CS014: Introduction to Data Structures and Algorithms

Hyuntae Jung

June 5, 2020

1 Algorithmic Complexity

sometimes efficiency does matter

Different algorithms can have different runtimes, and in turn, efficiency. For example, if a given list is sorted, **binary** search can find a value much quicker than **linear** search. This introduces the idea of *runtime complexity*.

1.1 Upper and Lower Bounds

An algorithm with runtime complexity $T(N)$ has:

- **Lower bound**: A function $f(n)$ that is $\leq$ the best case $T(N)$ for all values $N \geq 1$.
- **Upper bound**: A function $f(n)$ that is $\geq$ the worst case $T(N)$ for all value $N \geq 1$.

For example, a given algorithm with best case runtime $T(N) = 7N+36$ and worst-case runtime $T(N) = 3N^2+10N+17$ has a *lower* bound $f(N) = 7N$ and *upper* bound $f(N) = 30N^2$.

1.2 Growth Rates and Asymptotic Notation

Asymptotic notation uses only functions that indicate the growth rate of a function, excluding *constants*. The three generally used in complexity analysis are:

- $O \rightarrow T(N) = O(f(N))$: Prescribes a function for an **upper bound**
- $\Omega \rightarrow T(N) = \Omega(f(N))$: Prescribes a function for a **lower bound**
- $\Theta \rightarrow T(N) = \Theta(f(N))$: Prescribes a function that's both a lower and upper bound.

For any of these to be applicable, a constant **c** must exist that makes it possible for $\mathbf{c} * f(N)$ to be either a lower or upper bound.

2 Big O Notation

don't sweat the small stuff

Given a function describing the running time of an algorithm, the Big O notation is determined using the following rules:

1. If $f(N)$ is the sum of several terms, the highest order term is kept and all others are discarded.
2. If $f(N)$ is the product of several terms, all constants are omitted.

Concept. *Rules for Big O of composite functions:*

- $c \cdot O(f(N)) = O(f(N))$
- $c + O(f(N)) = O(f(N))$
- $g(N) \cdot O(f(N)) = O(g(N) \cdot f(N)))$
- $g(N) + O(f((N)) = O(g(N) + f(N))$

2.1 Recursive Time Complexity

The runtime complexity of a recursive function has a function T on both sides of the equation. The function is called a **recurrence relation**: defined in terms of the same function operating on an "updated" input $< N$. The recurrence relation essentially dictates how far the recursion extends, which is then extended into Big O Notation.

3 ADT: Doubly Linked List

now comes in reverse

A **doubly linked list** is essentially the same as a singly linked list, but each node now has both a `next` and `previous` pointer. Linked lists are **positional** lists because they contain pointers to subsequent and/or previous elements.

3.1 Core Operations

Operation. **Traversal**: Visits every node and performs an action.

```
        curNode = head; // curNode = tail;
        while (curNode) {
            do something;
            curNode = curNode->next; // curNode = curNode->previous;
        }
```

Operation. **Search**: Returns first value in a list that matches the given key.

```
    curNode = head; // linear implementation
    while (curNode) {
        if (curNode == key) {
            return curNode;
        }
        curNode = curNode->next;
    }
    return nullptr;
```

Operation. **Append**: Inserts a node after the tail node.

- `list empty`: point both `head` and `tail` to **`new`**
- **`else`**: point `tail->next` to **`new`**, **`new`**`->previous` to `tail`, then `tail` to **`new`** .

Operation. **Prepend**: Inserts a new node before `head` and points `head` to the new node.

- `list empty`: point both `head` and `tail` to **`new`**.
- **`else`**: point `head->previous` to **`new`** and `head` to **`new`**.

Operation. **Insert**: Inserts a new node after an existing node, `*curNode`.

- `list empty`: point `head` and `tail` to **`new`**.
- `curNode == tail`: point `tail->next` to **`new`**, **`new`**`->previous` to `tail`, then `tail` to **`new`**.
- `curNode in middle of list`: point **`new`**`->next` to `sucNode`, **`new`**`->previous` to `curNode`, `curNode->next` to **`new`**, then `sucNode->previous` to **`new`**.

Operation. **Remove**: Removes a specified node, `*curNode`. Uses 4 separate checks.

- `successor exists`: point `sucNode->previous` to `prevNode`.
- `predecessor exists`: point `prevNode->next` to `sucNode`.
- `curNode == head`: point `head` to `sucNode`.
- `curNode == tail`: point `tail` to `prevNode`.

Much of the functionality above can also be implemented **recursively**. Have the function call itself with an iterated input of `cur->next` or `cur->previous`, depending on traversal direction.

3.2 Dummy Nodes

An implementation can use a **dummy node**, a node with unused data that is never removed. Dummy nodes can be set at both the `head` and `tail`. Using dummy nodes can reduce the amount of checking that needs to occur for a function, since it doesn't need to check whether `head` and `tail` "exist," so to say.

4 ADTs: Stacks, Queues, and Array-Based Lists

might as well make all your elements wait in line too

4.1 Stack

A **stack** is an ADT where items are only inserted on or removed from the *top* of the stack. It's referred to as a **last-in first-out** ADT, and can be implemented using a linked list, array, or vector.

Operation. **Push**(`stack`, `x`): inserts `x` on **top** of the stack.

Operation. **Pop**(`stack`): Returns and removes the item at the **top** of the stack.

Operation. **Peek**(`stack`): Returns the item at the top of the stack.

Operation. **IsEmpty**(`stack`): Returns **`true`** if the stack has no items.

Operation. **GetLength**(`stack`): Returns the number of items in the stack.

`Pop` and `Peek` should not be applied to an empty stack; resulting behavior may be *undefined*.

4.2 Queue

A **queue** is an ADT wherein items are inserted at the *end* and removed from the *front*. This makes a queue a **first-in first-out** ADT. Queues can be implemented with linked lists, arrays, or vectors.

Operation. **Push**(`queue, x`): Inserts `x` at the **end**.

Operation. **Pop**(`queue`): Returns and removes item at the **front**.

Operation. **Peek**(`queue`): Returns the item at the front.

Operation. **IsEmpty**(`queue`): Returns **`true`** if `queue` has no items.

Operation. **GetLength**(`queue`): Returns the number of items in the `queue`.

`Pop` and `Peek` should not be applied to an empty queue; resulting behavior may be undefined.

A **deque** (short for *double-ended* queue) is a special implementation of the queue wherein items can be inserted and removed at both the front and the back. It has all the same fundamental operations of the queue, but has both **front** and **back** variations of `push`, `pop`, and `peep`.

4.3 Array Based Lists

An **array-based list** is a list ADT implemented using an array. It's essentially a *dynamically-allocated* array, which means it needs to be reallocated each time the capacity requirement is exceeded.

Operation. **Resize**(`list, newSize`): Transfer all elements to a newly allocated array of size `newSize`.

Operation. **Append**(`list, x`): Inserts `x` at the end.

Operation. **Prepend**(`list, x`): Inserts `x` at the start.

Operation. **InsertAfter**(`list, index, x`): Inserts item `x` after specified position `index`.

Operation. **Search**(`list`, key): Returns the index of the first element with data matching `key` or returns `-1` if no element is found.

Operation. **RemoveAt**(`list, index`): Removes the item at position `index` and moves all elements after `index` down 1 position afterwards.

With the exception of `Resize` and `Prepend`, all these operations have a best case runtime of $\Omega(1)$ and worst case runtime of $O(N)$ based on where they're called and subsequently how much of the list they have to sift through.

5 Exceptions
not quite baseball

Exceptions are circumstances a program wasn't designed to handle, such as negative height. **Exception-handling constructs** are used to try, throw, and catch exceptions in a specific sequence to keep error-checking code separate and reduce redundant checks.

1. A **try** block surrounds normal code, which aborts immediately if a **throw** executes.
2. A **throw** statement resides in a **try** block and skips to the end if tripped. It then provides an *object* of some type, for example: type `runtime error` (read: "throw an exception of the particular type") to *catch*.
3. The **catch** clause must *immediately* follow a **try** block, and *handles* a specific type of error. Also called a **handler** because it handles the exception.

The `stdexcept` library is commonly used to provide exception types.

5.1 Within Functions

As long as there is a *try-catch* sequence for a particular section, `throw` calls can be embedded into functions that can later be caught by handler outside of its definition. If *no* handler is found, then `terminate()` is called, which typically aborts the program.

```
try {
    ...
    if (exception condition) {
        throw exception_type("specific error message");
    }
}
catch (specific exception type &excpt) {
    stream << excpt.what() << endl; // prints "specific error message"
}
```

A `throw` in a `try` block causes an immediate jump to the corresponding `catch` block. If a `throw` is not in a `try` block, it causes an *immediate* exit without any `return` statement.

