Jacob Chmura

Introduction

Theorem 0.1. *Master Theorem for Divide and Conquer Recurrences:*

$$T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n), a \geq 1, b > 1, k \geq 0, p \in \mathbb{R} \tag{1}$$

then:

- $a > b^k \implies T(n) = \Theta(n^{log_b^a})$
- $a = b^k$
  - $p > -1 \implies T(n) = \Theta(n^{log_b^a} log^{p+1} n)$
  - $p = -1 \implies T(n) = \Theta(n^{log_b^a} loglogn)$
  - $p < -1 \implies T(n) = \Theta(n^{log_b^a})$
- $a < b^k$
  - $p \geq 0 \implies T(n) = \Theta(n^k log^p n$
  - $p < 0 \implies T(n) = \mathcal{O}(n^k)$

1 Linked Lists

ADT

- Insert: insert element into list
- Delete: removes and returns the specified positiong element from list
- Delete list: removes all elements of list
- Count: returns numbers of elements in list
- Find *nth* node from the end of the list

Linked Lists

- Expanded in constant time (array needs to double size and copy all elements
- Disadvantage is mainly access time is $\mathcal{O}(n)$ compared to random constant access for static arrays and dynamic arrays

Arrays

- Constant access
- Preallocates all needed memory up front
- Fixed size
- Position based insertion is slow
- Contiguous in memory helps cpu cache

Dynamic Arrays

- random access, variable size list structure

1.1 Singly Linked Lists

```
struct ListNode:
    int data;
    struct ListNode *next;

int ListLength(struct ListNode* head){
    // O(n) time
    // O(1) space

    struct ListNode *curr = head;
    int length = 0;

    while(curr != NULL){
        length++;
        curr = curr->next;
    }
    return length;
}

void Insert(struct ListNode** head, int data, int idx){
    // Send double pointer to modify the head pointer
    // O(idx) time
    // O(1) space

    struct ListNode *newNode;
    newNode = (ListNode)*malloc(sizeof(struct ListNode));
    newNode->data = data;

    if(idx == 0){
        newNode->next = *head;
        *head = newNode;
    }
    else{
        int k = 0;
        struct ListNode *curr = *head;
        struct ListNode *prev;
        while(k < idx && curr != NULL){
            k++;
            prev = curr;
            curr = curr->next;
        }
        prev->next = newNode;
        newNode->next = curr;

    }
}

void Delete(struct ListNode** head, int idx){
    // Send double pointer to modify the head pointer
    // O(idx) time
    // O(1) space
    if (*head == NULL) return;
```

```

        if(idx == 0){
            struct ListNode *tmp = *head;
            *head = (*head)->next;
            free(tmp);
        }
        else{
            int k = 0;
            struct ListNode *curr = *head;
            struct ListNode *prev;
            while(k < idx && curr != NULL){
                k++;
                prev = curr;
                curr = curr->next;
            }
            if(curr == NULL) return;
            prev->next = curr->next;
            free(curr);

        }
    }

    void DeleteList(struct ListNode**head){
        // O(n) time
        struct ListNode *iterator, *tmp;
        iterator = *head;
        while(iterator != NULL){
            tmp = iterator->next;
            free(iterator);
            iterator = tmp;
        }
        *head = NULL;
    }
```

1.2 Doubly Linked Lists

- Can delete a node directly given a pointer to it since we have pointer to predecessor
- Each node requires more space by 4 bytes
- More pointer operations
- Memory efficient implementation with single pointer called *pointer diff* which is given as the *xor* of previous node and next node adress

```
    struct DLLNode:
        int data;
        struct DLLNode *next
        struct DLLNode *prev;

    int ListLength(struct DLLNode* head){
        // O(n) time
        // O(1) space

        struct DLLNode *curr = head;
        int length = 0;

        while(curr != NULL){
            length++;
            curr = curr->next;
        }
        return length;
    }

    void Insert(struct DLLNode** head, int data, int idx){
        // Send double pointer to modify the head pointer
        // O(idx) time
        // O(1) space

        struct DLLNode *newNode;
        newNode = (DLLNode)*malloc(sizeof(struct DLLNode));
        newNode->data = data;

        if(idx == 0){
            newNode->next = *head;
            newNode->prev = NULL;
            if (*head) head->prev = newNode; // handles case for insertion into empty list
            *head = newNode;
        }
        else{
            int k = 0;
            struct DLLNode *curr = *head;
            while(k < idx - 1 && curr != NULL){
                k++;
                curr = curr->next;
            }

            newNode->next = curr->next;
            newNode->prev = curr;

            if(curr->next) curr->next->prev = newNode;
            curr->next = newNode;
        }
    }

    void Delete(struct DLLNode** head, int idx){
        // Send double pointer to modify the head pointer
        // O(idx) time
        // O(1) space
        if (*head == NULL) return;

        if(idx == 0){
            struct DLLNode *tmp = *head;
            *head = (*head)->next;
            if(*head != NULL){
                (*head)->prev = NULL;
                free(tmp);
            }
        }
        else{
            int k = 0;
            struct DLLNode *curr = *head;
            struct DLLNode *prev;
            while(k < idx && curr != NULL){
                k++;
                prev = curr;
                curr = curr->next;
            }

            if(curr == NULL) return;
            prev->next = curr->next;
            if(curr->next) curr->next->prev = prev;
            free(curr);
        }
    }

    void DeleteList(struct DLLNode**head){
        // O(n) time
        struct DLLNode *iterator, *tmp;
        iterator = *head;
        while(iterator != NULL){
            tmp = iterator->next;
            free(iterator);
            iterator = tmp;
        }
        *head = NULL;
    }
```

1.3 Circly Linked Lists

- Do not have ends with NULL like singly and double LL
- Use case: round robin
- implement stacks and queues

```
    struct CLLNode:
        int data;
        struct DLLNode *next

    int ListLength(struct CLLNode* head){
        // O(n) time
        // O(1) space

        struct DLLNode *curr = head;
        int length = 0;
        if(head == NULL) return 0;
        while(curr != head){
            length++;
            curr = curr->next;
        }
        return length;
    }

    void InsertAtEnd(struct CLLNode** head, int data){
        // Send double pointer to modify the head pointer
        // O(n) time
        // O(1) space

        struct CLLNode *newNode;
        newNode = (CLLNode)*malloc(sizeof(struct CLLNode));
        newNode->data = data;
        newNode->next = newNode;

        structCLLNode *curr = *head;
        while(curr->next != head){
            curr = curr->next;
        }
        if(*head == NULL){
            *head = newNode;
        }
        else{
            newNode->next = *head;
            curr->next = newNode;
        }
    }

    void DeleteAtEnd(struct CLLNode** head){
        // Send double pointer to modify the head pointer
        // O(n) time
        // O(1) space
        if (*head == NULL) return;

        struct CLLNode *curr = *head;
        struct CLLNode *prev;
        while(curr->next != *head){
            prev = curr;
            curr = curr->next;
        }
        prev->next = head;
        free(curr);
    }
```

1.4 Unrolled Linked List

- Inserting element in LL is constant time operation, but requires traversing $\mathcal{O}(n)$ nodes
- unrolled link list partitions LL into blocks of nodes. Each block is connected by a singly linked list. The nodes within a block are connected with circular LL. The size of each block can be $\sqrt{n}$, then we can do traversals in $\sqrt{n}$.

- Shift operation used for insertion done in constant time for each block, with at most $\sqrt{n}$ blocks implies insertion is also $\sqrt{n}$.

1.5 Skip Lists

- Probabilistic alternatives to balanced tress which allow quick seach, insertion and deletion
- use probabilistic balancing instead of stricly enforcing balancing
- sorted linked lists
- double the number of pointers by having each node skip in max multiples ahead, we get $\mathcal{O}(logn)$ performance.

