Data Structures

Stacks and Queues

Chapter

Abstract Data Types

An abstract data type is a data type where there is a set of things the data type has to do, but it is up to the developer to implement those details

Queues

A queue is a data structure that follows FIFO (*F*irst *I*n *F*irst *O*ut).

It has the following properties: - `enqueue`: to go into the queue - `dequeue`: to leave the queue

Stacks

A stack is a data structure that follows LIFO (*L*ast *I*n *F*irst *O*ut)

It has the following properties: - `push`: to add to the stack - `pop`: to remove from the stack

Resizing Arrays

Chapter

An array is a list of contigous data (back to back).

The issue of arrays, is that they have a finite number on initialisation, using either [] or `malloc`. This is what is called a static array.

Dynamic Array

Instead, why don't we copy and paste the entire array into a new block of memeory with a bit more space.

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

void print_array(int* arr, int size);

int main(){
    // Initalise a new array of length three
    int* list = malloc(3 * sizeof(int));
    if(list == NULL){
        return 1;
    }

    // Add values 1, 2, 3 to list
    for(int i = 0; i < 3; i++){
        list[i] = i + 1;
    }

    // Print array
    print_array(list, 3);

    // Add new value to dynamic array
    int new_value = 4;
    int* temp = malloc(4 * sizeof(int));
    if(temp == NULL){
        free(list);
        return 1;
    }

    // Add list values to temp
    for(int i = 0; i < 3; i++){
        temp[i] = list[i];
    }
    temp[3] = new_value;
    free(list);   // Forget where list used to be
    list = temp;  // And make list point to where temp is pointing

    // Print array
    print_array(list, 4);

    // Free memory and quit
    free(list);
    return 0;
}

void print_array(int* arr, int size){
  printf("[");

  for(int i = 0; i < size; i++){
    printf("%i", arr[i]);
    if(i != size - 1){
      printf(", ");
    }
  }

  printf("]\n");
}


// Output:
// [1, 2, 3]
// [1, 2, 3, 4]
```

This can be slow however, as each time you will need to copy and paste the entire array, getting to the point where you may be looping 100000 times just to add one more value. A solution to this could be to scale the size change, for example doubling the capacity of the array each time, however another solution could be to use a linked list

Linked List

Chapter

A linked list is a data structure in which data is not stored contigously, instead it is a structure in which you have the data, and you have a pointer to the next set of data. Therefore, it kind of looks like this:

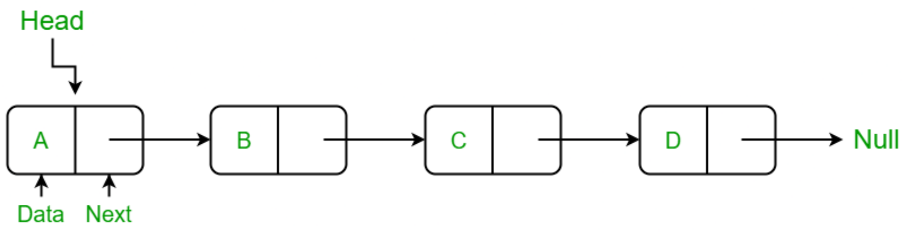

Figure 1: Linked List Diagram

Here's how we can implement a Linked List in C

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

typedef struct node{
    int number;
    struct node* next;
} node;

int main(int argc, char* argv[]){
    // Initalise 0 length list
    node* list = NULL;

    for(int i = 0; i < argc; i++){
        int number = atoi(argv[i]);

        node* n = malloc(sizeof(node));
        if(n == NULL){
            // free memory
            return 1;
        }

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

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

// Input: ./linked_list 1 2 3
//
// Output:
// 3
// 2
// 1
```

As you can see, this linked list prepends all values to the beginning (simmilar to a stack).

Adding to the linked list is $O(1)$ as it doesn't matter the size of the list. However, searching will take $O(n)$ as, at worst, it will have to look through the entire list to find it and it can't make any skips.

