Comp 2521 Notes

Lec 1.1 – Recursion

Compiling

The basic compilation command we will use is

```
clang -Wall -Werror -g -o prog prog.c
```

- -Wall enables almost all warnings → warnings will show when compiling, catching syntax errors like %s when you mean %d for printing an integer
- -Werrors turns warnings into errors, preventing the program from compiling
- -g preserves information useful for debugging when compiler produces errors → line numbers, function call stack, variable names etc

Sanitizers are compiler plugins that instrument executing code with sanity checks

- Checks for use-after-free, array overruns, value overflows, uninitialized values, etc.
- dcc uses Asan and UBSan

AddressSanitizer

```
-fsanitize=address
```

- compiled with -fsanitize=address to detect invalid memory accesses, such as out-of-bounds array accesses, use-after-free, double-free errors, etc.

LeakSanitizer

```
-fsanitize=leak
```

- detects memory leaks

MemorySanitizer

```
-fsanitize=memory
```

- detects uninitialized memory access

UndefinedBehaviourSanitizer

```
-fsanitize=undefined
```

- detects a wide range of undefined behaviours?
    - Integer overflow (integer bigger than max number int can store)

Address, and memory sanitizers are not compatible, so often need to compile multiple times

```
valgrind ./prog
```

- Finds memory leaks (not freeing after malloc) and memory errors (illegally trying to access memory
- NOT comptabile with Asan

Makefiles

- With a makefile, all we need to do to compile a program is type *make* in the terminal
    - Make asan
    - Make msan

    - Make clean → removes any previous compilations
- Make compiles all programs in a directory

Concepts

Recursive functions are those that call themselves within their own code to recursively repeat a certain programming operation

- The base case (or stopping case) is the case where we don't need a recursive call anymore and can stop the recursion loop
- The recursive case is the case where we will need to call the function itself on a smaller version of the problem
- Each function call is a separate instance that creates a new stack frame
    - Stack frames hold all of the data needed by the function and are removed when the function returns
- When the stack is growing, it is called winding, and when it is shrinking, it is called unwinding
- NOTE: as stacks are created, memory may be used up for large recursive operations

To write a recursive function:

- Consider whether recursion is appropriate
- Identify the base case
- Think about how the problem can be reduced → how can we express the problem as a smaller part of itself
- Think about how results can be built from the base and recursive cases

Recursive solutions usually require recursive helper functions when

- The data structure uses a "wrapper" struct → AKA "nodeRep"
    - A struct that contains a pointer to the head of the list (THIS IS NOT THE SAME AS THE HEAD CONTAINING A POINTER TO THE NEXT NODE)
    - Useful as we can add more information about the linked list (e.g. the size, a pointer to the last node, etc.) to this wrapper struct
- The recursive function needs to take in extra information

Running "time ./prog" runs the program, and also tells you the time it takes to run the program

- Recursive functions can become rather slow if there are numerous recursive calls, which is bad → inefficient → in such cases, iterative solutions (loops) are much more efficient as stack frames do not need to be created with each loop → iterative solutions generally use less memory than recursive solutions

Recursive function examples:

- Factorials → 1:22:50
- Sum of a linked list → 1:33:45
- Appending to end of list → 1:44:15 → 1:48:31 (with a recursive helper function)

Code
Typically, a recursive function uses an if else statements. E.g.

```
struct node *listAppend (struct node *list, int value) {
	if (list == NULL) {
		return newNode(value);
	} else {
		list->next = listAppend(list->next, value);
		return list;
	}
}
```

- This is the 1:44:15 lec example
- The if is the base case
- The else is the recursive case

Lec 1.2 – Performance Analysis

Concepts

Measuring literal program run time can be done by adding "time" before a command

- time ./prog
- Real: the real, physical time elapsed → often around user + sys
- User: the time taken for the CPU to process your program
- Sys: the time taken by the operating system for special actions (malloc, file open, file close)
- As the time it takes for user input is also counted when using time, we can feed files input data using '< file.txt'
- Program output also consumes time, so we can use '> dud.txt' to print all output to the dud file instead of printing them to terminal, which is decently slower
    - '>' is called the output redirect → '> dud.txt' redirects output from stdout to dud.txt

We can use the runtime of a program when given different sizes (number) of inputs to measure their efficiency

