CS3334 Review

Linked List

Singly Linked List

```
struct Node {
    int data;
    Node* next;
    Node(int data, Node* next) : data(data), next(next) {}
    Node(int data) : data(data), next(nullptr) {}
};
class LinkedList {
    Node* head;
public:
    LinkedList() : head(nullptr) {}
    void insert(int);
    void remove(int);
    int size();
    void print();
    Node* search(int);
};
```

Maintain a ascending order linked list.

```
void insert(int data) {
    // if list is empty
    if (head == nullptr) {
        head = new Node(data);
        return;
    }
    // if data is smaller than head
    if (data < head->data) {
        head = new Node(data, head);
        return;
    }
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        // if data is smaller than cur->next, or reach the end
        if (cur->next == nullptr || data < cur->next->data) {
            cur->next = new Node(data, cur->next);
            return;
        }
    }
}

void remove(int data) {
    if (head == nullptr) return;
    if (head->data == data) {
        Node* tmp = head;
        head = head->next;
        delete tmp;
```

```
        return;
    }
    for (Node* cur = head; cur->next != nullptr; cur = cur->next) {
        if (cur->next->data == data) {
            Node* tmp = cur->next;
            cur->next = cur->next->next;
            delete tmp;
            return;
        }
    }
}

int size() {
    int cnt = 0;
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        cnt++;
    }
    return cnt;
}

void print() {
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        cout << cur->data << " ";
    }
    cout << endl;
}

Node* search(int data) {
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        if (cur->data == data) return cur;
    }
    return nullptr;
}
```

Doubly Linked List

```
struct DNode {
    int data;
    DNode* prev;
    DNode* next;
    DNode(int data, DNode* prev, DNode* next) : data(data), prev(prev),
next(next) {}
    DNode(int data) : data(data), prev(nullptr), next(nullptr) {}
};
```

Swap a node with its next. Assume the node is not the last node.

```
void swapWithNext(DNode* node) {
    DNode* pre = node->prev;
    DNode* nxt = node->next;

    node->next = nxt->next;
    node->prev = nxt;
    next->next = node;
    next->prev = pre;
}
```

Remove a node.

```
void remove(DNode* node) {
    if (node->prev != nullptr) node->prev->next = node->next;
    if (node->next != nullptr) node->next->prev = node->prev;
    delete node;
}
```

Exercises

Exercise 1. Swap every pair of adjacent nodes in a linked list. If there is a last odd node, leave it in place.

Example: Input `1->2->3->4->5`, Output `2->1->4->3->5`.

Exercise 2. Find the middle node of a linked list. If there are even number of nodes, return the second middle node.

Example: Input `1->2->3->4->5`, Output `3`. Input `1->2->3->4->5->6`, Output `4`.

Exercise 3. Given a linked list, reverse it.

Example: Input `1->2->3->4->5`, Output `5->4->3->2->1`.

Exercise 4. Given a linked list, reverse from range `m` to `n`. Nodes are 1-indexed. Assume the range is valid.

Example: Input `1->2->3->4->5`, `m = 2`, `n = 4`, Output `1->4->3->2->5`.

Solution:

1.

```
void swapPairs(Node* head) {
    Node* cur = head;
    Node* pre = nullptr;
    while (cur != nullptr && cur->next != nullptr) {
        Node* nxt = cur->next;
        cur->next = nxt->next;
        nxt->next = cur;
        if (pre != nullptr) pre->next = nxt;
        else head = nxt; // update head to originally 2nd node
        pre = cur;
        cur = cur->next;
    }
}
```

2.

```
void findMiddle(Node* head) {
    Node* slow = head;
    Node* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

Explanation: `fast` moves twice as fast as `slow`, so when `fast` reaches the end, `slow` is at the middle.

More specifically, if there are $2n+1$ nodes, `fast` will stop at $2n+1$ and `slow` will stop at $n+1$. If there are $2n$ nodes, `fast` will stop at $2n-1$ and `slow` will stop at n.

3.

```
void reverse(Node* head) {
    Node* pre = nullptr;
    Node* cur = head;
    Node* nxt = nullptr;
    while (cur != nullptr) {
        nxt = cur->next;
        // do not update nxt->next here
        cur->next = pre;
        pre = cur;
        cur = nxt;
    }
    head = pre;
}
```

4.

```
void rangeReverse(Node* head, int m, int n) {
    Node* pre = nullptr;
    Node* cur = head;
    Node* nxt = nullptr;
    for (int i = 1; i < m; i++) {
        pre = cur;
        cur = cur->next;
    } // pre = [m-1], cur = [m]
    Node* preRange = pre;
    Node* rangeEnd = cur; // will be positioned at [n] after loop
    for (int i = m; i <= n; i++) {
        nxt = cur->next;
        cur->next = pre;
        pre = cur;
        cur = nxt;
    }
    // 1...m-1 n...m n+1...end
    // m-1=preRange m=rangeEnd n=pre n+1=cur
```

```
    if (preRange != nullptr) preRange->next = pre;
    else head = pre; // range involves head
    rangeEnd->next = cur;
}
```

Stack

Array Implementation

```
class Stack {
    int* data;
    int top;
    int cap;
public:
    Stack(int cap=100): cap(cap), top(0) {
        data = new int[cap];
    }
    void push(int);
    int pop();
    int peek();
    int size();
    bool isEmpty();
    bool isFull();
}

bool Stack::isEmpty() {
    return top == 0;
}

bool Stack::isFull() {
    return top == cap;
}
// can store from [0] to [cap-1]

void Stack::push(int x) {
    if (isFull()) return;
    data[top++] = x;
}

int Stack::pop() {
    if (isEmpty()) return -1;
    return data[--top];
}

int Stack::peek() {
    if (isEmpty()) return -1;
    return data[top-1];
}

int Stack::size() {
    return top;
}
```

Dynamic Array Implementation

```
void Stack::realloc(int newCap) {
    if (newCap < top) return;
    int* newData = new int[newCap];
    memcpy(newData, data, sizeof(int) * top);
    delete[] data;
    data = newData;
}

void Stack::push(int x) {
    if (isFull()) realloc(cap * 2);
    data[top++] = x;
}

int Stack::pop() {
    if (isEmpty()) return -1;
    int ret = data[--top];
    if (top <= cap / 4) realloc(cap / 2);
    return ret;
}
```

Linked List Implementation

Design: Add the top element to the head of the linked list.

