Summary of CPSC 221

Tom Wang

Spring, 2024

1 Complexity

A "good" algorithm should:

- Be correct (or at least mostly correct)
- finish in a reasonable amount of time (time complexity)
- uses a reasonable amount of memory (space complexity)

1.1 Time Complexity

Running time is expressed as a function $T(n) : \mathbb{Z}^0 \to \mathbb{R}^0$, where n is the size of the input. We are interested in the asymptotic behavior of $T(n)$ as $n \to \infty$. We use big-O notation to express this.

1.1.1 Asymptotic Notation

Definition 1. $T(n) \in O(f(n))$ *if there exists* $c > 0$ *and* $n_0 > 0$ *such that* $T(n) \leq cf(n)$ *for all* $n \geq n_0$.

$T(n) \in \Omega(f(n))$ *if there exists* $c > 0$ *and* $n_0 > 0$ *such that* $T(n) \geq cf(n)$ *for all* $n \geq n_0$.

$T(n) \in \Theta(f(n))$ *if* $T(n) \in O(f(n))$ *and* $T(n) \in \Omega(f(n))$.

$T(n) \in o(f(n))$ *if for all* $c > 0$*, there exists* $n_0 > 0$ *such that* $T(n) \leq cf(n)$ *for all* $n \geq n_0$.

$T(n) \in \omega(f(n))$ *if for all* $c > 0$*, there exists* $n_0 > 0$ *such that* $T(n) \geq cf(n)$ *for all* $n \geq n_0$.

The reason we use big-O notation and omit the details is that it is easy to work with. For example, if $T(n) \in O(f(n))$ and $S(n) \in O(g(n))$, then $T(n) + S(n) \in O(f(n) + g(n))$ and $T(n)S(n) \in O(f(n)g(n))$.

We only care about the dominant term in the function (When n gets really large). We also ignore constant factors because we are only interested in the asymptotic behavior.

1.1.2 Analyze loops

```
bool hasDuplicate(int arr[], int size){
    for(int i=0; i<size-1; i++){
        for(int j=+1; j<size; j++){
            if(arr[i]==arr[j]){
                return true;
            }
        }
    }
    return false;
}
```

Now we analyze the running time of this algorithm. Basically, counts for the worst-case scenario.

It is easy to tell that the outer loop runs $n-1$ times, thus $O(n)$.

The inner loop runs from $n-1$ to 1 times if we run the loop. However, we can use the fact that the sum of the first n integers is $\frac{n(n-1)}{2}$. So the inner loop runs $\frac{n(n-1)}{2}$ times, thus $O(n^2)$. This is the overall complexity of the nested loop.

Give more examples:

```
void candyapple(int n) {
    for (int i = 1; i < n; i *= 3)
    cout << "iteration:■" << i << endl;
}
void caramelcorn(int n) {
    for (int i = 0; i * i < 6 * n; i++)
    cout << "iteration:■" << i << endl;
}
```

If the loop variable is multiplied by a constant, then the loop runs in $O(\log n)$ time. If the loop variable is squared, then the loop runs in $O(\sqrt{n})$ time.

So candyapple runs in $O(\log n)$ time and caramelcorn runs in $O(\sqrt{n})$ time.

Proof. in candyapple, i is multiplied by 3 each time, so $i = 3^k$ after k iterations. So $3^k = n$, so $k = \log_3 n$. So the loop runs $\log_3 n$ times, thus $O(\log n)$.

in caramelcorn, $i^2 = 6n$, so $i = \sqrt{6n}$. So the loop runs $\sqrt{6n}$ times, thus $O(\sqrt{n})$. □

lastly, we can analyze the following code:

```
int i, j;
for (i = 1; i < 9*n; i = i*2) {
    for (j = n*n; j > 0; j--) {
        ...
    }
}
```

The outer loop runs $\log_2 9n$ times, thus $O(\log n)$. The inner loop runs n^2 times, thus $O(n^2)$. So the overall complexity is $O(n^2 \log n)$.

2 Correctness

Former methods to argue Correctness:

2.1 Loop Invariant

Definition 2.
- *A loop invariant is a property that holds before and after each iteration of a loop.*
- *Within a loop body, these properties may be violated briefly but must be fixed for the next iteration.*
- *A loop invariant proof must use features of the code being analyzed to demonstrate that any violations of loop invariants are fixed.*

2.1.1 Proving a loop invariant

Generally, we use induction to prove it:

1. Find an induction variable or property.
2. Base case: show that the loop invariant holds before the first iteration.

3. Inductive hypothesis: assume that the loop invariant holds before the kth iteration. k can be unspecified.

4. Inductive step: show that the loop invariant holds for the $(k+1)$th iteration.

5. Termination: show that the loop invariant holds after the last iteration.

3 Memory

4 area of memory:

1. Code: the compiled machine code of the instructions.

2. Stack: local variables, function parameters, and return addresses. Automatically allocated and deallocated. (Usually about few MB)

3. Heap: dynamically allocated memory. Must be manually allocated and deallocated. (Usually about few GB)

4. NULL: the area of memory that is not used. Address 0x0.

3.1 Call by value and call by reference

```
void swap_wrong(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}
void swap_true(int & a, int & b){
    int temp = a;
    a = b;
    b = temp;
}
int main(){
    int x = 3;
    int y = 5;
    swap(x, y);
    cout << x << "and" << y << endl;
}
```

The output of the above code is 3 5. This is because the swap function is called by value, so the original x and y are not changed.

If we change the swap function to swap_true, then the output will be 5 3. This is because the swap function is called by reference, so the original x and y are changed.

3.2 Pointers

Pointers are variables that store the address of another variable. It is a powerful tool but also dangerous. It is powerful because it can access the memory directly. It is dangerous because it can access the memory directly.

```
int main(){
    int x = 3;
    int * p = &x;
    cout << *p << endl;
    *p = 5;
    cout << x << endl;
}
```

The output of the above code is 3 5. This is because the pointer p is pointing to the address of x. So *p is the value of x. So *p = 5 is the same as x = 5.

Note that the type of p is int*, which means it is a pointer to an integer. The type of *p is int, which means it is the value of the integer that p is pointing to.

3.2.1 new and delete

```
int main(){
    int * p = new int;
    *p = 5;
    cout << *p << endl;
    delete p;
}
```

The above code is an example of dynamic memory allocation. The new operator allocates memory for an integer and returns a pointer to the memory. The delete operator deallocates the memory. It will give memory space in the heap instead of the stack. Thus it is a good way to avoid stack overflow if we want some huge data structure.

3.2.2 Memory leak

```
int main(){
    int * p = new int;
    p = new int;
    delete p;
}
```

The above code is an example of a memory leak. The first new operator allocates memory for an integer and returns a pointer to the memory. The second new operator allocates memory for another integer and returns a pointer to the memory. The delete operator deallocates the memory for the second integer. However, the memory for the first integer is not deallocated. This is a memory leak.

4 Linked List

A linked list is a data structure that stores a sequence of elements. Each element is a node that contains a value and a pointer to the next node. The last node points to NULL.

4.1 Node Pointers

```
struct Node{
    //Can be any type of data stored in the node
    type value;
    Node * next;
    //Constructor, by default next is NULL
    Node(type value, Node * next = NULL){
        this->value = value;
        this->next = next;
    }
};
```

example of the implementation of a node.

```
Node * head = new Node(3);
Node * second = new Node(5);
Node * third = new Node(7);
```

```
head->next = second;
//head->next->next = third;
second->next = third;
//Both of the above are equivalent
```

4.1.1 Traversing a linked list

```
Node * current = head;
while(current != NULL){
    cout << current->value << endl;
    current = current->next;
}
```

The above code traverses the linked list and prints out the value of each node.

4.1.2 Inserting a node

```
//Insert a node after the current node with the value
void insert(Node * & current_head, int value){
    Node * new_node = new Node(value);
    new_node->next = current_head->next;
    current_head->next = new_node;
}
```

4.1.3 Deleting a node

```
//Delete the node after the current node
void delete(Node * & current_head){
    Node * temp = current_head->next;
    current_head->next = current_head->next->next;
    delete temp;
}
```

4.2 Other features with linked list

List out other possible add-on features of linked list to improve the efficiency of the linked list.