To append the nodes, you can do the following:

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

typedef struct node{
    int number;
    struct node* next;
} node;

int main(int argc, char* argv[]){
    node* list = NULL;

    for(int i = 1; i < argc; i++){
        int number = atoi(argv[i]);

        node* n = malloc(sizeof(node));
        if(n == NULL){
            printf("Could not allocate memory for temporary node\n");
            return 1;
        }

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

        if(list == NULL){
            list = n;
        } else {
            for(node* ptr = list; ptr != NULL; ptr = ptr -> next){
                if(ptr -> next == NULL){
                    ptr -> next = n;
                    break;
                }
            }
        }
    }

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

// Input: ./linked_list 1 2 3
//
// Output:
// 1
// 2
// 3
```

This now means that, by appending, the time complexity is $O(n)$, as you have to go through the entire list to append to the end.

We can also do this in sorted order by adding a conditional during the pointer loop to check to see if the next number is smaller than the current.

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

typedef struct node{
    int* number;
    struct node* next;
} node;

int main(int argc, char* argv[]){
    node* list = NULL;

    for(int i = 0; i < argc, i++){
        int number = argv[i];

        node* n = malloc(sizeof(node));

        if(n == NULL){
            printf("Could not allocate memory for temporary node\n");
            return 1;
        }

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

        if(list == NULL){
            list = n;
        } else {
            for(node* ptr = list; ptr != NULL; ptr = ptr -> next){
                if(ptr -> next == NULL){
                    ptr -> next = n;
                    break;
                }

                if(n -> number < ptr -> next -> number){
                    n -> next = ptr -> next;
                    ptr -> next = n;
                    break;
                }
            }

        }
    }

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

// Input: ./linked_list 3 1 2 4
//
// Output:
// 1
// 2
// 3
// 4
```

This is still $O(n)$ as, worst case, it will need to go to the end to append each time, however this now changes the Ω to $\Omega(1)$ as best case it goes at the front of the list.

Trees

Chapter

A tree is a structure in which you have a *root* node, and many *child* nodes.

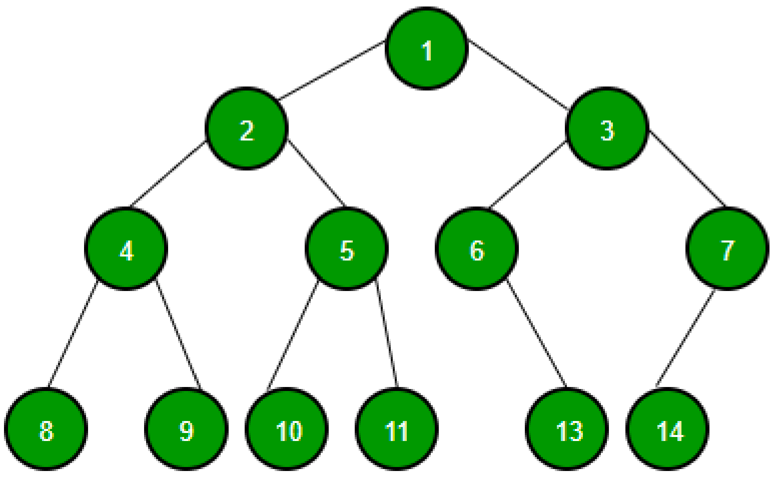

Figure 2: Tree Data Structure

Binary Search Tree

A binary search tree is a tree that is split in the same way that an array would be split in a binary search. This gives the benefit of using a sorted, easily searchable, data structure. The best of arrays and linked lists.