```
void push(int x) {
    Node* newNode = new Node(x, head);
    head = newNode;
}

int pop() {
    if (isEmpty()) return -1;
    Node* tmp = head;
    head = head->next;
    int ret = tmp->data;
    delete tmp;
    return ret;
}

bool isEmpty() {
    return head == nullptr;
}
```

Exercises

For simplicity, use `std::stack` in `<stack>`.

Exercise 1. Find and remove the largest element in a stack.

Exercise 2. Check if a bracket sequence is balanced.

Exercise 3. Evaluate a postfix expression.

Exercise 4. Convert a infix expression to postfix expression.

Exercise 5. Check if a expression contains duplicate brackets.

Exercise 6. Find the smallest integer that can be formed by removing k digits from a number.

Solution:

1.

```
int findAndRemoveLargest(stack<int> &s) {
    if (s.empty()) return -1;
    stack<int> tmp;
    int largest = s.top(); s.pop();
    while (!s.empty()) {
        int cur = s.top(); s.pop();
        if (cur > largest) largest = cur;
        tmp.push(cur);
    }
    while (!tmp.empty()) {
        int cur = tmp.top(); tmp.pop();
        if (cur != largest) s.push(cur);
    }
    return largest;
}
```

2.

```
bool isBalanced(string s) {
    stack<char> st;
    for (char c : s) {
        switch(c) {
            case '(':
            case '[':
            case '{':
                st.push(c);
                break;
            case ')':
                if (st.empty() || st.top() != '(') return false;
                st.pop();
                break;
            case ']':
                if (st.empty() || st.top() != '[') return false;
                st.pop();
                break;
            case '}':
                if (st.empty() || st.top() != '{') return false;
                st.pop();
                break;
        }
    }
    return st.empty(); // unclosed brackets
}
```

3.

```
double compute(char op, double a, double b) {
```

```
    switch(op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
}

double evaluatePostfix(string s) {
    stack<double> st;
    for (char c : s) {
        if (isdigit(c)) {
            st.push(c - '0');
        } else {
            double b = st.top(); st.pop();
            double a = st.top(); st.pop();
            st.push(compute(c, a, b));
        }
    }
    return st.top();
}
```

4.

We define the precedence of operators as follows:

Operator	Precedence
`#`	0
`(`	1
`+` `-`	2
`*` `/`	3

When a `)` is encountered, pop all operators until a `(` is encountered.

When a operator is encountered, pop all operators with higher precedence.

5.

When a `)` is encountered, pop all characters until a `(` is encountered.

If the top of the stack is a `(`, then the expression contains duplicate brackets. (e.g. `((a+b))`)

```
bool containsDuplicateBrackets(string s) {
    stack<char> st;
    for (char c : s) {
        if (c == ')') {
            if (st.top() == '(') return true;
            while (st.top() != '(') st.pop();
            st.pop();
        } else {
            st.push(c);
        }
    }
    return false;
}
```

6.

Maintain an ascending order stack. When a digit is encountered, pop all digits that are larger than it, then push it.

```
string removeKDigits(string s, int k) {
    stack<char> st;
    for (char c : s) {
        while (k > 0 && !st.empty() && st.top() > c) {
            st.pop();
            k--;
        }
        st.push(c);
    }
    while (k > 0) {
        st.pop();
        k--;
    }
    string ret;
    while (!st.empty()) {
        ret += st.top();
        st.pop();
    }
    reverse(ret.begin(), ret.end());
    return ret;
}
```

Complexity Analysis

- O-notation (Big-O): asymptotic upper bound
- Ω-notation (Big-Omega): asymptotic lower bound
- Θ-notation (Big-Theta): asymptotic tight bound
- o-notation (Little-O): asymptotic upper bound, but strictly less than

Mathematically,

- $f(n) = O(g(n))$ if $\exists c > 0, n_0 > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- $f(n) = \Omega(g(n))$ if $\exists c > 0, n_0 > 0$ such that $f(n) \geq cg(n)$ for all $n \geq n_0$.
- $f(n) = \Theta(g(n))$ if $\exists c_1, c_2 > 0, n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
- $f(n) = o(g(n))$ if $\forall c > 0, \exists n_0 > 0$ such that $f(n) < cg(n)$ for all $n \geq n_0$.

Recurrence relation: $T(n) = aT(n/b) + f(n)$

- If $aT(n/b)$ dominates $f(n)$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n)$ dominates $aT(n/b)$, then $T(n) = \Theta(f(n))$.
- If $aT(n/b)$ and $f(n)$ are of the same order, then $T(n) = \Theta(f(n) \log n)$. (simplified)

Example:

- $T(n) = 2T(n/2) + n^2, a = 2, b = 2, f(n) = n^2, n^{\log_b a} = n^{\log_2 2} = n$, $f(n)$ dominates $aT(n/b)$, so $T(n) = \Theta(n^2)$.
- $T(n) = 2T(n/2) + n, a = 2, b = 2, f(n) = n, n^{\log_b a} = n$, $aT(n/b)$ and $f(n)$ are of the same order, so $T(n) = \Theta(n \log n)$.
- $T(n) = 2T(n/2) + 1, a = 2, b = 2, f(n) = 1, n^{\log_b a} = n$, $aT(n/b)$ dominates $f(n)$, so $T(n) = \Theta(n)$.

Analyse the following functions:

1.

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i * i; j++) {
        for (int k = 0; k < j; k++) {
            sum++;
        }
    }
}
```

Solution: $O(n^5)$. `i` = $O(n)$, `j` = $O(n^2)$, `k` = $O(n^2)$.