- We repeat the timing 3 times and measure the same type of time (typically real or user) for scientific validity
- Where reasonable, we can plot out the relationship between runtime and input size
- We can do this with multiple programs to compare their efficiency
- If runtimes for minutely different sizes are all less than 0.01 and relatively similar, don't look too deep into it, because all programs have an overhead in just setting up

To estimate the amount of memory a program uses when running:

- We can look at the number of variables and the memory (bytes) they require
- A program typically always uses a constant amount of memory → unless they are recursive

For sorting algorithms, we have to consider the input size, AND the 'sortedness' of the original set of items

- Worst case: items in reverse order
- Average case: items are not in any order (random)
- Best case: items are already sorted

We need a theoretical way of analysing algorithms in a non-C language because:

- Measuring literal program runtime requires implementation of the algorithm
- Program runtime is influenced by the specific machine's hardware and O/S

Pseudocode is a method of describing the behaviour of a program without needing to worry about the language it's written in or the machine it runs on

- More structured than English, but less structured than code
- Removes all language-specific features and minute details
    - We don't need to care about variable declarations and types

    - Can use words to describe conditions for while/if
    - Don't need semicolons
    - Don't need to follow exact library functions (e.g. printf("%d", sum) → print(sum))
- There are no exact style or syntax rules → we just need to describe thoughts and logic of code in a consistent and clear way

Theoretical performance analysis involves looking at a program's pseudocode to evaluate its complexity regardless of the hardware or software environment

- For time complexity, we do this by counting how many 'operations' occur in a program and then express the complexity of the program as an equation based on the number of input elements 'n'

To analyse simple programs:

- Analyse each line, noting the number of operations on it (we generally treat each line as one operation only)
- When you enter a loop, multiply the cost of each line within the loop by the number of times its being looped
- For algorithms, we can analyse for best and worst case input (sometimes they are the same)
- Then we sum the cost of all lines

We usually assume primitive operations are "constant time" (1 step of processing)

- This differs from reality, but it doesn't really matter because Big-O notation gets rid of all constants and lower order terms → hence why we don't need to be too finicky about the cost of each line and can just consider it as 1
    - After dropping constants and lower order terms, we just take the result and write it as O(result)
    - HOWEVER, if its two terms with n being multiplied, we don't drop one of the terms if their product is the highest order → 2n*log(n) + 4 becomes O(n*log(n))
    - Notes on order:
        - n > log(n)
        - x^n > n^x
- Primitive operations include:
    - Indexing into an array (array[i])
    - Comparisons (<, >, ==)
    - Variable assignment (vName = x)

Binary search

- Searches an ordered array for a number X by going to the middle element of the array and checking if it is greater than, less than, or equal to X
    - If it is greater than X, it goes to the element halfway between the middle element and the upper bound of the array and does the same check
    - If it is less than X, it goes to the element halfway between the middle element and the lower bound of the array and does the same check
    - If it is that element, it has found X and ends

    - Repeats this until either X is found, or X cannot be in the array (it can't be found in the ordered section it should be in)
- With every loop, we are halving the part of the array we are searching → hence, binary search has time complexity $\log_2 n$ = O(log(n))
- EXAMPLE: [1, 2, 3, 4, 5, 6, 7] and X == 7
    1. Middle element = 4 < 7, so check between 4 and upper
    2. Middle element = 5 < 7, so check between 5 and upper
    3. Middle element = 6 < 7, so check between 6 and upper
    4. Only one element between 6 and upper which is 7 == X, so return found

Amortisation refers to the process of doing some work up front to reduce the work later on
- With regards to search algorithms, think about how binary search needs the array to be sorted first:
    - Linear: O(n)
    - Binary: O(n^2) (sorting) + O(log(n)) (binary searching)
- We need to think about the context of the program → is it worth sorting the list (doing some work up front) to get the ability to search quickly? (to do less work later) → depends on how often searches need to be done in the scenario

EXAMPLES:
- bubble_sort_pseudo.txt → theoretical analysis of pseudo code for bubble sort (bubble sort variation from hayden smith lec 2.1)
- binary_search.c → binary_search_pseudo.txt
-