4.2.1 Linkage

- Singly linked list: each node only points to the next node.
- Doubly linked list: each node points to the next node and the previous node.

4.2.2 Access/Entry

- Front/Head pointer: a pointer to the first node.
- Back/Tail pointer: a pointer to the last node.

4.2.3 Termination

- Circular linked list: the last node points to the first node.
- Sentinel node: a dummy node that is always there. The first node points to the head sentinel node. The last node points to the last sentinel node.
- NULL: the last node points to NULL.

4.3 double linked list

Links to the previous node and the next node. It is useful for traversing the linked list backward. However, it takes more space and time to maintain the links.

```
struct Node{
    //Can be any type of data stored in the node
    type value;
    Node * next;
    Node * prev;
    //Constructor, by default next is NULL
    Node(type value,
         Node * next = NULL,
         Node * prev = NULL){
         this->value = value;
         this->next = next;
         this->prev = prev;
    }
};
```

```
//Insert a node after the current node with the value
void insert(Node * & current_head, int value){
    Node * new_node = new Node(value);
    //Note the order of the following two lines
    new_node->next = current_head->next;
    new_node->prev = current_head;
    current_head->next->prev = new_node;
    current_head->next = new_node;
}
//Delete the given node
void delete(Node * & target){
    target->prev->next = target->next;
    target->next->prev = target->prev;
    delete target;
}
```

4.4 Recursive nature of linked list

Consider that we need to print the value of each node in a signly linked list.

If we access each node independently, then we have to start from the head each time. Each visit is $O(n)$, so the total running time is $O(n^2)$.

If we use recursion, then we can start from the head and print the value of the current node. Then we can recursively call the function on the next node. Each visit is $O(1)$, so the total running time is $O(n)$.

```
void print(Node * current){
    if(current == NULL){
        return;
    }
    cout << current->value << endl;
    print(current->next);
}
```

4.4.1 Reverse a linked list and print odd-numbered nodes

```
    void PrintReverseOdds(Node * current){
        //Base case, last one or second last one
```

```
        if(current == NULL || current->next == NULL){
            return;
        }
        //Recursive step
        //recurse first to print the last one first
        PrintReverseOdds(current->next->next);
        cout << current->value << endl;
    }
```

5 Stack

Stack is a data structure that stores a sequence of elements. It is a LIFO (Last In First Out) structure. It has two operations: push and pop. Push adds an element to the top of the stack. Pop removes the top element of the stack.

5.1 Stack implementation

Here is the public interface of the stack class.

```
//the type of the elements in the stack
template <typename T>
class Stack {
    public:
        Stack();
        bool IsEmpty() const;
        void Push(T value);
        T Pop();
    private:
        //the rest of the implementation
}
```

We can implement using a linked list or an array.

- Linked list: all operatoins are $O(1)$.
- Array: Each push and pop is $O(1)$, but we need to resize the array if it is full which is $O(n)$. So the amortized running time is $O(1)$. However, we get a better cache performance.

6 Queue

Queue is a data structure that stores a sequence of elements. It is a FIFO (First In First Out) structure. It has two operations: enqueue and dequeue. Enqueue adds an element to the back of the queue. Dequeue removes the front element of the queue.

Like stacks, the operations are expected to be $O(1)$.

6.1 Queue implementation

6.1.1 Linked list

```
template <typename T>
class Queue {
    public:
        Queue();
        bool IsEmpty() const;
        void Enqueue(T value);
        T Dequeue();
    private:
        struct Node{
            T value;
            Node * next;
            Node(T value, Node * next = NULL){
                this->value = value;
                this->next = next;
            }
        };
        Node * front, * back;
}
bool Queue<T>::IsEmpty() const{
    return front == NULL;
}
bool Queue<T>::Enqueue(T value){
    Node * new_node = new Node(value);
    if(IsEmpty()){
        front = new_node;
    }else{
```

```
        back->next = new_node;
    }
    back = new_node;
}
T Queue<T>::Dequeue(){
    if(IsEmpty()){
        throw "Queue is empty";
    }
    T value = front->value;
    Node * temp = front;
    front = front->next;
    delete temp;
    return value;
}
```

Really easy to implement using a linked list. All operations are $O(1)$.

6.1.2 Array

Array size is fixed, so the front and the end cannot move physically. Copying the array is $O(n)$, so the enqueue and dequeue are $O(n)$. We can use a circular array to avoid copying the array.

The idea of a circular array is that the end of the array "wraps around" to the beginning of the array. We use the modulo operator to calculate the index of the next element.

The only problem it resizing. When resizing, we need to copy the other. But note that we should put the front of the queue at the beginning of the array instead of the original order of the queue. It is because the capacity changed, we need to start from the beginning of the array.

6.2 Priority Queue

A priority queue is a data structure that stores a sequence of elements. Each element has a priority. Enqueue adds an element to the priority queue. Dequeue removes the element with the highest priority.

- Prioritisation is a weaker condition then ordering
- Order of insertion is not important

- A priority queue is an ADT that maintains a multiset of items: allows duplicates
- Two or more distinct items in a priority queue may have the same priority
- If all items have the same priority, the priority queue behavior is subject to the implementation

6.2.1 Complexity

- Enqueue: $O(\log n)$
- Dequeue: $O(\log n)$
- Peek: $O(1)$

6.2.2 Implementation

See heap section for the implementation of priority queue.

7 Sort

7.1 Selection Sort

```
void selectionSort(Vector<int> & arr) {
    for(int i=0; i<arr.size(); i++){
        int min = indexOfMin(arr, i);
        swap(arr[i], arr[min]);
    }
}
```

This algorithm finds the smallest element in the array and swap it with the first element. Then it finds the second smallest element and swap it with the second element. And so on.

For each iteration, we need to compare $n - i$ elements to find the smallest element. So a total of $\sum_{i=0}^{n-1} n - i = \frac{n(n-1)}{2}$ comparisons. Swap is just a function to perform each iteration, which could be seen as $O(n)$. So the running time is $O(n^2)$.

Note that the running time is independent of the input. So it is a **deterministic** algorithm. Running time is the same for best, average, and worst case.

Space complexity is $O(1)$ because we only need to store the array and use a few variables on the stack to keep track of the indices.

7.1.1 Invariant proof of selection sort

Proof. Claim that the loop invariant is that the first i elements are sorted. All elements before i are smaller than or equal to all elements after i.

Base case: before the first iteration, $i = 0$, so the first i elements are empty, so the invariant holds.

Inductive hypothesis: assume that the invariant holds before the kth iteration.

Inductive step: after the kth iteration, the smallest element in the array is swapped with the kth element. So the first k elements are sorted. All elements before k are smaller than or equal to all elements after k.

Termination: after the last iteration, $i = n$, so the first i elements are the whole array, which is sorted. All elements before i are smaller than or equal to all elements after i. □

7.2 Insertion Sort

```
void slide(Vector<int> & arr, int p) {
    //need arr is in ascending order up to p exclusive
    int temp = arr[p];
    int j = p;
    while(j>0 && arr[j-1]>temp){
        arr[j] = arr[j-1];
        j--;
    }
    arr[j] = temp;
}
void insertionSort(Vector<int> & arr) {
    for(int i=1; i<arr.size(); i++){
        slide(arr, i);
    }
}
```

7.2.1 Slide

Slide checks the pth element and moves to the front of the array if it is smaller than the previous elements. It is a $O(n)$ operation (worst case) and $\Omega(1)$ (best case) space complexity.

loop invariant:

- the first p elements are sorted.
- $arr[0 : j-1] \cup arr[j+1 : p] \cup temp$ is the same value **set** as $arr[0 : p]$.

7.2.2 Insertion Sort

Insertion sort is a $O(n^2)$ algorithm since it calls slide $n-1$ times where slide is $O(n)$.

Best case is a sorted array, so the running time is $O(n)$.

Worst case is a reverse sorted array, so the running time is $O(n^2)$.

If random data is given, the running time is closer to the worst case than the best case. Around $\frac{n(n-1)}{4}$ comparisons and $\frac{n(n-1)}{4}$ swaps.