2.

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i * i; j++) {
        if (j % i == 0) {
            for (int k = 0; k < j; k++) {
                sum++;
            }
        }
    }
}
```

Solution: $O(n^4)$.

- `i` = $O(n)$
- `j` = $O(n^2)$
  - n times `j % i == 0`, with a total of $i + 2i + 3i + \cdots + ni = O(n^3)$ iterations
  - $n^2 - n$ times `j % i != 0`, with a total of $O(n^2)$ iterations
  - therefore, the total iterations of `j` and `k` is $O(n^3)$

Total: $O(n^4)$.

Queue

Array Implementation

```
class Queue {
    int* data;
    int head; // (index of front)
    int tail; // (index of tail) + 1
    int cap;
public:
    Queue(int cap=100): cap(cap), head(0), tail(0) {
        data = new int[cap];
    }
    void enqueue(int);
    int dequeue();
    int getFront();
    int size();
    bool isEmpty();
    bool isFull();
} // can store up to (cap-1) elements

bool Queue::isEmpty() {
    return head == tail;
}

bool Queue::isFull() {
    return (tail + 1) % cap == head;
}

int Queue::size() {
    return (tail - head + cap) % cap;
}

void Queue::enqueue(int x) {
    if (isFull()) return;
    data[tail] = x;
    tail = (tail + 1) % cap;
}

int Queue::dequeue() {
    if (isEmpty()) return -1;
    int ret = data[head];
    head = (head + 1) % cap;
    return ret;
}
```

Linked List Implementation

```
Queue::Queue() {
    head = nullptr;
    tail = nullptr;
}

bool Queue::isEmpty() {
    return head == nullptr;
}
```

```
void Queue::enqueue(int x) {
    Node* newNode = new Node(x);
    if (isEmpty()) {
        head = newNode;
    } else {
        tail->next = newNode;
    }
    tail = newNode;
}

int Queue::dequeue() {
    if (isEmpty()) return -1;
    Node* tmp = head;
    head = head->next;
    int ret = tmp->data;
    delete tmp;
    if (head == nullptr) tail = nullptr; // empty queue
    return ret;
}
```

Exercises

Exercise 1. Reverse a stack using a queue.

Exercise 2. Implement a queue using two stacks.

Solution:

1.

```
void reverseStack(stack<int> &s) {
    queue<int> q;
    while (!s.empty()) {
        q.push(s.top());
        s.pop();
    }
    while (!q.empty()) {
        s.push(q.front());
        q.pop();
    }
}
```

2.

(1) Push $O(1)$, Pop $O(n)$

Design: Use `s1` as a cache, and `s2` as the queue. If `s2` is empty, pop all elements from `s1` and push them to `s2`.

Proof:

- If both `s1` and `s2` contain elements, then all elements in `s2` arrived before all elements in `s1`. Therefore, the elements in `s1` cannot be popped before `s2` is empty.
- If `s2` is empty, then transferring all elements from `s1` to `s2` will reverse the order of elements.

```
class Queue {
    stack<int> s1;
    stack<int> s2;
public:
    void enqueue(int);
    int dequeue();
    int size();
    bool isEmpty();
}

bool Queue::isEmpty() {
    return s1.empty() && s2.empty();
}

int Queue::size() {
    return s1.size() + s2.size();
}

void Queue::enqueue(int x) {
    s1.push(x);
}

int Queue::dequeue() {
    if (isEmpty()) return -1;
    if (s2.empty()) {
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }
    }
    int ret = s2.top();
    s2.pop();
    return ret;
}
```

(2) Push $O(n)$, Pop $O(1)$

Design: Use `s1` as the queue, and `s2` as an auxiliary stack.

```
void Queue::enqueue(int x) {
    while (!s1.empty()) {
        s2.push(s1.top());
        s1.pop();
    }
    s1.push(x);
    while (!s2.empty()) {
        s1.push(s2.top());
        s2.pop();
    }
}

int Queue::dequeue() {
    if (isEmpty()) return -1;
    int ret = s1.top();
    s1.pop();
```

```
    return ret;
}
```

Hash Table

- **Hash function**: $h : U \to \{0, 1, \cdots, m-1\}$ where m is the size of the hash table (usually a prime number).
- **Load factor**: $\alpha = \frac{\text{number of elements}}{m}$
- **Rehashing**: whenever α exceeds a threshold, set the size of table to a prime number that is roughly twice as large, and rehash all elements.
- **Linear probing**: when a collision occurs at slot i, try slot $i+1$, $i+2$, $\cdots$ until an empty slot is found.
- **Quadratic probing**: when a collision occurs at slot i, try slot $i+1^2$, $i+2^2$, $\cdots$ until an empty slot is found.
- **Double hashing**: when a collision occurs at slot i, try slot $i+h_2(k)$, $i+2h_2(k)$, $\cdots$ until an empty slot is found. $h_2 : U \to \{1, 2, \cdots, m-1\}$ is a second hash function. Value of $h_2(k)$ must not be 0.
- **Separate chaining**: each slot is a linked list. When a collision occurs at slot i, insert the element to the linked list at slot i.

If $\alpha \leq 0.5$ and m is a prime number, then quadratic probing will always find an empty slot.

Time complexity per operation: average $O(1)$, worst $O(n)$.

Exercise: Refer to slides.