Examples in 23T3 lecs:
- Search algorithm at -1:25:05 → function returns index of a searched number in an array → -1:17:52 shows how data can be collected, summarised, and have a conclusion drawn from it (note the worst case for the linearSearch algorithm was when the searched int was not in the array, which was what was tested) → -1:03:24 shows an example of pseudocode for the linearSearch algorithm, and how primitive operations are counted
- Binary search algorithm at -5:33

Code

Vid 2.1 – Sorting Algorithms Overview

Hayden Smith

Concepts

Sorting involves arranging a collection of items in order based on some property (usually the key) of the items using an ordering relation on that property (e.g. <, >, etc)

- Sorting speeds up subsequent searching (binary search can be used)
- Arranges data in a human and computationally useful way
- Sorting occurs in many data contexts (arrays, linked lists, files), and different contexts generally require different approaches
- We generally sort arrays in ascending order of the property we want to sort by, and can sort the whole array or just a part (slice) of it → e.g. numbers are sorted numerically in value, and characters/strings are sorted alphabetically

A sorting algorithm needs an array (a), the lower bound (lo) and upper bound (hi) as input

- lo and hi tell us between which indices of the array we want to sort → if we want to sort the whole array, we set lo = 0 and hi = N – 1, where N is the size of the array
- Before we begin, we need $0 \leq lo < hi \leq N - 1$ to ensure we are actually working within the array → lo and hi must be valid indexes and lo < hi
- We also need to make sure that a[lo..hi] contains defined values of type Item → no uninitialized values
- After the sort, we need to make sure that a[lo..hi] contains the same set of values as it did before the sort, and that for each i in lo to hi – 1, a[i] <= a[i + 1]

Generally, most algorithms do a sort "in-place"

- This means they do the sort inside the original array —> they do not create temporary arrays

We sort arrays of items that can be simple values (e.g. int, char, float) or structured values (e.g. struct)

- Each item contains a key which can be a simple value or collection of values, and the order of key values determines the order of the sort
- Duplicate key values are allowed
- In 2521 lecs, the key value of an item usually refers to the whole item
    - For ints, floats and chars we just use <=, and for strings we use strcmp for lexicographical (alphabetic) sorts

Sorting algorithms are stable and adaptive

- Stable → If we have a duplicate key value, the order those values are found in the original array is the order they will be put into the sorted array → Given that x == y, if x precedes y in the unsorted list, then x precedes y in the sorted list
- Adaptive → the behaviour/performance of the algorithm is affected by data values → the performance of the algorithm in the best/average/worst cases differ → (e.g. if the array is already sorted, it will be fast, but if it isn't it will be slower)

When analysing sorting algorithms, we are interested in:

- Three main cases:
    - Random order (average case)
    - Sorted order (best case)
    - Reverse order (worst case)
- When we look at an algorithm, we try to minimise the number of comparisons (C) and swaps (S) between items
- Most sorts fall into two types of complexity:
    - $O(n^2)$ → slow but simple algorithms that are fine for small arrays (size within the hundreds)
    - $O(n \log n)$ → faster but more complex → this is ideal

Code

Concrete framework of a sorting algorithm → 16:21 of vid 2.1:

```
// dealing with generic Items
typedef SomeType Item;

// abstractions to hide details of Items
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define swap(A,B) {Item t; t = A; A = B; B = T;}

// function protoypes

// function to sort a slice of an array of Items
void sort(Item a[], int lo, int hi);

// function to check if slice of array is sorted
int isSorted(Item a[], int lo, int hi) {
        for (int i = lo; i < hi; i++) {
                if (!less(a[i], a[i + 1])) {
                        return 0;
                }
        }
        return 1;
}
```

- SomeType is just a generic data type
- #define key(A) (A) → the key of an item is just the item itself
- #define less(A,B) (key(A) < key(B)) → defines comparison operation → whenever we put less(A,B) we get the evaluation of the comparison between A and B
- #define swap(A,B) {Item t; t = A; A = B; B= t;} → defines the swap operation -> whenever we put swap(A,B) we swap the positions of A and B
- The isSorted function checks that the slice lo to hi of array a is sorted, returning 1 (TRUE) if it is, and 0 (FALSE) if it isn't

Vid 2.2 – Basic Sorts

USE Structs.sh

Concepts

For small arrays, sorting algorithms of the $O(n^2)$ class are sufficient

- They have an $O(n^2)$ worst- case time complexity

Bubble Sort

- Bubble sort moves through the list pair-wise, swapping pairs in the wrong order → repeats this until the list is fully sorted (when there is an iteration where no swaps are made)
- Is adaptive
- John Shep diagram at 12:29 of vid 2.2