Insertion sort is good for small data sets and nearly sorted data. Also, it barely uses any space (memory), so it is good for embedded systems.

7.2.3 correctness proof

Proof. **Base case:** before the first iteration, $i = 1$, so the first i elements are sorted. All elements before i are smaller than or equal to all elements after i.

Inductive hypothesis: assume that the invariant holds before the kth iteration.

Inductive step: after the kth iteration, the kth element is inserted into the first k elements. So the first $k+1$ elements are sorted. All elements before $k+1$ are smaller than or equal to all elements after $k+1$.

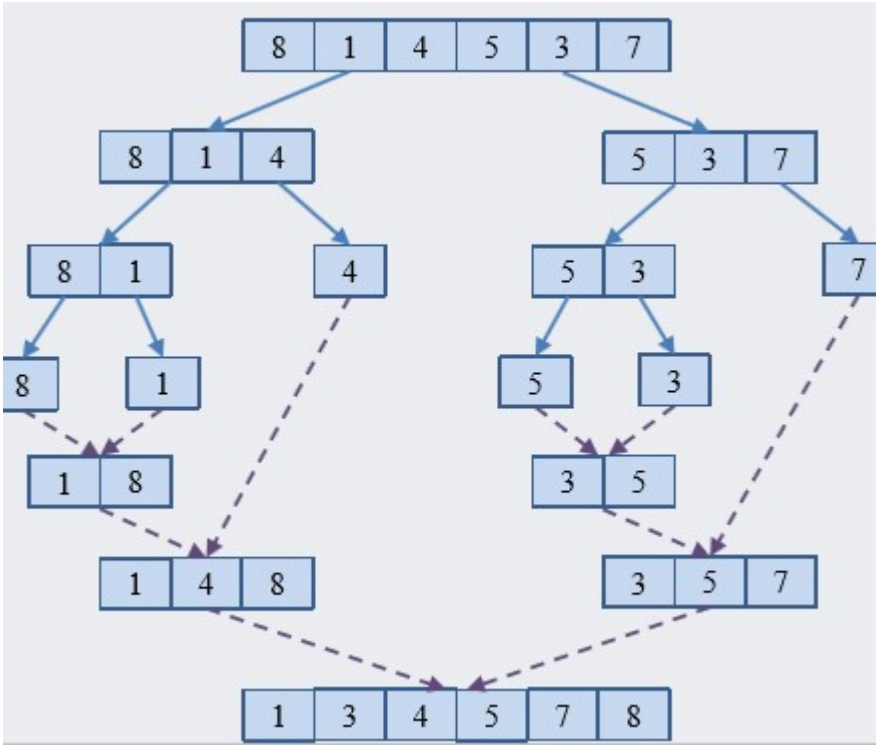
Termination: after the last iteration, $i = n$, so the first i elements are the whole array, which is sorted. All elements before i are smaller than or equal to all elements after i.

All of the above is true because of the loop invariant of slide. □

7.3 Merge sort

Merge sort is a divide and conquer algorithm. It divides the input into smaller parts, solves the smaller parts, and then combines the solutions to solve the original problem.

The running time of merge sort is $O(n \log n)$, which is the best possible running time for a comparison-based sorting algorithm.



7.3.1 implementation

```
void MergeSort(vector<T> & arr){
    Msort (arr, 0, arr.size()-1);
}
void Msort(vector<T> & arr, int low, int high){
    if(low < high){
        int mid = (low + high) / 2;
        Msort(arr, low, mid);
        Msort(arr, mid+1, high);
        Merge(arr, low, mid, high);
    }
}
void Merge(vector<T> & arr, int low, int mid, int high){
    vector<T> temp;
```

```
    int i = low;
    int j = mid+1;
    while(i <= mid && j <= high){
        if(arr[i] < arr[j]){
            temp.push_back(arr[i]);
            i++;
        }else{
            temp.push_back(arr[j]);
            j++;
        }
    }
    while(i <= mid){
        temp.push_back(arr[i]);
        i++;
    }
    while(j <= high){
        temp.push_back(arr[j]);
        j++;
    }
    for(int k = low; k <= high; k++){
        arr[k] = temp[k-low];
    }
}
```

7.3.2 Analysis

For merge, worst case we need to compare $n-1$ times, and best case $n/2$ times. So the running time is $O(n)$.

For Msort, the running time is $O(\log n)$ since we are dividing the array into half each time.

So the overall running time is $O(n \log n)$.

7.3.3 Correctness

Proof. Base case: if the array has only one element, then it is already sorted.

Inductive hypothesis: assume that the subarray from low to mid and the subarray from mid+1 to high are sorted.

Inductive step: merge the two sorted subarrays into one sorted array.

Termination: after the last merge, the whole array is sorted. □

7.4 Lower bound of sorting

A CORRECT algorithm must transform every possible input into the correct output.

For comparison-based sorting algorithms, the specific sequence can be represented as a binary tree.

- The height of the tree is the worst-case running time of the algorithm.
- The number of leaves is the number of possible outputs. Note that it is because each input case needs to be transformed into a unique output case.
- The height of the tree is at least $\log_2 n!$ because each layer can only have at most 2^h leaves where h is the height of the layer.

7.4.1 Lower bound of comparison-based sorting

A perfect tree with h levels has $2^{h+1} - 1$ nodes and 2^h leaves on the bottom level. So need to solve for $n! \geq 2^h$.

$$\begin{aligned} h &= \lceil \log_2 n! \rceil \\ &\geq \sum_{i=1}^{n} \log_2 i \\ &\geq \sum_{i=1}^{n/2} \log_2 \frac{n}{2} \\ &= \frac{n}{2} \log_2 \frac{n}{2} \\ &\in \Omega(n \log n) \end{aligned}$$

So the longest decision path can be no shorter than $n \log n$.

Note that non-comparison-based sorting algorithms can have a lower bound of n. But that has more restrictions on the input.

7.5 Topological Sort

Topological sort is an algorithm that sorts the vertices of a directed acyclic graph (DAG) in a linear order. It is useful for scheduling tasks that have dependencies.

Given

- a collection of vertexes V
- a collection of edges E

Want

- a collection of vertexes $V' = V$
- a collection of edges $E' \in E$ such that E' is a topological sort of V' meaning that for every edge $(u, v) \in E'$, u comes before v in V'.

Definition 3. *Topological sort: Given a graph $G = (V, E)$, output all vertices v_i in V such that for every edge $(v_i, v_j) \in E$, v_i comes before v_j in the output.*

7.5.1 Algorithm

function TOPOLOGICALSORT(G)
    Label each vertex's **in-degree** (number of incoming edges)
    Create a queue Q and enqueue all vertices with in-degree 0
    Create an empty list L
    while Q is not empty **do**
        Dequeue a vertex v from Q
        Add v to the end of L
        for each vertex u adjacent to v **do**
            Decrement u's in-degree
            if u's in-degree is 0 **then**
                Enqueue u to Q

Note that if the graph has a cycle, then the in-degree of some vertices will never be 0. So the algorithm will not terminate. So the graph must be a DAG (Directed Acyclic Graph).

7.5.2 Complexity

- Labeling in-degree: $O(V + E)$

- Enqueue all vertices with in-degree 0: $O(V)$
- Dequeue a vertex: $O(1)$ but need to iterate through all adjacent vertices.
- Decrement in-degree: $O(E)$
- Enqueue a vertex: $O(1)$ but need to iterate through all adjacent vertices.
- Total: $O(V + E)$

8 Dictionary/Map

A dictionary is a data structure that stores a set of key-value pairs.

- Values may be any (homogeneous) type.
- Keys may be any (homogeneous) **comparable** type.
- This is a motivation for an efficient search algorithm.

Since values rarely affect our design choices, we ignore them and focus on the keys that is handled by **Sets**

9 Sets

A set is a data structure that stores a set of keys.

- Keys may be any (homogeneous) **comparable** type.

9.1 Set implementation

9.2 Disjoint Set

9.2.1 Disjoint set ADT

No element is the member of different sets. The sets are disjoint. The ADT has the following operations:

- MakeSet(x): create a new set with the element x.
- Find(x): return the representative of the set that contains x.
- Union(x, y): merge the sets that contain x and y.