5.2 Using Multiple Handlers

Different `throws` in a `try` block may require varying exception types, which may then necessitate the use of **multiple** handlers. In such a situation, the first matching handler executes and the remaining handlers are skipped.

```
try {
    ...
    if (exception condition 1) {
        throw excptType1("optional error message");
    }
    ...
    if (exception condition 2) {
        throw excptType2("optional error message");
    }
}
catch(excptType1 &excpt) {
    some exception indication;
}
catch(excptType2 &excpt) {
    some exception indication;
}
catch(...){
    general exception indication;
}
```

`catch(...)` is a catch-all handler that catches *any* type, which makes it useful listed as the last handler.

A thrown exception can sometimes be caught by a handler meant only to handle exceptions of a *base* class. To avoid this, it's best to place handlers of *derived* classes before the handler of their *base* class.

6 Templates

sometimes you just can't decide

Sometimes multiple functions may nearly be identical, differing only in data type. A **function template** is a function definition that replaces different types with a special type parameter that can then reduce redundancy.

6.1 Function Templates

When using templates, the function return type is preceded by `template<typename newType>` as such:

```
template<typename newType>
newType funcName (newType parameters) {
    newType someVar;
    do something;
    return someVar;
}
```

The new type placeholder is known as the **type parameter** and can be used throughout the function for any parameter, local variable, or return type. The identifier is known as a **template parameter** and can be various items, such as a data type, a pointer or reference, or even another template parameter.

The compiler automatically generates a unique function definition for each type that appears in function calls that is never seen.

A type can also be explicitly specified as a special argument, as in `TripleMin<int>(num1, num2, num3);` and a function template can also have multiple parameters:

```
template<typename T1, typename T2>
returnType FunctionName(parameters) {
    do something;
}
```

6.2 Class Templates

A **class template** is a class definition with a special type parameter that can be used in place of types in the class. The type is then specified when declaring a variable. Most of the principles of function templates carry over, such as the use of multiple parameters.

```
template <typename newType>
class item {
public:
    item(newType val1 = 0, newType val2 = 0);
    newType SomeFunction();
private:
    newType val1;
    newType val2;
}

int main() {
    item<int> someVal(1, 2);
    item<string> someName("eric", "andre");
    ...
}
```

Any function definitions outside of the class declaration must also be preceded by the template declaration, such as with:

```
template<typename newType>
newType item<newType>::SomeFunction() {
    definition;
}
```

7 Binary Trees
because who doesn't love trees?

7.1 Basic Organization

The goal of a **binary tree** is to create a defined hierarchy for data. As opposed to a list where each node has up to one successor, a binary tree's node can have up to 2 children, known as *left child* and *right child*.

Everything starts at the top **root** node. The root then splits into two children, which can either become **internal** nodes with children or **leaf** nodes with no children. An internal node is said to be its child node's **parent** node. Every node that has children is considered internal, *even* the root node.

The link from a node to child is called an **edge**, and its **depth** is the number of edges on the path from the root (depth 0) to the node. All nodes with the same depth form a tree **level**, and a tree's **height** is the largest depth of a node in the tree.

7.2 Special Types

There are 3 different types of special binary trees:

1. **Full**: If every node has no children or 2 children.
2. **Complete**: If all levels, but not necessarily the last, contain all possible nodes and all nodes in the last level are as far left as possible.
3. **Perfect**: If all internal nodes have 2 children and all leaves are at the same level.

A tree *cannot* be perfect if it is not full or complete.

7.3 Basic Applications

File systems are commonly implemented as trees, since directories and files are intuitively analogously to trees and their children, although in a file system the divisions aren't necessarily binary since they can be more than 2 files in a directory.

Binary space partitioning (BSP) is repeatedly separating a region of space into 2 parts and cataloging the objects contained within and is commonly used when rendering computer graphics.

8 Binary Search Trees

simplifying parenting

A **binary search tree** (BST) is a tree with an ordering property such that **left** children are always *smaller* than the parent and **right** children are always larger. This is then extended across the entire tree.

A node's **predecessor** and **successor** are the nodes smaller and larger than that node. If a node has a right subtree, its *successor* is that right subtree's leftmost child, which is reached traversing left children until reaching a node with no left children. If a node doesn't have a right subtree, its successor is the first ancestor with the node in a left subtree.

8.1 Essential Algorithms

The fundamental operation of the BST is **searching**, where it's much more efficient than a general linear search. Given a tree with with height H and N nodes, its worst case runtime is O(H) but can be minimized with careful organization. By keeping all except possibly the last level full, the heigh can be reduced to $H = \lfloor \log_2 N \rfloor$.

Operation. **Search**(`tree`, `key`): returns the first node that matches `key`, returning `null` otherwise.

```
cur = tree->root; // start from the root
while (cur) {
    if (key == *cur) {
        return cur;
    if (key < *cur) { // traverses down left or right path depending on the value
        cur = cur->left;
    }
    else {
        cur = cur->right;
    }
}
return null;
```

Insertion has to adhere to the ordering property of the BST and so requires some comparisons.

Operation. **Insert**(`node`): inserts `node` using the following logic:

```
if (!root) { // empty tree special case
    root = node;
    set left and right = null
    return;
}
cur = root;
while (cur) {
    if (*node < *cur) {
        if (!cur->left) { // only inserts if there's an open spot
            cur->left = node;
        }
        else {
            cur = cur->left; // if none, then go to next node
        }
    }
    else {
        if (!cur->right) {
            cur->right = node;
        }
        else {
            cur = cur->right;
        }
    }
}
set left and right = null
```

Insertion has a worst-case runtime complexity of O(N) and a best-case of $\Omega(\log N)$. Space complexity will always be O(1) because only a single pointer is used for traversal.

Remove removes the first node matching a given key and has to restructure the tree to preserve the ordering

Operation. **Remove**(`key`): removes the node matching `key`; there are 3 situations:

```
par = null;
cur = root;
while (cur) {
    if (key != *cur) { // traverse until a match is found
        par = cur;
        if (key < *cur) {
            cur = cur->left;
        }
        else {
            cur = cur->right;
        }
    }
    else {
        if (!cur->left && !cur->right) { // case 1: node has no children
            if (!par) { // check if root
                root = null;
            }
            else if (par->left == cur) { // set the parent's left/right to null
                par->left = null;
            }
            else {
                par->right = null;
            }
        }
        else if (cur->left && !cur->right) { // case 2a: node only has left child
            if (!par) { // check if root
                root = cur->left;
            }
            else if (cur == par->left) { // set the parent's left/right to the child
                par->left = cur->left;
            }
            else {
                par->right = cur->left;
            }
        }
        else if (cur->right && !cur->left) { // case 2b: node only has right child
            same operations as above, with cur->right in place of cur->left
        }
        else { // case 3: node with 2 children
            suc = cur->right;
            while (suc) { // find successor
                suc = suc->left;
            }
            sucData = *suc;
            Remove(*suc); // remove successor and replace cur with successor data
            cur = sucData;
        }
    }
}
```

If the node being removed is full, then the successor has to be found in a recursive call. This means that the tree would would to potentially be traversed twice. This means means a runtime complexity worst-case O(N) and best-case $\Omega(\log N)$. Since 2 pointers, 3 if removing a full node, are used, the space complexity is always O(1).

Traversal in order starts from the root and recursively visits the nodes in sorted order.

Operation. **Traverse**(`node`):

```
if (!node) {
    return;
}
Traverse(node->left);
do something
Traverse(node->right);
```

The order of the `Traverse` calls can be switched to traverse the list in reverse order.

8.2 Insertion Order and Height

As seen above, a large factor in determining runtime complexity of operations for a given tree depends on how it's ordered, which determines its height. *Minimum* height is achieved when all levels are practically full and *maximum* height when each node only has one child. This is all dependent on the **order** in which elements are *inserted*. Inserting in a **sorted** order is the best way to ensure **maximum** height since it just becomes a chain of right children.

Height can be computed by recursively hunting down the deepest path.

Operation. **GetHeight**(node):

```
if (!node) {
    return -1;
}
leftHeight = GetHeight(node->left);
rightHeight = GetHeight(node->right);
return (1 + max(leftHeight, rightHeight);
```

The worst-case time complexity is O(N), where N is the number of nodes. The function would also work with the left and right calls switched, since it's just a matter of comparison.