Selection Sort

- NOT ADAPTIVE
- Finds the smallest element and puts it into the first array slot, then finds the second smallest element and puts it into the second array slot, and so on and so on
- Swaps the ith smallest element in the array with the element in the ith slot
- Each iteration improves sortedness by one element
- John Shep diagram at 3:22 of vid 2.2

NOTE → a pass refers to an iteration of the outer loop

Insertion Sort

- We take the first element and treat it as a sorted array of length one, then we take the next element and insert it into the sorted part of the array in the correct order, repeating until the whole array is sorted
- Each pass increases the length of the sorted array by 1
- Diagram at 23:41
- Basically, we take the next element called val from the original array, then starting from the end of the sorted part of the array we move everything larger than val right by one until we either reach the front of the array or an Item smaller than it, at which point we insert it in that slot

ShellSort → improvement of insertion sort → MIGHT NOT BE STUDIED

- An h-sorted array is one where every hth element yields a sorted array → a h-sorted array is made of interleaved sorted arrays
- The basic idea for ShellSort is that if we start with h as the largest value () and make sure that the array is h-sorted, then reduce h and make sure it is h-sorted again, and repeat until h == 1, we get a sorted array
- The h-values we use are 701, 301, 132, 57, 23, 10, 4, 1 → start with the largest h-value < the size of the array → however, there are other sequences of h-values
- Diagram at 33:18

COMPLEXITY SUMMARY AT 37:38 JOHN SHEP VID 2.2

FOR LINKED LISTS

- The complexity of these algorithms are the same, but actually using them will be more expensive since it involves the heap and pointers
- Selection sort works by finding the largest value V in the original list, then unlinking it and adding it to the start of a new sorted list (initially empty)
- Bubble sort works by traversing the list, then swapping node values (NOT ENTIRE NODES, JUST THE NODE VALUE) if current->value > next->value, and repeating until no swaps are required in one traversal
- Selection sort works by scanning the original list then inserting each item into S in order → scan an item and add it into the new sorted list in order

Code

```
void bubbleSort (int a[], int lo, int hi) {
        for (int i = 0; i < hi; i++) {
                int nswaps = 0;
                for (int j = hi; j > i; j--) {
                        if (less(a[j], a[j – 1])) {
                                swap(a[j], a[j – 1]);
                                nswaps++;
                        }
                }
                if (nswaps == 0) {
                        break;
                }
        }
}
```

- Basic structure for a bubble sort
- Outer loop moves the lowest position we want to continue sorting from until we have sorted all the positions we want to sort
- nswaps keeps track of how many swaps are made in a pass, so that the function can end if there are none since no swaps means everything is sorted
- Inner loop moves an element up (towards the start) the array until it meets an element smaller than it, then moves that smaller element up until it meets an element smaller than it and so on, provided the smaller element is not already part of the sorted section, which i keeps track of
- The j > i condition of the inner for loop means the function does not check and swap elements that have already been sorted → bubble sort always sorts at least one element correctly by floating the nth largest into the SIZE – n th slot on the nth loop
- If we did not include nswap, the algorithm would be non-adaptive

```
void selectionSort (int a[], int lo, int hi) {
        for (int i = lo; i < hi – 1; i++) {
                int min = i;

                for (int j = i + 1; j <= hi; j++) {
                        if (less(a[j], a[min])) {
                                min = j;
                }
                swap(a[i], a[min]);
        }
}
```

- Basic structure for a selection sort
- Outer loop runs the ith position swap for each element in a → i < hi – 1, since the last element is trivially the largest element (it would have already been swapped into order otherwise)
- Inner loop finds the minimum element amongst the unsorted part of the array and stores its index in min so that it can be swapped into the ith position

```
void insertionSort (int a[], int lo, int hi) {
        for (int i = lo + 1; i <= hi; i++) {
                int val = a[i];

                for (int j = i; j > lo; j--) {
                        if (!less(val, a[j – 1]) {
                                break;
                        }
                        a[j] = a[j – 1];
                        // shifts everything right one
                }

                a[j] = val;
        }
}
```