1.6 Problems

Find nth element from the end of linked list

- We can scan once to get length of LL and then perform second scan to correct position
- One scan algorithm: two pointers, pTmp and pNthNode. Move pTemp n times. Then move each pointer one step until pTemp is at the end of list. At this point pNthNode points to the required node.

```

    struct ListNode* NthNodeFromEnd(struct ListNode* head, int n){
        struct ListNode *pTemp = head, pNthNode = NULL;
        for(int i = 0; i < n; i++){
            if(pTemp) pTemp = pTemp->next;
        }
        while(pTemp){
            if(pNthNode == NULL){
                pNthNode = head;
            }
            else{
                pNthNode = pNthNode->next;
            }
            pTemp = pTemp->next;
        }
        return pNthNode;

```

Detect Cycle

- Can store a hash set of node adresses and iterate the list and check if current node is seen. If no loop we will reach Null and say no cycle
- A constant space algorithm exists *Floyd Cycle Finding*. Fast pointer and slow pointer iterate list at different speeds. If cycle exists then they must eventually meet, otherwise fast pointer hits the end of the list

```

    bool FloydLLCycleDetect(struct ListNode* head){
        struct ListNode *slow = head, *fast = head;
        while(slow && fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) return True;
        }
        return False; // no cycle
    }

```

Reverse a linked list

- We can do it recursively but iteration has better space complexity.

```

        struct ListNode* reverse(struct ListNode* head){
            struct ListNode *curr = NULL, *tmp = NULL;
            while(head){
                curr = head->next;
                head->next = tmp;
                tmp = head;
                head = curr;
            }
            return head;

        }
```

Intersection of two linked lists

- Scan a list at random putting each node address into hash set. Then scan second list and check for occurrence in hash set. Return first occurrence.
- More efficient approach in $\mathcal{O}(max(m, n))$ time and $\mathcal{O}(1)$ space. Find length of both lists, m and n. Take difference of lengths $d = n - m$. Take d steps in the longer list. Now step in parallel on both lists and exist when node addresses match.

Print linked list from end

- Recursive solution

```

        void reversePrint(struct ListNode* head){
            if(!head) return;
            reversePrint(head->next);
            cout << head->data << endl;
        }
```

Delete element from linked list given point to element

- We don't have previous point access to redirect its pointer to our current next. Instead, we should copy over data from the next node to current, and delete the next node.

Stacks

Definition 1. A **stack** is an ordered list in which insertion and deletion are done at one end, called the top. It implements *last in fast out* behaviour.

Stack ADT

- Push: inserts data onto top of stack
- Pop: removes and returns the last inserted element from the stack
- Top: returns the last inserted element without removing it
- IsEmpty / IsFull: determines if the stack is empty or full

Applications

- Implementing function calls including recursion
- Page visited history in web browser
- Undo sequence in a text editor
- Use for tree traversal algorithms

Implementation

We can implement using a static array. All operation are constant time but the size of the stack must be defined and not changed.

```

    #define MAXSIZE = 10;
    struct ArrayStack {
        int top;
        int capacity;
        int *array;
    }

    struct ArrayStack *Create(){
        struct ArrayStack *S = malloc(sizeof(struct ArrayStack));
        S->capacity = MAXSIZE;
        S->top = -1;
        S->array = malloc(S->capacity * sizeof(int));
        return S;
    }

    bool isEmpty(struct ArrayStack *S){
        return S->top == -1;
    }

    bool isFull(struct ArrayStack *S){
        return S->top == S->capacity - 1;
    }

    void Push(struct ArrayStack *S, int data){
        if(isFull(S)){
            cout << "StackOverFlow" << endl;
        }
        else{
            S->array[++S->top] = data
        }
    }

    int Pop(struct ArrayStack *S){
        if(isEmpty(S)){
            cout << "Alreadu Empty << endl;
            return INT_MIN;
        }
        return S->array[S->top--]
    }
```

We can remedy by using **array doubling** to get amortized constant complexity of the push operation.

Alternatively, we can implement using a linked list where the top element is head node.

Problems

Design stack s.t. getting minimum is $\mathcal{O}(1)$

- Create auxilary stack called *min stack*
- When we push to our stack, push to the min stack when the value being pushed onto the main stack is less than or equal to the current min
- When we pop from our stack, we only pop from the min stack when the value we pop from the main stack is equal to the one on the min stack.

Reverse stack using only stack ops

- Recursive solution

```

        void reverse(struct Stack *S){
            // O(n) space and O(n^2) time
            if(isEmpty(S)) return;
            int data = Pop(S);
            reverse(S);
            insertAtBottom(S, data);
        }

        void insertAtBottom(struct Stack *S, int data){
            if(isEmpty(S)){
                Push(S, data);
                return;
            }
            int temp = Pop(S);
            insertAtBottom(S, data);
            Push(S, temp);
        }


```

Finding Spans

- Given an array A, the span $S[i]$ of A is the maximum number of consecutive elements $A[j]$ immediately preceding $A[i]$ such that $A[j] < A[i]$

```

        void FindSpans(int A[], int n, int *S){
            // O(n) time and space
            struct Stack *D = Stack();
            int P;
            for(int i = 0; i < n; i++){
                while(!isEmpty(D) && A[i] > A[top(D)]){
                    pop(D);
                }
                P = isEmpty(D) ? -1 else top(D);
                S[i] = i - P;
                Push(D, i);
            }
            return S;
        }
        }
```

Largest Rectangle under histogram using stacks

- Given histogram of rectangle heights, find the largest area rectangle possible.
- This can be solved in linear time using stacks.

```

        struct StackItem {
            int height;
            int index;
        }

        int MaxRectangleArea(int *A, int n){
            int maxArea=-1, top=-1, left, currentArea;
            struct StackItem *S = (struct StackItem *) malloc(sizeof(struct StackItem) * n);
            for(int i = 0; i <=n; i++){
                while(top >= 0 && (i==n || S[top]->height > A[i]{
                    left = (top > 0) ? S[top-1]->index : -1;
                    currentArea = (i - left - 1) * S[top--]->height;
                    maxArea = max(maxArea, currentArea)
                }
                if(i < n){
                    top++;
                    S[top]->height = A[i];
                    S[top]->index = i;
                }
            }
            return maxArea;
        }
```

Recursively remove all adjacent duplicates

- Given a string of characters, recursively remove adjacent duplicate characters from string.

```

        struct StackItem {
            int height;
            int index;
        }

        void removeAdjDuplicates(char *str){
            int stkptr=-1, i=0, len=strlen(str);
            while(i<len){
                if(stkptr == -1 || str[stkptr] != str[i]){
                    str[++stkptr] = str[i++];
                }
                else{
                    while(i < len && str[stkptr] == str[i]){
                        i++;
                        stkptr--;
                    }
                }
            }
        }
```

Queues

Definition 2. A **queue** is an ordered list in which insertions are done at one end (rear) and deletions are done at the other end (front). The queue implements *first in first out* behaviour.

Queue ADT

- Push: inserts an element at the back of the queue
- Pop: removes and returns the element at the front of the queue
- isEmpty: indicates whetheer the queue is empty

Applications

- Job scheduling with uniform priority
- Asynchronous data transfer
- Used on other data structure algorithms

Queues are implemented using static circular array, dynamic circular array, or linked list.

```

    struct ListNode {
        int data;
        struct ListNode *next;
    }
    struct Queue {
        struct ListNode *front, *rear;
    }

    struct Queue* create(){
        struct Queue *Q = malloc(sizeof(struct Queue));
        Q->front = Q->rear = NULL;
        return Q;
    }

    bool isEmpty(struct Queue *Q){
        return Q->front == NULL;
    }

    void push(struct Queue *Q, int data){
        struct ListNOde * node = malloc(sizeof(struct ListNode));
        node->data = data;
        node->next = NULL;
        if(Q->rear) Q->rear->next = newNode;
        Q->rear = newNode;

        if(Q->front == NULL) Q->front = Q->rear;
    }

    void pop(struct Queue *Q){
        if (isEmpty(Q)) return;
        struct ListNode *tmp = Q->front;
        data = Q->front->data;
        Q->front == Q->front->next;
        free(tmp);
        return data;
    }

```

All operations are constant time.