Find(x) = Find(y) if and only if x and y are in the same set. Union(x, y) = Union(y, x) causes their sets to be merged permanently.

9.2.2 Simple Array implementation

Index is the element ID, value is the set ID. Find is $O(1)$, Union is $O(n)$. Union need to update all elements in the set by linear search.

```
int parent[MAX_SIZE];
void MakeSet(int x){
    parent[x] = x;
}
int Find(int x){
    return parent[x];
}
void Union(int x, int y){
    int set_x = Find(x);
    int set_y = Find(y);
    for(int i=0; i<MAX_SIZE; i++){
        if(parent[i] == set_x){
            parent[i] = set_y;
        }
    }
}
```

9.3 Tree based array implementation

Still array based, but the array is a tree. Find is $O(\log n)$, Union is $O(\log n)$.

- if array value is -1, then it is the root of the tree.
- if array value is i, then the parent is i.
- $x, y \in S$ if and only if $Find(x) = Find(y)$ which means they are in the same tree.

```
int parent[MAX_SIZE];
void MakeSet(int x){
    parent[x] = -1;
}
int Find(int x){
    if(parent[x] == -1){
        return x;
    }
    return Find(parent[x]);
}
void Union(int x, int y){
    int set_x = Find(x);
    int set_y = Find(y);
    parent[set_x] = set_y;
}
```

9.3.1 Smarter Union

The naive union is $O(n)$ because we need to update all elements in the set. We can make it $O(\log n)$ by making the tree height smaller. We can make the tree height smaller by making the smaller tree a subtree of the larger tree. We can do this by comparing the size of the trees.

- So when union, we need to compare the size of the trees.
- We can store the size of the tree in the root node.
- We can make the tree with the smaller size a subtree of the tree with the larger size.

- At the same time, instead of comparing size, we can compare the height of the trees. We can store the height of the tree in the root node.

10 Trees

A tree will have a parent node connecting to multiple child nodes. The connection is called an edge.

Most basic implementation:

```
template <typename T>
class Tree {
    public:
        // ... all the other methods
    private:
        struct Node {
            T value;
            vector<Node *> children;
        };
        Node * root;
};
```

10.1 Terminology

- **Root**: the top node of the tree.
- **Leaf**: a node with no children.
- **Internal node**: a node with at least one child.
- **Path**: a sequence of nodes where each node is connected to the next node.
- **Height**: the length of the longest path from the root to a leaf.
- **Depth**: the length of the path from the root to a node.
- **Level**: the set of all nodes at the same depth.
- **Subtree**: a tree that is a subset of another tree.
- **Ancestor**: a node on the path from the root to a node.

- **Descendant**: a node that is on the path from a node to a leaf.
- **Siblings**: nodes that share the same parent.
- **Degree**: the number of children of a node.
- **Binary tree**: a tree where each node has at most two children.
  - A **perfect binary tree** is a binary tree where all leaves are at the same level and all internal nodes have two children. So there will be $2^{h+1}-1$ nodes and 2^h leaves where h is the height of the tree.
  - A **complete binary tree** is a binary tree where all levels are completely filled except for the last level, which is filled from left to right.
  - A **full binary tree** is a binary tree where all nodes have either 0 or 2 children.
- A tree is a **connected graph** with no cycles.
  - There is a unique path between any two nodes.
  - All nodes are connected to the root through a path.
  - A tree

10.2 Tree traversals

10.2.1 Preorder

```
void Preorder(Node * current){
    if(current == NULL){
        return;
    }
    cout << current->value << endl;
    for(int i=0; i<current->children.size(); i++){
        Preorder(current->children[i]);
    }
}
```

10.2.2 Postorder

```
void Postorder(Node * current){
    if(current == NULL){
        return;
    }
    for(int i=0; i<current->children.size(); i++){
        Postorder(current->children[i]);
    }
    cout << current->value << endl;
}
```

10.2.3 Inorder (only for binary tree)

```
void Inorder(Node * current){
    if(current == NULL){
        return;
    }
    Inorder(current->left);
    cout << current->value << endl;
    Inorder(current->right);
}
```

10.2.4 Find the height of a tree

```
int Height(Node * current){
    if(current == NULL){
        return -1;
    }
    int max = -1;
    for(int i=0; i<current->children.size(); i++){
        int h = Height(current->children[i]);
        if(h > max){
            max = h;
        }
    }
    return max+1;
```

}

10.3 Space complexity

Other than the data stored in the nodes, we need more pointers to keep track of the tree. Hence, there is some space complexity. Some pointers points to another node, some points to NULL.

Theorem: A finite tree of n nodes has exactly $n + 1$ null pointers.

Proof. Base case: a tree with only one node has exactly one null pointer.

Inductive hypothesis: assume that a tree with k nodes has exactly $k + 1$ null pointers.

Inductive step: add a node to the tree. The new node has exactly $k + 1$ null pointers. The new tree has exactly $k + 2$ null pointers.

Termination: After adding all n nodes, the tree has exactly $n+1$ null pointers. □

In a binary tree with n values:

- there are exactly n nodes.
- exactly 1 of those nodes has no parent
- every other node has exactly 1 parent
- every non-null pointer points from **a parent** to **a child**
- so there are exactly $n - 1$ non-null pointers
- The total number of pointers in a tree is exactly 2n since each node holds two pointers
- So the number of null pointers in a **binary tree** is exactly $2n - (n - 1) = n + 1$

10.3.1 General tree

Think about a k-ary tree. Each node has k children. So the number of null pointers is $kn - (n - 1) = kn - n + 1$. So the number of null pointers is $n(k - 1) + 1$.

10.4 Binary Search Tree

A binary search tree is a binary tree where each node has a key and a value.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree must also be binary search trees.
- Binary search trees are fully **ordered**.

10.4.1 Search

```
Node * Search(Node * current, T key){
    if(current == NULL){
        return NULL;
    }
    if(current->key == key){
        return current;
    }
    if(key < current->key){
        return Search(current->left, key);
    }else{
        return Search(current->right, key);
    }
}
```

Worst-case run time: height of the tree + 1. So the worst case is $O(\log n)$.

10.4.2 Insertion

The BST property must be maintained! The cost is about the same as the cost for the search algorithm.

```
Node * Insert(Node * current, T key, U value){
    if(current == NULL){
        return new Node(key, value);
    }
```

```
    if(key < current->key){
        current->left = Insert(current->left, key, value);
    }else{
        current->right = Insert(current->right, key, value);
    }
    return current;
}
```

Note the base case returns a new node when the current node is NULL. This is because we need to create a new node when we reach the end of the tree. And we updates the left or right pointer of the parent node to the new node.

10.4.3 Find the minimum and maximum

```
Node * FindMin(Node * current){
    if(current == NULL){
        return NULL;
    }
    if(current->left == NULL){
        return current;
    }
    return FindMin(current->left);
}
Node * FindMax(Node * current){
    if(current == NULL){
        return NULL;
    }
    if(current->right == NULL){
        return current;
    }
    return FindMax(current->right);
}
```

Note that the left-most node is the minimum and the right-most node is the maximum. It could be that the left-most node still contains a right child, but that child will be larger than the left-most node. Similarly for the right-most node.

10.4.4 Deletion

After removal, the BST property must be maintained! The cost is about the same as the cost for the search algorithm.

$$\begin{cases} \text{Case 1:The node has no children.} & \Longrightarrow \text{ Remove directly, assign null to parents} \\ \text{Case 2:The node has one child.} & \Longrightarrow \text{ Replace the node with its subtree} \\ \text{Case 3:The node has two children.} & \Longrightarrow \text{ Need a lot more care} \end{cases}$$

When a node has two children, we find its **in-order predecessor**:

- The in-order predecessor is the maximum node in the left subtree. (Right most node in the left subtree)
- It is the largest node that is smaller than the node to be deleted.
- The predecessor cannot have a right child because it is the maximum node in the left subtree.