8.3 Parent Nodes

A BST implementation can include a **parent** pointer inside each node, which comes in useful when use with balanced BSTs. They also allow for some added functionality, shown below.

Parent pointers can be used to implement a **Replacement** function:

Operation. **ReplaceChild**(parent, cur, new):

```
if (parent->left == cur) {
    parent->left = new;
}
else (parent->right == cur) {
    parent->right = new;
}
```

Which can be used to simplify Remove:

```
if (!node) { // empty  special case
    return;
}
if (node == root) { // root case
    if (node->left) {
        root = node->left;
    }
    else {
        root = node->right;
    }
    root->parent = null; // ensures the parent of root is always null
}
else if (node->left) { // internal left child only
    ReplaceChild(node->parent, node, node->left);
}
else if (node->right) { // internal right child only
    ReplaceChild(node-parent, node, node->right);
}
else { // full node
    suc = node->right;
    while (suc) {
        suc = suc->left;
    }
    node = *suc;
    Remove(suc);
}
```

9 Heaps

staying rooted

9.1 Core Ideas

For fast access and replacement of a maximum item in a changing set of items, a **max-heap**, is a tree that maintains an ordering property that a node's key is always greater or equal to the node's children s key. No relation between children are required. This means the root node will always be the maximum. It's the reverse idea for a **min-heap**. Both are commonly implemented as a binary tree.

The height of a max or min heap will always be filled left-to-right, resulting in a height for N nodes of $\lfloor \log N \rfloor$

9.2 Implementation

Heaps are typically stored using arrays. The formulas for parents and child indices for a node with index *i*:

$$\text{parent: } (i-1)/2$$

$$\text{children: } 2i+1,\ 2i+2$$

The parent formula cannot be used on the root node since the index is 0.

9.3 Essential Algorithms

Operation. **Insert**(`heap, node`): inserts a node in the last level and swaps the node with the parent until no ordering violation occurs. Also called *percolating*.

Operation. **Remove**(`heap`): Removes the root, replaces the root with the bottom node, and switches that node with is greatest child until ordering is restored.

The worst-case complexity of inserts and removals is O($\log N$)

An array based implementation of percolating up in a max heap is as follows:

```
while (node > 0) {
    parent = (node - 1) / 2;
    if (heap[node] <= heap[parent]) {
        return;
    }
    swap(heap[node], heap[parent]);
    node = parent;
}
```

And for percolating down:

```
child = 2 * node + 1;
val = heap[node];
while (child < arrSize) {
    maxVal = val;
    maxIdx = -1;
    for (i = 0; i < 2 && child < arrSize; i++) {
        if (heap[i + child] > maxVal) {
            maxVal = heap[i + child];
            maxIdx = i + child;
        }
    }
    if (maxVal == val) {
        return;
    }
    swap(heap[node], heap[maxIdx]);
    node = maxIdx;
    child = 2 * node + 1;
}
```

10 Heap Sort

heaps of fun

10.1 Heapify

Heap sort takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order. The **heapify** operation is first used to turn an unsorted array into a heap by percolating down every non-leaf node from the bottom-up.

Heapify in an array starts on the internal node with the largest index and continues backwards to the root node at index 0. The index of the largest internal node in a binary tree with N nodes is $(N/2) - 1$.

An array sorted in ascending order is not a valid max-heap.

10.2 Implementation

Heapsort begins by heapifying the array into a max-heap and initializing an end index value to the size of the array minus 1. Heapsort repeatedly swaps the maximum value with the lowest rightmost node, and decrements the end index; ending when end index = 0.

Operation. **Heapsort**(`arr, arrSize`):

```
for (i = arrSize / 2 - 1; i >= 0; i--) { // heapifies the array first
    PercolateDown(i, arr, arrSize);
}
for (i = arrSize - 1; i > 0; i--) { // executes actual sorting
    swap(arr[0], arr[1]);
    PercolateDown(0, numbers, i);
}
```

Heapsort's worst-case runtime is O$N \log N$

10.3 ADT: Priority Queue

A **priority queue** is a queue where each item has a priority and items with higher priority are close to the front. The **push** operation inserts an item such that the item is closer to the front than all items of lower priority and closer to the end than all items of equal or higher priority. **Pop** removes and returns the item at the front, which has the highest priority.

Operation. **Push**(`queue, x`): Inserts `x` after all equal or higher priority items.

Operation. **Pop**(`queue`): Returns and removes the item at the front.

Operation. **Peek**(`queue`): Returns but doesn't remove the item at the front.

Operation. **IsEmpty**(`queue`): Returns whether the queue has no items.

Operation. **GetLength**(`queue`): Returns the number of items in the queue.

Items can either be pushed with an additional parameter specifying priority or have priority data residing in the object itself.

Priority queues are commonly implemented using heaps, keeping the highest priority item in the root node. Adding and removing items operates with worst-case runtime O$\log N$.

11 Sorting Overview

not your regular bucket

11.1 Selection Sort

An array is partitioned into sorted and unsorted partitions, All elements in the unsorted section are searched to find the smallest element, which is then swapped with the element at the beginning of the partition.

```
for (i : arr) {
    minIdx = i; // set minimum index
    for (j = i + 1 : arr) {
        if (arr[j] < arr[minIdx]) { // find the minimum in unsorted section
            minIdx = j;
        }
    }
    swap(arr[i], arr[minIdx]); // swap minimum with initial minimum
}
```

Selection sort is very inefficient, with a runtime of O(N^2).

11.2 Insertion Sort

Insertion sort treats the input as two sections, one sorted and the other unsorted, repeatedly inserting the next value from the unsorted part into the correct location in the sorted part.

```
i, j = 0;
for (i : arr) {
    j = i;
    while (j > 0 && arr[j] < arr[j - 1]) {
        swap(arr[j], arr[j - 1]); // keeps swapping adjacent values until ordering holds
        j--;
    }
}
```

Insertion sort has a worst-case runtime of O(N^2).

If there are a relatively small amount C (less than half) of values unsorted, the runtime for a *nearly* sorted list is O(N).

11.3 Shell Sort

Shell sort treats the input as a collection of interleaved lists, sorting each list individually with a variant of insertion sort. It uses a **gap value**, representing the number distance between elements to determine the number of interleaved lists.

Shell sort starts by choosing a gap value K and sorting K interleaved lists in place. It finishes by performing an insertion sort on the entire array. The insertion sort within the interleaved lists *redefines* next and previous items as $x \pm K$ instead of $x \pm 1$.

```
i, j = 0;
for (i = idx + gap; i < size; i += gap) {
    j = i;
    while (j - gap >= idx && arr[j] < arr[j - gap]) { // insertion sorts jumping by gap
        swap(arr[j], arr[j - gap]);
        j -= gap;
    }
}
```

Shell sorts perform well with a descending sequence of gap values, ending at 1.

```
for (gap : gapList) {
    for (i : gap) { // sorts array with each gap in a list of gaps
        InterleavedInsertionSort(arr, arrSize, i, gap)
    }
}
```

Interleaved insertion is called the sum of all the gap values times during a shell sort.

11.4 Quick Sort

Quicksort repeatedly partitions the input into low and high parts and then recursively sorts each part. Quicksort uses a **pivot** value to divide the data, and can be any value within the array being sorted, commonly the value of the middle array element.

To partition the input, the array is divided into two parts where all values ≤ to the pivot are in the lower partition and ≥ in the higher partition.

Operation. **partition**(`arr, low, high`)

```
mid = low + (high - 1) / 2; // set midpoint
pivot = arr[mid]; // set pivot value from midpoint
while (low < high) { // iterate until fully partitioned
    while (arr[low] < pivot) { // iterate until order is wrong
        low++;
    }
    while (arr[high] > pivot) {
        high--;
    }
    if (low < high) { // switches items
        swap(arr[low], arr[high]);
        i++, h--; // continues traversing from next position
    }
}
return high;
```

Quicksort is then implemented recursively, using calls to quicksort to sort the low and high partitions. The process continues until a partition has one or zero elements.

Operation. **Quicksort**(`arr, low, high`):

```
if (low >= high) {
    return;
}
i = Partition(arr, low, high);
Quicksort(arr, low, i);
Quicksort(arr, i + 1, high);
```

Quicksort partitions an array into $\log N$ levels and does at most N comparisons moving the `low` and `high` indices. This results in an average runtime of $\Theta(N \log N)$. If partitioning yields unequally sized parts, the worst case runtime will be $O(N^2)$, but this rarely happens.