```
const int MAXN = 20017;
class HashTable {
    int data[MAXN], cnt[MAXN];
    HashTable() {
        memset(data, -1, sizeof(data));
        memset(cnt, 0, sizeof(cnt));
    }
    int hash(int x) {
        return (x + MAXN) % MAXN;
    }
    int find(int x) {
        int h = hash(x);
        while (data[h] != -1 && data[h] != x) {
            h = (h + 1) % MAXN;
        }
        return h;
    }
    void insert(int x) {
        int h = find(x);
        if (data[h] == -1) {
            data[h] = x;
            cnt[h] = 1;
        } else {
            cnt[h]++;
        }
    }
    int query(int x) {
        int h = find(x);
```

```
        if (data[h] == -1) return 0;
        return cnt[h];
    }
}
```

Tree

- **Tree**: a finite set T of one or more nodes such that (the tree cannot be empty)
  - There is a specially designated node called the **root**.
  - By removing the root, we obtain k disjoint sets $T_1, T_2, \cdots, T_k$ of nodes, each of which is a tree. ($T_1, T_2, \cdots, T_k$ are directly connected to the root.)
- **Degree**: the number of subtrees of a node.
- **Leaf**: a node with degree 0.
- **Internal node**: a node with degree ≥ 1.
- **Parent** and **child**: given a node u and one of its subtree v, u is the parent of v and v is the child of u.
- **Sibling**: two nodes with the same parent.
- **Path**: a path from u to v is a sequence of nodes $u, v_1, v_2, \cdots, v_k, v$ such that v_i is the parent of v_{i+1} for $1 \leq i < k$. (Can only go down the tree. The path can be empty.)
- **Ancestor** and **descendant**: given two nodes u and v, u is an ancestor of v and v is a descendant of u if there is a path from u to v. (u is an ancestor and a descendant of itself.)
- **Depth**: the length of the path from the root to the node. (The depth of the root is 0.)
- **Height**: the maximum depth of all nodes. (The height of a tree with only one node is 0.)

Binary tree: is a tree in which each node has at most two children.

- In binary tree, left and right child are distinguished.
- All nodes in a binary tree can have a degree of 0, 1, or 2.
- A binary tree can be empty. An empty tree has a height of -1.

Properties of binary tree:

- A binary tree of height n has at most $2^{n+1} - 1$ nodes.
- Level i has at most 2^i nodes.

More definitions:

- **Full binary tree**:
  - A full binary tree of height n has $2^{n+1} - 1$ nodes.
- **Complete binary tree**: a binary tree such that
  - Its first $n - 1$ levels are full.
  - The last level is filled from left to right.
  - The number of nodes in a complete binary tree of height n is between 2^n and $2^{n+1} - 1$.
- **Binary search tree**: a binary tree such that
  - 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.

Binary Search Tree

```
struct Node {
    int data;
    int cnt;
    Node* left;
    Node* right;
    Node(int data) : data(data), cnt(1), left(nullptr), right(nullptr) {}
};

class BST {
    Node* root;
    void insert(Node*&, int);
    void remove(Node*&, int);
    Node* search(Node*, int);
    void preorder(Node*);
    void inorder(Node*);
    void postorder(Node*);
public:
    BST() : root(nullptr) {}
    void insert(int);
    void remove(int);
    Node* search(int);
    void preorder();
    void inorder();
    void postorder();
};

void BST::insert(Node*& node, int data) {
    if (node == nullptr) {
        node = new Node(data);
        return;
    }
    if (data < node->data) {
        insert(node->left, data);
    } else if (data > node->data) {
        insert(node->right, data);
    } else {
        node->cnt++;
    }
}

void BST::insert(int data) { insert(root, data); }

void BST::remove(Node*& node, int data) {
    if (node == nullptr) return;
    if (data < node->data) {
        remove(node->left, data);
    } else if (data > node->data) {
        remove(node->right, data);
    } else {
        if (node->cnt > 1) {
            node->cnt--;
        } else {
            if (node->left == nullptr && node->right == nullptr) {
```

```
                delete node;
                node = nullptr;
            } else if (node->left == nullptr) {
                Node* tmp = node;
                node = node->right;
                delete tmp;
            } else if (node->right == nullptr) {
                Node* tmp = node;
                node = node->left;
                delete tmp;
            } else {
                Node* cur = node->right;
                while (cur->left != nullptr) cur = cur->left; // choose the
smallest node in the right subtree
                node->data = cur->data; // swap that node with the node to be
deleted
                node->cnt = cur->cnt;
                remove(node->right, cur->data); // delete that node
            }
        }
    }
}

void BST::remove(int data) { remove(root, data); }

Node* BST::search(Node* node, int data) {
    if (node == nullptr) return nullptr;
    if (data < node->data) {
        return search(node->left, data);
    } else if (data > node->data) {
        return search(node->right, data);
    } else {
        return node;
    }
}

Node* BST::search(int data) { return search(root, data); }

void BST::preorder(Node* node) {
    if (node == nullptr) return;
    cout << node->data << " ";
    preorder(node->left);
    preorder(node->right);
}

void BST::preorder() { preorder(root); cout << endl; }

void BST::inorder(Node* node) {
    if (node == nullptr) return;
    inorder(node->left);
    cout << node->data << " ";
    inorder(node->right);
}

void BST::inorder() { inorder(root); cout << endl; }
```

```
void BST::postorder(Node* node) {
    if (node == nullptr) return;
    postorder(node->left);
    postorder(node->right);
    cout << node->data << " ";
}

void BST::postorder() { postorder(root); cout << endl; }
```

Note: the traversal of a BST is always sorted.

Iterative Traversal

Preorder: use a stack to store the nodes whose right subtrees are to be visited.

```
void preorder(Node* root) {
    stack<Node*> st;
    st.push(root);
    while (!st.empty()) {
        Node* cur = st.top(); st.pop();
        cout << cur->data << " ";
        if (cur->right != nullptr) st.push(cur->right);
        if (cur->left != nullptr) st.push(cur->left); // left is to be visited
first
    }
    cout << endl;
}
```

Inorder: use a stack to store the nodes whose data and right subtrees are to be visited.

Go as left as possible, but don't print the internal nodes. When reaching a leaf, print it and return to its parent.

