Notes made while reading Kalicharan's Advanced C

Table of Contents

Source can be found at https://github.com/mtn/kalicharan-adv-c-notes

Generated using Asciidoctor.

5 Stacks and Queues

5.1 Abstract Data Types

We refer to implementations of data types whose operations can be performed by users without knowledge of the underlying implementation.

5.2 Stacks

A **stack** is a linear list where items are added and deleted from the same end. This means that stacks exhibit a "last in, first out" property. As an example, consider the following set of numbers:

```
36 15 52 23
```

If we want to print the following:

```
23 52 15 36
```

We would need to construct the following stack:

```
(top) 23 52 15 36 (bottom)
```

And we would remove numbers one at a time, printing them as we remove them. In psuedocode:

```
initialize empty stack S
read(num)
while a next number, num was read:
    push num onto S
    read(num)
while S is not empty:
    num = pop S
    print num
```

5.2.1 Implementing a Stack Using an Array

Stacks can be implemented with a struct containing an array and an int index of the top of the stack:

```
typedef struct {
    int top;
    int ST[MaxStack];
} StackType, *Stack;
```

Note that having StackType is useful for allocating a stack (general useful practice). When initialized, we'll set top to -1. Considering a stack variable, `Stack S`, we have access to `top` and the stack array, as we should based on our above definition of ADTs. Declaring a stack variable is therefore pretty simple:

```
Stack initStack(){
    Stack sp = malloc(sizeof(StackType));
    sp->top = -1;
    return sp;
}
```

And a user would simply initialize a new stack with `Stack S = initStack()`.

TODO Finish stack section

5.6 Queues

11 Hashing

11.1 Hashing Fundamentals

Hashing is a fast method for searching for an item in a table where each item's key determines its placement. Keys are converted to numbers (if non-numeric) and then **hashed** (mapped) to a table location. If multiple keys hash to the same location, we have a **collision**, which we must resolve.

11.1.1 The Search and Insert Problem

Problem statement: Given a (possibly empty) list of items, search for a given item. If not found, insert it.

There are many ways we can implement the list:

1. As an array, where integers are placed in the next available position and searching is sequential. Fast insertion by slow searching.
2. As an array, where each item is inserted such that the array remains sorted. Searching is faster than (1), but insertion is slower.
3. As an unsorted linked-list, which must be searched sequentially.
4. As a sorted linked list

Alternatively, we can use **hashing**, which allows for fast search *and* easy insertion.

11.2 Solving the Search and Insert Problem by Hashing

Suppose we have 12 spots (indexed 1-12) to place numbers and we want to insert 52. A simple hash function might be `x % 12 + 1`, where 52 would be placed in slot 5. If we try to add 16 next, however, we have a collision. One option (*linear probing*) is to just place it (16) in the next open spot. Our searching is thus slightly more complicated:

- Apply our hash function and check the slot
- If the slot is occupied by a different number, try the next location. Keep going until it's found or an empty slot is found.

Roughly as code:

```
loc = hash(key)
while(num[loc] && num[loc] != key)
    loc = loc % n + 1 //n == number of slots
if(!num[loc]){
    num[loc] = key;
}
```

Note that the above while loop never terminates if that table is full (which we never allow happen in practice). Also, rather than `!num[loc]`, we'll assign a default 'empty' value (ex. -1 or 0).

11.2.1 The Hash Function

Looking beyond the previous example, sometimes we encounter keys that are non-numeric. When handling these, we first need to convert them to some numeric value. Considering strings, we can add up numeric char values:

```
int h = 0, wordNum = 0;
while(word[h] != '\0') wordNum += word[h++];
loc = wordNum % n + 1; //loc between 1 and n
```

The problem here is that words with the same letters hash to the same location. To counter this, we can apply a weight to each letter based on position in the word.

```
int h = 0, wordNum = 0;
while(word[h] != '\0'){
    wordNum += w * word[h++];
    w = w + 2;
}
loc = wordNum % n + 1;
```

Generally, we want to spread our data out in the table but avoid having a costly function.

Deleting an Item from a Hash Table

We can't simply delete values, or our resolved collisions might erroneously not be found. Instead, we can *mark* values as deleted (ex. -1).

11.3 Resolving Collisions

The above collision-resolution process is an example of **linear probing**. Other options include **quadratic probing**, **chaining**, and **double hashing**.

11.3.1 Linear Probing

Essence: `loc += 1`. Downsides: Clustering (long chains tend to get longer, and connect to other chains). *Primary clustering* is when keys hash to different locations but trace the same path looking for an empty location (ex. keys that hashes to 5 and 6). *Secondary clustering* is when keys that hash to the same location trace the same path.

Trying to solve this problem by substituting `k` for `1` can be *worse* if not all locations end up being checked. However, s o long as table size `m` and `k` are coprime, we know all locations will be generated. Note: not that important in practice since we don't want full tables anyways.

When evaluating the speed that results from collision-resolution methods, we consider **search length**, which is a function of **load factor** *f* where:

Using this *f*, we have:

as the average number of steps for successful searches and:

as the average number for unsuccessful searches. Trying some figures, we can easily see that filling the table above 75% capacity is not optimal (0.75: 8.5/unsuccessful vs. 0.90: 50.5/unsuccessful).

11.3.2 Quadratic Probing

Essence: `loc += ai + bi^2`

Rough implementation:

```
loc  = hash(key);
int i = 0;
while(num[loc] && num[loc] != key){
    ++i;
    loc = loc + a*i + b*i*i;
    while(loc > n) loc = loc - n; // n is table length
}
if(!num[loc]) num[loc] = key;
```

Note that powers of 2 for *n* are quite bad, only a small fraction will be tried. For prime *n*, half can be reached (usually sufficient).

11.3.3 Chaining

Items that hash to the same index are held within a linked list. Thinking about implementation, we start with a basic linked list implementation:

```
typedef struct node{
    int num;
    struct node *next;
} Node, *NodePtr
```

and a function to create new nodes:

```
Node newNode(int n){
    Node temp;
    temp.num = n;
    temp.next = NULL;
    return temp;
}
```

and hash would be an array of these nodes as follows:

```
NodePtr hash[MaxItems]; // 0-index
```

which we initialize with NULLs:

```
for(int i = 0; i < MaxItems; ++h) hash[h] = NULL;
```

Putting this together, we might have an implementation that like this:

```
```

TODO: Add a full implementation with creation, searching, and search + insert.