Problems

Implement queue using two stacks

- Push op: just push onto stack 1 in constant time
- Pop op: if stack 2 nonempty, pop it. Else transform all from stack 1 to stack 2 and pop from stack 2. Amortized complexity is $\mathcal{O}(1)$.

Maximum sum in sliding window

- Given array A[] with sliding window of size w, return array B[] where B[i] is the maximum value from A[i] to A[i+w-1].
- Doubly ended queue works here

Trees

A **tree** is a non-linear data structure that can be used to represent hierarchical nature of structures. Ordering information is better necoded with linear data structures such as linked lists, stacks, queues, etc.

Definition 3. The **root** of a tree is the node with no parents.

An **edge** refers to a link from parent to child

A node with no parents is called a **leaf**.

Children of the same parent are called **siblings**.

A node p is an **ancestor** of node q if there exists a path from root to q and p appears on the path. The node q is called a **descendant of p**

The set of all nodes at a given depth is called the **level** of the tree. The root node is at level zero.

The **depth** of a node is the length of the path from the root to the node.

The **height** of a node is the length of the path from that node to the deepest node. The heigth of a tree is the length of the path from the root to the deepest node in the tree.

Binary Trees

A tree is called **binary tree** if each node has zero, one, or two children. A *full binary tree* is when each node has exactly two children, and all leaf nodes are at the level. A *complete binary tree* is when all leaf nodes are at height h or $h-1$ where h is the height of the tree.

Properties:

- The number of nodes n in a full binary tree is $2^{h+1}-1$
- The number of nodes n in a complete binary tree is $n \in [2^h, 2^{h+1}-1]$
- The number of leaf nodes in a full binary tree is 2^h
- The number of NULL links (wasted pointers) in a complete binary tree of n nodes is $n+1$.

Structure:

```

    struct BinaryTreeNode{
        int data;
        struct BinaryTreeNode *left;
        struct BinaryTreeNode *right;
    };

```

Operations:

- Insert into tree
- Delete from tree
- Search for an element in tree
- Traverse tree
- Find the size, height of tree

- Find the level of tree with maximum sum
- Find least common ancestor for a given pair of node

Binary Tree Traversals

Definition 4. Pre Order Traversal processes the current node, then the left and right subtrees.

```

    void PreOrder(struct BinaryTreeNode *root){
        // O(n) time and space
        if(root){
            print(root->data);
            PreOrder(root->left);
            PreOrder(root->right);
        }
    }

    void PreOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = Stack();
        while(1){
            while(root){
                print(root->data);
                push(S, root);
                root = root->left;
            }
            if(isEmpty(S)) break;
            root = Pop(S)->right;
        }
        delete S;
    }

```

Definition 5. In Order Traversal processes the left subtree, then the root, then the right subtree

```

    void InOrder(struct BinaryTreeNode *root){
        // O(n) time and space
        if(root){
            InOrder(root->left);
            print(root->data);
            InOrder(root->right);
        }
    }

    void InOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = Stack();
        while(1){
            while(root){
                push(S, root);
                root = root->left;
                print(root->data);

            }
```

```
                if(isEmpty(S)) break;
                root = Pop(S)
                print(root->data)
                root = root->right;
            }
            delete S;
        }

```

Definition 6. Post Order Traversal processes the left and right subtrees, and then the root.

In preorder and inorder traversals, after popping the stack element, we do not need to visit the same vertex again. In postorder traversal, each node is vitied twice.

```
        void PostOrder(struct BinaryTreeNode *root){
            // O(n) time and space
            if(root){
                InOrder(root->left);
                InOrder(root->right);
                print(root->data);
            }
        }

        void PostOrderNonRecursive(struct BinaryTreeNode *root){
            struct Stack *S = Stack();
            struct BinaryTreeNode *prev = NULL;

            do{
                while(root!=NULL){
                    push(S, root);
                    root = root->left;
                }

                while(root == NULL && !isEmpty(S)){
                    root = Top(S);
                    if(root->right == NULL || root->right == previous){
                        print(root->data)
                        Pop(S);
                        previous = root;
                        root = NULL;
                    }
                    else{
                        root = root->right
                    }
                }
            }while(!isEmptyStack(S));
        }

```

Definition 7. Level Order Traversal is defined as follows:

- visit the root

- while traversing level, keep all the elements at level +1 in queue
- go to the next level and visit all the nodes at that level
- repeat until all levels are complete

```
    void LevelOrder(struct BinaryTreeNode *root){
        // O(n) time and space
        struct BinaryTreeNode *tmp;
        struct Queue *Q = Queue();

        if(!root) return;
        Q.push(root);
        while(!isEmpty(Q)){
            tmp = Q.pop();
            print(tmp->data);
            if(tmp->left) Q.push(tmp->left));
            if(tmp->right) Q.push(tmp->right));
        }
        delete Q;
    }
```

Problems

- Find maximum, searching, size, height: all use recursive sub-substructure and recursion. Alternatively, do a level order traversal
- Finding level with max sum.

```
    int FindLevelMaxSum(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        int level = 0, maxlevel = 0;
        struct Queue *Q;
        int currentSum = 0, maxSum = 0;
        if(!root) return 0;
        Q.push(root);
        Q.push(NULL); // end of first level
        while(!isEmpty(Q)){
            temp = Q.pop();
            if(temp == NULL){// if current level complete, compare sums
                if(currentSum > maxSum){
                    maxSum = currentSum;
                    maxLevel = level;
                }
                currentSum = 0;
                if(!isEmpty(Q)) Q.push(NULL);
                level++
            }
            else {
                currentSum += temp->data;
                if(temp->left) Q.push(temp->left)
                if(temp->right) Q.push(temp->right)
            }
        }

        return maxLevel;

    }
```

- Find the least common ancestor of two nodes

```
    struct BinaryTreNode *LCA(struct BinaryTreeNode *root, struct BinaryTreeNode *a,
                              struct BinaryTreeNode *b){
        if(root == NULL) return root;
        if (root == a || root == b) return root;
        struct BinaryTreeNode * left = LCA(root->left, a, b);
        struct BinaryTreeNode * right = LCA(root->right, a, b);
        if(left && right) return root;
        return left? left: right;
    }
```

- Build a tree given inorder and preorder traversal arrays
  1. Select an element from pre order. Increment preorderidx. The element is root.
  2. Find this element in the inorder list. Elements to left belong in left subtree, and right belong right subtree.
  3. Call recursive for element before inOrderIndex and make result left subtree. Call recusrive for element after in order index, and make result right subtree.

```
    struct BinaryTreNode *BuildTree(int inOrder[], int preOrder[],
                                    int inOrderStart, int inOrderEnd){
        if(inOrderStart > inOrderEnd) return NULL;
        static int preOrderidx = 0;
        struct BinaryTreeNOde *newNode = new BinaryTreeNode;

        newNode->data = preOrder[preOrderIdx++];
        if(inOrderStart == inOrderEnd) return newNode;
        int inOrderIdx = search(inOrder, inOrderStart, inOrderEnd, newNode->data);
        newNode->left = BuildTree(inOrder, preOrder, inOrderStart, inOrderIndex-1);
        newNode->right = BuildTree(inOrder, prOrder, inOrderIndex+1, inOrderEnd);
        return newNode;
    }
```

We can uniquely construct a tree given two traversals ***as long as one of them is inorder***

N-ary trees

Representation

At each node link children of same parent (siblings) from left to right. Remove the links from parents to all children except the first child. This is called the **first child next siblings representation**.

```
    struct TreeNode{
        int data;
        struct TreeNode *firstChild;
        struct TreeNode *nextSiblings;
    }
```

which is a binary tree.