```
void inorder(Node* root) {
    stack<Node*> st;
    Node* cur = root;
    while (cur != nullptr || !st.empty()) {
        while (cur != nullptr) {
            st.push(cur);
            cur = cur->left;
        } // exhaust its left subtree
        cur = st.top(); st.pop();
        cout << cur->data << " "; // print the leaf -> next iteration will go to
its parent, print it and visit its right subtree
        cur = cur->right;
    }
    cout << endl;
}
```

Postorder (2 stacks): postorder (L->R->N) is the reverse of preorder (N->R->L), just with each subtree recursively reversed.

```
void postorder(Node* root) {
```

```
    stack<Node*> st1, st2;
    st1.push(root);
    while (!st1.empty()) {
        Node* cur = st1.top(); st1.pop();
        st2.push(cur);
        if (cur->left != nullptr) st1.push(cur->left);
        if (cur->right != nullptr) st1.push(cur->right);
    }
    while (!st2.empty()) {
        cout << st2.top()->data << " ";
        st2.pop();
    }
    cout << endl;
}
```

Heap

Take min heap as an example.

Min heap: a complete binary tree such that the value of each node is $\leq$ the value of its children.

Assume no duplicate values.

Note the left and right children of node at i are at $2i + 1$ and $2i + 2$.

```
class MinHeap {
    vector<int> data;
public:
    MinHeap() {}
    void build(vector<int>&);
    void insert(int);
    int extractMin();
    int getMin();
    void increaseKey(int, int);
    void decreaseKey(int, int);
};

void MinHeap::build(vector<int>& arr) {
    data = arr; // 0-indexed
    for (int i = data.size() / 2 - 1; i >= 0; i--) {
        heapify(i);
    }
}

int MinHeap::getMin() {
    return data[0];
}

int MinHeap::extractMin() {
    int ret = data[0];
    data[0] = data[data.size() - 1]; // use last element to replace the root
    data.pop_back();
    for (int i = 0, smallest = 0; i < data.size(); i = smallest) {
        smallest = i;
        if (i * 2 + 1 < data.size() && data[i * 2 + 1] < data[smallest])
smallest = i * 2 + 1;
```

```
        if (i * 2 + 2 < data.size() && data[i * 2 + 2] < data[smallest])
smallest = i * 2 + 2;
        if (smallest == i) break;
        swap(data[i], data[smallest]); // if the node is larger than its
children, swap it with the smallest child
    }
    return ret;
}

void MinHeap::insert(int x) {
    data.push_back(x); // push to the end
    for (int i = data.size() - 1, fa = (i - 1) / 2; i > 0; i = fa, fa = (i - 1)
/ 2) {
        if (data[i] >= data[fa]) break;
        swap(data[i], data[fa]); // if the node is smaller than its parent, swap
it with its parent
    }
}
```

Balanced BST

AVL Tree

```
struct Node {
    int data;
    int height;
    Node* ls;
    Node* rs;
    Node(int data) : data(data), height(1), ls(nullptr), rs(nullptr) {}
};

int h(Node* node) {
    return node == nullptr ? 0 : node->height;
}
void insert(Node* v, int x) {
    if (v == nullptr) {
        v = new Node(x);
        return;
    }
    else if (x < v->data) {
        insert(v->ls, x);
        if (h(v->ls) - h(v->rs) == 2) {
            if (x < v->ls->data) LL(v);
            else LR(v);
        }
    } else if (x > v->data) {
        insert(v->rs, x);
        if (h(v->rs) - h(v->ls) == 2) {
            if (x > v->rs->data) RR(v);
            else RL(v);
        }
    }
    v->height = max(h(v->ls), h(v->rs)) + 1;
}
```

```
void LL(Node*& v) { // left rotate
    Node* ls = v->ls;
    v->ls = ls->rs;
    ls->rs = v;
    v->height = max(h(v->ls), h(v->rs)) + 1;
    ls->height = max(h(ls->ls), h(ls->rs)) + 1;
    v = ls;
}
void RR(Node*& v) { // right rotate
    Node* rs = v->rs;
    v->rs = rs->ls;
    rs->ls = v;
    v->height = max(h(v->ls), h(v->rs)) + 1;
    rs->height = max(h(rs->ls), h(rs->rs)) + 1;
    v = rs;
}
void LR(Node*& v) { // double left rotate
    RR(v->ls);
    LL(v);
}
void RL(Node*& v) { // double right rotate
    LL(v->rs);
    RR(v);
}
```

An AVL tree with height n has at least $F_{n+2} - 1$ nodes, where F_n is the n-th Fibonacci number.

n	$F_{n+2} - 1$
1	1
2	2
3	4
4	7

$F_{n+2} - 1 \leq \alpha^n - 1$ where $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.618$.

Splay Tree

```
void zig(Node*& v) { // left rotate
    Node* ls = v->ls;
    v->ls = ls->rs;
    ls->rs = v;
    v = ls;
}
void zag(Node*& v) { // right rotate
    Node* rs = v->rs;
    v->rs = rs->ls;
    rs->ls = v;
    v = rs;
}
void zigzag(Node*& v) { // double left rotate for LR
    zag(v->ls);
```

```
    zig(v);
}
void zagzig(Node*& v) { // double right rotate for RL
    zig(v->rs);
    zag(v);
}
void zigzig(Node*& v) { // for LL
    zig(v); // root and 1st ls
    zig(v); // 1st ls and 2nd ls
}
void zagzag(Node*& v) { // for RR
    zag(v);
    zag(v);
}
void splay(Node*& v, int x) {
    if (v == nullptr) return;
    if (x < v->data) {
        if (v->ls == nullptr) return;
        if (x < v->ls->data) { // LL = zigzig
            splay(v->ls->ls, x);
            zigzig(v);
        } else if (x > v->ls->data) { // LR = zigzag
            splay(v->ls->rs, x);
            zigzag(v);
        } else { // L = zig
            zig(v);
        }
    } else if (x > v->data) {
        if (v->rs == nullptr) return;
        if (x < v->rs->data) { // RL = zagzig
            splay(v->rs->ls, x);
            zagzig(v);
        } else if (x > v->rs->data) { // RR = zagzag
            splay(v->rs->rs, x);
            zagzag(v);
        } else { // R = zag
            zag(v);
        }
    }
}
void split(Node*& v, int x, Node*& ls, Node*& rs) {
    splay(v, x); // splay the node with x (or largest node smaller than x) to
the root
    // simply assume that ls <= x, rs > x
    ls = v;
    rs = v->rs;
}
void insert(Node*& v, int x) {
    Node* ls, *rs;
    split(v, x, ls, rs);
    v = new Node(x);
    v->ls = ls;
    v->rs = rs;
}
void join(Node*& v, Node* ls, Node* rs) {
```