At the same time, we can find the **in-order successor**:

- The in-order successor is the minimum node in the right subtree. (Left most node in the right subtree)
- It is the smallest node that is larger than the node to be deleted.
- The successor cannot have a left child because it is the minimum node in the right subtree.

We can choose either the predecessor or the successor to replace the node to be deleted. It is a design choice. Just be consistent.

```
Node * Delete(Node * current, T key){
    if(current == NULL){
        return NULL;
    }
    if(key < current->key){
        current->left = Delete(current->left, key);
    }else if(key > current->key){
        current->right = Delete(current->right, key);
    }else{
        if(current->left == NULL
        && current->right == NULL){
```

```
            delete current;
            return NULL;
        }
        if(current->left == NULL){
            Node * temp = current;
            current = current->right;
            delete temp;
        }else if(current->right == NULL){
            Node * temp = current;
            current = current->left;
            delete temp;
        }else{
            Node * temp = FindMax(current->left);
            current->key = temp->key;
            current->value = temp->value;
            current->left = Delete(current->left,
            temp->key);
        }
    }
    return current;
}
```

The above code finds the predecessor to replace the node to be deleted.

10.4.5 BST efficiency

The efficiency of operations depends on the height of the tree. If a tree is complete, then the height is $\lfloor \log_2 n \rfloor$.

- Search, insertion, and deletion are all $O(\log n)$. Where n is the number of nodes in the tree since it just needs to traverse the height of the tree.

10.4.6 Balanced BST

Note that if a BST is not balanced, the worst case, it is just a sorted linked list with $O(n)$ height and a bunch of wasted NULL pointers. So we need to balance the BST to make it efficient.

A binary tree is balanced if:

- The left and right subtrees of every node differ in height by at most 1.

- Leaves are all **about** the same depth.
- The exact specification varies
- Sometimes trees are balanced by comparing the height of nodes: The height of a node's left subtree and the height of a node's right subtree differ by at most 1. (AVL tree)
- Sometimes trees are balanced by comparing the number of nodes: The number of nodes in a node's left subtree and the number of nodes in a node's right subtree differ by at most 1. (Red-black tree)

It would be ideal if a BST was always close to complete.

10.5 AVL tree

An AVL tree is a balanced BST

- The height of the left and right subtrees of every node differ by at most 1.
- Rebalancing is done by **rotations** when an insertion or deletion causes the tree to become unbalanced.
- AVL tree nodes contain an extra field to store the height of the node.

10.5.1 AVL tree height

The height of an AVL tree is $O(\log n)$ where n is the number of nodes in the tree.

Theorem 1. *The height of an AVL tree with n nodes is $O(\log n)$.*

Proof. Let N_h represent the minimum number of nodes in an AVL tree of height h.

Since AVL trees are balanced, the minimum number of nodes in an AVL tree of height h is the sum of the minimum number of nodes in the left and right

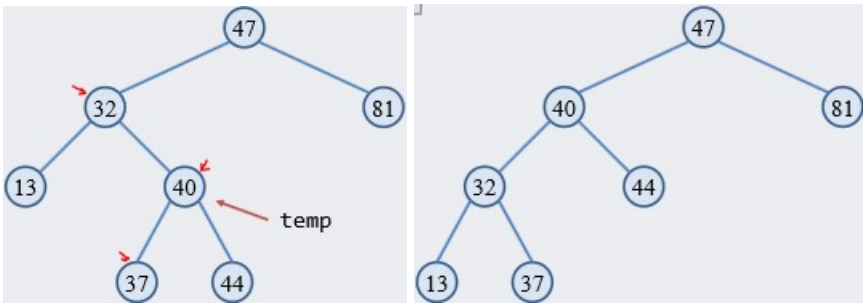
subtrees plus 1. So $N_h = N_{h-1} + N_{h-2} + 1$.

$$
\begin{aligned}
N_h &= N_{h-1} + N_{h-2} + 1 \\
&= 1 + (1 + N_{h-2} + N_{h-3}) + N_{h-2} + 1 \\
&= 2 + 2N_{h-2} + N_{h-3} \\
&> 2N_{h-2} + N_{h-3} \\
N_h &> 2N_{h-2} \\
&> 2(2N_{h-4}) \\
&> 2^2 N_{h-4} \\
&> 2^3 N_{h-6} \\
&> \cdots \\
&> 2^{\frac{h}{2}} N_0 \\
&> 2^{\frac{h}{2}} \\
\log N_h &> \frac{h}{2} \\
h &< 2 \log N_h \\
h &\in O(\log n)
\end{aligned}
$$

□

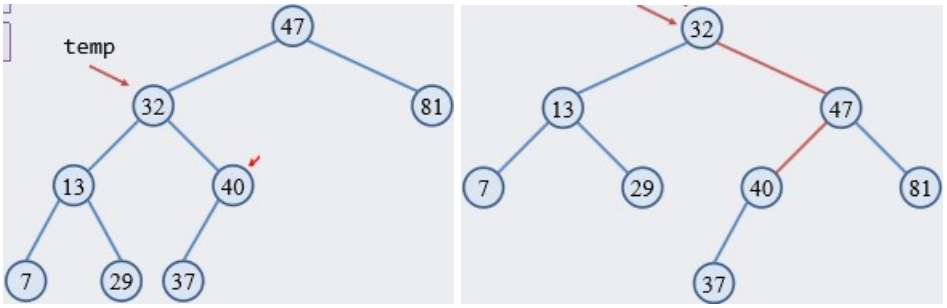
10.5.2 Rotation

- Left rotation: For example, I try to rotate node 32.



  Become the left child of the current right child and adopt the left child of the current right child as the right child.

- Right rotation: For example, I try to rotate node 47.



Become the right child of the current left child and adopt the right child of the current left child as the left child.

A sample implementation of rotations:

```
AVLNode * LeftRotate(AVLNode * current){
    AVLNode * new_root = current->right;
    current->right = new_root->left;
    new_root->left = current;
    current->height = 1 + max(Height(current->left),
    Height(current->right));
    new_root->height = 1 + max(Height(new_root->left),
    Height(new_root->right));
    return new_root;
}
AVLNode * RightRotate(AVLNode * current){
    AVLNode * new_root = current->left;
    current->left = new_root->right;
    new_root->right = current;
    current->height = 1 + max(Height(current->left),
    Height(current->right));
    new_root->height = 1 + max(Height(new_root->left),
    Height(new_root->right));
    return new_root;
}
```

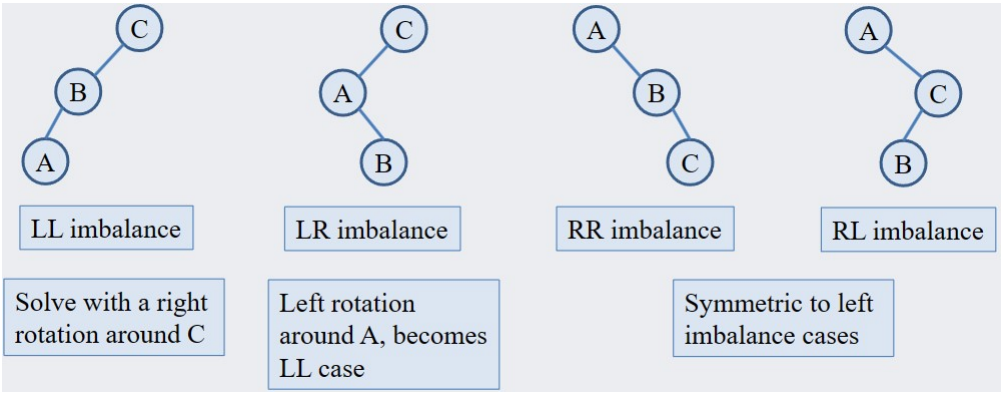
10.5.3 AVL nodes

AVL Nodes are almost the same as a binary tree node. The only difference is that it contains an extra field to store the height info of the node. There are lots of ways to store the height info.

```
enum BalanceFactor {LEFT_HEAVY=-1,
 BALANCED=0, RIGHT_HEAVY=1};
template <typename T>
class AVLNode {
     public:
          T value;
          AVLNode * left;
          AVLNode * right;
          int height;
          BalanceFactor balance;
};
AVLNode (T value, AVLNode * left = NULL,
AVLNode * right = NULL){
     this->value = value;
     this->left = left;
     this->right = right;
     this->height = 0;
     this->balance = BALANCED;
}
```

There are four types of imbalance in an AVL tree.

