void iterativeReverse(sllist *linkedList) {

    if (linkedList == NULL || linkedList->head == NULL) {
        return;
    }

    node *prev, *curr, *next;
    prev = NULL;
    next = NULL;
    curr = linkedList->head;

    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }

    linkedList->head = prev;
}
typedef struct _student {
    char *firstName;
    char *lastName;
    int age;
    float gpa;
} student;

typedef struct _listElement {
    student *data;
    struct _listElement *next;
    struct _listElement *prev;
} listElement;

typedef struct _llist {
    listElement *head;
    listElement *tail;
} llist;
The efficiency of insert and find

• How can we keep track of whether or not we’ve seen a number?
• First case: we know the range of the values in question (e.g. an integer between 0 and 25)
• How efficient would it be if we used a linked list to keep track of our numbers?
• What’s a better way (assuming range bound on the number)?
  – If you know the range, just have an array of length 26 (with index values from 0 through 25) and update the corresponding cell when you see the corresponding value appear.
• What if you no longer assume the user will type a number within a specified range? Say we’re only restricted to non-negative numbers.
  – That would require an awful lot of memory given our array approach.
  – Also, numbers might be in a big range, but the numbers are sparse (say 20, 100 total max)
Hash tables

- Hash table is a data structure that associates *keys* with *values*.
- Primary operation that hash tables support efficiently is a lookup: given a key, (say a person’s name), return the corresponding value (say the person’s email address).
- A *hash function* takes the key and transforms it into a *hash* or *hash value*, a number that’s used to index into an array location (“bucket”) where the value should be.
Hash functions and collisions

• The hash function is usually chosen to have pseudo-random properties, so that small changes of a key give a large and seemingly random effect on the hash value returned.

• Due to this random effect, the calculated index can sometimes be the same for two different keys (a collision)

• Hash tables support the efficient lookup, insertion and deletion of elements in constant time on average (O(1)) that does not vary with the number of elements stored in the table; although this may vary somewhat depending on how full the table is.
Open addressing

- **Open addressing or closed hashing** is a method of resolving collisions by *probing*.
- If a collision happens, probe array for the next available space.
- Example of *linear probing*:
  - \( h(key1) = 2 \)
  - \( h(key2) = 2 \) but location 2 is full. So go to next empty location and store key2 there.
- Other types of methods of probing: quadratic probing, double hashing (using a second hash function).
- All methods of open addressing become linear in time as the hash table approaches maximum capacity. The only solution to this is to rehash to a larger table.
The Birthday Problem

A graph showing the probability of at least two people sharing a birthday among a certain number of people.


<table>
<thead>
<tr>
<th>$n$</th>
<th>$p(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12%</td>
</tr>
<tr>
<td>20</td>
<td>41%</td>
</tr>
<tr>
<td>23</td>
<td>50.7%</td>
</tr>
<tr>
<td>30</td>
<td>70%</td>
</tr>
<tr>
<td>50</td>
<td>97%</td>
</tr>
<tr>
<td>100</td>
<td>99.99996%</td>
</tr>
</tbody>
</table>
Separate chaining

- One simple way of dealing with collisions: have each array index point to a list.
- If there’s a collision, just add the key to the linked list at the array index.
- If there’s a uniform distribution, none of these chains will be much longer than the others.
- Given a bad hash function, chains in theory can get very long (search time increases to $O(n)$ in the worst case)
- Advantages: removal operation is simple and resizing the table can be postponed for a much longer time.
- Given a good distribution, what’s the running time given a hash function $h(k) = k \% 26$?

Figure from Lewis and Denenberg’s *Data Structures and Their Algorithms*
Coalesced hashing (coalesced chaining)

- Say we have a key, “J. Smith” that we want to add to our table, and it hashes to index 19.
- We would find the first open bucket in our table at index 18 (we’d keep track of where the last open space was so we don’t have to keep searching from the beginning of the table every time).
- “J. Smith” would go in index 18.
- Then we traverse the list starting at the original hash index 19 (1, 7, 8, and 9 in that order).
- When we reach 9, we change the uninitialized pointer there to point to our new value at index 18.
Note on hash functions

• Good hash functions
  – Quick to compute
  – Produces all possible indices
  – Minimizes collisions
• The size of the table should be a prime number (they work better for technical reasons).
• For integer keys, \( h(key) = key \mod \text{primeNumber} \), where \text{primeNumber} is the size of the array, is typically used.
• For string keys, some method of converting the string to an integer is usually used, followed by taking \( \text{result} \mod \text{primeNumber} \) to reduce the integer value down to an index within the table.
Binary search trees

- Other data structures
  - Branching data structures
  - Binary search tree, for example (logarithmic performance)
Binary tree example

• Exercise: building a binary tree
• Printing out a binary tree

See tree.c
Tries

- Tries (short for retrieval)
- Allows you to implement lookups in a hash table for $O(L)$ where $L$ is the length of the string you’re storing in a dictionary
- Implement a tree, each of whose nodes is an array of size 26.

Figure from Lewis and Denenberg's *Data Structures and Their Algorithms*