* **Linked Lists**
  + Startup
    - Nodes
      * Singly-linked lists
        + struct node{

int data;

struct node\* next;

* + - * + };
      * Doubly-linked lists
        + struct node{

int data;

struct node\* prev;

struct node\* next;

* + - * + };
      * Circular linked list
        + Pointer of the last node points to the first.

End of the list: when head->next == front.

* + - * + Must keep a link between both.
    - Head node
      * struct node\* head = (struct node \*)malloc(sizeof(struct node));
      * //Empty list
      * head = NULL;
    - Initializing a node pointer
      * struct node\* current = (struct node \*)malloc(sizeof(struct node));
      * current->next = NULL;
      * current->prev = NULL; //If doubly-linked
  + Traversal [O(*n*)]
    - current = head;
    - while (current != NULL) //End of list
      * current = current->next;
  + Access data [O(1)]
    - \*(current).data
    - current->data
  + Insertion
    - Startup
      * //Pointer to be saved
      * struct node\* current = (struct node \*)malloc(sizeof(struct node));
      * current->data = x;
      * current->next = NULL;
      * //Empty list
      * if (head == NULL) return current;
      * //Sometimes necessary
      * struct node\* front = head;//Saved front of list
      * struct node\* current2 ;//Used to save pointer
    - To front [O(1)]
      * Point current->next to head.
      * Point head to current.
      * Return head.
    - To back [O(*n*)]
      * Iterate head before head->next is NULL.
      * Point head->next to current.
      * Return front.
    - In order (low to high) [O(*n*)]
      * if(x < head->data) return current;
      * Iterate head before head->next is NULL && before head->next->data > x.
      * Point current->next to head->next.
      * Point head->next to current.
      * Return front.
  + Delete [O(*n*)]
    - Startup
      * struct node\* front = head; //Saved front of list
      * struct node\* temp; //Pointer to be deleted
      * if(head != NULL) //Included before deletions
    - First case [O(1)]
      * if(head->data == x)
        + Point temp to head->next.
        + Free head.
        + Return temp.
    - All other cases [O(*n*)]
      * Before head->next is NULL.
        + if(head->next->data == x)

Point temp to head->next.

Point head->next to head->next->next.

Free temp.

Return front.

* + - * + Iterate head to head->next.
    - Return front.
  + Reverse [O(*n*)]
    - while (nextptr != NULL) {
      * cur->next = prev;
      * prev = cur;
      * cur = nextptr;
      * nextpr = nextpr->next;
    - }
  + Recursion
    - Startup
      * struct node\* temp = (struct node\*)malloc(sizeof(struct node));
    - Insert
      * Base case: node is at the front
        + if(head == NULL || x <= head->data)

Save x to temp->data.

Point temp->next to head.

Return temp.

* + - * Recursion
        + Point head->next to recInsert(head->next, x).
        + Return head.
    - Deletion
      * Simple case: empty list
        + if(head == NULL) return NULL;
      * Base case
        + if(head->data == x){

Point temp to head->next.

Free head.

Return temp.

* + - * + }
      * Recursion
        + Point head->next = recDel(head->next, x).
        + Return head.
    - Sum all values
      * Base case: empty list
        + Return 0.
      * Return (ptr->data + sum(ptr->next));
  + Count frequency of a particular value
    - int freq(struct node\* mylist, int value){
      * int res = 0;
      * while (ptr != NULL) {
        + if (ptr->data == value) res++;
        + ptr = ptr->next;
      * }
    - }
  + Check if sorted
    - int isSorted(struct node\* ptr) {
      * if (ptr == NULL) return 1;
      * while (ptr->next != NULL) {
        + if (ptr->data > ptr->next->data) return 0;
        + ptr = ptr->next;
      * }
      * return 1;
    - }
* **Queues**
  + Array
    - Implementation
      * struct queue {
        + int\* elements; //Array of elements
        + int front; //Number at front of queue
        + int numElements; //Number of elements
        + int queueSize; //Max queue size
      * };
    - Setup
      * struct queue\* myQ = (struct queue\*)malloc(sizeof(struct queue));
      * myQ->elements = (int\*)malloc(sizeof(int)\*INIT\_SIZE);
      * myQ->front = 0;
      * myQ->numElements = 0;
      * myQ->queueSize = INIT\_SIZE;
    - Full: myQ->numElements == myQ->queueSize
    - Empty: myQ->numElements == 0
    - Enqueue [O(1)]
      * If not full
        + myQ->elements[(myQ->front + myQ->numElements) % myQ->queueSize] = x;
        + Increment myQ->numElements.
      * If full
        + //Reallocate more memory
        + myQ->elements = (int\*)realloc(myQ->elements, (myQ->queueSize)\*sizeof(int) \* 2);
        + //Copy values
        + for(i = [0, myQ->front)

myQ->elements[i+myQ->queueSize] = myQ->elements[i];

