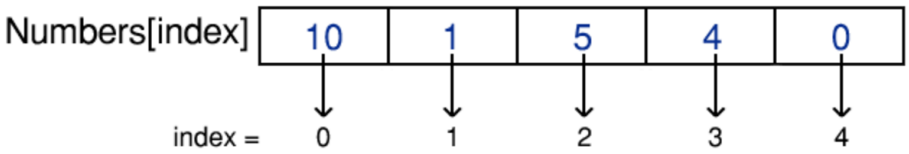
1 Algorithms I

1.1 Just a List of Numbers



1.1.1 Static Arrays

Java: `type[]`

C++: `type[] or <array>`, they should perform equally well. Array just contains a wrapper around the $< T >$ []. Array is just more easier to pass around.

1.1.2 Dynamic Arrays

Java: `java.util.ArrayList or java.util.Vector`

C++: `<vector>`

1.1.3 Bitset

An array of 0 and 1. For Java, this is dynamically sized. C++ is fixed size.

Java: `java.util.BitSet`

C++: `<bitset>`

1.2 Sliced-Bread Model

1.2.1 Queue

When you think of a queue, think of people standing in a line (queue). People enter from the back, people exit through the front. FIFO.

Java: `java.util.Queue`.
C++: `<queue>`

1.2.2 Priority Queue

Think of priority queue as a todo list, where the highest priority element always surfaces to the top. We have the same methods as a queue, but the first element is just what is greatest.

Java: `java.util.PriorityQueue`
C++: `<priority_queue>`

1.2.3 Deque

Think of a double ended queue as sliced bread. Bread is removed or added from the sides only.

Java: `java.util.Deque`
C++: `<deque>`

1.2.4 Stack

Think of a stack as a stack of pancakes. You don't reach for the bottom pancake. Instead, you only access what is on top.

Java: `java.util.Stack`
C++: `<stack>`

1.3 Bag or Box Model

1.3.1 Sets

Everything that goes in a set will be ordered and unique. Think of it as lining up to take a headcount in elementary school. We are assigned a number and we must get in that order. There will be no duplicate "Curtis-es".

They are internally implemented as a balanced BST. They are ordered via the internal `set::key_comp` (strict weak ordering) criterion in C++. They are ordered either by natural ordering or by `Comparator` provided at creation time.

Java: `java.util.SortedSet`

C++: `<set>`

1.3.2 Unordered Sets

Unordered sets are unordered and unique. Think of it as a room full of people. Note: the keys are the actual values, not the index.

Java: `java.util.Set`

C++11: `<unordered_set>`

1.3.3 Multisets

Multisets are ordered and not unique (multi-keys). Think of it as a toolbox. You can have multiple sets of the same tools, but they are organized.

C++: `<multiset>`

1.3.4 Unordered Multisets

Unordered Multisets are unordered and not unique. Think of it as a paper bag. You put anything you want in there.

C++11: `<unordered_multiset>`

1.3.5 HashSet

They are basically like normal sets except that they are ordered by their hash code, which is basically random. You can initialize with capacity (how many elements) and load factor (percent at which to copy elements in a new sized array).

Java: `java.util.HashSet`

Linked Hash Sets and Hash Sets both implement sets. Linked lists are doubly linked lists internally. Linked Hash Sets will retain an insertion order, while Hash Sets just order elements according to hashes. Linked Lists are slower and take more memory.

Java: `java.util.LinkedHashSet`

1.3.6 EnumSet

EnumSets is a set but we can only store values of an enum. For example, there can be four clubs and an enum set shows which clubs you are apart of.

Java: `java.util.EnumSet`

1.3.7 TreeSet

This is a set that is organized into a tree. It sorts ascending, but you can add your own comparator. It provides internal methods for navigation.

Java: `java.util.TreeSet`

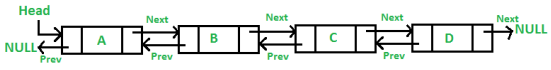
1.4 Linked Lists

1.4.1 Singly Linked List

Linked lists are dispersed throughout memory.

C++: `<forward_list>`

1.4.2 Doubly Linked List



The Java LinkedList implements the List and Deque interfaces.

Java: `java.util.LinkedList`

C++: `<list>`

1.5 Table Model (Maps)

1.5.1 Maps

Basically, this is just a json object where the keys are ordered. This is basically a dictionary with unique words only.

Java: `java.util.Map or java.util.HashMap`
C++: `<map>`

1.5.2 MultiMaps

Basically this is just a json object that allows multiple keys that are the same. This is kinda like a dictionary where a word can have multiple meanings.