11.5 Merge Sort

Merge sort divides a list into two halves, recursively each half, then merges the sorted halves to produce a sorted list. The recursive partitioning continues until a list of 1 element is reached.

Merge sort uses three indices to keep track of the elements to sort for each recursive function call.

Operation. **Merge**(`arr, low, mid, high`):

```
while (i <= mid && j <= high) { // starting with i = low and j = mid
    if (arr[i] <= arr[j]) { // increases the left or right position accordingly
        mergeArr[idx] = arr[i];
        i++;
    }
    else {
        mergeArr[idx] = arr[j];
        j++;
    }
    idx++;
}
while (i <= high) { // adds the rest of whatever partition is left
    mergeArr[idx] = arr[i];
    i++, idx++;
}
while (j <= high) {
    mergeArr[idx] = arr[j];
    j++, idx++;
}
for (idx : mergeArr) {
    arr[low + idx] = mergeArr[idx];
}
```

Operation. **MergeSort**(`arr, low, high`):

```
if (i >= k) {
    return;
}
mid = (low + high) / 2;
MergeSort(arr, low, mid);
MergeSort(arr, mid + 1, high);
Merge(arr, low, mid, high);
```

Merge sort partitions input into $\log N$ levels and performs about N comparisons selecting and copying elements from left and right partitions, yielding a runtime of $O(N \log N)$. Since it uses a temporary dynamically allocated array, it requires an additional $O(N)$ memory components. A temporary array with the same size as the working array can also be passed as an argument.

11.6 Radix Sort

Radix sort is a type of bucket sort designed specifically for integers. The array is subdivided into **buckets** that all share a particular digit. It processes one digit at a time starting with the least significant digit and ending with the most. All array elements are place into buckets based on the current digit's value, then the array is rebuilt by removing all elements from buckets in order from lowest bucket to highest.

If an array contains *negative* integers, the algorithm performs two separate buckets for the negative and positive integers, then reverses the order of the negative bucket after sorting to yield a sorted array.

Operation. **RadixSort**(`arr, arrSize`):

```
create array of 10 buckets
maxDigits = RadixGetMaxLength(arr, arrSize); // determine max length of a digit
pow10 = 1; // start with least significant digit, 1
for (digitIdx : maxDigits) { // iterates through each level of significance
    for (i : arrSize) { // sorts each value into the appropriate bucket
        bucketIdx = GetLowestDigit(arr[i] / pow10);
        buckets[bucketIdx] = arr[i];
    }
    arrIdx = 0;
    for (i : 10) { // rebuilds array with sorted significant digit
        for (j : buckets[i]) {
            arr[arrIdx++] = buckets[i][j];
        }
    }
    pow10 *= 10; // increase to next significant digit
    clear buckets
}
negatives[] = all negative values
positives[] == all positive values
reverse order of negatives
concatenate negative and positives into array
```

Operation. **RadixGetMaxLength**(`arr, arrSize`):

```
max = 0;
for (i : arrSize) {
    curDigits = RadixGetLength(arr[i]);
    if (curDigits > max) {
        max = curDigits;
    }
}
return max;
```

Operation. **RadixGetLength**(`val`):

```
if (val == 0) {
    return 1;
}
digits = 0;
while (val != 0) {
    digits++;
    val =/ 10;
}
return digits;
```

Radix sort has a constant number of iterations in the outer loop and *N* in the inner, making its runtime O(N) and space complexity O(N).

12 Balanced Trees
complicating parenting

12.1 Core Ideas

A **B-tree** with order K is a tree where nodes can have up to $K - 1$ keys and up to K children. The **order** is the maximum number of children a node can have. They maintain the follow properties:

1. All keys must be distinct
2. All leaves must be at the same level
3. All internal nodes with N keys must have $N + 1$ children
4. Keys in a node are stored in ascending order
5. Each key in an internal node has one left subtree and one right subtree

A 2-3-4 tree is an order 4 tree with specific organizational guidelines, where the number of children mandated is always related to the number of keys in the node. Keys in a 2-3-4 tree node are labeled A, B, and C. The children are labeled left, middle1, middle2, and right. A 2-3-4 tree node containing 3 keys is **full**, and as such uses all keys and children. Nodes are labeled according to how many children they have, so a node with 1 key is a **2-node**, and so forth.

12.2 Basic Algorithms

Search is performed recursively starting from the root node with the following logic:

Operation. **Search**(`node, key`):

```
if (node) {
    if (key == node->A or node->B or node->C) { // checks if key is in current node
        return node;
    }
    if (key < node->A) { // checks if smaller than leftmost key
        return search(node->left, key);
    }
    if (key < node->B or !node->B) { // checks if between A and B
        return search(node->middle1, key);
    }
    if (key < node->C or !node->C) { // checks if between B and C
        return search(node->middle2, key);
    }
    return search(node->right, key); // last case where key is larger than C
}
return null
```

Given a new key, an **insert** operation inserts the key in a new leaf whilst maintaining ordering properties. An important part of insertion is the **split** operation, which is performed on every full node encountered during insertion. Split moves the middle key from a child node into the child's parent node and the first and keys into two separate nodes, returning the parent node that received the middle key. This results in 2 new nodes, each with a single key.

Operation. **Split**(`node`):

```
if (node is not full) { // only executes for full nodes
    return null;
}
newLeft = new bnode(node->A, node->left, node->middle1); // allocates new left/right subtrees
newRight = new bnode(node->C, node->middle2, node->right);
if (node->parent) { // inserts middle node B now with new children A and C
    InsertIntoParent(node->parent, node->B, newLeft, newRight);
}
else { // sets new root if new top node will be the root
    node->parent = new bnode(node->B, newLeft, newRight);
    tree>root = node->parent;
}
```

Operation. **InsertIntoParent**(`parent, key, leftChild, rightChild`):

```
if (key < parent->A) { // inserts at leftmost position if less than A
    parent->C = parent->B; // shift all data to accommodate new key
    parent->B = parent->A;
    parent->A = key;
    parent-right = parent->middle2;
    parent->middle2 = parent->middle1;
    parent->middle1 = rightChild;
    parent->left = leftChild;
}
else if (key < parent->B  or  !parent->B) {
    parent->C = parent->B;
    parent->B = key;
    parent-right = parent->middle2;
    parent->middle2 = rightChild;
    parent->middle1 = leftchild;
}
else {
    parent->C = key;
    parent->right = rightchild;
    parent->left = leftChild;
}
```

Full insertion can be done with a **preemptive split** scheme that splits any full node encountered during insertion traversal, ensuring that any time a full node is split, the parent node has room to accommodate the child's middle value.

Operation. **Insert**(`node, key`):

```
if (key is in node) {
    return null
}
if (node is full) {
    node = split(node);
}
if (node != leaf) { // checks where to insert the new node
    if (key < node->A) {
        return Insert(node->left, key);
    }
    if (key < node->B or !node->B) {
        return Insert(node->middle1, key);
    }
    if (key < node->C or !node->C) {
        return Insert(node->middle2, key);
    }
    else {
        return Insert(node->right, key);
    }
}
else {
    InsertIntoLeaf(node, key); // if none of the above, insert into a newly split node
    return node;
}
```

12.3 Rotation

Removing an item from a 2-3-4 tree may require rearranging keys to maintain tree properties. A **rotation** transfers keys between sibling nodes to maintain ordering. A node donates its key to the root and the root transfers its corresponding node to its right sibling in a **right rotation** and to its left sibling in a **left rotation**.

Rotations can utilize several utility functions to facilitate operation:

Operation. **GetLeftSibling**(): returns a pointer to the left sibling of a node or null if there's no left sibling

Operation. **GetRightSibling**(): returns a pointer to the right sibling of a node or null if there's not left sibling

Operation. **GetParentKeyLeftOfChild**(`parent, child`): returns the key in the parent immediately left of the child

Operation. **SetLeftParentKeyLeftOfChild**(`parent, child`): sets the key in parent immediately left of the child

Operation. **AddKeyAndChild**(`node`): adds one new key and one new child to the node where the new key must be greater than all keys in the node and all keys in the new child subtree must be greater than the new key.