* + - * + myQ->elements[i+myQ->queuesize] = x;
        + (myQ->queueSize) \*= 2;
        + Increment myQ->elements.
    - Dequeue [O(*n*)]
      * If empty, return EMPTY.
      * Store myQ->elements[] at index front in retval.
      * Adjust myQ->front to (myQ->front + 1)%myQ->queueSize.
      * Decrement numElements.
      * Return retval.
    - Breadth First Search [O(*n*)]
      * A counter of steps starts at 0.
      * Nodes visited in order.
      * Each node enqueues adjacent spots that are valid and have not yet been visited.
      * The node is dequeued and marked as visited by the number of steps to get to it.
  + Linked List
    - Implementation
      * struct node {
        + int data;
        + struct node\* next;
      * };
      * struct queue {
        + struct node\* front;
        + struct node\* back;
      * };
    - Setup
      * struct queue\* myQ = (struct queue\*)malloc(sizeof(struct queue));
      * myQ->front = NULL;
      * myQ->back = NULL;
      * struct node\* temp = (struct node\*)malloc(sizeof(struct node));
      * //Check that memory was allocated correctly
    - Empty: myQ->front = NULL
    - Front: myQ->front->data
    - Enqueue [O(*n*)]
      * Store x in temp->data.
      * Set temp->next to NULL.
      * If myQ->back != NULL, set myQ->back->next to temp.
      * Set myQ->back to temp.
      * If myQ->front == NULL, set myQ->front to temp.
    - Dequeue [O(1)]
      * If empty, return EMPTY.
      * Store myQ->front->data in retval.
      * Set temp to myQ->front.
      * Set myQ->front to myQ->front->next.
      * If myQ->front == NULL, set myQ->back to NULL.
      * Free temp.
      * Return retval.
* **Stacks**
  + Array
    - Implementation
      * struct stack {
        + int items[SIZE]; //Stack array
        + int top; //Index to top of stack

//-1: empty

//SIZE: full

* + - * }
    - Setup
      * struct stack\* stackptr = (struct stack\*)malloc(sizeof(struct stack));
      * stackptr->top = -1;
    - Full/Empty
      * Empty: top = -1
        + Evaluate if (stackptr->top == -1).
      * Full: top = SIZE – 1
        + Evaluate if (stack->top == SIZE - 1).
    - Top [O(1)]
      * If empty, return EMPTY.
      * Return items[] at top index.
    - Push [O(*n*)]
      * If full, return 0;
      * Insert x into stackptr->items[] at index top.
      * Increment top index.
      * Return 1.
    - Pop [O(1)]
      * If empty, return -1.
      * Save items[] at top index to retval.
      * Decrement top index.
      * Return retval.
  + Linked List
    - Implementation
      * struct stack {
        + int data;
        + struct stack\* next;
      * };
    - Setup
      * struct stack\* stackptr = (struct stack\*)malloc(sizeof(struct stack));
      * //Pass &stackptr into functions as a double-pointer
      * //Ensure memory was allocated dynamically before continuing
      * struct stack\* temp = (struct stack\*)malloc(sizeof(struct stack));
      * //Init
      * &(\*stackptr) = NULL;
    - Empty: stackptr == NULL.
    - Top
      * If empty, return -1.
      * Else, return stackptr->data.
    - Push [O(*n*)]
      * Save x into temp->data.
      * Point temp->next to \*(&stackptr).
    - Pop [O(1)]
      * Point temp to NULL.
      * Point temp to \*(&stackptr).
      * Point \*(&stackptr) to \*(&stackptr)->next.
      * Point temp->next to NULL.
      * Return temp.
  + Operator stacks
    - Read an operand 🡪 push onto stack
    - Read an operator 🡪 pop last 2 items off the stack op2, followed by op1. Calculate op1 op op2 and push this value onto the stack.
      * If you ever try to pop an empty stack, the expression is invalid postfix expression.
      * If after running operations you end up with a stack size > 1, it’s also invalid.
        + Should only end up with 1 answer.
  + Infix to postfix
    - Open parenthesis 🡪 push onto stack
    - Operand 🡪 place into expression in order.
    - Close parenthesis 🡪 pop items off stack, placing each in the expression until we hit the first open parenthesis.
    - Operator 🡪 pop off the stack each operator of equal or higher precedence, placing each into the expression.
      * Stop popping when you reach…
        + an operator of lower precedence.
        + a parenthesis.
        + the end of the stack.
      * Push this operator onto the stack.
    - End 🡪 pop off remaining operators and place in the expression.
* **Binary Trees**
  + Implementation
    - struct tree\_node {
      * int data;
      * struct tree\_node\* left;
      * struct tree\_node\* right;
    - };
  + Setup
    - struct tree\_node\* root = (struct tree\_node\*)malloc(sizeof(struct tree\_node));
    - root->data = VALUE;
    - root->left = NULL;
    - root->right = NULL;
    - struct tree\_node\* element;
    - struct tree\_node current\_ptr;
  + Empty: root == NULL.
  + Traversal [O(*n*)]
    - Inorder: left🡪root🡪right
      * if (current\_ptr != NULL) {
        + inorder(current\_ptr->left);
        + //access current\_ptr->data
        + inorder(current\_ptr->right);
      * }
    - Preorder: root🡪left🡪right
      * if (current\_ptr != NULL) {
        + //access current\_ptr->data
        + preorder(current\_ptr->left);
        + preorder(current\_ptr->right);
      * }
    - Postorder: left🡪right🡪root
      * if (current\_ptr != NULL) {
        + preorder(current\_ptr->left);
        + preorder(current\_ptr->right);
        + //access current\_ptr->data
      * }
  + Search
    - If empty, return element.
    - //Go left
    - If (x < root->data)
      * Return search(root->left, x);
    - //Go right
    - If (x>root->data)
      * Return search(root->right, x).
    - Return 1.
  + Insert
    - if empty, return element.
    - //Item should be inserted to the right
    - if (element->data > root->data)
      * if (root->right != NULL)
        + Set root->right to insert(root->right, element);
      * else
        + Set root->right to element.
    - //Item should be inserted to the left
    - else
      * If(root->right != NULL)
        + Set root->left to insert(root->left, element).
      * Else
        + Set root->left to element.
    - Return root.