C++: `<multimap>`

1.5.3 Unordered Maps

Basically this json object without ordered keys

C++: `<unordered_map>`

1.5.4 Unordered MultiMaps

Basically this a json object no unique or sorted keys.

C++: `<unordered_multimap>`

1.5.5 HashMap

Basically a Map, except the key is

Java: `java.util.HashMap`

1.5.6 LinkedHashMap

HashMap that maintains a doubly linked list internally. Hence, it maintains insertion order. It is thus slower and uses more memory.

Java: `java.util.LinkedHashMap`

1.5.7 TreeMap

A HashMap that maintains a tree datastructure. Comes with a ton of internal features

Java: `java.util.TreeMap`

1.6 Union Find

1.6.1 Quick Union

Basically, a way to create groups with an array. Note that we treat the array as if it were a key-value pair. Index and the value at the index are two useful information at our advantage. If two indexes are in the same group, their values

will be the same. So everytime you join groups, one group will have to reassign all of its values to the other group's value.

```
[0, 1, 1, 2, 2] // 0 1-2 3-4
```

1.6.2 Quick Find

We add on QuickUnion by creating trees. All nodes are -1 by default to signify a root node. If two nodes are grouped, the root for one of the node's tree becomes a branch of the other node's tree root. We can check if they are connected if they contain the same root node.

```
[-1, 3, -1, -1] // 0 1-3 2
[1, 3, -1, -1] // 0-1-3 2
```

1.6.3 Weighted Quick Union

We build off of QuickFind. Randomly grouping trees is not optimal. We want a fat tree. Weighted Quick Union stores the tree size/height via the absolute value of the root node's value. When we group nodes, we make the smaller tree a branch of the larger tree.

```
[1, 3, -1, -3] // 0-1-3 2

// NOT OPTIMAL (Basically a singly linked list)
[1, 3, -4, 2] // 0-1-2-3

// BETTER
[1, 3, 3, -4] // 0-1-2-3
```

1.7 Binary Trees

An ubiquitous used datastructure.

1.7.1 Binary Search Trees

Binary search is the most optimal search algorithm for a sorted array.

```
int binarySearch(int a[], int item, int low, int high) {
    if (high <= low)
        return (item > a[low]) ? (low + 1) : low;

    int mid = (low + high) / 2;

    if (item == a[mid])
        return mid + 1;
```

```
        if (item > a[mid])
            return binarySearch(a, item,
                                mid + 1, high);
        return binarySearch(a, item, low, mid - 1);
    }
```

1.7.2 B(ushy) Trees

Tree height. Tall trees are naturally harder to climb than shorter ones. As tree height grows, your datastructure gets closer to a linear array. A shorter tree height means less traversals, less branches, less searching.

An AVL is $O(log_2(n))$ while a super tall tree is roughly $O(n)$

1.7.3 2-3 Trees

Spindly trees are our worst enemy. We need to retain the bushiest trees. We can do this by overstuffing. A node can be overstuffed with multiple values as a list.

2-3 Trees: If a node has three values, the juicy node bursts. The middle value moves up and the node splits into two children. (This preserves the sorted balanced binary search tree).

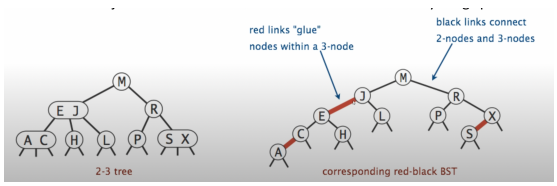
2-3-4 Trees means a L=3 before bursting and a maximum of 4 children. 2-3 Tree means L=2 before bursting and a max of 3 children. 2-55 Tree can have up to 2 elements and up to 55 children.

1.7.4 Rotation and Balanced Binary Trees

Temporarily merging and getting sent down. For `rotateLeft();` You merge with the right child and get sent down into the left. For `rotateRight();` You merge with the left child and get sent down into the right. You get sent down in a L shape. Remember that you must retain strictly two children.

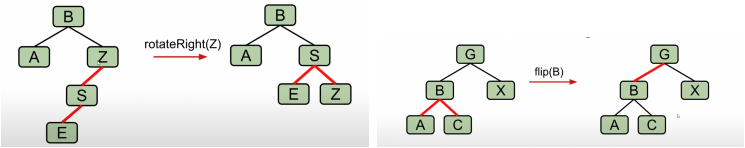
1.7.5 Red Black Trees

Left-Leaning Red Black Trees (LLRB). We force a 2-3 tree into the shape of a Balanced BST via glue links. Glue links are red and normal links are black. Glue links connect elements within the same node.



