are huge / Horrible for lists that have very few inversion because pivots will always land close to front causing O(N) time
- Mergesort - Worst case of O(nlogn), even though slightly slower empirically than quicksort, O(n²) worst case for quicksort is avoided / Requires O(N) memory for creation of new array

Iterative Algorithms:
- Heap sort - Slowest between Mergesort and Quicksort / Does not need deep recursion or auxiliary arrays in order to work
- Insertion sort - Strong for short list/lists lists with few number of inversions (≤ cN inversions) / Kinda bad for list with large inversions / Good when size < 15 (empirical fact)
- Selection sort - Performs Θ(N) swaps in the worst case / Always Θ(N²) because even if array is sorted we still must find the minimum and place it in its place (even if array is sorted)

Radix Sorts:
- Counting Sort - Good when alphabet size is low → Counting sort is usually used with LSD or MSD / Atrocious for large alphabets - Count sorting integers from 0-1,000,000 will take long
- LSD Sort - In order to combate larger alphabet size we can count sort each digit individual digit (least sig. digit) / Annoying because runtime is dependant on length of longest key
- MSD sort - Similar to LSD but sort from right to left / Best case is when last digits are different requiring one pass / Degrades to LSD sort in the worst case / Good if lengths vary

ALL COUNTING SORTS DON'T REQUIRE COMPARISONS HOWEVER THEY ALSO NEED A STABLE SUBROUTINE

Java Quirks:

Casting rules:

Assume we have the following cast: `A a = (C) d`

Rule 1 (Compile time checking):
The type of d and C must have some relation (child to parent or parent to child). If no such relationship exists throw compiler error (inconvertable types)

Rule 2 (Compile time checking):
C must be either the same type or derived type (subclass) of A otherwise we will get a compile error

Rule 3 (Runtime Exception):
Runtime object of d must be same or derived type of C otherwise we will get a ClassCastException

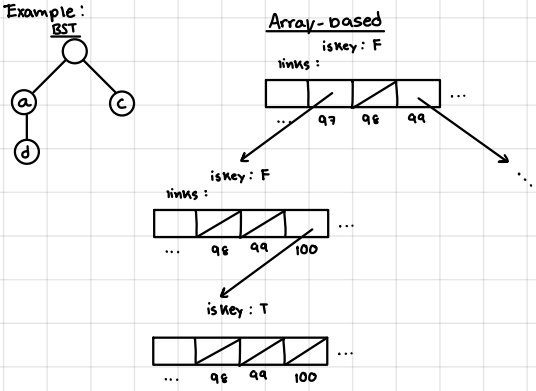
After casting the casted object will use the object that it was casted to's methods (if they are in there) however, the same does not applied to its instance variables. See dynamic method selection table above.

Tries: Every node only stores one letter and node can be shared by multiple keys

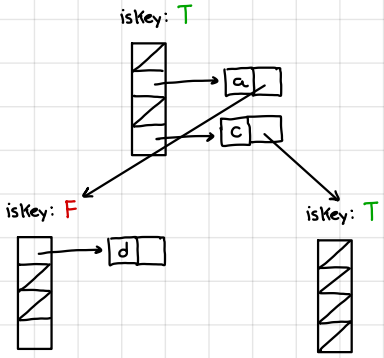
Tries can also be maps

Underlying implementation can be anything such as BST, Array, or Hashtable → The problem with this is that our arrays are very sparse utilizing alot of space

Example:

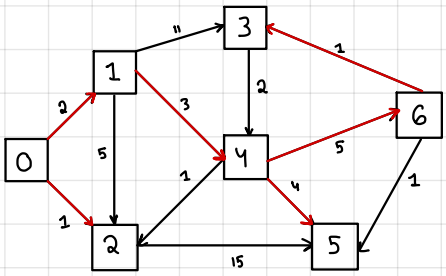


Hashtable based Trie



Inserting N strings length M → Θ(NM)
Finding all keys length L with prefix of another key → Θ(NL)
Finding the longest key that is the prefix of another key → Θ(NL)

A* algorithm and Dijkstra's Algorithm runthrough:



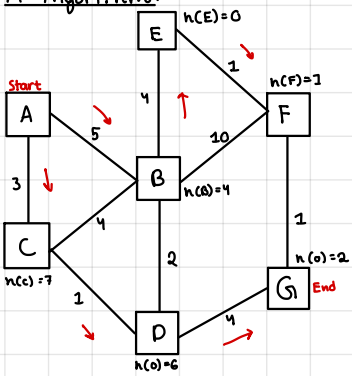
Dijkstras Algorithm:

Start from vertex 0
Set up table of all other vertices which will represent to distance from the starting vertex to the other vertex at each row, circle the vertices that you have relaxed

1	2	3	4	5	6
2	(1)	∞	∞	∞	∞
(2)	(1)	∞	∞	16	∞
(2)	(1)	13	(5)	16	∞
(2)	(1)	13	(5)	(9)	10
(2)	(1)	13	(5)	(9)	(10)
(2)	(1)	(11)	(5)	(9)	(10)

→ Dijkstras is done when all edges have been circled
The last row in the table is the distance to each node from the starting vertex

A* Algorithm:



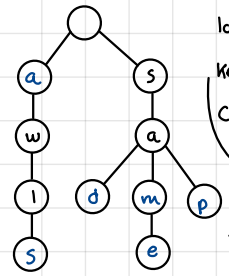
B	C	D	E	F	G
(9)	10	∞	∞	∞	∞
(9)	10	13	(9)	16	∞
(9)	(10)	13	(9)	11	∞
(9)	(10)	(10)	(9)	11	∞
(9)	(10)	(10)	(9)	11	(10)

G is selected ∴ Algorithm is stopped

Distance to a node is calculated as the actual distance to get to the node plus the hueristic of the node your are going to visit.

main appeal of tries is prefix matching

Given the following graph:



longestPrefixOf("sample") → sam

keysWithPrefix("sa") → [sad, sam, same, sap]

collect() → ["a", "awls", "sad", "same", "sap"]

→ DFS preorder adding the word built so far to a list passed in as a parameter every time a node that is a key is passed

→ The collect operation starting from the children of the last character in the prefix (when found)

Trie traversals can be simplified to a question about tree traversals and graph traversals.

When your key is a String you can use a Trie:

→ Theoritically better performance than hashtable or a search tree

→ Have to decide on mapping from letter to node
- DataIndexedCharMap
- Bushy BST
- Hashtable

→ All three choices are fine, though hash table is probably the most natural

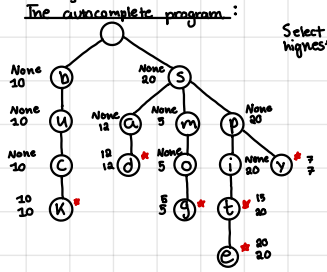
→ Supports special string operations like longestPrefixOf and keysWithPrefix.
- keysWithPrefix is the heart of important teannology like autocomplete
- Optimal implementation of Autocomplete involves use of a priority queue

```
class Trie {
    private TrieNode root = new TrieNode();
    class TrieNode {
        public HashMap<Character, TrieNode> children;
        public boolean isWord;
        public TrieNode(){
            children = new HashMap<>();
            isWord = false;
        }
    }
    public void insert(String word){
        TrieNode curr = root;
        for (char c : word) {
            if (curr.children.containsKey(c)) {
                curr.children.put(c, new TrieNode());
            }
            curr = cur.children.get(c);
        }
        curr.isWord = true;
    }
    public boolean contains(String word){
        TrieNode curr = root;
        for (char c : word){
            if(!curr.children.containsKey(c)){
                return False;
            }
            curr = curr.children.get(c);
        }
        return cur.isWord;
    }
}
```

```
public List<String> collect(){
    return collectHelper("", new ArrayList<>(), root);
}
private List<String> collectHelper(String word, List
    <Integer> wordList, TrieNode root){
    if(root.children == null){
        return wordList;
    }
    for (char c : root.children.keySet()){
        if (root.children.get(c)){
            wordList.append(word+c);
        }
        return collectHelper(word+c, wordList,
            root.children.get(c));
    }
}
```

The autocomplete program:

Select the word with the highest priority



Graph Theorems

The last edge added to the MST by Prims Algorithm is always the highest weight edge in the MST. False
The largest edge in a graph is never a part of an SPT False
On a graph of 4 or more nodes DFS and BFS never visit vertices in the same order from the same start vertex False
Dijkstra's algorithm always finds the shortest path in a directed acyclical graph even if the graph contains negative values False
In any undirected graph, the shortest paths tree always has total weight less than or equal to the weight of the MST for that graph. False
Given a graph G with unique edge weights, the shortest paths from start vertex s to all other vertices are unique False
A graph with unique edge weights will have exactly one MST True
If you take any graph G with positive edge weights an square all edges, the resulting graph will have the same MSTs True