- Left-Left (LL): the left subtree of the left child is taller than the right subtree of the left child.
- Left-Right (LR): the right subtree of the left child is taller than the left subtree of the left child.
- Right-Left (RL): the left subtree of the right child is taller than the right subtree of the right child.
- Right-Right (RR): the right subtree of the right child is taller than the left subtree of the right child.



10.5.4 Insertion

For each insertion, we need to check the balance of the tree. If the tree is unbalanced, we need to do a rotation to balance the tree.

if root is NULL **then**
    Create new node containing the value, assign it to root and return true.
else
    if value is equal to `root->value` **then**
        value exists already Return false.
if value is less than `root->value` **then**
    recursively insert value into left subtree.
    if left subtree height is greater than right subtree height by 2 **then**
        Perform a right rotation.
else
    recursively insert value into right subtree.
    if right subtree height is greater than left subtree height by 2 **then**
        Perform a left rotation.

```
AVLNode * Insert(AVLNode * current, T key){
if(current == NULL){
    return new AVLNode(key);
}
if (key == current->value){
    throw "Duplicate key";
}
if(key < current->key){
    current->left = Insert(current->left, key);
```

```
}else{
    current->right = Insert(current->right, key);
}
current->height = 1 + max(Height(current->left),
Height(current->right));
current->balance = Balance(current);
if(current->balance == LEFT_HEAVY){
    if(current->left->balance == LEFT_HEAVY){
        return RightRotate(current);
    }else{
        current->left = LeftRotate(current->left);
        return RightRotate(current);
    }
}else if(current->balance == RIGHT_HEAVY){
    if(current->right->balance == RIGHT_HEAVY){
        return LeftRotate(current);
    }else{
        current->right = RightRotate(current->right);
        return LeftRotate(current);
    }
}
return current;
}
```

For each insertion, we need to perform a BST insertion $O(\log n)$ and then check the balance of the tree. If the tree is unbalanced, we need to do a rotation to balance the tree. The rotation is $O(1)$ since at most 2 rotations are needed. So the overall running time is $O(\log n)$.

10.5.5 Deletion

Same idea as insertion.

- begin with a standard BST deletion (using the in-order predecessor or successor)
- local root balance is adjusted for removing a node
- if the local root becomes unbalanced, a rotation is performed

Complexity is $O(\log n)$ since we need to perform a BST deletion $O(\log n)$ and then check the balance of the tree. If the tree is unbalanced, we need to do a rotation to balance the tree. The rotation is $O(1)$ since at most 2 rotations are needed. So the overall running time is $O(\log n)$.

11 B-trees

Observations:

- AVL trees are great for small data sets.
- AVL balance invariant guarantees worst case $O(\log n)$ find, insert, delete.
- But nodes are allocated one at a time in dynamic memory. No guarantee that parents/children are close in memory. Thus, cache misses are likely.
- Realities: For large data sets, **disk accesses** dominate running time.

11.1 Disk access

Suppose we have a very large data set stored in a BST that it does not fit in RAM. Thus part of the tree must be on disk.

Different level of memory are accessed in blocks

- BST nodes may all reside in different pages on disk.
- A tree operation may require many disk accesses.

Goal: put more relevant data together in the same block.

- Increase the number of keys in each node.
- Minimize the number of disk accesses.

11.2 B-trees nodes

A B-tree node defines an m-ary tree

- Each node has up to $m - 1$ keys and up to m children.
- Nodes are still fully ordered.

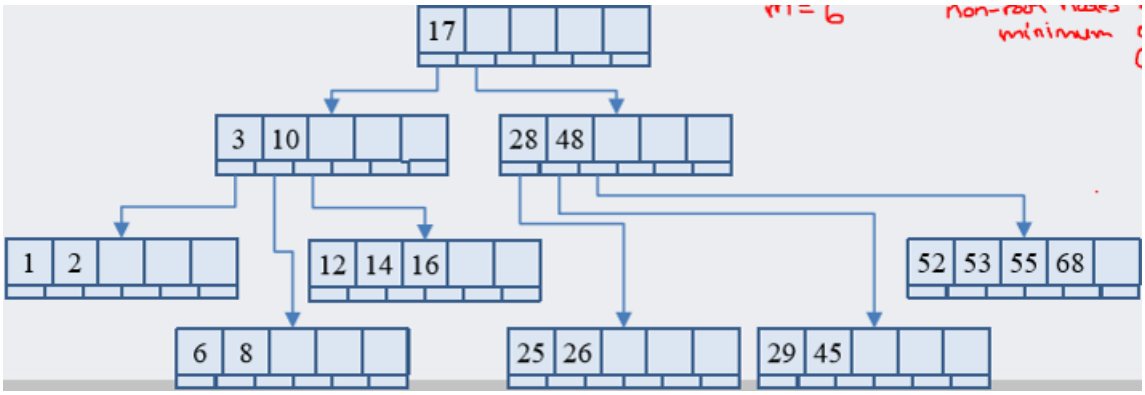
- Additional pointer: parent, left sibling, right sibling.

Ideally, maximize the size of a B-tree node to fill a disk block. In practice, branching factors are greater than 10000 usually.

11.3 B-tree properties

B-tree of order m is an ordered m-ary tree

- Internal nodes: d keys and $d+1$ children where $m/2 \leq d \leq m-1$.
- All leaves are at the same level.
- All leaves hold no more than $m-1$ keys.
- All non-root internal nodes have at least $m/2$ children.
- The root has at least 2 children unless it is a leaf.



For a tree of height h, what is the minimum number of nodes?

- root has 1 key and at least 2 children.
- each internal node has at least $\lceil m/2 \rceil$ children. ($\lceil m/2 \rceil - 1$ keys)
- Let $t = \lceil m/2 \rceil$.
  1. level 0 (root): 1 node two children.
  2. level 1: $2 \times t^0$ nodes with t children each and $t-1$ keys each.
  3. level 2: $2 \times t^1$ nodes with a total of $2 \times t^1 \times t$ children and $2 \times t^1 \times (t-1)$ keys.

4. level h: $2 \times t^{h-1}$ nodes with a total of $2 \times t^{h-1} \times t$ children and $2 \times t^{h-1} \times (t-1)$ keys.

- The total number of nodes is $1 + 2\sum_{i=0}^{h-1} t^i$.

11.4 B-tree operations

11.4.1 Search

Start at the root.
Find the smallest key greater than or equal to the search key.
If the key is found, return the key.
If the key is not found, follow the child pointer to the next node.
Repeat until the key is found or the leaf is reached.

```
T Search(BNode * current, T key){
    while(current != NULL){
        int i = 0;
        while(i < current->keys.size()
        && key > current->keys[i]){
            // Linear search within a node
            i++;
        }
        if(i < current->keys.size()
        && key == current->keys[i]){
            return current->keys[i];
        }
        // Find the child on RAM or disk.
        current = current->children[i];
    }
    return NULL;
}
```

The linear search within a node makes use of spatial locality.

- For each node visit, we need to perform a linear search $O(m)$.
- Worst case we need to search $O(\text{height})$ nodes.
- Then the running time is $O(m \times \text{height})$.

- The height of a B-tree is $O(\log n)$.
- So the running time is $O(m \log n)$.
- m is usually a constant, so the running time is $O(\log n)$.

11.4.2 Insertion

List BST, we first search for the insertion location. Then we insert the key into the leaf node. If the leaf node is full, we need to split the node.

```
void Insert(BNode * current, T key){
    BNode * leaf = FindLeaf(current, key);
    leaf->keys.push_back(key);
    sort(leaf->keys.begin(), leaf->keys.end());
    if(leaf->keys.size() == m){
        Split(leaf);
    }
}
```

We will split the node like this:

- The median key is moved up to the parent.