Threaded binary tree traversals (stack or queue-less traversals

Idea: store useful information in the NULL pointers of a tree. Typically we put the predecessor/successor information.

- If we store predeccesor information in NULL left pointers only, it is a *left threaded binary tree*. Alternatively we have *right threaded binary tress* and *fully threaded binary trees*

- **PreOrder Threaded Binary Tree**: NULL left pointers contain PreOrder predecessor info, and NULL right pointer contain PreOrder successor info
- **Inorder Threaded Binary Tree**: NULL left pointers contain Inorder predecessor info, and NULL right pointer contain Inorder successor info
- **PostOrder Threaded Binary Tree**: NULL left pointers contain PostOrder predecessor info, and NULL right pointer contain PostOrder successor info

```
    struct ThreadedBinaryTreeNode{
        int data;
        int LTag, RTag; // if 0, pointer to predecessor / successor info.
                        // if 1, points to left/right child
        struct ThreadedBinaryTreeNOde *left, *right
    }
```

Leftmost and rightmost pointers point to a special dummy node which is always present. Right tag of dummy node is 1, and right child points to itself.

Finding Inorder Successor in Inorder Threaded Binary Tree

```
    struct TBTN *InorderSucc(struct TBTN *P){
        if(P->RTag == 0) return P->right;
        struct TBTN *pos = P->right;
        while(Position->LTag == 1) pos = pos->left;
        return pos;

    }
```

InOrder Traversal in Inorder Threaded Binary Tree

```
    void InOrderTraversal(struct TBTN *root){
        struct TBTN *P = InOrderSucc(root);
        while(P!= root){
            P = InOrderSucc(P);
            print(P->data);
        }
    }
```

XOR Trees

Similar to memory efficient doubly linked lists.

- Each nodes left will have the xor of its parent and its left children
- Each nodes right will have the xor of its parent and its right children
- The root nodes parent is NULL and also leaf nodes children are NULL nodes

Binary Search Trees

Binary search tree is a representation used for *searching*, reducing the worst case average seacrch operation to $\mathcal{O}(n)$.

BST Property

- Left subtree of a node contains only nodes with keys less than node key
- Right subtree of a node contains only nodes with keys greater than the node key
- Both left and right subtrees also have the BST property

Operations

- Find, find min, find max in BST
- Insert, delete
- Find kth smallest
- Sort elements

Notes

- Since root data is always between left subtree data and right subtree data *inorder traversal on BST produces a sorted list*
- If we search element, we only need to consider left *or* right subtree but not both due to the ordering
- Basic ops on a BST take time proportional to height of the tree. For a complete tree, $log(N)$, for a linear chain, $O(n)$.

Finding Element

```
    struct BSTNode *Find(struct BSTNode *root, int data){
        // O(n) worst case time for a skew tree, O(n) space for stack
        if(root == NULL) return NULL;
        if(data < root->Data) return Find(root->left, data);
        if(data > root->Data) return Find(root->right, data);
        return root;
    }

    struct BSTNode *FindNonRecurse(struct BSTNode *root, int data){
        while(root){
            if(data == root->data) return root;
            else if(data > root->data) root = root->right;
```

```c
            else root = root->left;
        }
        return NULL;
    }
```

<u>Finding Minimum / Max</u>

```c
    struct BSTNode *FindMin(struct BSTNode *root){
        // O(n) worst case for left skew tree, O(n) space for stack
        if(root == NULL) return NULL;
        if(root->left == NULL) return root;
        return FindMin(root->left);
    }

    struct BSTNode *FindMinNonRecuse(struct BSTNode *root){
        if(root == NULL) return NULL;
        while(root->left != NULL) root = root->left;
        return root;
    }
```

Symmetric code for finding max by descending right subtrees

<u>Inorder Predecessor, Succesor</u>

- If X has two children, inorder predecessor is max value in left subtree, inorder successor is min value in right subtree
- If X has no left child, inorder predecessor is first left ancestor,

<u>Insertion</u>

```c
    struct BSTNode *Insert(struct BSTNode *root, int data){
        // O(n) time worst case, O(n) space for recurse
        if(root == NULL){
            root = (struct BSTNode)*malloc(sizeof(struct BSTNode));
            root->data = data;
            root->left = root->right = NULL;
        }
        else{
            if(data < root->data){
                root->left = Insert(root->left, data);
            }
            else if( data > root->data){
                root->right = Insert(root->right, data);
            }
        }
        return root;
    }
```

<u>Deletetion</u>

- Find node to be deleted

- If node is leaf: return NULL to its parent
- If node has one child: send the current node's child to its parent
- If node has two children: replace the node with the largest element of the left subtree, and recursively delete that node (which is now empty). The largest node in the left subtree cannot have a right child, so the seocnd delete is an easy one. *We can replace with minimum element in right subtree also*

```
    struct BSTNode *Delete(struct BSTNode *root, int data){
        // O(n) time worst case, O(n) space for recurse
        struct BSTNode *temp;
        if(root == NULL) return root;
        if(data < root->data){
            root->left = Delete(root->left, data);
        }
        else if (data > root->data){
            root->right = Delete(root->right, data);
        }
        else{
            // found element
            if(root->left && root->right){
                // replace with largest in left subtree
                temp = FindMax(root->left);
                root->data = temp->data;
                root->left = Delete(root->left, root->data);
            }
            else{
                // one child
                temp = root;
                if(root->left == NULL) root = root->right;
                if(root->right == NULL) root = root->left;
                free(temp);
            }
        }
        return root;
    }
```

Problems

- Given pointers to two nodes in a BST, find the least common ancestor LCA. Assume both values already exist in the tree.
  - While traversing the BST from root to bottom, the first node we encounter with the value between α and β is the LCA of α, β where $\alpha < \beta$.
  - Traverse the BST in pre-order, and if we find a node with value in between α, β then the node is the LCA. If its value is greater than both α, β, LCA is left side, if smaller than both α, β, LCA is right side.

```
    struct BSTNode *LCA(struct BSTNode *root, struct BSTNode *a, struct BSTNode *b){
        while(1){
            if(a->data < root->data && b->data > root->data) ||
              (b->data < root->data && a->data > root->data) return root;
             if(a->data < root->data) root = root->left;
            else root = root->right;
        }
    }
```

- Shortest path between two nodes in a BST. Find LCA of two nodes in BST.
- Check if bst is satisfied
  - For each node, check if max value in left subtree is smaller than the current node data, and min value in right subtree greater than node data.

```
    int prev = INT_MIN;
    int isBST(struct BSTNode *root, int *prev){
        if(!root) return 1;
        if(!isBST(root->left, prev)) return 0;
        if(root->data < *prev) return 0;
        *prev = root->data;
        return isBST(root->right, prev);
    }
```

- Given sorted array, convert to BST

```
    struct BSTNode *BuildBST(int A[], int left, int right){
        if(left > right) return NULL;
        int mid;
        struct BSTNode *newNode = (struct bstNode *)malloc(sizeof(struct BSTNode));
        if(left == right){
            newNode->data = A[left];
            newNode->left = newNode->right = NULL;
        }
        else{
            mid = left + (right-left) / 2;
            newNode->data = A[mid];
            newNode->left = BuildBST(A, left, mid-1);
            newNode->right = BuildBST(A, mid+1, right);
        }
        return newNOde;
    }
```

- Find kth smallest element in BST
  - Inorder traversal of BST produces sorted lists. While travering, keep track of the number of elements visited.