```
    if (rs == nullptr) {
        v = ls;
        return;
    } // ls cannot be nullptr
    splay(ls, INT_MAX); // choose the largest node in ls
    ls->rs = rs;
    v = ls;
}
void remove(Node*& v, int x) {
    splay(v, x);
    if (v->data != x) return;
    Node* ls = v->ls;
    Node* rs = v->rs;
    delete v;
    join(v, ls, rs);
}
```

Remember: splay tree on LL or RR is up-to-bottom, while AVL tree on L, R, LR, or RL is bottom-to-up.

Disjoint Set

```
class DisjointSet {
    int fa[MAXN];
    int height[MAXN];
public:
    DisjointSet() {
        for (int i = 0; i < MAXN; i++) {
            fa[i] = i;
            height[i] = 1;
        }
    }
    int find(int);
    void merge(int, int);
};

int DisjointSet::find(int x) {
    if (fa[x] == x) return x;
    return fa[x] = find(fa[x]);
} // path compression

void merge(int x, int y) {
    int fx = find(x);
    int fy = find(y);
    if (fx == fy) return;
    if (height[fx] < height[fy]) {
        fa[fx] = fy;
    } else if (height[fx] > height[fy]) {
        fa[fy] = fx;
    } else {
        fa[fx] = fy;
        height[fy]++;
    }
} // union by rank
```

Time complexity: $O(\alpha(n))$ per operation, where $\alpha(n)$ is the inverse Ackermann function.

String Matching

1. Trie:

- Time complexity $O(T^2)$ (preprocessing), $O(P)$ (matching)
- Space complexity $O(T^2)$

2. Suffix tree:

- Use substring to replace single character in trie.
- Space complexity $O(T)$

3. Suffic array with doubling algorithm:

- Time complexity $O(T \log T)$ (preprocessing), $O(P \log T)$ (matching)
- Space complexity $O(T)$

L-order: Define $str_{i:n} <_L str_{j:n}$ if the first L characters of str_i are lexicographically smaller than those of str_j.

Doubling algorithm: sort by 1-order, 2-order, 4-order, 8-order, $\cdots$.

From L-order to $2L$-order:

- If $str_{i:n} <_L str_{j:n}$, then $str_{i:n} <_{2L} str_{j:n}$.
- If $str_{i:n} >_L str_{j:n}$, then $str_{i:n} >_{2L} str_{j:n}$.
- If $str_{i:n} =_L str_{j:n}$, then the order of $str_{i:n} \overset{?}{=}_{2L} str_{j:n}$ is the same as $str_{i+L:n} \overset{?}{=}_L str_{j+L:n}$.

Example: string `mississippi`

- $str_{3:n}$ = `ssissippi`, $str_{6:n}$ = `ssippi`
- $str_{5:n}$ = `issippi`, $str_{8:n}$ = `ippi`
- $str_{3:n} =_2 str_{6:n}$, $str_{5:n} >_2 str_{8:n}$
- Therefore, $str_{3:n} >_4 str_{6:n}$ because $str_{3+2:n} >_2 str_{6+2:n}$

```
vector<string> suffixArray(string s) {
    int n = s.size();
    vector<int> sa(n), rank(n);
    for (int i = 0; i < n; i++) {
        sa[i] = i;
        rank[i] = s[i];
    }
    for (int k = 1; k < n; k *= 2) {
        sort(sa.begin(), sa.end(), [&](int i, int j) {
            if (rank[i] != rank[j]) return rank[i] < rank[j];
            if (i + k < n && j + k < n) return rank[i + k] < rank[j + k];
            return i > j;
        });
        vector<int> tmp(n);
        for (int i = 1; i < n; i++) {
            tmp[sa[i]] = tmp[sa[i - 1]] + (rank[sa[i]] != rank[sa[i - 1]] ||
rank[sa[i] + k] != rank[sa[i - 1] + k]);
        }
        rank = tmp;
    }
```

```
    vector<string> ret(n);
    for (int i = 0; i < n; i++) {
        ret[i] = s.substr(sa[i]);
    }
    return ret;
}
```

Graph

A graph $G = (V, E)$ consists of a non-empty set V of vertices and a set E of edges. (i.e. cannot have 0 vertices)

- **Degree**: the number of edges incident to a vertex.
- **Weight**: a number assigned to an edge.
- **Weighted/unweighted graph**: a graph with/without weights. If unweighted, all edges have the same weight.
- **Directed/undirected graph**: a graph with/without directions.
- **Simple graph**: an unweighted, undirected graph with no self-loops and no multiple edges.
- **Complete graph**: a simple graph in which every pair of vertices is connected by an edge. K_n denotes a complete graph with n vertices. It has $\frac{n(n-1)}{2}$ edges.

Representations of graphs:

Adjancency matrix

use A_{ij} to denote the weight (or existence) of the (directed) edge from i to j.

$A_{ij} = 0$ if $i = j$.

Memory: $O(V^2)$.

Very fast to check if an edge exists. $O(1)$.

Slow to list the neighbors of a vertex. $O(V)$.

Adjancency list