Operation. **RemoveKey**(`node, key`): removes a key from a node using a key index in the range [0, 2], which may require moving keys and children to fill the location left by removing the key:

```
if (key == 0) { // displaces A by shifting everything to the left
    node->A = node->B;
    node->B = node->C;
    node->C = null;
    node->left = node->middle1;
    node->middel1 = node->middle2;
    node->iddle2 = node->right;
    node->right = null;
}
else if (key == 1) { // displaces B
    node->B = node->C;
    node->C = null;
    node->middle2 = node-right;
    node->right = null;
}
else if (key == 2) { // remove C
    node->C = null;
    node-right = null;
}
```

Operation. **RotateLeft**(`node`): causes a net decrease of 1 key in a node by removing a key, moving it into the parent, and moving the displaced key into a sibling.

```
leftSib = GetLeftSibling(node);
newLeftSibKey = GetParentKeyLeftOfChild(node->parent node); // finds new key
AddKeyAndChild(leftSib, newLeftSibKey, node->left); // inserts parent-side key into left sib
SetParentKeyLeftOfChild(node->parent, node, node->A); // inserts old left key into parent
RemoveKey(node, 0);
```

A node being rotated must have at least 2 keys and its adjacent sibling must have no more than 2 keys.

12.4 Fusion

When rearranging values during deletion, rotations are not an option for nodes that don't have a sibling with 2 or more keys. A **fusion** combines 2 keys from adjacent siblings nodes each with 1 key and a third key between the two adjacent siblings from the siblings' parent to form a new node.

Fusion of the root is a special case that happens only when the root and the root's 2 children all have only 1 key. In this case, the 3 keys are combined into a single new root.

Operation. **FuseRoot**(`root`):

```
oldLeft = root->left;
oldMiddle1 = root->middle1;
root->B = root->A; // shift old A into middle position
root->A = oldLeft->A; // fill in A and C with subtree values
root->C = oldMiddle1->A;
root->left = oldLeft->left; // reassign child paths
root->middle1 = oldLeft->middle1;
root->middle2 = oldMiddle1->left;
root->right = oldMiddle1->left;
root->right = oldMiddle1->middle1;
return root // return the newly fused root
```

In the *non-root* case, fusion operates on 2 adjacent siblings that each have 1 key. The key in the parent node that is between the 2 adjacent siblings is combined with the 2 sibling keys to make a single fused node.

Operation. **GetKeyIndex**(`node, key`): returns an integer in the range [0, 2] that indicates the index of the key within the node

Operation. **SetChild**(`parent, newChild, keyIdx`): sets the child pointer based on an index value within [0, 3].

Operation. **Fuse**(`leftNode, rightNode`): performs fusion using the utility functions above

```
parent = leftNode->parent;
if (parent == root and has 1 key) { // performs root fuse if conditions met
    return FuseRoot(parent);
}
middleKey = GetParentKeyLeftOfChild(parent, rightNode); // finds the middle reference
fusedNode = new node(leftNode->A, middleKey, rightNode->A); // allocate new fused node
fusedNode->left = leftNode->left; // reassign child paths
fusedNode->middle1 = leftNode->middle1;
fusedNode->middle2 = rightNode->left;
fusedNode->right = right->Node->middle1;
key = GetKeyIdx(parent, middleKey);
RemoveKey(parent, key); // remove leftmost (old) child
SetChild(parent, fusedNode, key); // set new fused node as leftmost child, displacing old right
return fusedNode;
```

12.5 Merge

Operation. **Merge**(`node`): Increases the number of keys of a node with 1 key using rotation or fusion.

```
leftSib = GetLeftSibling(node);
rightSib = GetRightSibling(node);
if (!leftSib and leftSib->numKeys >= 2) { // rotate if adjacent siblings have 2 or more keys
    RotateRight(leftSib);
}
else if (rightSib and rightSib->numKeys >= 2) {
    RotateLeft(rightSib);
}
else { // if inadequate number of keys, fusion is used
    if (!leftSib) {
        node = fuse(node, rightSib);
    }
    else {
        node = fuse(leftSib, node);
    }
}
return node;
```

12.6 Removal

Several utility functions are used for removal:

Operation. **GetMinKey**(`node`): returns the minimum key in a subtree

Operation. **GetChild**(`node, idx`): returns a pointer to a child based on the index passed in

Operation. **NextNode**(`node, key`): returns the child of a node that would be traversed next to search for a key

Operation. **KeySwap**(`node, cur, new`): swaps one key with another in a subtree, provided the new key doesn't violate ordering, returning a bool value depending on if the switch was applicable

```
if (!node) {
    return false;
}
key = GetKeyIndex(node, cur);
if (key == -1) { // iterate through the tree until the key is found (or not)
    next = NextNode(node, cur);
    return KeySwap(next, cur, new);
}
if (key == 0) {
    node->A = new;
}
if (key == 1) {
    node->B = new;
}
else {
    node->C = new;
}
return true;
```

There are two possible cases when removing a key, depending on if the key resides in a leaf or internal node. A key can only be removed from a leaf node with 2+ keys. The **preemptive merge** removal scheme in turns increases the number of keys in all single-key non-root nodes encountered during traversal, always occurring before any key removal is attempted. The key is replaced with the minimum key in the right child subtree or the maximum key in the leftmost child subtree.

Operation. **Remove**(`key`):

```
if (root is leaf with 1 key and root->A == key) { // special root removal case
    root = null;
    return true
}
cur = root;
while (cur) {
    if (cur has 1 key and cur != root) { // merge all single key nodes
        cur = Merge(cur);
    }
    keyIdx = GetKeyIndex(cur , key);
    if (keyIdx != -1) { check for key
        if (cur is leaf) { // easy removal if key in a leaf
            RemoveKey(cur, keyIdx);
            return true;
        }
        oldChild = GetChild(cur, keyIdx + 1);
        oldKey = GetMinKey(oldChild);
        Remove(tmpKey);
        KeySwap(root, key, oldKey);
        return true;
    }
    cur = NextNode(cur, key);
}
return false;
```

13 The AVL Tree

balancing parenting

13.1 Core Ideas

An **AVL tree** is a BST with a height balance property and specific operations to rebalance the true upon insertion or removal of a node. A BST is **height balanced** if for any node, the heights of the left and right subtrees differ by 0 or 1. The **balance factor** is then the height of the left subtree minus the height of the right subtree, which in an AVL Tree can be -1, 0, or 1. A non-existent subtree has a height of -1. An AVL tree can provides heights no worse than 1.5x that of the minimum height as established by the perfect form of that particular BST. This means that the height still stays in the O(log N) range.

An AVL tree implementation can store the subtree height as a member of each node, which may be recomputed in subsequent operations. Since balance factor is exclusively determined by the height of subtrees, any insertion and subsequent update of balance factors throughout the tree *must* be updated starting from the inserted node's parent and going upwards.

Several utility functions can be defined to maintain these core properties and simplify insertion and removal:

Operation. **UpdateHeight**(`node`): updates a node's height by taking the maximum subtree height and adding 1

```
leftHeight, rightHeight = -1;
if (node->left) {
    leftHeight = node->left->height;
}
if (node->right) {
    rightHeight = node->right->height;
}
node->height = max(leftHeight, rightHeight) + 1;
```

Operation. **GetBalance**(`node`): computes a node's balance factor by subtracting the right subtree eight from the left

```
leftHeight, rightHeight = -1;
if (node->left) {
    leftHeight = node->left->height;
}
if (node->right) {
    rightHeight = node->right->height;
}
return leftHeight - rightHeight;
```

Operation. **SetChild**(`parent, whichChild, child`): sets new left/right child and updates the parent's height

```
check inputs, return false if invalid or child is null
if (whichChild == 0) {
    parent->left = child;
}
else {
    parent->right = child;
}
child->parent = parent;
UpdateHeight(parent);
```

Operation. **ReplaceChild**(`parent, curChild, newChild`): uses `SetChild` to replace an existing child

```
if (curChild == parent->left) {
    return SetChild(parent, 0, newChild);
}
if (curChild == parent->right) {
    return SetChild(parent, 1, newChild);
}
return false;
```

13.2 Rotations and Balancing

The **right rotation** and **left rotation** algorithms are defined on a subtree root, which must have a left child or right child, respectively. Rotations locally rearrange the tree to balance it while maintaining ordering.

Operation. **RotateLeft**(node):

```
rightLeftChild = node->right->left; // identify new "right" child
if (node->parent) {
    ReplaceChild(node->parent, node, node->right); // reassign parent pointer
}
else { // root case
    root = node->right;
    root->parent = null;
}
SetChild(node->right, 0, node); // "rotation" where current node becomes a right child
SetChild(node, 1, rightLeftChild); // assignment of new "right" child
```

Operation. **RotateRight**(node):

```
leftRightChild = node->left->right; // identify new "left" child
if (node->parent) {
    ReplaceChild(node->parent, node, node->left); // reassign parent pointer
}
else { // root case
    root = node->left;
    root->parent = null;
}
SetChild(node->left, 1, node); // "rotation" where current node becomes a right child
SetChild(node, 0, leftRightChild); // assignment of new "left" child
```

When an AVL node has a balance factor of |2|, which only occurs after insertion or removal, the node must be **rebalance** using rotations.