11.4.3 Remove

- Find node containning the key.
- If the key is in a leaf node, remove the key.
- If the key is in an internal node, replace the key with the predecessor or successor.
- If the leaf node is less than half full, borrow a key from a sibling or merge with a sibling.

12 Hashing

Use arrays to store data. The key is used to calculate the index of the array.

Fix the array size based on the amount of data to be stored.

- map the key to an index in the array.
- store the value at that index.
- Need a good hash function to map keys to indices.

12.1 Hash table

A hash table consists of an array to store data

- Data often consists of complex types or pointers to such objects.
- One attribute of the object is designated as the table's key.

A hash function maps a key to an array index in 2 steps

- The key should be converted to an integer.
- And then that integer is mapped to an array index using some function (often the modulo function).

12.1.1 Collosions

A hash function may map two different keys to the same index(**collision**).

- A good hash function minimizes collisions.
- Collisions can be resolved by chaining or open addressing.
- Necessary to have a policy for resolving collisions since collisions are not avoidable due to the pigeonhole principle.

12.1.2 Hash function

Hash functions should have the following properties:

- Fast to compute.
- Deterministic: no randomness.
- Uniform distribution: keys are mapped to indices uniformly.

12.2 General principle

- Use the entire search key in the hash function.
- If the has function uses modulo m, then m should be a prime number.
- A simple and usually effective function is
  - Convert the key to an integer.
  - Use the modulo function to map the integer to an index.
  - $h(k) = k \mod m$ where m is the size of the array.
  - m should be a prime number.

12.3 Hash Table Efficiency

When analyzing the efficiency of hashing, it is necessary to consider *load factor*, λ

- $\lambda = \frac{n}{m}$ where n is the number of keys and m is the size of the array.
- As the table fills up, the number of collisions increases, λ increases.
- The table size should be selected so that λ is does not exceed a certain value like 0.5.

12.4 Collision handling

12.4.1 Open addressing

Idea: when a collision occurs, find another open slot in the array.

- Linear probing: search for the next open slot.
  - The hash table is searched sequentially starting from the original hash value.
  - The search wraps around to the beginning of the table if necessary.
  - Linear probing leads to primary clustering. Gathering of keys in the same area. Reducing the efficiency of the hash table.
  - Searching goes through the same sequence of slots.

- Quadratic probing: search for the next open slot using a quadratic function.
- Double hashing: use a second hash function to find the next open slot.
  - Double hashing produces *key dependent* probing sequences. Call it h_2 and h_1 is the original hash function. Then the next slot is $h_1(key) + i \times h_2(key)$, where i is the number of probes.
  - $h_2(key) \neq 0, h_2(key) \neq h_1(key)$. A typical h_2 is $p - (key \mod p)$ where p is a prime number less than the size of the array.

Removal is difficult in open addressing. We need to mark the slot as deleted and then reinsert the key since if we just remove the key, the empty slot will break the probing sequence.

We put a tombstone in the slot to mark it as deleted which helps to maintain the probing sequence.

However, it will add extra overhead after a large amount of insertions and deletions.

So it makes sense to rehash the table after a large amount of insertions and deletions.

12.4.2 Separate Chaining

Each entry in the hash table is a pointer to a linked list.

- When a collision occurs, the new key is added to the linked list
- The linked list is searched for the key.
- Performance degrades less rapidly as the load factor increases compared to open addressing.
- Note that it adds extra overhead for the pointers. And the worst case is $O(n)$.

13 Heaps

A heap is a binary tree with the following properties:

- It is a complete binary tree.

  - All levels are filled except possibly the last level.
  - The last level is filled from left to right.
- It is partially ordered:
  - For a max heap, the value of each node is greater than or equal to the value of its children.
  - For a min heap, the value of each node is less than or equal to the value of its children.

It is totally possible that two binary heaps have the same values but different structures.

13.1 Implementation

A heap can be implemented using an array.

- The root is at index 1.
- The left child of a node at index i is at index $2i + 1$.
- The right child of a node at index i is at index $2i + 2$.
- The parent of a node at index i is at index $i/2$.

```
template <class LIT>
class MinHeap{
    private:
        int size;
        int capacity;
        LIT * arr;
    public:
        ...
};

template <class LIT>
MinHeap<LIT>::MinHeap(int capacity){
    this->capacity = capacity;
    this->size = 0;
    this->arr = new LIT[capacity];
}
```

13.1.1 Insertion

To maintain the heap property, we need to perform a **percolate up** operation.

- Insert the new value at the end of the array.
- Compare the new value with its parent.
- If the new value is less than its parent, swap the new value with its parent.
- Repeat until the new value is greater than or equal to its parent.

Finding the last index is $O(1)$ and percolating up is $O(\log n)$. So the overall running time is $O(\log n)$.

13.1.2 Deletion

To maintain the heap property, we need to perform a **percolate down** operation.

- Replace the root with the last value in the array.
- Compare the new root with its children.
- If the new root is greater than its children, swap the new root with the smaller child. (This is for a min heap, for a max heap, swap with the larger child)
- Repeat until the new root is less than or equal to its children. (This is for a min heap, for a max heap, repeat until the new root is greater than or equal to its children)

Finding the last index is $O(1)$ and percolating down is $O(\log n)$. So the overall running time is $O(\log n)$.

13.1.3 Resize

When the array is full, we need to resize the array. The resizing operation is $O(n)$. Very similar to the resizing operation in a dynamic array.

13.2 Build a heap

To create a heap from an unordered array, repeatedly call **heapifyDown**

- Any subtree in a heap is itself a heap.
- Call heapifyDown on elements in the upper $\frac{1}{2}$ of the array. Since the lower half are leaf nodes which are already heaps.
- Start from index $\frac{n}{2}$ and go down to 0. (From the last non-leaf node to the root)

```
template <class LIT>
void MinHeap<LIT>::BuildHeap(){
    for(int i=size/2; i>=0; i--){
        PercolateDown(i);
    }
}
```

13.2.1 Complexity of build heap

- The cost for heapifyDown is $O(\log n)$ which is the height of the tree. We need to perform heapifyDown for each non-leaf node which is $\frac{n}{2}$.
- It would look like $O(n \log n)$ but it is actually $O(n)$.
- It is because heapifyDown must still follow a single path down to the bottom level.
- There are exactly a maximum of $n-1$ edges that can perform heapifyDown since each node can only perform heapifyDown once.

13.3 Heap sort

- Build a heap from the array.
- Repeatedly remove the root from the heap and insert it into the end of the array.
- The array is now sorted.

```
template <class LIT>
void MinHeap<LIT>::HeapSort(){
    for(int i=size-1; i>0; i--){
        swap(arr[0], arr[i]);
        size--;
        PercolateDown(0);
    }
}
```

Algorithm	Best	Average	Worst	Space
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n) + O(\log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

Note that Merge Sort is recursive and requires additional space on the call stack. Heap Sort is iterative and does not require additional space on the call stack.

14 Graphs

A graph is a collection of nodes and edges.

- Nodes are also called vertices.
- Edges connect nodes.
- Edges can be directed or undirected.

  - In a directed graph, edges have a direction. (u, v) is different from (v, u).
  - In an undirected graph, edges do not have a direction. (u, v) is the same as (v, u).
- Edges can have weights.

14.1 Terminology

- Vertices adjacent to $v : N(v) = \{u|(u, v) \in E\}$.
- Edges incident to $v : I(v) = \{(u, v)|u \in E\}$.
- Degree of $v : \deg(v) = |I(v)|$.
  - Handshaking theorem: $\sum_{v \in V} \deg(v) = 2|E|$ if $G = (V, E)$ is an undirected graph. Basically saying that if you count the number of edges each vertex has, you count each edge twice.
  - The *in-degree* of a vertex $v \in V$ is denoted as $\deg^-(v)$ which means the number of edges entering the vertex.
  - The *out-degree* of a vertex $v \in V$ is denoted as $\deg^+(v)$ which means the number of edges leaving the vertex.
  - For a directed graph, the handshaking theorem is $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$.
- Path: a sequence of vertices where each vertex is adjacent to the next connected by edges.
- Simple path: a path where no vertex is repeated.
- Cycle: a path that starts and ends at the same vertex.
- Simple graph: a graph with no self-loops or multiple edges between the same pair of vertices.
- Subgraph: a graph where the vertices and edges are a subset of the original graph. $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
  - We often care about the structure of graphs not the names of its vertices:

  - **Isomorphic graphs** have the same structure but different names for the vertices. Check by trying to match the vertices via a bijection.
  - **Subgraph isomorphism** is the problem of finding a subgraph of a graph that is isomorphic to another graph.