Balanced BST

Goal: reduce worst case complexity to log time by imposing restrictions on the height.

In general, the height balanced trees are represented $HB(k)$ whree k is the difference between the left subtree height and right subtree heigtht, the *load factor*.

Fully Balanced Binary Trees

These are HB(0), and hence a full binary tree.

AVL Trees

The **avl property is** :

- It is a BST
- For any ndoe X, the height of left subtree of X and height of right subtree of X differ by at most 1

```
    struct AVLNode{
        struct AVLNode *left, *right;
        int data, height;
    }
```

Rotations

When tree structure changes (insertion or deletion), we need to modify tree to keep AVL property. This can be done using single rotations or double rotations.

If AVL property is violated at Node X, it means that height of left(X), right(X) differ *exactly* by 2.

Only nodes that are on the path from the insertion point to the root might have their balances altered, because only those nodes have their subtrees altered. To restore the AVL property, we start at the insertion point, and keep going to the root of the tree.

While moving to the root, we need to consider the first node that is not satisfying the AVL property. From that node onwards, every node on the path to the root will have an issue.

If we fix the issue for the first messed up node, all the nodes onwards in the path will be restored. So we just have to fix one node.

Problems are either:

- An insertion into left subtree of left child of X
- An insertion into right subtree of left child of X
- An insertion into right subtreee of right child of X
- An insertino into right subtree of right child of X

Cases 1 and 4 are symmetric and easily solved with a single rotations. Cases 2 and 3 are symmetric and can be solved with double rotations.

```
    struct AVLnode *SingleRotLeft(struct AVLNode *X){
        // for cas #1
        struct AVLNode *W = X->left;
        X->left = W->right;
        W->right = X;
        X->height = max(Height(X->left), Height(X->right)) + 1;
        W->height = max(Height(W->left), X->height) + 1;
        return W; / new root
    }
```

SingleRotRight (case #4)is symmetric.

```
    struct AVLnode *LeftRightRot(struct AVLNode *Z){
        // for case #2
        Z->left = SingleRotateRight(Z->left);
        return SingleRotateLeft(Z);
    }
```

RightLeftRot (case #3) is symmetric.

Inserting Into AVL:

- Insert as in BST
- Depending on insertion case, call appropriate rotation

Red-black Trees

Each node is associated with an extra attribute: color, either red or block.

The *red-black property* is:

- Root property: the root is black
- External property: every leaf is black
- Internal property: children of a red node are black
- Depth property: all the leaves have the same black

Splay trees

A BST with a self-adjusting property. Any sequence of K operations from an empty tree is $Klogn$.

Augmented Trees

Add additional properties to a balanced tree, that allows us to fast compute properties of interest. Need to ensure we can maintain properties through insertion and deletion efficiently.

Interval Trees (Segment Trees)

Interval trees are binary trees that store interval information in the node structure. We maintain a Set of n intervals $[i_1, i_2]$ s.t. one of the intervals containing a query point Q (if any) can be found efficiently. Used for performing range queries efficiently.

Heap-like data structure that can be used fro making update/query operations upon array intervals in log time.

Recursive definition for a segment tree for interval $[i, j]$:

- The root (first node in array) will hold the information for interval $[i, j]$
- If $i < y$ the left and right children will hold the information for intervals $[i, \frac{i+j}{2}$ and $\frac{i+j}{2} + 1, j]$

Time complexity is $T(nlogn)$ with linear time required to build tree, and each query takes log time.

- Sort 2n endpoints in intervals
- Let X_med be the median
- Root stores intervals that cross X_mid
- Left child contains intervals that are completely to the left of X_mid
- Right child contains intervals that are completely to the right of X_mid.

Priority Queues and Heaps

Priority Queues

A priority queue ADT is a data structures that support the operations Insert, DeleteMin (or DeleteMax), that helps us find the minimum or maximum element among a collection of eleements.

Equivalent to queue push and pop, but in priority queue, the order is not determined by insertion time, but by priority.

Applications

- Data compression (huffman coding)
- Shortest path algorithms (Dijkstras)
- Minimum spanning trees (Prims)
- Event driven simulation
- Selection problem: finding kth smallest element

Note: we can implement logn insertion and deletion using balanced BST. The **Binary heap** will allow us to get logn for search, insertion, deletion, and constant time for max or min element.

Heap and Binary Heap

A heap is a tree with special properties.

- Value of a node must be $\geq$ ($\leq$) than the values of its children. This is the **heap property**
- Heap should be a complete binary tree

A **min heap**: the value of a node must be less than or equal to the values of its children (smallest at root) A **max heap**: the value of a node must be greater than or equal to the values of its children (largest at root)

Representing Heaps

```
    struct Heap {
        int *array;
        int count; // number of elements in heap
        int capacity; // size of heap
        int heap_type; // min heap or max heap
    }

    struct Heap *CreatHeap(int capacity, int heap_type){
        struct Heap *h = (struct Heap *)malloc(sizeof(struct Heap));
        h->heap_type = heap_type;
        h->count = 0;
        h->capacity = capacity;
        h->array =(int*)malloc(sizeof(int) * h->capacity));
        return h;
    }

    int Parent(struct Heap *h, int i){
        // For a node at idx i, parent is at idx (i-1)/2
        if(i <= 0 || i >= h->count) return -1;
        return i-1/2
```

```c
    }

    int LChild(struct Heap *h, int i){
        int left = 2 * i + 1;
        if(left >= h->count) return -1;
        return left;
    }

    int RChild(struct Heap *h, int i){
        int right = 2 * i + 2;
        if(right >= h->count) return -1;
        return right;
    }

    int getMax(Heap *h){
        // assume h is a max heap
        if(h->count == 0) return -1;
        return h->array[0];
    }
```

Heapifying

After insertion into heap, we may break the heap property.

- In max heap, to heapify an element, we find the max of children and swap it with the current element, and continue this process until the heap property is satisfied
- In min heap, to heapify an element, we find the min of children and swap it with the current element, and continue this process until the heap property is satisfied

```c
    void Heapify(struct Heap *h, int i){
        int l = LeftChild(h, i);
        int r = RightChild(h, i);
        int max, temp;

        if(l != -1 && h->array[l] > h->array[i])
            max = l;
        else
            max = i;
        if(r != -1 && h->array[t] > h->array[max])
            max = r;
        if(max != i){
            // swap h->array[i] and h->array[max]
            temp = h->array[i];
            h->array[i] = h->array[max];
            h->array[max] = temp;
        }
        Heapify(h, max);
    }
```

Deletion

We only support deleting from the root (either min in min heap, or max in max heap). After deleting the root element, copy the last element of the heap and delete the last element. After this replacement, heapify to preserve heap property.

```c
    int deleteMax(struct Heap *h){
        if(h->count == 0) return -1;
        int data = h->array[0];
        h->array[0] = h->array[h->count-1];
        h->count--;
        Heapify(h, 0);
        return data;
    }
```

Insertion

- Increase heap size
- Keep new element at the end of the heap
- Heapify the elemnt from bottom to top

```c
    int Insert(struct Heap *h, int data}{
        if(h->count == h->capacity) ResieHeap(h);
        h->count++;
        int i = h->count-1;
        while(i>+0 && data > h->array[(i-1)/2]{
            h->array[i] = h->array[(i-1)/2];
            i=i-1/2;
        }
        h->array[i] = data;
    }

    void ResizeHeap(struct Heap *h){
        int *array_old = h->array;
        h->array = (int *)malloc(sizeof(int) * h->capacity * 2);
        for(int i = 0;i < h->capacity; i++){
            h->array[i] = array_old[i];
        }
        h->capacity *=2;
        free(array_old);
    }
```

Heapify The Array

```c
   void BuildHeap(struct Heap *h, int A[], int n){
    // O(n) time!
    if(h == NULL) return;
    while(n > h->capacity) ResizeHeap(h);
    for(int i = 0; i < n; i++){
        h->array[i] = A[i];
    }
    h->count = n;

    // Start at first non-leaf node and heapify
    for(int i = (n-1)/2; i>=0;i--){
        Heapify(h, i)
    }
   }
```

Heapsort

Insert all elements of unsorted array into heap, then remove them from the root of a heap until the heap is empty.