- Basic structure for insertion sort
- Start from the second element (i = lo + 1) in outer loop since we treat the first as its own sorted array of length 1, and loop until we have added every element into the sorted part
- Inner loop moves the element to the left of the current position j one to the right until either the start of the slice is reached or an element <= than val is encountered (break will exit the inner loop) → upon exiting the inner loop, val is moved into the position j
- Val keeps track of the current element being inserted into the sorted part of the array ONE

```
void shellSort (int a[], int lo, int hi) {
        int hvals[8] = {701, 301, 132, 57, 23, 10, 4, 1};

        for (int g = 0; g < 8; g++) {
                int h = hvals[g];
                int start = lo + h;

                for (int i = start + 1; i <= hi; i++) {
                        int val = a[i];

                        for (int j = I; j >= start; j -= h) {
                                if (!less(val, a[j – h]) {
                                        break;
                                }
                                a[j] = a[j – h];
                        }
                        a[j] = val;
                }
        }
}
```

- Basic structure for a shellsort algorithm
- The hvals array is just the h-values we use in shellsort, and the outermost loop iterates through the h-values until we have one smaller than the size of the array → the condition of the middle loop makes it so it will not run the inner loop if the hval >= size
- The middle and inner loops h-sort the original array into interleaved h-sorted arrays for the current hval in the same way as the usual insertion sort → the j -= h incrementation means the array is h-sorted and not normal sorted
- MIGHT NOT BE STUDIED

Vid 2.3 – Quicksort

NOTE: 1hr 40 min of vid lecs remaining for sorting algorithms

Concepts

Quicksort

- The standard quicksort has time complexity $O(n^2)$ and is not stable

- Quicksort chooses an item to be a pivot, then it partitions the array such that all elements left of the pivot are smaller than it, and all elements right of the pivot are larger than it → DOES NOT mean the partitions are sorted
- Quicksort recursively sorts the partitions
- John Shep diagram at 2:50 of vid 2.3
- Partitioning is done by having two indexes i and j which start from the beginning and the end of the array → these indexes loop through their half of the array until they find elements that don't belong (elements > pivot in the RHS and elements < pivot in the LHS) → once both loops have either found an element that doesn't belong or reaches the end/start of their partition, they swap the elements indexed at i and j so that the smaller one goes to the smaller side and the larger one goes to the larger side → after the swap, it continues from the current index of i and j until i == j, which means all elements belong
- This i == j condition will activate a break ending the loop, at which point the element at the i == j index will be swapped with the pivot (as that is the index at which the partitions are split), and i == j will be returned as the pivot for the quicksort algorithm
- John Shep diagram at 6:16 of vid 2.3
-

To improve quicksort → worst case → Median-of-three

- The worst case for quicksort occurs when the chosen pivot is the smallest or largest value in the array
- The median-of-three method of choosing the pivot avoids the worst case → we take the values at the start (x), middle (z) and end (y) of the array, then order them from smallest to largest → then we take the new middle value and swap it with the lo + 1th slot since we want to use that as the pivot
    - We want it so that we have the values at the start, middle and end as
      Start < Middle < End
      Doing this will also mean we have 2 less elements to compare since we know the start and end are on the correct side
- This ensures the partitions end up being roughly the same size
- Diagram at 16:00 → kind of confusing with the < stuff, look at sub-second dot point

To improve quicksort → recursion for small partitions

- The cost of recursive function calls is not worth the benefit of partitioning arrays of size < some threshold (5 is reasonable)

- It is more efficient to handle small partitions differently by either switching to insertion sort immediately or not sorting it yet and then using insertion sort after the quicksort → using insertion sort to sort the remaining un-sorted parts of the array

To improve quicksort → many duplicate keys

- If we have multiple duplicate keys of the pivot, instead of just placing them in the partitions, we can create a third pivot partition so that we work with smaller partitions on the left and right of the pivot partition
- We find all elements of the same key as the pivot and move them together while retaining their order, then treat that as a third pivot partition
- Diagrams 23:11 → including Bentley/McIlroy approach to three-way partition
- The Bentley/McIlroy approach involves moving any duplicates found in the left partition to the start of the array, and any duplicates found in the right partition to the end of the array, then using those chunks of duplicates at the start and end to form a third partition that is moved to its spot in the middle of the left and right partitions → NOT STUDIED???
-