Operation. **Rebalance**(node): updates the height, computes the balance factor, and rotates if warranted

```
UpdateHeight(node);
if (GetBalance(node) == -2) {
    if (GetBalance(node->right) == 1) {
        RotateRight(node->right);
    }
    return RotateLeft(node);
}
else if (GetBalance(node) == 2) {
    if (GetBalance(node->left) == -1) {
        RotateLeft(node->left);
    }
    return RotateRight(node);
}
return node;
```

13.3 Insertion

Insertion into a tree may cause imbalance, which would require a rotation to rebalance. An insertion on the *inside* of a subtree may necessitate a *double* rotation.

After insertion, there exists 4 possible cases for how a tree may become unbalanced if inserted on a leaf node. The newly inserted child could be the left or right child of either the original left or right child. If the new child is inserted along the same path (*left-left* or *right-right*), then only one rotation in the other direction is needed. Otherwise, an initial rotation in the direction of the path is needed to "align" the new path, and then a rotation in the other direction can be performed to fully rebalance.

As such, insertion insoles searching for the insert location, inserting, then updating balance factors and rebalancing if needed.

Operation. **Insert**(node): standard BST insertion, but rebalancing occurs afterward

```
if (!root) {
    root = node;
    node->parent = null;
    return;
}
// BST insertion
.
.
.
node = node->parent;
while (node) { // iterate through path to rebalance all nodes that need it
    Rebalance(node);
    node = node->parent;
}
```

13.4 Removal

Removal uses the standard BST removal algorithm and simply traverses back through the node's ancestors and checks if anything needs to be rebalanced.

Operation. **Remove**(node): standard BST remove with rebalancing at the end

```
if (!node) {
    return false;
}
parent = node->parent; // take copy of parent node
// BST removal
.
.
.
node = parent;
while (node) {
    Rebalance(node);
    node = node->parent;
}
return true;
```

14 The Red-Black Tree

B-trees, now with colors

A **red-black tree** is a balanced BST with 2 node types: red and black. The nodes follow the guidelines:

1. every node is either red or black
2. the root is black
3. a red node's children cannot be red
4. a null child is a black leaf node
5. all paths from a node to any null leaf descendant must have the same number of black nodes

14.1 Insertion

Insertion recolors and balances nodes in the originally traversed path to maintain ordering properties. Both insertion and rotation logic carry over from BST and AVL trees. Utility functions are first defined to simplify balancing:

Operation. **GetGrandparent**(node):

```
if (!node->parent) {
    return null;
}
return node->parent->parent;
```

Operation. **GetUncle**(node):

```
grandparent = null;
if (!node->parent) {
    grandparent = node->parent->parent;
}
if (!grandparent) {
    return null;
}
if (grandparent->left == node->parent) {
    return grandparent->right;
}
else {
    return grandparent->left;
}
```

Insertion itself is relatively simple; `BSTInsert` is used to insert the node using BST logic and then colored red:

Operation. **Insert**(node):

```
BSTInsert(node);
node->color = red;
Balance(node);
```

Balancing is accordingly defined:

Operation. **Balance**(node):

```
if (!node->parent) {
    node->color = black;
    return;
}
if (node->parent->color == black) {
    return;
}
parent = node->parent;
grandparent  GetGrandparent(node);
uncle = GetUncle(node);
if (uncle && uncle->color == red) {
    parent->color = uncle->color = black;
    grandparent->color = red;
    Balance(grandparent);
    return;
}
if (node == parent->right && parent == grandparent->left) {
    RotateLeft(parent);
    node = parent;
    parent = node->parent;
}
else if (node == parent->left && parent = grandparent->right) {
    RotateRight(parent);
    node = parent;
    parent = node->parent;
}
parent->color = black;
grandparent->color = red;
if (node == parent->left) {
    RotateRight(grandparent);
}
else {
    RotateLeft(grandparent);
}
```

14.2 Removal

Removal is a little more involved. `BSTSearch` is first used to find the node, and `RemoveNode` is recursively called to execute the actual removal.

Operation. **Remove**(key):

```
node = BSTSearch(key);
if (node) {
    RemoveNode(node);
}
```

Operation. **RemoveNode**(node): removes a node from a tree while maintaining ordering

```
if (node->left && node->right) {
    preNode = GetPredecessor(node);
    preKey = *preNode;
    RemoveNode(preNode);
    *node = preKey;
    return;
}
if (node->color == black) {
    RemovalPrep(node);
}
BSTRemove(*node);
```

Operation. **GetPredecessor**(node): utility function

```
node = node->left;
while (node->right) {
    node = node->right;
}
return node;
```

Additional utility functions can further simplify removal code:

Operation. **GetSibling**(node):

```
if (node->parent) {
    if (node == node->parent->left) {
        return node->parent->right;
    }
    return node->parent->left;
}
return null;
```

Operation. **NonNullAndRed**(node):

```
if (!node) {
    return false;
}
return (node->color == red);
```

Operation. **NullOrBlack**(node):

```
if (!node) {
    return true;
}
return (node->color == black);
```

Operation. **BothChildrenBlack**(node):

```
return (!NonNullAndBlack(node->left) && !NonNullAndNlack(node->right));
```

Preparation for removing a black node requires altering the number of black nodes along paths to preserve the black-path-length property. This can be done by using 6 utility functions that make appropriate alterations where needed:

Operation. **TryCases**(): if a case doesn't apply, it return **false**

```
TryCase1(node) { // node is red or parent is null
    return (!node->parent or node->color == red);
}

TryCase2(node, sib) { // sibling node is red
    if (sib->color == red) {
        node->parent->color = red;
        sib->color = black;
        if (node = node->parent->left) {
            RotateLeft(node->parent);
        }
        else {
            RotateRight(node->parent);
        }
        return true;
    }
}

TryCase3(node, sib) { // parent is black and both of sibling's children are black
    if (node->parent->color == black and BothChildrenBlack(sib)) {
        sib->color = red;
        RemovalPrep(node->parent);
        return true;
    }
}

TryCase4(node, sib) { // parent is red and both of sibling's children are black
    if (node->parent->color == red and BothChildrenBlack(sib)) {
        node->parent->color = black;
        sib->color = red;
        return true;
    }
}

TryCase5(node, sib) { // sibling's left+right children are red+black, node is left child
    if (NonNullAndRed(sib->left) and NullOrBlack(sib->right) and node == node->parent->left) {
        sib->color = red;
        sib->left->color = black;
        RotateRight(sib);
        return true;
    }
}

TryCase6(node, sib) { // sibling's left+right children are black+red, node is right child
    if (NonNullAndRed(sib->right) and NullOrBlack(sib->left) and node == node->parent->right) {
        sib->color = red;
        sib->right->color = black;
        RotateLeft(sib);
        return true;
}
```

The `RemovalPrep` algorithm then combines these functions:

Operation. **RemovalPrep**(`node`):

```
if (TryCase1(node)) {
    return;
}

sib = GetSibling(node);
if (TryCase2(node, sib)) {
    sib = GetSibling(node);
}
if (TryCase3(node, sib)) {
    return;
}
if (TryCase4(node, sib)) {
    return;
}
if (TryCase5(node, sib)) {
    sib = GetSibling(node);
}
if (TryCase6(node, sib)) {
    sib = GetSibling(node);
}

sib->color = node->parent->color;
node->parent->color = black;
if (node == node->parent->left) {
    sib->right->color = black;
    RotateLeft(node->parent);
}
else {
    sib->left->color = black;
    RotateRight(parent);
}
```

15 Hash Tables

A **hash table** stores ordered items by *hashing* each item to a location in an array or vector. A hash table's main advantage is that searching may require only O(1). Each hash table array element is a **bucket**, and a **hash function** is used to compute the index of a bucket from the item's key, which is ideally unique. A **collision** occurs when an inserted item maps to the same bucket as an existing item in the hash table.

15.1 Chaining

Chaining handles collisions by implementing a linked list in each bucket, where each list stores items that map to the same bucket.