Can be done in place. Instead of deleting an element, exchange the first element (maximum) with the last element, and reduce the heap size (array size). Then we heapify the first element. Continue this process unilt the number of remaining elements is one.

```
void HeapSort(int A[], int n){
 struct Heap*h = CreateHeap(n);
 BuildHeap(h, A, n);

 int old_size = h->count;
 for(int i = n-1; i > 0; i--){
     // h->array[0] is largest element
     int temp = h->array[0];
     h->array[0] = h->array[h->count-1];
     h->array[h->count-1] = temp;
     h->count--;
     Heapify(h, 0);
 }
 h->count = old_size;
}
```

Disjoint Sets

- Makeset(X): creates a new set containing a single element X
- Union(X, Y) creates a new set containing union of X, Y
- Find(X): return the name of the set containing the element X (representative)

There are two ways to implement Find and Union:

1. Fast Find (quick find)
2. Fast Union (quick union)

Quick Find Use an array. The element at index i have set name A[i]. To perform Union(a, b), where a is in set i, b is in set j, we scan the array and change all i's to j. Linear time.

Quick Union

There are three versions.

1. Slow Find
2. Quick Find
3. Quick Find with path compression

1. Quick Union Slow Find

- Use an array which stores the parent of each element. The root's parent is the root.
  - Makeset: create a new set containing a single element X, and in the array, udpate parent of X is X
  - Union: replaces the set containing X, Y by their union, and in the array, updates the parent of X as Y.
  - Find: traverse to get to the top of the tree

Problem: we can end up with large skew trees making find slow after many joins

2. Quick Union Quick Find

- Union by size: make the smaller tree a subtree of the larger tree
- Union by height: make the tree with less height a subtree of the tree with more height

A sequence of m unions and finds is $\mathcal{O}(mlogn)$

3. Path compression

We can make later FIND operations efficient by making each of the vertices point directly to the root.

This is compatible with Union by size, but not with union by height, as there is no efficient way to change the height of the tree.

Graphs Algorithms

Definition 8. A **spanning tree** of a connected graph is a subgraph that contains all of that graphs vertices and is a single tree

A **spanning forest** of a graph is the union f spanning trees of its connected components

A **bipartite graph** is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in another set

Graph Representations

Adjacency Matrix

```
        struct Graph{
         int V;
         int E;
         int **Adj; // V x V boolean matrix
        };
```

- Adj[u, v] = 1 if there is an edge from u to v, and 0 otherwise
- We can process only upper triangular in undirected graphs due to symmetry
- Good if graphs are dense. $O(V^2)$ space and $O(V^2)$ time for initialization
- Bad for sparse graphs

Adjacency List

For each vertecx v, we use a linked list and list nodes represents the connections between v and other vertices to which v has an edge:

```
        struct Graph{
         int V;
         int E;
         int *Adj; // head pointers to linked list
        };
```

- The order of edge inputs is important, since it determines the order of the vertices on the adjacancy lists
- Using disjoint sets instead of linked lists improves
- $O(E + V)$ space, and $O(deg(v))$ time to check if edge between $v \to w$ compared to constant time for adj matrix (reduce to $O(log(deg(v))$ for disjoint set adj list)
- $O(deg(v))$ time to iterate over edges incident to v, compared to $O(V)$ for adj matrix

Graph Traversals

DFS

Similar to pre-order traversal, uses a stack internally.

We encounter the following edges:

- Tree edge: encounter new vertex
- Back edge: from descendent to ancestor
- Forward edge: from ancestor to descendent
- Cross edge: between a tree of subtrees

We store vertex information :

- false: vertex is unvisited
- true: vertex is visited

Terminates when backtracking leads back to the start vertex

```
int Visited[G->V];
void DFS(struct Graph *G, int u){
    Visitied[u] = 1;
    for(int v = 0; v < G->V; ++v){

        // for each unvisited adjacent node v of u
        if(!Visited[v] && G->Ad[u][v]){
            DFS(G, v);
        }
    }
}

void DFSTraversal(struct Graph *G){
    for(int i = 0; i < G->V; ++i) Visited[i] = 0;

    // loop required if more than one connected component
    for(int i = 0; i < G->V; ++i){
    if(!Visited[i]) DFS(G, i);
}
```

- DFS traversal creates a tree (without back edges) called a DFS tree
- $O(V + E)$ with adjacency list, $O(V^2)$ with adjacency matrix
- Used for topological sorting, finding connected components, finding strongly connected components

BFS

Works similar to level order traversal of trees, uses queue as internal structure.

Initially all vertices are marked unvisited. Vertices that have been processed and removed from the queue are marked visited.

```c
    int Visited[G->V];
    void BFS(struct Graph *G, int u){
        struct Queue *Q = CreateQueue();
        Q.push(u);

        while(!Q.isempty()){
            u = Q.pop();
            //process u
            Visited[u] = 1;

            // for each unvisited adjacent node v of u
            for(int v = 0; v < G->V; v++){
                if(!Visited[v] && G->Adj[u][v]) Q.push(v);
            }
        }
    }


    void BFSTraversal(struct Graph *G){
        for(int i = 0; i < G->V; ++i) Visited[i] = 0;

        // loop required if more than one connected component
        for(int i = 0; i < G->V; ++i){
        if(!Visited[i]) BFS(G, i);
    }
```

- $O(V + E)$ if using adjacency list and $O(V^2)$ for adjacency matrix
- Find all connected components, find shortest path between two nodes

DFS has lower memory requirements than BFS since it's not required to store all the child pointers at each level. DFS goes depth first, BFS goes bredeth first, so depending on the termination condition of a problem, one might be very slow.

Topological Sort

Definition 9. Topological sort is an ordering of vertices in a directed acyclic graph in which each node comes before all nodes which it has outgoing edges.

- Every DAG may have one or more topological ordering
- Topological ordering not possible if the graph has a cycle
- If All pairs of consecutive vertices in the sorted order are connected by edges, a hamiltonian path exists
- If a hamiltonian path exists, the topological sort is unique

```c

    void TopologicalSort(struct Graph *G){
        struct Queue *Q = CreateQueue();
        int counter = 0;
        // start with all vertices with indegree 0 (no prerequisites)
        for int v = 0; v < G->V; ++v){
            if indegree[v] == 0: Q.push(v);
        }

        // pop from the queue
        while(!Q.isEmpty()){
            int v = Q.pop();
            topologicalOrder[v] = ++counter;
            // add all adjacent edges to v and push to q if such a node has indegree 0
            for each int w adjacent to v{
                if(--indegree[w] == 0) Q.push(w);
            }
        }

        if(counter != G->V) print(Graph has cycle)

    }
```

- Runtime $O(V + E)$

Shortest Path Algorithms

Suppose we want shortest path to all nodes from vertex s.