Code

```
void quicksort (Item a[], int lo, int hi) {
        if (hi <= lo) {
                return;
        }
        medianOfThree(a, lo, hi);
        int i = partition(a, lo, hi);
        quicksort(a, lo, i - 1);
        quicksort(a, i + 1, hi);
}
```

- Basic structure of recursive quicksort algorithm
- The if condition activates when hi <= lo, which occurs when the hi and lo, which are passed into as i – 1 and i + 1 in the recursive calls, overlap → when the partition has been sorted, exiting the recursive loop
- medianOfThree is a supplementary function used to avoid the worst case for the quicksort algorithm
- i is the index of the chosen pivot given by a supplementary partition function

```
void quicksort (Item a[], int lo, int hi) {
        if (hi – lo <= Threshold) {
                insertionSort(a, lo, hi);
                return;
        }
        medianOfThree(a, lo, hi);
        int i = partition(a, lo, hi);
        quicksort(a, lo, i - 1);
        quicksort(a, i + 1, hi);
}
```

- Quicksort with insertionSort
- NOT LOOKED AT IN 2521??? → FKN JOHN SHEP
- This structure of the recursive quicksort algorithm avoids recursive calls for the quicksort (and partition) function, and instead uses the insertionSort algorithm once the size of a partition is below a certain Threshold

```
int partition (Item a[], int lo, int hi) {
        Item v = a[lo];
        int i = lo + 1;
        int j = hi;

        for(;;) {
                while (less(a[i], v) && i < j) {
                        i++;
                }
                while (less(v, a[j]) && j > i) {
                        j--;
                }
                if (i == j) {
                        break;
                }
                swap(a, i, j);
        }

        if (less(a[i], v) {
                j = i;
        } else {
                j = i – 1;
        }

        swap(a, lo, j);
        return j;
}
```

- Basic partition function
- for(;;) means infinite loop → same as while(1) → the loop keeps going until a break is called
- The first while loop keeps looping right through the array from the beginning until it either encounters an element larger than the pivot or j → stopping the loop there means i is the index of the larger element or i == j
- The second while loop keeps looping left through the array from the end until it either encounters an element smaller than the pivot or i → stopping the loop there means j is the index of the smaller element or j == i

- j == i when there are no more elements in the wrong partition (all elements < pivot are on the LHS and elements > pivot are on the RHS), and so the if statement breaks the loop since there are no more elements to move
- If j != i, both i and j are indexes at a non-belonging element, and thus swap then continue the loop until i == j
- The if else statement after the loop checks to see whether the current i == j indexed element is less than the pivot or not, updating j so that it will be indexed on the side of the split where the element is less than the pivot so that the pivot can swap with it without ruining the partition
- We then return j, which is the current position of the pivot after the entire partitioning phase

```
void medianOfThree (Item a[], int lo, int hi) {
        int mid = (lo + hi) / 2;

        if (less(a[mid], a[lo])) {
                swap(a, lo, mid);
        }
        if (less(a[hi], a[mid])) {
                swap(a, mid, hi);
        }
        if (less(a[mid], a[lo])) {
                swap(a, lo, mid);
        }
        swap(a, mid, lo + 1);
}
```

- Basic median-of-three pivot choosing function
- mid is the index of the middle element in the array
- The if statements order the elements at lo, mid and hi
- The final swap moves the current middle element, which is the median of the original lo, mid and hi elements, to the lo + 1th slot to be used as the pivot in the partition function

Lec 2.4 – Mergesort

Concepts

Mergesort → non-adaptive

- Works by recursively splitting the array (partitions of the array in recursive calls) into two equal-sized partitions then merging the partitions into a new sorted array in an ordered fashion before copying it back to the original array
    - Hayden diagram 4:05
    - Doesn't actually physically split the array, just conceptualise it as if it does
- Merging works by comparing elements (from start to end) in the two sorted partitions, and adding the smallest of the two to the new sorted array, repeating until one of the two sorted partitions are exhausted at which point we copy the rest of the unexhausted partition
    - Hayden diagram at 16:20
- John Shep diagram at 3:38 and 7:39
- Mergesort can also be done non-recursively by computing the partition boundaries iteratively
- The disadvantage of mergesort compared to quicksort is that mergesort needs extra storage for the heap allocations
- n loops in merge, and log n from partitions → O(n logn)