Operation. **Search**(`key`): determines the bucket, then searches the bucket's list

```
bucketList = table[hash(key)]; // hash into the right bucket
itemNode = ListSearch(bucketList, key); // searches for the key in the list
if (itemNode) {
    return itemNode->data;
}
return null;
```

Operation. **Insert**(`item`): uses the item's key to determine the bucket, then inserts in the bucket's list

```
if (!Search(item->key)) { // only executes if item doesn't already exist
    bucketList = table[Hash(item->key)]; // hash into the right bucket
    new node(item)
    listAppend(bucketList, node); // add a newly allocated node to the bucket list
}
```

Operation. **Remove**(`item`): uses an item's key to determine the bucket then removes from the bucket's list

```
bucketList = table[Hash(key)]; // hash into the right bucket
itemNode = Search(item->key);
if (itemNode) { // checks if the item exists and executes
    ListRemove(bucketList, itemNode);
}
```

15.2 Linear Probing

Linear probing handles collisions by starting at a mapped bucket and then linearly searching subsequent buckets until an **empty-since-start** or **empty-after-removal** bucket is found. Search only stops for *empty-since-start* buckets.

Operation. **Insert**(`item`): determines the initial bucket and probes, cycling back at 0

```
bucket = Hash(item->key); // hash into initial bucket
bucketsProbed = 0; // initialize counter
while (bucketsProbed < N) { // ends if every bucket has been traversed
    if (table[bucket] is empty) { // inserts if an empty bucket is found
        table[bucket] = item;
        return true;
    }
    bucket = (bucket + 1) % N // update and keep moving
    bucketsProbed++;
}
return false;
```

Operation. **Search**(`key`): probes until a matching item, empty-since-start bucket, or all are traversed

```
bucket = Hash(key); // set up initial conditions
bucketsProbed = 0;
while (table[bucket] != EmptySinceStart and bucketsProbed < N) {
    if (table[bucket] ->key == key) { // found condition
        return table[bucket];
    }
    bucket = (bucket + 1) % N; // update and continue
    bucketsProbed++;
}
return null;
```

Operation. **Remove**(key): probes until either a matching bucket, empty-since-start, or all are traversed and marks a bucket empty-after-removal if executed

```
bucket = Hash(key);
bucketsProbed = 0;
while (table[bucket] != EmptySinceStart and bucketsProbed < N) {
    if (table[bucket]->key == key) {
        table[bucket] = EmptyAfterRemoval;
        return;
    }
    bucket = (bucket + 1) % N;
    bucketsProbed++;
}
```

15.3 Quadratic Probing

Quadratic probing handles collisions similarly to linear probing, except it uses the formula $(H + Ai + Bi^2)$ % N for a bucket H to determine the item's index in the hash table, where A and B are programmer-defined constants for probing. It uses a **probing sequence** wherein iteration starts at $i = 0$ and the index is incremented by 1 each time an empty bucket is not found.

Operation. **Insert**(item): inserts an item by following a quadratic probing sequence

```
i = 0, bucketsProbed = 0; // initial conditions
bucket = Hash(item->key);
while (bucketsProbed < N) {
    if (table[bucket] is Empty) {
        table[bucket] = item;
        return true;
    }
    i++;
    bucketsProbed++;
    bucket = (Hash(item->key) + A*i + B*i*i) % N // updated probing sequence
}
return false;
```

Operation. **Search**(key): uses probing until key is found or empty-since-start bucket is found

```
i = 0, bucketsProbed = 0; // initial conditions
bucket = Hash(key);
while (table[bucket] !+ EmptySinceStart and bucketProbed < N) {
    if (table[bucket]->key == key) {
        table[bucket] = EmptyAfterRemoval;
        return true;
    }
    i++;
    bucketsProbed++;
    bucket = (Hash(key) + A*i + B*i*i) % N; // update probing sequence
}
return false;
```

Operation. **Remove**(key): searches for the key to remove and marks if executed

```
i = 0, bucketsProbed = 0;
bucket = Hash(key);
while (table[bucket] != EmptySinceStart and bucketsProbed < N) {
    if (table[bucket]->key == key) {
        table[bucket] = EmptyAfterRemoval;
        return true;
    }
    i++;
    bucketsProbed++;
    bucket = (Hash(key) + A*i + B*i*i) % N;
}
return false;
```

15.4 Double Hashing

Double hashing is an open-addressing collision resolution technique where given two hash functions A, B, a key k's index in a table is then computed $(Ak + iBk) \ \% \ N$. The probing sequence accordingly starts at $i = 0$ and iterates through sequential values of i incremented by 1.

Insertion, removal, and search are accordingly done in the probing sequence defined by two hash functions.

15.5 Resizing

A hash table **resize** operation increases the number of buckets while preserving existing items. A hash table with N buckets is commonly resized to the next prime number $\geq N \cdot 2$. A new array is allocated and all items for the old array are re-hashed into the new array, giving resize operations $O(N)$ time complexity.

A hash table's **load factor** is the number of items in the hash table divided by the number of buckets, and can be used to decide when to resize the hash table, alongside considerations like the number of collisions encountered when hashing or the size of a bucket's linked list if chained.

15.6 Hash Functions

A *good* hash function *minimizes* collisions and *uniformly distributes* items into buckets. Ideal hash functions in conjunction with chaining result in short bucket lists or minimal average linear probing length if linear probing is used. A **perfect hash function** results in no collisions and can be created if the number of items and all possible item keys are known beforehand.

A **direct hash** uses the item's key as the bucket index, and is used in a **direct access table**. Direct hashing can have no collisions provided every key is unique, but is limited in the sense that it requires positive keys and the hash tables size equals the largest key +1, which could be prohibitively large.

A *binary* **mid-square hash** squares the key, extracts the middle R bits and returns the remainder of the middle bits divided by table size N, where $R \geq \lceil \log_2 N \rceil$

```
MidSquareHash(int key) {
    squaredKey = key * key;
    lowBitsToRemove = (32 - R) / 2;
    extractedBits = squaredKey >> lowBitsToRemove;
    extractedBits = extractedBits & (0xFFFFFFFF << (32 - R));
    return extractedBits % N;
}
```

A **multiplicative string hash** repeatedly multiples the hash value and adds the ASCII value of each character in the string, starting with a large initial value. For each character, the hash function multiples the current hash value by an often prime and adds the character's value. The function then returns the remainder of the sum divided by table size N.

```
MultiplicativeHash(string key) {
    stringHash = InitiaValue;
    for (strChar : key) {
        stringHash = (stringHash * HashMultiplier + strChar);
    }
    return stringHash % N;
}
```

16 Graphs

A **graph** consists of **vertices** connected by **edges**. Vertices are **adjacent** if connected by an edge, and a **path** is a sequence of edges connecting a source and a destination vertex where the **path length** is the number of edges in the path. The **distance** is the shortest number of edges between two vertices.

16.1 Representing Adjacency

In an **adjacency list**, each vertex is attached to a list of adjacent vertices, where each list item represents an implicit edge. Adjacency lists require a space of $O(V+E)$, since each vertex V appears once and each edge E appears twice. Determining adjacency is done in $O(V)$ time for a list with V item, but this time is much smaller for a **sparse graph**, where there are far fewer edges than possible which tends to be the case for real-world representations.

In an **adjacency matrix**, a $V \times V$ matrix is used to denote connection. While elements are accessible in $O(1)$ time, the matrix accordingly requires $O(V^2)$, which would be very inefficient for a sparse graph, in which most elements would be 0. An adjacency matrix only represents edges; vertices with data need to be stored separately.

16.2 Searching

A **breadth-first search** (BFS) visits a starting vertex, then all vertices of sequentially increasingly distance without revisiting a vertex.

Operation. **BFS**(`startV`): uses a queue to implement search

```
frontierQueue.push(startV);
add startV to discoveredSet
while (frontierQueue is not empty) {
    currentV = frontierQueue.pop();
    do something with currentV
    for (each vertex adjV adjacent to currentV) {
        if (adjV not in discoveredSet) {
            frontierQueue.push(adjV);
            add adjV to discoveredSet
        }
    }
}
```

When a vertex is first encountered, it is **discovered**. It is then queued into the **frontier**, being vertices thus discovered but not yet visited. An already-discovered vertex is not pushed to the frontier after visiting.

A **depth-first search** (DFS) traverses along each path from a starting vertex to the path's end before backtracking.

Operation. **DFS**(`startV`): uses a stack to implement search

```
stack.pop(startV);
whlile (!stack.empty()) {
    currentV = stack.pop();
    if (currentV is not in visitedSet) {
        do something with currentV
        add currentV to visitedSet
        for (each vertex adjV adjacent to currentV) {
            stack.push(adjV);
        }
    }
}
```

A recursive DFS can also be implemented using the program stack instead of an explicit stack.

