CS50 Lecture 5: Data Structures

Jordan Mandel

2021_08_04

Array Example

In this case we are allocating more memory and copying the contents of the old memory over and then we are freeing the memory that list originally pointed to. Note: `NULL` is actually `0x0`

```
#include <stdio.h>
#include <stdlib.h>
#can't resize a statically allocated array;

int main(void)
{
    int *list = malloc(3 * sizeof(int));
    //could be int list[3];
    if (list == NULL)
    {
        return 1;
    }

    list[0] =1;
    list[1] = 2;
    list[2] = 3;

    int *tmp = malloc(4 * sizeof(int));
    // if tmp is null free list and return

    for (int i = 0; i < 3; i++)
    {
    tmp[i] = list[i];
    }

    tmp[3] = 4;
    free(list);
    list = tmp;
    //print the list;
    free(list);
}
```

We can reallocate memory using realloc; can't do this if you statically declare the array

```
int *tmp = realloc(list, 4 * sizeof(int));
```

Linked Lists

In this type definition we include 'node' both at the begining and at the end; supposedly this allows us to refer to the struct name in the definition itself; allows for recursion.

Recall that malloc gives you the first address of a contiguous block of memory. you say `node* list = NULL` to give it `NULL` as opposed to garbage.

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct node \\ we use the word node twice here
{
    int number;
    struct node *next;
}
node;

int main(void)
{
    node *list = NULL; // can't leave it as a garbage value
    node *n = malloc(sizeof(node)); //this is just a temporary; try to implement without?
    if (n == NULL)
    {return 1;}

    n -> number=1;
    n -> next = NULL;
    list = n;

    n = malloc(sizeof(node));
    if(n == NULL)
    {
        free(list);
        return 1;
    }
    n->number = 2;
    n->next = NULL;

    list->next=n;
    n = malloc(sizeof(node));
    if (n==NULL)
    {
        free(list->next);
        free(list);
        return 1;
    }
    n->number=3;
    n->next=NULL;
    list->next->next=NULL

        for (node *tmp = list; tmp != NULL; tmp->next) //print list
        {
            printf("%i\n, tmp->number);
        }

    while (list != NULL)
    {
        node *tmp = list ->next;
        free(list);
        list=tmp;
    }
}
```

Now we have an example of inserting a node into the middle of a linkedlist:

```
node *n = malloc(sizeof(node));
if(n != NULL){

n->number=1;
n->next=NULL;
}
```

```
//now want to connect it to the linkedlist: make sure not to orphan
n->next=list;
list=n;
// we can find a way to add it to the middle of the linked list
```

`a->b` is the same as `(*a).b`

1. `node* list = NULL`
   `list ---> NULL(node: (number, *next))`
2. `node* n = malloc(sizeof(node))`
   `list ---> NULL(node: (number, *next)), n ---> node(number, *next)`
3. `n->number=1,n->next=NULL`
   `list ---> NULL(node: (number, *next)), n ---> node(number = 1, *next = NULL)`
4. `list = n`
   `(list, n) ---> node(number = 1, *next = NULL)`
5. `n = malloc(sizeof(node))`
   `(list) ---> node(number = 1, *next = NULL), n ---> node(number , *next )`
6. `n->number = 2, n ->next = NULL`
   `(list) ---> node(number = 1, *next = NULL), n ---> node(number = 2, *next = NULL)`
7. `List->next=n`
   `(list) ---> node(number = 1, *next = n), n ---> node(number = 2, *next = NULL)`
8. `n = malloc(sizeof(node))`
   `(list) ---> node(number = 1, *next = [second]), [second] ---> node(number = 2, *next = NULL)  n ---> node(number, *next)`
9. `n->number=3,n->next=NULL`
   `(list) ---> node(number = 1, *next = [second]), [second] ---> node(number = 2, *next = NULL)  n ---> node(3, NULL)`
10. `list->next->n`
    `(list) ---> node(number = 1, *next = [second]), [second] ---> node(number = 2, *next = n)  n ---> node(3, NULL)`

Inserting at the beginning of a linked-list is O(1).

These are shown above: Then we can print each element of the linked list:

```
for (node *tmp = list; tmp!=NULL; tmp = tmp->next)
{
    printf("%i\n", tmp->number
}
```

We can use a while loop to free the list.

```
while (list != NULL)
{
    node *temp = list->next;
    free(list);
    list=tmp;
}
```

Trees

- With an array we can use binary search
- In a treee every node points two two other nodes: a left node and a right node; the left one is smaller
- The tree object itself is a pointer to the root (as opposed to the 'beginning' of a linkedlist)
- Here is how we define it:

```
1 2 3 4 5 6 7
```

```
       4
   2       6
 1   3   5   7
```

```
typedef struct node
{
    int number;
    struct node *left
    struct node *right
}
```

- We can recursively search a tree

```
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else
    {
        return true;
    }
}
```

- adding in a leaf can be done but we want to keep the binary tree balanced; there are more advanced datastructures and algorithms to do this
- can find a way to add a point
- adding in a node is O(log(n))

Other data structures

Hash Table

- A hash table is an array of linked lists.
- A hash function tells which index we put it in: for example by letter of the alphabet. so for the harry potter example they take a string and outputs an index
- Trade off of time and memory based on how specific our hash function is.
- We just append to the end of whatever linked list is in the index of the hash table.
- Hash tables are O(n) because the average number of things in buckets increases linearly; but memorywise they are expnonential; in real world it will be ~constant if you do it right

Trie

- A tree with arrays as nodes; each array holds an alphabet
- Every valid letter will point to a child alphabet, and every valid next letter will point to yet another alphabet. Any last letters will point to `true`.
- Fast but takes up a lot of memory

Others

- queue (enqueue, dequeue) stack (push, pop) ; we use a linked-list
- dictionary maps keys to values: could do this with a hash table or an array

- sometimes the thing that is theoretically less efficient (asymptotically) is more efficient in the real world

Linked List Short

Creation of Node in Linked List

a. Dynamically allocae space fora new sllnode

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

b. Check for null

c. initialize `val` field

d. initialize `next` fielda.

e. return pointer to the new `sllnode`

Finding a Value in a Linked List

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

Might want to always try keeping first pointer global

a. create traversal pointer
b. compare to val field
c. if not go to next
d. if we reach the end then we didn't find it.

Might call `bool exists = find(list, 6);`

Insert Node Into List

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

a. Allocate space for new node
b. Check to make sure we didn't run out of memory
c. Populate and isert node at the beginning fo the list
d. Return pointer to new head of the list

start by pointing new node to head of list

Destroy List

'void destroy(sllnode* head);

a. stop at `null`
b. delete rest of list
c. free current node

Can do this recursively

Delete a single node

Difficult: we have to use a doubly linked list

Hash Table Video

Has a hash function and an array

```
int x = has("John");
hashtable[x] = "John"
int y= hash("Paul");
hashtable[y] = "Paul";
```

A hash function should - only and all of the data being hashed - be determinitic - uniformly distribute data - generate very different codes for similar data

Example of bad hash function:

```
unsigned int hash(char* str)
{
    int sum= 0;
    for (int j =0; str[j] != '\0`; j++)
    {
        sum += str[j];
    }
    return sum % HASH_MAX;
}
```

But what about hash collision?
To solve we use linear probing; we just store at the next available location
This problem is called clustering when we have to do a lot of linear probing.

A better solution is ***chaining***

Trie Video

```
typedef struct_trie
{
    char university[20];
    struct_trie* paths[10];
}
trie;
```

Globally declare root Set a pointer called trav to root node