Bottom-up mergesort

- This version of mergesort does not require recursion → instead, we compute the partition boundaries iteratively
- On each pass of the array, we view the array as sorted runs of length m
    - At the start, we treat the n size array as an array of n sorted runs of length 1
    - On the first pass, we merge these adjacent sorted runs of length 1 (elements) into sorted runs of length 2
    - On the second pass, we merge the sorted runs of length 2 to get sorted runs of length 4
    - Continue until a single sorted run of length n
- Diagram at 17:50
-

Mergesort on linked lists

- We split a linked list by using three pointers a, b and c
- a remains at the head of the linked list
- c starts at the head of the linked list and moves down one node with every iteration
- b starts at the second node, and moves down two nodes with every iteration
- The iterations continue until b is either at the end (b->next == NULL) or beyond the end (b == NULL)
- Diagram at 31:25

Code

```
void mergesort (Item a[], int lo, int hi) {
        int mid = (lo + hi) / 2;
        if (hi <= lo) {
                return;
        }
        mergesort(a, lo, mid);
        mergesort(a, mid + 1, hi);
        merge(a, lo, mid, hi);
}
```

- Basic mergesort → uses recursion
- mid is the mid point of the array since we are splitting the array in half every time
- if condition to break recursion
- mergesort the left half, then mergesort the right half
- Merge the first and second half back together

```
void merge (Item a[], int lo, int mid, int hi) {
        int nitems = hi – lo + 1;
        Item *tmp = malloc(nitems * sizeof(Item));

        int i = lo;
        int j = mid + 1;
        int k = 0;

        while (i <= mid && j <= hi) {
                if (less(a[i], a[j])) {
                        tmp[k++] = a[i++];
                } else {
                        tmp[k++] = a[j++];
                }
        }

        while (i <= mid) {
                tmp[k++] = a[i++];
        }
        while (j <= hi) {
                tmp[k++] = a[j++];
        }

        for (int i = lo, k = 0; i <= hi; i++, k++) {
                a[i] = tmp[k];
        }
```

```
    free(tmp);
}
```

- Basic merge function
- Creates a new temporary array for the sorted elements called tmp
- First while loop compares elements in the two partitions using the if else statements, and adds the smaller of the two into the tmp array, moving the corresponding counter and the tmp array counter → loops until one of the partitions are exhausted
- The next two while loops each loop through the uncopied segments of both partitions, adding them to the tmp array → only one of the while loops should actually run since one should already be exhausted of elements
- The for loop copies the tmp array back into the original array
- The tmp array is then freed
- NOTE: tmp[k++] = a[j++] → we use the current values of k and j as the index and update the data at those indexes, THEN we increment them → equivalent to

  ```
          tmp[k] = a[j];
          k++;
          j++;
  ```
- THE CODE SHOULD PRIORITISE THE LEFT SIDE FIRST TO BE STABLE

```
#define min(A,B) ((A < B) ? A: B)

void mergesort (Item a[], int lo, int hi) {
    int m;
    int len;
    Item c[];

    for (m = 1; m <= lo – hi; m = 2 * m) {
        for (int i = lo; i <= hi – m; i += 2 * m) {
            len = min(m, hi – (i + m) + 1);
            merge(&a[i], m, &a[i + m], len, c);
            copy(&a[i], c, m + len);
        }
    }
}
```

- The define before the algorithm just means if A < B then return A, else return B
- Basic structure for a bottom-up mergesort
- m is the length of the runs, len is the end of the 2nd run, and c[] is the array to merge into
- The outer for loop makes it so that with each loop we deal with runs twice as large (the increment is double)
- The inner for loop finds the start and end position of one of the m-length sorted arrays, and iterates through the pairs
    - len = min → accounts for when there are less than m more elements in the array

- Then we merge each pair of adjacent sub arrays and copy the merged array back into the original → the merge function for bottom-up is different from the usual one
-

```
void merge (Item a[], int aN, item b[], int bN, Item c[]) {
        for (int i = 0, int j = 0, int k = 0; k < aN + bN; k++) {
                if (i == aN) {
                        c[k] = b[j++];
                } else if (j == bN) {
                        c[k] = a[i++];
                } else if (less(a[i], b[j])) {
                        c[k] = a[i++];
                } else {
                        c[k] = b[j++];
                }
        }
}
```

