1. **Lists**: sequence of similar elements in some order.
   1. No constraints upon where to start/delete its elements.
   2. Operations
      1. Important
         1. Create a list
         2. Search for an item in the list
         3. Insert a new item into the list
            1. Ordered 🡪 in a specific spot
            2. Not ordered 🡪 at the beginning
         4. Delete an item from a list
      2. Less important
         1. Count number of items in list
         2. Retrieve data in item in list
         3. Change value of data in list
         4. Clear list
         5. Determine if list is full/empty
         6. Is an item the first/last on the list?
   3. Types
      1. Contiguous Lists (Arrays)
         1. Advantages
            1. Pre-defined structure in most languages
            2. Random cells can easily be…

Accessed

Declared

Defined

* + - * 1. No pointers
      1. Disadvantages
         1. If an item is to be inserted at the beginning, room must be made by shifting items over one by one.
         2. Deleting leaves a “hole” of wasted memory.
         3. Array size must be determined at compile time.

Set aside extra memory

Wastes memory when array is not fully populated

Can crash program or overwrite old data if insufficient memory is allocated

* + - * 1. Finding a block of contiguous memory large enough may sometimes be difficult.
    1. Linked Lists (Dynamically-allocated memory blocks)
       1. Grow easily, shrink easily, and only use the amount of memory needed.
       2. No need to know how much memory a priori.
       3. Advantages
          1. Insertions in middle do not require making room
          2. Deletions do not leave holes
       4. Disadvantages
          1. Expensive random access

Cannot access a particular cell in one operation as in arrays

Nodes require more memory because of pointer in every node

Most operations require traversal of list and can be expensive

* + - * 1. Linear

Cannot access items randomly

Traversals begin from head to tail

* + - 1. Created by linking chunks through malloc()
         1. Memory allocated at runtime
         2. Memory deallocated at runtime through delete and free operators (C and C++ respectively).
      2. Implementation
         1. Link blocks allocated dynamically through pointers.
         2. **Node**: each element of the list; complex structs containing pointer to structure of same data type.

Data type defined by typedef

* + - * 1. Each pointer that is a member of a node actually points to the next pointer.

**Head**: first node.

Pointed at by **head pointer**

Can be labeled as the name of the list

**Tail**: last node; always points (and contains) NULL.

NULL: indicated by a large X or electrical ground symbol.

* + - * 1. Process

Head 🡪 element 1 🡪 … 🡪 NULL

* + - 1. Doubly-Linked Lists: allow forward and backward movements through a list.
         1. Make insert-before = insert-after
         2. Singly-linked lists achieve this by starting from the beginning again and again.
         3. Require more memory.
         4. Only used for special purposes.

1. Operations on a List
   1. search(): involves *traversal* of list item by item from start to end to look for something and process it.
      1. Means for permanent storage
      2. Looking for a particular item is easier if the list is ordered and not random.
      3. Processing of data
         1. Delete
         2. Modify
         3. Retrieve
         4. Insert
         5. Print
         6. Define a node in a list as a self-referencing structure
            1. struct node {
            2. int data;
            3. struct node \* next;
            4. };
            5. //Self-referencing structure - pointer next points to a data type of itself (“node”)
            6. //C makes an exception, but the data type must be defined before the end of the program
   2. Traversal: going from 1 node to the next, following next pointers.
      1. Process
         1. Have a pointer that points to the node you are currently at.
            1. current = head;
         2. Going to the next node
            1. current = current->next;
         3. When current->next == NULL, you have reached the end of the list.
   3. Creation
      1. struct node {
      2. int data;
      3. struct node \* next;
      4. };
      5. typedef struct node Node;
      6. Node \* head = NULL;
   4. Insertion
      1. Requires traversal of list to look for location where insertion is to occur.
      2. Create 3 pointers
         1. newnode: utility pointer; points to node being inserted.
         2. prior: points to node before where insertion is to happen.
         3. current: points to node after where insertion is to happen.
         4. prior and current always move in tandem in the traversal, with prior trailing current by one node.
      3. Locations
         1. Inserting into an empty list
            1. Create a block of memory
            2. Point to it with newnode pointer

newnode->next //pointer points to NULL

head = newnode; //to insert first item

* + - 1. Inserting at the beginning
         1. Set newnode->next to point to the same as head
         2. Set head to same address as newnode
      2. Inserting in the middle
         1. Insert after the node pointed to by current.

newnode-> next points to where current->next now points.

current->next is then shiften to point to where newnode points

* + - 1. Inserting at the end
         1. Identical to inserting at the middle
         2. current-> next points to NULL instead of another node
         3. Simply set newnode->next to NULL instead of to where current->next was pointing
  1. Deletion
     1. Find the item to be deleted.
     2. Item is deleted, and pointers are reconnected.
        1. Maintain a pointer pointed to the node being deleted so that the memory in which it is located can be freed up.
     3. Locations
        1. Deleting from the front
           1. Pointer to temp points to node being deleted, which is where head would have pointed
           2. head pointer is then set up to point to head->next (the second node)

head = head->next;

* + - * 1. Free the node you removed that is being pointed to by temp.

free(temp);

* + - 1. Deleting from the middle
         1. Need to know the item to be deleted (pointed at by temp) and the item just before it (pointed at by crnt)

crnt->next = temp->next

free(temp);

* + - 1. Deleting from the end
         1. Same as deleting from the middle
         2. Second node is NULL

crnt->next = NULL