```
struct Edge {
    int to;
    int weight;
    Edge(int to, int weight) : to(to), weight(weight) {}
};
struct Vertex { // use a linked list to store all edges
    vector<Edge> edges;
};
Vertex adj[MAXN];
void add(int u, int v, int w) {
    adj[u].push_back(Edge(v, w));
}
```

Memory: $O(V + E)$.

BFS

```
void bfs(int s) {
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (Edge& e : adj[u]) {
            int v = e.to;
            // do something
            q.push(v);
        }
    }
}
```

Time complexity: $O(V + E)$.

DFS

Recursive implementation:

```
bool vis[MAXN];
void dfs(int u) {
    vis[u] = true;
    for (Edge& e : adj[u]) {
        int v = e.to;
        if (!vis[v]) dfs(v);
    }
}
```

Iterative implementation:

```
bool vis[MAXN];
void dfs(int s) {
    stack<int> st;
    st.push(s);
    while (!st.empty()) {
        int u = st.top(); st.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (Edge& e : adj[u]) {
            int v = e.to;
            st.push(v);
        }
    }
}
```

Minimum Spanning Tree

- **Spanning tree**: a subgraph $G' = (V, E')$ of $G = (V, E)$ such that G' is a tree, thus
    - contains all vertices of G.
    - contains $|V| - 1$ edges.

- **Minimum spanning tree**: a spanning tree with the minimum total weight.

Kruskal's algorithm:

Choose the edge with the minimum weight that does not form a cycle with the existing edges.

Implementation: disjoint set.

```
struct Edge {
    int from, to, weight;
    Edge(int from, int to, int weight) : from(from), to(to), weight(weight) {}
    bool operator<(const Edge& rhs) const {
        return weight < rhs.weight;
    }
};
vector<Edge> edges;
int prim(int n) {
    int ret = 0;
    sort(edges.begin(), edges.end());
    DisjointSet ds(n);
    for (Edge& e : edges) {
        if (ds.find(e.from) != ds.find(e.to)) {
            ds.merge(e.from, e.to);
            ret += e.weight;
        }
    }
    return ret;
}
```

Time complexity: $O((V+E)\log E)$.

Prim's algorithm:

Choose the edge with the minimum weight that connects a vertex in the tree to a vertex not in the tree.

Implementation: priority queue.

```
struct Edge {
    int to, weight;
    Edge(int to, int weight) : to(to), weight(weight) {}
    bool operator<(const Edge& rhs) const {
        return weight > rhs.weight;
    }
};
vector<Edge> adj[MAXN];
int prim(int n) {
    int ret = 0;
    priority_queue<Edge> pq; // sort by weight ascending order
    // priority_queue is a max heap by default; we want to use it as a min heap
so define operator < in a reverse way
    vector<bool> vis(n);
    vis[0] = true;
    for (Edge& e : adj[0]) {
        pq.push(e);
    }
```

```
    while (!pq.empty()) {
        Edge cur = pq.top(); pq.pop();
        int u = cur.to;
        if (vis[u]) continue;
        vis[u] = true;
        ret += cur.weight;
        for (Edge& e : adj[u]) {
            pq.push(e);
        }
    }
    return ret;
}
```

Time complexity: $O((V+E)\log V)$.

Shortest Path

Dijkstra's algorithm:

- Maintain a set of visited vertices S.
- While S does not contain all vertices, choose the vertex v that is not in S and has the minimum distance from the source, and add it to S.
- Use v's distance and its incident edges to update the distances of its neighbors.

Does not work with negative cycles.

Implementation: priority queue.

```
vector<Edge> adj[MAXN];
int dist[MAXN];
typedef pair<int, int> pii;
void dijkstra(int n, int s) {
    memset(dist, 0x3f, sizeof(dist));
    dist[s] = 0;
    priority_queue<pii, vector<pii>, greater<pii>> pq; // (dist, vertex) in dist
ascending order
    pq.push(make_pair(0, s));
    while (!pq.empty()) {
        pii cur = pq.top(); pq.pop();
        int u = cur.second; // vertex
        if (dist[u] < cur.first) continue; // visited
        for (Edge& e : adj[u]) {
            int v = e.to;
            if (dist[v] > dist[u] + e.weight) { // relaxations
                dist[v] = dist[u] + e.weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }
}
```

Sorting

Heap Sort

Build a heap, then repeatedly extract the minimum element.

```
void min_heapify(int a[], int i, int n) {
    int smallest = i;
    if (i * 2 + 1 < n && a[i * 2 + 1] < a[smallest]) smallest = i * 2 + 1;
    if (i * 2 + 2 < n && a[i * 2 + 2] < a[smallest]) smallest = i * 2 + 2;
    if (smallest == i) return;
    swap(a[i], a[smallest]);
    min_heapify(a, smallest, n);
}
void heap_sort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        min_heapify(arr, i, n);
    }
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        min_heapify(arr, 0, i);
    }
}
```

Note: with a min heap, the result will be in descending order.

Note: with a max heap, heapify should swap with the largest child.

- Average time complexity: $O(n \log n)$
- Worst time complexity: $O(n \log n)$
- Auxiliary space: $O(1)$
- Heap sort is not stable.

Merge Sort

Split the array into two halves, sort each half, then merge them.