- Complete graph: a graph where every pair of vertices is connected by an edge. So maximum number of edges.

- Connected graph: a graph where there is a path between every pair of vertices.

  - undirected graphs are connected if there is a path between every pair of vertices.
  - directed graphs are strongly connected if there is a directed path between every pair of vertices.
  - directed graphs are weakly connected if the underlying undirected graph is connected.

- Connected component: a maximal connected subgraph.

- Acyclic graph: a graph with no cycles.

- Spanning tree: a subgraph that is a tree and contains all the vertices of the original graph. It is a **connected acyclic** subgraph.

- Weighted graph: a graph where each edge has a weight.

- Graph density:

  - A **sparse** graph has few edges compared to the number of vertices. Typically has $O(|V|)$ edges.
  - A **dense** graph has many edges compared to the number of vertices. Typically has $O(|V|^2)$ edges.
  - Anything in between is considered **moderate**, depending critically on context!
  - Analysis of graph operations typically must be expressed in terms of both $|V|$ and $|E|$.

14.2 Graph representation

14.2.1 Analysis Critiria

Running times are often reported in terms of n (the number of vertices) and m (the number of edges).

- Number of edges:
  - For connected graph, $m \geq n - 1$.
  - For not connected graph, $m \geq 0$.
  - For simple graph, $m \leq n(n-1)/2$. (Complete graph with no repeated edges)
  - For not simple graph, not bounded.
  - $\sum_{v \in V} \deg(v) = 2m \in O(m)$.

14.2.2 Adjacency matrix

- A matrix A where $A_{ij} = 1$ if there is an edge between i and j.
- For a weighted graph, A_{ij} is the weight of the edge.
- For an undirected graph, A is symmetric.
- For a directed graph, A is not symmetric.
- The space complexity is $O(n^2)$.
- The time complexity to check if there is an edge between i and j is $O(1)$.
- The time complexity to find the neighbors of a vertex is $O(n)$.

14.2.3 Adjacency list

- An array of linked lists where each element in the array is a vertex and the linked list contains the neighbors of the vertex.
- For a weighted graph, the linked list contains the weight of the edge.
- The space complexity is $O(n + m)$.
- The time complexity to check if there is an edge between i and j is $O(\deg(i))$.
- The time complexity to find the neighbors of a vertex is $O(\deg(i))$.

14.2.4 Edge list

- A list of edges where each edge is a tuple (u, v).
- For a weighted graph, the edge is a tuple (u, v, w) where w is the weight of the edge.
- The space complexity is $O(m)$.
- The time complexity to check if there is an edge between i and j is $O(m)$.
- The time complexity to find the neighbors of a vertex is $O(m)$.

14.2.5 Asymptotic performance

n vertices, m edges no parallel edges no self-loops bounds are big-Θ	Edge list	Adjacency list	Adjacency matrix
Space	$n + m$	$n + m$	n^2
`incidentEdges(v)`	m	$\deg(v)$	n
`areAdjacent(v, w)`	m	$\min(\deg(v), \deg(w))$	1
`insertVertex(o)`	1	1	n (amortized)
`insertEdge(v, w, o)`	1	1	1
`removeVertex(v)`	m	$\deg(v)$	n (amortized)
`removeEdge(e)`	1	1	1

14.3 BFS

Breadth-first search is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

It looks like a level order traversal in a tree, we use a queue to store the nodes.

Algorithm BFS(G,v)
Input: A graph G and a vertex v of G to start.

Output: The vertices of G reachable from v labeled as discovered.
Queue q
SetLabel(v,"discovered")
q.enqueue(v)
while q is not empty **do**
    v = q.dequeue()
    for each edge (v, w) in G **do**
        if Vertex w is not labeled **then**
            SetLabel(w,"discovered")
            q.enqueue(w)

14.3.1 Complexity

- There are one while loop to run for each vertex, so the time complexity is $O(n)$.
- For each while loop, there is a for loop to run for each vertex's degree, so the time complexity is $O(m)$.
- So the overall time complexity is $O(n + m)$.

14.4 DFS

Depth-first search is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores as far as possible along each branch before backtracking.

It looks like a pre-order traversal in a tree, we use a stack to store the nodes.

Since programs run on a stack, we can use the call stack to store the nodes.

Algorithm DFS(G,v)
Input: A graph G and a vertex v of G to start.
Output: The vertices of G reachable from v labeled as discovered.
SetLabel(v,"discovered")
for each edge (v, w) in G **do**
    if Vertex w is not labeled **then**
        DFS(G, w)
    else
        SetEdge((v, w),"Back")

14.4.1 Complexity

- There are one for loop to run for each vertex, so the time complexity is $O(n)$.
- For each for loop, there is a recursive call to run for each vertex's degree, so the time complexity is $O(m)$.
- So the overall time complexity is $O(n + m)$.

However, if we use Adjacency matrix, the time complexity is $O(n^2)$ since we need to check all the vertices for each vertex.

14.5 Spanning tree

A spanning tree of a graph G is a subgraph that is a tree and contains all the vertices of G. It is a **connected acyclic** subgraph.

Minimum spanning tree is a spanning tree with the smallest sum of edge weights.

14.5.1 Kruskal's algorithm

Builds a spanning tree from several *connected components*, repeatedly chooses the *lightest* edge that connects two components, which **does not** form a cycle until edge set has $|V| - 1$ edges.

Algorithm Kruskal(G)
Input: A graph G
Output: A minimum spanning tree of G
set $E' = \emptyset$
while E' has less than $|V| - 1$ edges **do**
    Choose the lightest edge (u, v) from G
    if (u, v) does not form a cycle with E' **then**
        $E' = E' \cup \{(u, v)\}$

Find minimum weight edge: Priority queue.
Find cycle: Disjoint set.
Complexity:

- For heap based approach:
  - Build: $O(|E| \log |E|)$

See his lecture here!!!!!!!!!!!!

14.5.2 Prim's algorithm

- Based on Partition Property in graphs
- Builds a spanning tree from initially one vertex
- Repeatedly chooses the *lightest* edge that connects the tree to a new vertex until the tree contains all vertices.

Algorithm Prim(G)
Input: A graph G
Output: A minimum spanning tree of G
set $V' = \emptyset$
set $E' = \emptyset$
Choose a vertex v from G
$V' = V' \cup \{v\}$
while V' has less than $|V|$ vertices **do**
    Choose the lightest edge (u, v) from G where $u \in V'$ and $v \notin V'$
    $V' = V' \cup \{v\}$
    $E' = E' \cup \{(u, v)\}$

Complexity: $O(|E| \log |V|)$ with a priority queue.

14.6 Dijkstra's algorithm

Traversal using priority queue to find the shortest path from a source vertex to all other vertices in a weighted graph.

Note that if it is unweighted, we can use BFS to find the shortest path.

If negative weights are allowed, Dijkstra's algorithm might produce correct results but it is not guaranteed.

14.6.1 Algorithm

Algorithm Dijkstra(G,s)
Input: A graph G and a source vertex s
Output: The shortest path from s to all other vertices
set $dist$ to ∞ for all vertices
set $dist[s]$ to 0

set pq to a priority queue
pq.enqueue(s)
while pq is not empty **do**
    v = pq.dequeue()
    for each edge (v, w) in G **do**
        if $dist[v] + weight(v, w) < dist[w]$ **then**
            $dist[w] = dist[v] + weight(v, w)$
            pq.enqueue(w)

If we want to also collect the path, we can use a parent array to store the parent of each vertex.

Each of $|E|(m)$ edges has to be processed once

- Looking up (and changing) the current cost of a vertex in a heap takes $O(|V| = n)$ for an unindexed heap
- Cost: $O((|V| + |E|) \log |V|)$