Shortest Path in unweighted graph

```

    void UnWeightedShortestPath(struct Graph *G, int s){
        struct Queue *Q = CreateQueue();
        Q.push(s);
        for(int i = 0; i < G->V; ++i) Distance[i] = -1;
        Distance[s] = 0
        while(!Q.isEmpty()){
            int v = Q.pop();
            for each w adjacent to v:
                if(Distance[w] == -1) // each vertex examined at most once
                    Distance[w] = Distance[v] + 1;
                    Path[w] = v;
                    Q.push(w); // each vertex queued at most once
        }
    }
```

- $O(V + E)$ with adjacency list, and $O(V^2)$ with adjacancy matrix

Shortest Path in weighted graph *Dijkstras*

Dijkstras is a generalization of BFS, that is greedy (almost pick the next closest vertex to the source0

- Uses a priority queue to store unvisited vertices by distance from s

- it does not work with negaitve weights

```

    void Dijkstra(struct Graph *G, int s){
        struct PriorityQueue *PQ = CreatePriorityQueue();
        PQ.push(s);
        for(int i = 0; i < G->V; ++i) Distance[i] = -1;
        Distance[i] = 0;
        while(!PQ.isEmpty()){
            int v = DeleteMin(PQ);
            for all adjacent w to v:
                // compute new distance
                int d = Distance[v] + weight[v][w];

                // we don't have a best distance so far, push into priority q
                if(Distance[w] == -1){
                    Distance[w] = d;
                    PQ.push(w, priority=d);
                    Path[w] = v;
                }

                // found a better distance, update priority queue
                if(Distance[w] > d){
                    Distance[w] = d;
                    PQ.updatePriority(w, priority=d);
                    Path[w] = v
                }
        }
    }
```

- If standard binary heap is used we have $O(ElogV)$ for E updated each taking logV
- does not work for negative edges

Shortest Path in weighted graph with negative edges: Bellman Ford

Bellman Ford computes single source shortest path in a weighted graphs, and can handle negative edges (but does not work for negative cost cycles)

```

    void BellmanFord(struct Graph *G, int s){
        struct Queue *Q = CreateQueue();


        // assume distance table is filled with INT_MAX
        for(int i = 0; v < G->V; ++i) Distance[i] = INT_MAX;
        Distance[s] = 0;
        Q.push(s);
        while(!Q.isEmpty()){
            int v = Q.pop();
            for all adjacent vertices w of v:
                // compute new distance
                int d = Distance[v] + weight[v][w];

                if(distance[w] > d){
                    distance[v] += weight[v][w];
                    path[w] = v;
                    if
                }
            }
        }
```

- $O(E \cdot V)$

Minimal Spanning Tree Prims Algorithm and Kruskals Algorithm

Prim

```

        void Prims(struct Graph *G, int s){
            struct PriorityQueue *PQ = CreatePriorityQueue();
            PQ.push(s);
            for(int i = 0; i < G->V; ++i) Distance[i] = -1;
            Distance[i] = 0;
            while(!PQ.isEmpty()){
                int v = DeleteMin(PQ);
                for all adjacent w to v:
                    // compute new distance
                    int d = Distance[v] + weight[v][w];

                    // we don't have a best distance so far, push into priority q
                    if(Distance[w] == -1){
                        Distance[w] = weight[v][w];
                        PQ.push(w, priority=d);
                        Path[w] = v;
                    }

                    // found a better distance, update priority queue
                    if(Distance[w] > d){
                        Distance[w] = weight[v][w];
                        PQ.updatePriority(w, priority=d);
                        Path[w] = v
                    }
            }
        }
```

- Running time is $O(V^2)$ without heaps (good for dense graphs)
- And $O(ElogV)$ using binary heaps (good for sparse graphs)

Kruskal

Start with V different trees, and by greedy as long as we don't add cycle

```

    void Kruskal(struct Graph *G){
        S = []; // contains edges of minimum spanning trees
        for(int v = 0; v < G->V; ++v) MakeSet(v);

        sort edges of E by increasing weights w;
        for each edge (u, v) in E{
            if(FIND(u) != FIND(v)){
                S.append((u, v));
                UNION(u, v);
            }
        }
        return S
    }
```

- O(ElogE)

Sorting

Bubble sort

The simplest algorithm works by iterating the input array from first to last, comparing each element, and swapping if needed.

```

    void BubbleSort(int A[], int n){
        for(int pass = n-1; pass >= 0; --pass){
            for(int i = 0; i <= pass - 1; ++i){
                if (A[i] > A[i+1]){
                    int temp = A[i];
                    A[i] = A[i+1];
                    A[i+1] = temp;
                }
            }
        }
    }
```

- $O(n^2)$ even in the best case

Selection Sort

An inplace sorting algorithm, works well for small files. It is used for sorting teh files with very large values and small keys. Called selection sort because it repeatedly selects the smallest element.

```

    void SelectionSort(int A[], int n){
        for(int i = p; i < n-1; ++i){
            int min = i;
            for(int j = i + 1; j < n; ++j){
                if (A[j] < A[min]): min = j;
            }

            temp = A[min];
            A[min] = A[i]
            A[i] = temp;
        }
    }
```

- $O(n^2)$ even in the best case

Insertion Sort

Each iteration removes an element from the input data, and inserts it into the correct position in the list being sorted. The choice of the element being removed is random, and the process is repeated until all input elements have been inserted.

After k iterations, the first $k+1$ entries are sorted.

```

    void InsertionSort(int A[], int n){
        for(int i = 1; i < n-1; ++i){
            int v = A[i];
            int j = i;
            while(A[j-1] > v && j >= 1){
                A[j] = A[j-1];;
                --j;
            }
            A[j] = v;
    }
```

- $O(n^2)$ even in the best case

Insertion sort is used when the data is nearly sorted (due to its adaptiveness) or when the input size is small (due to low overhead). For these reasons and due to stability, insertion sort is used as the recursive base case for higher overhead divide and conquer sorting algorithms like merge sort or quick sort.

Merge Sort

Merge sort

- Accesses data in sequential manner
- Used for sorting linked list
- Insensitive to initial order of input
- It is more stable than quicksort

```

    void MergeSort(int A[], int temp[], int left, int right){
        int mid;
        if(right > left){
            mid = (right + left) / 2;
            MergeSort(A, temp, left, mid);
            MergeSort(A, temp, mid + 1, right);
            Merge(A, temp, left, mid+1, right)
        }
    }

    void Merge(int A[], int temp[], int left, int mid, int right){
        int left_end = mid - 1;
        int temp_pos = left;
        int size = right - left + 1;

        while((left <+ left_end) && (mid <+ right)){
            if(A[left] <+ A[mid]){
                temp[temp_pos] = A[left];
                temp_pos++;
                left++;
            }
            else{
                temp[temp_pos] = A[mid];
                temp_pos++;
                mid++
            }
        }
        while(left <= left_end){
            temp[temp_pos] = A[left];
            left++;
            temp_pos++;
        }
        while(mid <= right){
            temp[temp_pos] = A[mid];
            mid++;
            temp_pos++;
        }

        for(int i = -; i <= size; ++i){
            A[right] = temp[right];
            right--;
        }
    }

```

- $O(nlogn)$ worst case, best acse, and average case

HeapSort

Although somewhat slower in practice than a good implementation of quicksort, it has the advantage of having $O(nlogn)$ runtime. It is inplace, but not stable

Quicksort

```

    void Quicksort(int A[], int low, int hight){
        if(high > low){
            int pivot = Partition(A, low, high);
            Quicksort(A, low, pivot-1);
            Quicksort(A, pivot+1, high);
        }
    }

    int Partition(int A, int low, int hight){
        int left = low;
        int right = high;
        int pivot_item = A[low]; // our pivot

        while(left < high){
            // move left while item < pivot
            while(A[left] <= pivot_item) left++;

            // move right while item > pivot
            while(A[right] > pivot_item) right--;

            if(left < right) swap(A, left, right);
        }

        // right is final position for pivot
        A[low] = A[right];
        A[right] = pivot_item;
        return right;
    }

```

- $O(n^2)$ worst case
- $O(nlong)$ best case and average case

Often we add randomization by randomly choosing the pivot element. We can improve worse case by choosing pivot as the median of a set of 3 elements randomly selected from the array.