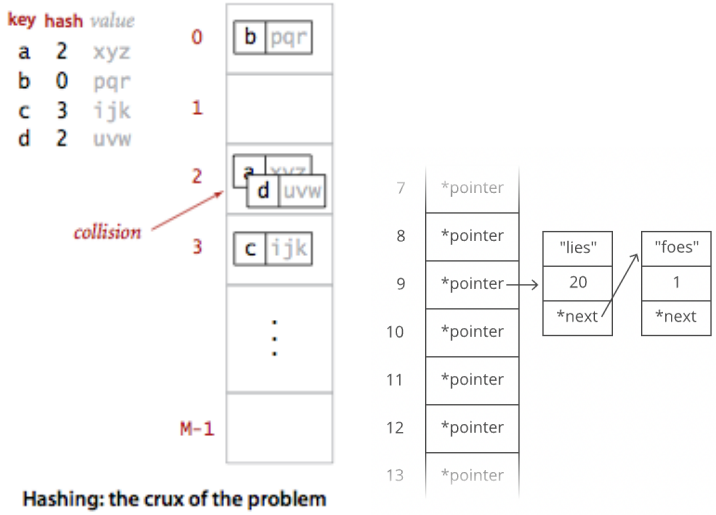
Hence, binary search applies.

How does bursting nodes get represented in LLRBs. (There is both the LLRB and 2-3 Tree visual interpretation). For creating 4-nodes, we can use rotation to temporarily get a node with two red links. This is meant to just get everything in the right place. To burst that node, we create a red link with the parent and the children become black links.



1.8 Hash Tables

In reality, this is an array where a hash function determines the index or key. Hence, they are randomized within the array.



Java: `java.util.Hashtable, java.util.HashMap`
C++: `<unordered_map>`

1.8.1 Collisions

One of the concerns with Hashtables are collisions. If you had an short array and kept inputting values, you'd inevitably find collisions.

There are many ways to handle collisions. One example is to create a linked list for indices with a collision.

Another way is to dynamically resize the array. When collisions start to occur, reassign all the hash values into a larger array in O(n) time.

1.9 Elementary Sorting

1.9.1 Selection Sort

Linearly go through the array. Swap with the next smallest value in the array.

```
 for (i = 0; i < n-1; i++) {
    min_idx = i;
    for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

    if(min_idx!=i)
        swap(&arr[min_idx], &arr[i]);
}
```

1.9.2 Insertion Sort

Basically how you'd sort a deck of cards. Assume a deck is mostly shuffled. Go through, when you see a card that is out of order, go in the reverse order to check where it would belong.

```
int i, key, j;
for (i = 1; i < n; i++) {
    key = arr[i];
    j = i - 1;

    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
}
```

1.9.3 Quick Sort

Pick a random pivot. Everything that is greater goes to the right branch of the binary tree (aka array). Everything less goes to the left branch. Repeat for each branch until you reach a leaf.

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
```

```
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}
```

1.9.4 Merge Sort

We basically disassemble the array into a binary tree and reassemble based on a comparator.

```
void sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        sort(arr, l, m);
        sort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

```
    void merge(int arr[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[] = new int[n1];
        int R[] = new int[n2];

        /*Copy data to temp arrays*/
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];

        int i = 0, j = 0;
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
```

```
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
```

1.9.5 Heap Sort

Create a maxHeap. Delete largest element and add it to the array we will return. Repeat.

```
void heapify(int arr[], int N, int i) {
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, N, largest);
    }
}
```

```java
public void sort(int arr[]) {
    int N = arr.length;

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element from heap
    for (int i = N - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

1.10 Others

1.10.1 Heap's Algorithm

```java
void heapPermutation(int a[], int size, int n) {
    if (size == 1) {
        printArr(a, n);
        return;
    }

    for (int i = 0; i < size; i++) {
        heapPermutation(a, size - 1, n);

        if (size % 2 == 1)
            swap(a[0], a[size - 1]);
        else
            swap(a[i], a[size - 1]);
    }
}
```

1.10.2 Euclidean Algorithms

```java
int gcd(int a, int b) {
    if (a == 0)
        return b;
    return gcd(b % a, a);
```

```
}
int lcd(int a, int b) {
    return (a * b) / gcd(a, b);
}
```