```
void merge_sort(int arr[], int l, int r) {
    if (l >= r) return; // if array size <= 1, it is already sorted
    int mid = (l + r) / 2;
    merge_sort(arr, l, mid);
    merge_sort(arr, mid + 1, r);
    int* tmp = new int[r - l + 1];
    int i = l, j = mid + 1, k = 0;
    while (i <= mid && j <= r) {
        if (arr[i] < arr[j]) tmp[k++] = arr[i++];
        else tmp[k++] = arr[j++];
    }
    while (i <= mid) tmp[k++] = arr[i++];
    while (j <= r) tmp[k++] = arr[j++];
    for (int i = l; i <= r; i++) {
        arr[i] = tmp[i - l];
    }
    delete[] tmp;
}
```

- Average time complexity: $O(n \log n)$
- Worst time complexity: $O(n \log n)$
- Auxiliary space: $O(n)$
- Merge sort is stable.

Linked list implementation:

```
void merge_sort(Node*& head, int r) { // only handle the first r elements
    if (head == nullptr || head->next == nullptr || r <= 1) return;
    Node* mid = head;
    for (int i = 0; i < r / 2; i++) mid = mid->next;
    Node* tmp = mid->next;
    mid->next = nullptr;
    merge_sort(head, r / 2);
    merge_sort(tmp, r - r / 2);
    Node* cur = nullptr;
    while (head != nullptr && tmp != nullptr) {
        if (head->data < tmp->data) {
            if (cur == nullptr) cur = head;
            else {
                cur->next = head;
                cur = cur->next;
            }
            head = head->next;
        } else {
            if (cur == nullptr) cur = tmp;
            else {
                cur->next = tmp;
                cur = cur->next;
            }
            tmp = tmp->next;
        }
    }
    if (head != nullptr) cur->next = head;
    if (tmp != nullptr) cur->next = tmp;
    head = cur;
}
```

Quick Sort

Choose a pivot, partition the array into two parts. Left part is smaller than the pivot, right part is larger than the pivot. Recursively sort the two parts.

During each partition:

- Choose a pivot. In this example, we choose the first element as the pivot.
- Maintain two pointers i and j.
- While $i < j$, move i to the right and j to the left until $a[i] > a[0]$ and $a[j] \leq a[0]$.
- Swap $a[i]$ and $a[j]$.
- When $i \geq j$, the partition is done.
- Swap $a[0]$ and $a[j]$. The pivot is now at its final position.

```
void quick_sort(int arr[], int l, int r) {
    if (l >= r) return;
```

```
    int pivot = partition(arr, l, r);
    quick_sort(arr, l, pivot - 1);
    quick_sort(arr, pivot + 1, r);
}

int partition(int arr[], int l, int r) {
    int i = l, j = r;
    int pivot = arr[l];
    while (i < j) {
        while (i < j && arr[i] <= pivot) i++;
        while (i < j && arr[j] > pivot) j--;
        swap(arr[i], arr[j]);
    }
    arr[l] = arr[j];
    arr[j] = pivot;
    return j;
}
```

- Average time complexity: $O(n \log n)$
- Worst time complexity: $O(n^2)$ (When the array is ascendingly sorted. Can be mitigated by choosing a random pivot, but cannot be avoided.)
- Auxiliary space: $O(1)$
- Quick sort is not stable.

Bucket Sort

Count the apperance of each element, then output them in order.

```
int cnt[MAXV];
void bucket_sort(int arr[], int n) {
    memset(cnt, 0, sizeof(cnt));
    for (int i = 0; i < n; i++) {
        cnt[arr[i]]++;
    }
    for (int i = 0, j = 0; i < MAXV; i++) {
        while (cnt[i]--) {
            arr[j++] = i;
        }
    }
}
```

- Average time complexity: $O(n + V)$ (`memset` is $O(V)$)
- Worst time complexity: $O(n + V)$
- Auxiliary space: $O(V)$
- Bucket sort is stable.

Radix Sort

Per-digit bucket sort.

Sort the numbers, starting from least significant digit till most significant digit.

```
// d = log10(MAXV), i.e. the number of digits of MAXV
int digit(int x, int d) {
```

```
    while (d--) x /= 10;
    return x % 10;
}
void radix_sort(int arr[], int n, int d) {
    for (int i = 0; i < d; i++) {
        queue<int> buckets[10];
        for (int j = 0; j < n; j++) {
            buckets[digit(arr[j], i)].push(arr[j]);
        }
        for (int digit = 0, j = 0; digit < 10; digit++) {
            while (!buckets[digit].empty()) {
                arr[j++] = buckets[digit].front();
                buckets[digit].pop();
            }
        }
    }
}
```

- Average time complexity: $O(d(n+B))$
  - where B is the base of the number system, $d = \log_B(MAXV)$ is the number of digits of MAXV under base B.
- Worst time complexity: $O(d(n+B))$
- Auxiliary space: $O(n+B)$
- Radix sort is stable.
- Compared to bucket sort, radix sort is faster and memory efficient when $MAXV$ is large.

Complexity Analysis

- Linked list:
  - Head insertion: $O(1)$
  - Random insertion/deletion/access: $O(n)$
- Stack:
  - Push/pop: $O(1)$
- Queue:
  - Enqueue/dequeue: $O(1)$
- Binary search:
  - $O(\log n)$
- Hash table:
  - Insert/delete/search: average $O(1)$, worst $O(n)$
- Binary search tree:
  - Insert/delete/search: average $O(\log n)$, worst $O(n)$
  - AVL tree: $O(\log n)$
  - Splay tree: worst single operation $O(n)$, amortized $O(\log n)$
- Heap:
  - Insert/delete: $O(\log n)$
  - Build: $O(n)$
  - Extract min/max: $O(\log n)$
- Disjoint set:

    - Find/merge (without optimization): $O(n)$
    - Find/merge (with union by rank / union by size): $O(\log n)$
    - Find/merge (with union by rank and path compression): $O(\alpha(n))$
- Graph:
    - BFS: $O(V + E)$
    - DFS: $O(V + E)$
    - MST (Prim): $O((V + E) \log E)$
    - Shortest path (Dijkstra): $O((V + E) \log V)$
- Sorting:
    - Heap sort: $O(n \log n)$
    - Merge sort: $O(n \log n)$
    - Quick sort: average $O(n \log n)$, worst $O(n^2)$
    - Bucket sort: $O(n + V)$
    - Radix sort: $O(d(n + B))$
- String matching:
    - Trie: $O(T^2 + P)$, space $O(T^2)$
    - Suffix tree: $O(T)$, space $O(T)$
    - Suffix array: $O(T \log T + P \log T)$, space $O(T)$