Operation. **RecrusiveDFS**(`currentV`):

```
if (currentV not in visitedSet) {
    add currentV to visitedSet
    do something with currentV
    for (each vertex adjV adjacent to currentV) {
        RecursiveDFS(currentV);
    }
}
```

17 Graphs, But Spicier

17.1 Directed Graphs

A **directed graph** or **digraph** consists of vertices connected to **directed edges** between a starting and terminating vertex. In a digraph, a vertex Y is *adjacent* to a vertex X if there is an edge from X to Y.

A **path** is a sequence of directed edges leading from a source vertex to a destination vertex and a **cycle** is a path that starts and ends at the same vertex. A graph is **cyclic** if it contains a cycle and **acyclic** if it doesn't.

17.2 Weighted Graphs

A **weighted graph** associates a **weight** or **cost** between vertices. The **path length** in this case is the sum of edge weights in a path. The **cycle length** is the sum of edge weights in a cycle. A **negative edge weight** has a cycle weight < 0. A shortest path doesn't exist for a graph with a negative edge weight cycle since each loop around the negative edge weight cycle decreases the cycle length, so no minimum exists.

17.3 Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm determines the shortest path from a start vertex to each vertex in a graph. For each vertex, the algorithm determines the vertex's shortest path **distance** fro the start and a **predecessor pointer** that points to the previous vertex along the shortest path from the start.

All distances are initialized to ∞, all predecessors are initialized to 0, and all vertices are pushed to a queue of unvisited vertices. The start vertex's distance is then set to 0.

Operation. **DijkstraSP**(`startV`):

```
for (each vertex v in graph) { // iterate through and initialize every vertex
    v->distance = Infinity;
    v->pred = 0;
    unvisitedQueue.push(v);
}
startv->distance = 0;
while (!unvisitedQueue.empty()) {
    v = PopMin unvisitedQueue // visit closest adjacent vertex
    for (each vertex adjv adjacent to v) {
        edgewt = weight of edge from v to adjv
        altDistance = v->distance + edgewt;
        if (altDistance < adjv->distance) { // update if shorter path found
            adjv->distance = altDistance;
            adjv->pred = v;
        }
    }
}
```

After execution of Dijkstra's algorithm, the shortest path from the start to a destination vertex can be determined using predecessor pointers. If the destination vertex's predecessor pointer is 0, then a path to the start doesn't exist. For a digraph with negative edge weights, the algorithm may not find the shortest path for some vertices, and so should not be used if negative weights are used. If the unvisited vertex queue is implemented using a list, the runtime is $O(V^2)$. Using a binary heap instead reduces the runtime to $O((E + V) \log V)$.

17.4 Bellman-Ford's Shortest Path Algorithm

The **Bellman-Ford shortest path algorithm** traverses the graph in arching iterations instead of the specific order used by Dijkstra's algorithm, updating distances and predecessors as necessary with each traversal.

Operation. **BellmanFordSP**(`startV`): iteratively checks and updates distances and predecessor nodes

```
for (vertex v in graph) { // initialize all vertices
    v->distance = Infinity;
    v->pred = 0;
}
startV->distance = 0;
for (i = 1 : number of vertices - 1) { // main number of iterations
    for (each vertex v in graph) {
        for (each vertex adjv adjacent to v) {
            edgewt = weight of edge from v to adjv
            altDistance = v->distance + edgeWeight;
            if (altDistance < adjv->distance) { // update if shorter path found
                adjv->distance = altDistance;
                adjv->pred = v;
            }
        }
    }
}
for (each vertex v in graph) { // checking for negative weight cycles
    for (each vertex adjv adjacent to v) {
        edgewt = weight of edge from v to adjv
        altDistance = v->distance + edgewt;
        if (altDistance < adjv->distance) { // nwc exists if another shorter path found
            return false;
        }
    }
}
return true;
```

17.5 Topological Sort

A **topological sort** of an acyclic digraph produces a list of vertices such that for every edge from a vertex X to a vertex Y, X comes before Y in the list; the ordering must respect the directions of each vertex.

Operation. **TopologicalSort**():

```
results = empty list of vertices
noIn = list of all vertices with no incoming edges
remaining = list of all edges in the graph
while (!noIn.empty()) {
    v = remove any vertex from noIn
    add v to results // vertices added in progressive no-incoming order
    outgoing = remove v's outgoing edges from remaining // build list of outoing edges
    for (each edge e in outgoing) {
        inCount = GetInCount(remaining, e->toVertex);
        if (inCount == 0) { // added to no-incoming list if no incoming
            add e->toVertex to noIn
        }
    }
}
return results;
```

Operation. **GetInCount**(`edgeList, vertex`): counts the number of remaining edges that are incoming to a vertex

```
count = 0;
for (each edge e in edgeList) {
    if (edge->toVertex == vertex) {
        count++;
    }
}
return count;
```

The space complexity of topological sorting is O($|V| + |E|$). If a graph implementation allows retrieval of incoming and outgoing edges in constant time, then the time complexity is also O($|V| + |E|$).

17.6 Minimum Spanning Trees

A weighted and connected graph's **minimum spanning tree** is a subset of edges that connect all vertices in the graph together with the minimum sum of edge weights. A **connected** graph contains a path between *every* pair of vertices.

Kruskal's minimum spanning tree algorithm determines the subset of edges that connect all vertices in an *undirected* graph with the minimum sum of edge weights.

Operation. **KruskalMST**(graph):

```
edges = all edges of the graph
vSets = collections of initially empty vertex sets
for (each vertex v in graph) {
    add new set containing only v to vSets
}
results = new, empty set of edges
while (vSets->size > 1 and edges->size > 0) {
    nextEdge = remove edge with minimum weight from edges
    vSet1 = set in vSets containing nextEdge->v1
    vSet2 = set in vSets containing nextEdge->v2
    if (vSet1 != vSet2) { // vertex sets are only merged if distinct
        add nextEdge to results
        remove vSet1 and vSet2 from vSets
        merged = union(vSet1, vSet2);
        add merged to vSets
    }
}
return results;
```

Kruskal's minimum spanning tree algorithm's use of the edge list, collection of vertex sets, and resulting edge list results in a space complexity of $O(|E| + |V|)$. If the edge list is sorted at the beginning, then the minimum edge can be removed in constant time. Combined with a mechanism to map a vertex to the containing vertex set in constant time, the algorithm has a runtime complexity of $O(|E| \log |E|)$.

17.7 All Pairs Shortest Path

An **all pairs shortest path** algorithm determines the shortest path between all possible pairs of vertices in a graph. A $|V| \times |V|$ matrix represents the shortest path lengths between all vertex pairs in the graph, with each row corresponding to a start vertex and each column corresponding to a terminating vertex for each path.

The **Floyd-Warshall all-pairs shortest path algorithm** generates a $V \times V$ matrix of values representing the shortest path lengths between all vertex pairs in a graph. A graph with negative cycles is not supported.

Operation. **FloydWarshallAPSP**(graph): computes shortest path lengths for all vertex pairs, recomputing as needed.

```
numVertices = graph->vertexCount;
set all values in distMat to infinity
set each distance for vertex to itself to 0
for (each edge in graph) { each immediate edge is initialized with its weight
    distMat[edge->fromVertex][edge->toVertex] = edge->weight;
}
for (i = 0 : numVertices) {
    for (toIdx = 0 : numVertices) {
        for (fromIdx = 0 : numVertices) {
            curLength = distMat[fromIdx][toIdx];
            altLength = distMat[fromIdx][i] + distMat[i][toIdx];
            if (altLength < curLength) {
                distMat[fromIdx][toIdx] = altLength;
            }
        }
    }
}
return distMat;
```

17.8 Path Reconstruction

The shortest-path matrix can also be used to reconstruct a path sequence. Given the shortest length from a start to an end vertex L, an edge from vertex X to the end vertex must exist such that the shortest length from the starting vertex to X plus the edge weight equals L. Each such edge is found and the path is reconstructed in reverse.

Operation. **FloydWarshallReconstructPath**(`graph`, `startV`, `endV`, `distMat`):

```
path = new empty path
v = endV;
while (v != startV) {
    inEdges = all edges in the graph incoming to current vertex
    for (each edge in incomingEdges) {
        expected = distMat[startV][v] - edge->weight;
        actual = distMat[startV][edge->fromVertex];
        if (expected == actual) {
            v = edge->fromVertex;
            prepend edge to path
            break;
        }
    }
}
return path;
```

The algorithm builds a $V \times V$ matrix and therefore has a O(V^2) space complexity constructed with a runtime complexity of O(V^3).