Linked List

Derrick

So far

- We’ve only had one kind of data structure for representing collections of like values.
  - structs give us “containers” for holding variables of different data types.

- Arrays are great for element lookup, but unless we want to insert at the very end of the array, inserting elements is quite inefficient. (shifting) Also problem with resizing...

Idea

- Through clever use of **pointers**, **dynamic memory allocation**, and **`struct`**s, we can put the pieces together to develop a new kind of data structure that gives us the ability to grow and shrink a collection of like values to fit our needs.

Singly-Linked Lists

- We call this combination of elements, when used in this way, a **linked list**.
- A linked list **node** is a special kind of struct with two members:
  - Data of some data type (`int`, `char`, `float`, `…`)
  - A pointer to another node of the same type
- In this way, a set of nodes together can be thought of as forming a chain of elements that we can follow from beginning to end.

`typedef`

The `typedef` is an advance feature in C language which allows us to create an alias or new name for an existing type of user defined type.

Syntax : `typedef data_type new_name;`

Example : `typedef char * string;`

Singly-linked lists

```
typedef struct sllist
{
    TYPE val;
    struct sllist *next;
} sllnode;
```

Operations

- In order to work with linked lists effectively, there are a number of operations that we need to understand:
  - ***Create*** a linked list when it doesn't already exist.
  - ***Search*** through a linked list to find an element.
  - ***Insert*** a new node into the linked list.
  - ***Delete*** a ***single*** element from a linked list.
  - ***Delete*** an ***entire*** linked list.

Create

```
sllnode *create(VALUE val);
```

- Dynamically allocate space for a new `sllnode`.
- Check to make sure we didn't run out of memory.
- Initialize the node's `val` field.
- Initialize the node's `next` field.
- Return a pointer to the newly created `sllnode`.

Search

```
bool find(sllnode *head, VALUE val);
```

- Create a traversal pointer point to the list's `head`. (we don't want to change the head).
- If the current node's `val` field is what we're looking for, return `true`.
- If not, set the traversal pointer to the next pointer in the list.
- If you've reached the end of the list, return `false`.

Insert

```
sllnode *insert(sllnode *head, VALUE val);
```

- Dynamically allocate space for a new sllnode.
- Check to make sure we didn't run out of memory.
- Populate and insert the node *at the beginning of the linked list*.
- Return a pointer to the new head of the linked list.

Delete

- Delete an entire linked list.

```
void destroy(sllnode *head);
```

- If you’ve reached a null pointer, stop.
- Delete ***the rest of the list***.
- Free the current node.

```
Recursion!
```

Delete a single element

- Delete a single element from a linked list.

  How?

  << Exercise >>

Arrays vs Linked Lists

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]