1. Always draw pictures of what is going on for tracing, linked lists, and binary trees.
2. **Linked list**
   1. Code
      1. typedef struct Node {
         1. int data;
         2. struct Node\* next;
      2. };
      3. struct Node\* list = NULL;
   2. Diagram
      1. list🡪[3][]🡪[2][]🡪[7][x]🡪NULL
   3. Always check if head pointer is NULL
3. Accessing nodes
   1. Code
      1. Struct Node\* help\_ptr = NULL;
      2. help\_ptr = mylist;
      3. //Access help\_ptr->data;
      4. help\_ptr = help\_ptr->next;
4. Structural changes
   1. Add to front
      1. Steps
         1. Create a node storing the element to insert.
         2. Attach this node to the rest of the list.
         3. Return a pointer to this newly created node.
      2. list is limited to changing values in nodes and changing list->next ptrs.
   2. Add to back
      1. Steps
         1. Create a node storing the element to insert.
         2. Use a temporary pointer to iterate to the last node of the list.
         3. Attach this last node to the newly created node.
         4. Return a pointer to the front of the original list.
      2. Keep in mind the case of the empty list.
   3. Add in order
      1. Steps
         1. Create a node storing the element to insert.
         2. Use a temporary pointer to iterate through the list, making sure to stop at the node RIGHT BEFORE(call this x), the insertion needs to be made.
         3. Save a pointer to the node RIGHT AFTER(call this y) where the inserted node needs to be placed.
         4. Attach x to the newly created node.
         5. Attach the newly created node to y.
   4. Reverse
      1. Steps
         1. Recursively
            1. Recursively reverse rest of the list
            2. Iterate to the end of new list
            3. Attach end of reversed list to 1st item in original list
            4. Return new front
         2. Iteratively
            1. while (nextptr != NULL) {

cur->next = prev;

prev = cur;

cur = nextptr;

nextpr = nextpr->next;

* + - * 1. }
    1. Base case
       1. Size 1
          1. list->next == NULL
       2. Size 0
          1. list == NULL
  1. Delete
     1. Steps
        1. 1) Use a temporary pointer to iterate through the list, stopping at the node RIGHT BEFORE the one storing the value to delete.
        2. 2) Store pointers to the node to delete.
        3. 3) Patch the node RIGHT BEFORE the deleted node to the one RIGHT AFTER it.
        4. 4) Free the memory for the deleted node.

1. Non-structural changes
   1. Print
      1. Code
         1. void print(struct node\* mylist){
            1. while(mylist != NULL){

printf(“%d”, mylist->data);

mylist = mylist->next; //Advance to next element

* + - * 1. }
      1. }
      2. void printRec(struct node\* mylist) {
         1. if (ptr != NULL){

print(“%d”, ptr->data);

printRec(ptr->next);

* + - * 1. }
      1. }
  1. Sum all values
     1. Code
        1. int sumRec(struct node\* mylist){
           1. if (mylist == NULL) return 0;
           2. return (ptr->data + sum(ptr->next));
        2. }
  2. Count frequency of a particular value
     1. Code
        1. int freq(struct node\* mylist, int value){
           1. int res = 0;
           2. while (ptr != NULL) {

if (ptr->data == value) res++;

ptr = ptr->next;

* + - * 1. }
      1. }
  1. Check if sorted
     1. Code
        1. int isSorted(struct node\* ptr) {
           1. if (ptr == NULL) return 1;
           2. while (ptr->next != NULL) {

if (ptr->data > ptr->next->data) return 0;

ptr = ptr->next;

* + - * 1. }
        2. return 1;
      1. }