Counting Sort

This is a linear sorting algorithm, which assumes that each of teh elements is an integer in the range 1 to K, and we assume $K = O(n)$, so the numbers aren't too big. It works by determining for each input element, the number of elements less than it.

```

    void CountingSort(int A[], int n, int B[], int K){

        int C[K];

        for(int i = 0; i < K ++i) C[i] = 0

        for(int j = 0; j < n; ++j){
            C[A[j]] = C[A[j]] + 1;
        }

        // C[i] now contains the number of elements equal to i

        for(int i = 1; i < K; ++i) C[i] += C[i-1]l;

        // C[i] not contains the number of elements <= i


        for(j = n-1; j >= 0; --j){
            B[C[A[j]]] = A[j];
            C[A[j]] = C[A[j]] - 1;
        }
    }

```

Bucket Sort

Same as Counting sort but generalizes to assume the input comes from a fixed size set (not necessarily range of integers). We need a good hash function that is ordered so that $i < k \implies hash(i) < hash(k)$. Second, the elements to be sorted must be uniformly distributed.

Radix Sort

Similar in nature this linear algorithm assumes the input values to be sorted are all d-digit numbers (base d).

First sort the elements based on the last digit. These result sare again sorted by second digit, and continue until reaching the most significant digit. Use some stable sort to sort them by last digit, then stable sort them by second least significatn digit etc.

Hashing

Balanced trees support insert, delete and search in $O(logn)$. Hashing reduced average case to constant time. This achieved by mapping a set of possible keys to a limited memory location.

Hash Table

Hash table is a generalization of an array. With an array we store the elemnt whose key is k at position k. This is called **direct addressing**. This does not work is we have more keys than locations.

A hash table stores keys and values using a hash function ot map keys to values. This is great when the number of keys actually stored is small relative to the number of possible keys.

Hash function

Hash function transforms key to index. It should map each possible key uniformly across the possible indices. We want an efficient **collision resolution** mechanism so that we can compute an alternative index for a key whose hash index is already taken.

The **load factor** is $\frac{numberofelementsinhashtable}{hashtablesize}$.

Collision Resolution

Direct chaining: seperate chaining: an array of linked lists. When two or more records hash to the same location, these records are constituted into a single linked list called a chain.

Open addressing In open addresisng all keys are stored in the hash table itself. A collision is resolved by **probing**. Open addressing cannot be used if the data does not have unique keys.

- Linear probing: we search the hash table sequentially starting from hash location, if occupied go to next. Clustering is a problem where over time hashes end up in similar regions. We choose a step size relatively prime to table size, but this can stil be bad in worst case
- Quadratic probing: solves clustering; we start at hash location i, then check $i + 1^2, i + 2^2, i + 3^3, ...$
- Double hashing: the interval between probes is computed by another hash function.

Hashing techniques

Static hashing: set of keys is kept fixed and given in advance

Dynamic hashing: data not fixed, so static hashing does not work.

Bloom filters: probabilistic data structure designed to check whether an element is present in a set. It tells us that the element either is definitely **not** in the set or **may be** in the set, using a *bit vector*.

- Start with bit vector initialized to zero
- To store a data value, apply k different hash function and set each of the resulting k index values in bit vector to 1
- Repeat for each element
- To check element, hash k times and return **not in set** if any values are off, else return **may be in set**.

String Algorithms

Check whether a pattern P is a substring of another string T.

Rabin-Karp String Matching Algorithm

Idea: check whether the hashing of P and the hashing of m characters of T give the same result.

What we need: a hash function that takes linear time to hash m characters.

KMP

Use a table called a **prefix table** that stores the knowledge about how the pattern matches against shifts of itself, to avoid useless shifts.

```

	void PrefixTable(int P[], int m){
	    int i = 1, j = 0, F[0] = 0;
	    while(i <= m){
	        if(P[i] == P[j]){
	        F[i] = j + 1;
	        ++i;
	        ++j;
	        }
	    } else if (j > 0){
	        j = F[j-1];
	    } else{
	        F[i] = 0;
	        ++i;
	    }
	}
	}

	int KMP(char T[], int n, int P[], int m){
	PrefixTable(P, m);
	while(i < n){
	    if(T[i] == P[j]){
	        if(j == m-1) return i-j;
	        else{
	            ++i;
	            ++j;
	        }
	    } else if (j > 0){
	        j = F[j-1];
	        ++i;
	    } else ++i;

	    return -1;
	}
	}
```

Boyer-Moore

Like KMP, algorithm scans cahracters of the pattern from right to left

Tries

A **trie** is a tree and each node it contains the number of pointers equal to the number of characters of the alphabet.

```

	struct TrieNode{
	    char data; // current node character
```

```
        int isEnd; // whether the string formed from root is a string
        struct TrieNode *child[26]; // pointers to other tri nodes
    }

    struct TrieNode *TrideNode subNode(struct TrieNode (root, char c){
        if(root){
            for(int i = 0; i < 26; ++i){
                if(root.child[i]->data == c) return root.child[i];
            }
        }
        return NULL;
    }
```

Trie can insert and find string in $O(L)$ time where L represenrts length of word. Much faster than hash table (no ordering) and BST. The two major methods are insert and delete.

Insertion

To insert a string we just start at root and follow the path given by prefix. Once we hit NULL, we create a skew of tail nodes for the remaining characters.

```

    void Insert(struct TrieNode*root, char *word){
        // O(length of word)
        if(root){
            struct TrieNode *newNode = (struct TrieNOde *)malloc(sizeof(struct TrieNode *));
            newNode->data = *word;
            for(int i = 0; i < 26; ++i){
                newNode->child[i] = NULL;
            }
            if(!*(word+1)) newNode->isEnd = 1;
            else newNode->child[*word] = Insert(newNode->child[*word], word+1);
            return newNode;
        }
        root->child[*word] = Insert(root->child[*word], word+1);
        return root;
    }
```

Search

```

    int Search(struct TrieNode *root, char*word){
        if(!root) return -1;
        if(!*word)
            return root->isEnd;
        if(root->data == *word) return Search(root->child[*word], word+1);
        else return -1;
    }
```

The main disadvantage of tries is they need a lot of memory.

Ternery Trees

Combines memory efficiency of BST with speed of tries.

```

    struct TSTNode{
        char data;
        int isEnd;
        struct TSTNode *left;
        struct TSTNode *eq;
        struct TSTNode *right;
    }
```

- Left pointer points to TST containing strings alphabeticall less than data
- RIght pointer points to TST containing strings alphabetically greater than data
- Eq points to TST containing strings alphabetically equal to data.

We can proint all strings in TST in sorted order via a inorder traversal.

We can find largest word in TST by finding height of TST.

Suffix Trees

We can answer queries very fast, and answer many string related queries in linear time.

Suffix tree is a tree for a single string (whereas Tries, TST store a set of strings)

The **suffix tree** of text T is a trie-like data structure that represents the suffixes of T.

- A suffix tree will contain n leaves numbered 1 to n
- Each internal node except root should have at least 2 children
- Each edge in a tree is labeled by a nonempty substring of T
- No two edges of a node begin with the same character
- Path from root to leaves represent all the suffixes of T

Construction

- Let S be the set of all suffices of T. Append $ to each
- Sort suffixes in S based on first character
- For each group if S_c has only one element, create leaf node. Otherwise find longest common prefix of suffixes in S_c, create internal node, and recurse to step 2, with S being the set of remaining suffixes form S_c after splitting off longest common prefix.

Applications

- Exact string maching
- Longest repeated substring
- Longest palindrome
- Longest common substring
- Longest common prefix