A way to implement a search algorithm for this would be:

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

int search(node* tree, int target){
    if(tree -> value == NULL){
        return 1;
    } else if(target < tree -> value){
        search(tree -> left);
```

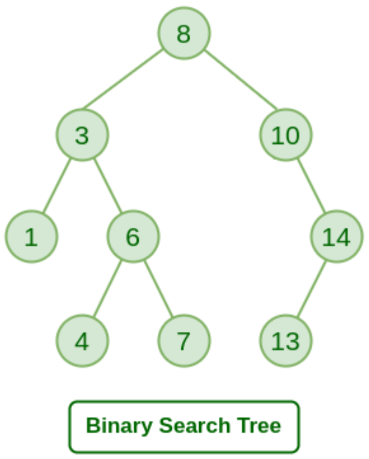

Figure 3: Binary Search Tree

```
    } else if(target > tree -> value){
        search(tree -> right)
    } else {
        return 0;
    }
}
```

The problem with binary search trees is that you have to input the nodes in the correct order, otherwise you could end up with one side having a larger amount of nodes than the other, or, worst case, you could have the user input in sorted order and end up with a linked list.

Dictionaries

Chapter

A dictionary is a data stucture that stores keys and values

Hashing

A hashing function is a reducer functions, in which it takes an infinite amount of inputs, and reduces them to a finite amount.

An example of a simple hashing function, in C, would be:

```c
#include <ctypes.h>

unsigned int hash(const char* in){
    return toupper(in) - 'A';
}
```

This converts the first letter of a string, to its place in the alphabet.

Hash Tables

A hash table is an array of linked lists, where each index is the reduction of the key, and the linked list is the values of the reduced key. For example, given some Harry Potter characters, this is a hashtable of their names, where the array on the right reduces their name to the first character, `[0]` is A, and `[25]` is Z.

This isn't as good as the time complexity is still $O(n)$ as, worst case, you have everyone with the same letter trying to be added (Think Anne, Amy, Andrew. It would take n steps to get to the last person in that linked list).

To solve this issue, we can increase the size of the reduced array, so instead of the starting character of each name, instead use the starting 3 letters of each name. This will reduce the amount of conflicts, and eventually you will regain the $O(1)$ of arrays. This does increase the memory size to allow us the constant time complexity.

Tries

A Trie is a tree of hash arrays. The issue of this is you are having to store a lot of hash arrays, therefore this takes a lot of memory. However, this is the closest that we get to $O(1)$ for dictionaries.

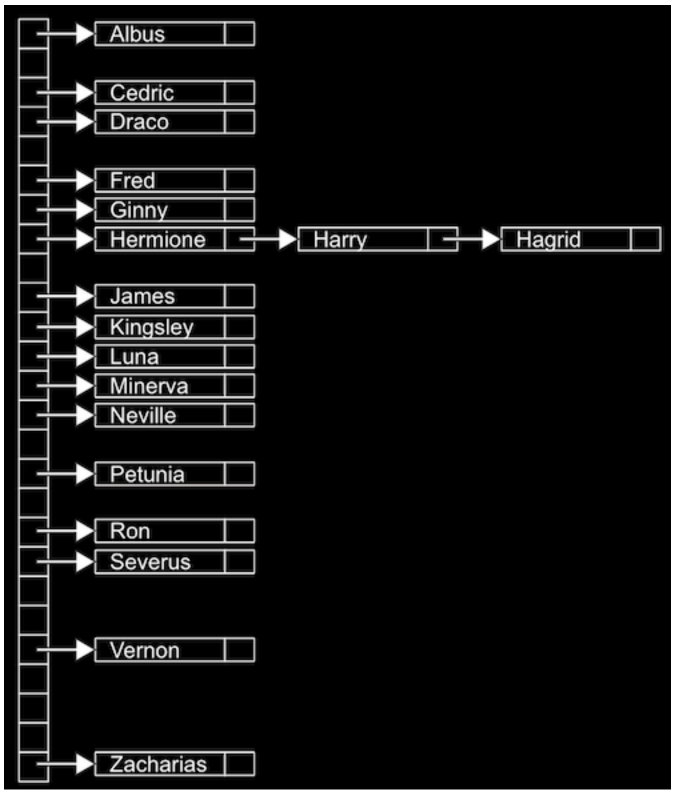

Figure 4: Hash Table of Harry Potter Characters