- Merge function for bottom-up mergesort
- aN and bN are the lengths of array a and b
- the for loop condition means we scan until we have scanned everything in both array a and b
- First two if else statements are for when either a or b have been exhausted of elements
- The last two if else statements are to compare the elements and add the smaller one

```
List merge (List a, List b) {
        List new = newList();
        while (a != NULL && b != NULL) {
                if (less(a->item, b->item)) {
                        new = ListAppend(new, a->item);
                        a = a->next;
                } else {
                        new = ListAppend(new, b->item);
                        b = b->next;
                }
        while (a != NULL) {
                new = ListAppend(new, a->item);
                a = a->next;
        }

        while (b != NULL) {
                new = ListAppend(new, b->item);
                b = b->next;
        }

        return new;
}
```

- First while loop runs until either the a or b list is exhausted
    - If else statements compares the elements of the lists, adding the smaller one to the end of the new list
- Second and third while loop copies over the elements in the list that is not yet exhausted → again, only one while loop should iterate
- NOTE: because a and b are passed in as parameters, changing a and b within the function will not change them in main or other functions

Lec 2.5 – Radix Sort

Concepts

Non-comparison-based sorting algorithms exploit specific properties of the data to sort items without comparing them

- This can allow time complexities better than O(n log(n))

Radix sort is a non-comparison-based sorting algorithm → $O(mn)$

- It is stable, non-adaptive and not in-place (uses additional space in memory to create the buckets)
- It requires keys to be decomposed into individual symbols (digits, characters, bits, etc.) → e.g. 372 becomes (3, 7, 2) and “Sydney” becomes (‘s’, ‘y’, ‘d’, ‘n’, ‘e’, ‘y’)
- The number of possible symbols is known as the radix, denoted R
    - Ideally, the radix is reasonably small → e.g. 10 (0 to 9) for numeric and 26 (a to z) for alphabetic
- If the item keys have different lengths, we pad them with a suitable character → typically 0 for numbers (15 becomes 015 if longest number has 3 digits) and _ for chars (“z” becomes “z _ _” if longest string has 3 chars)

Radix sort (LSD version) works by (explanation at -8:45 of lec 4 23T3):

1. Padding all keys with the necessary blank symbols
2. Create ‘buckets’ (a dynamic array or linked list) for each symbol in order, starting with the blank character
3. Place the items into the corresponding bucket based on their rightmost symbol (we place them in as we go, so without any real order
4. Then clear the original array and go through the buckets in order and add the items back as you go
5. Repeat with the next symbol (the one left of the previous one) until there is no symbol on the left (we have done the first character as well

Radix sort has complexity $O(mn)$

- n is the number of keys in the array, and m is the number of symbols in each key
- We use $R + 1$ buckets
- Each stable sort has time complexity $O\big(n + (R + 1)\big)$ since we go through the array and add the n elements into the bucket then we got through the $R + 1$ buckets and add the elements there back into the array
- We run the stable sort m times (once for each symbol), so we have $O(m(n + (R + 1)))$, but R is typically small so we can just say $O(mn)$

Examples:

- radix_sort_pseudo.txt

Code

Lec 3.1 – Abstract Data Types

Concepts

Data types are sets of values (atomic or structured) that have a collection of operations that can be done on them

- int is the set of integer values → addition, subtraction, multiplication, etc. are operations that can be done on ints
- arrays are sets of value(s) of repeated data types → index look up, index assignment, etc. are operations that can be done on arrays

Abstraction: hiding details about how a system is built so that we can focus on the high level behaviours, or inputs and outputs, of the system

- e.g. C abstracts away assembly code

Abstract Data Types (ADTs) are descriptions of data types that focus on its high level behaviour, without regard for how it is implemented underneath

- They are a fundamental concept of writing robust software, and of being able to work with other people
- Interface refers to what users can do on the ADT, and the implementation refers to the code that enables users to do things on the ADT
  - E.g. users use while loops in C, and the implementation is the assembly code that allows while loops to work
- The interface is separated from implementations
  - Users of the ADT only see the interface
  - Builders of the ADT provide an implementation
  - Both parties, however, need to agree on the ADTs interface, as it is what allows people to agree at the start and then work separately
- However, both parties need to agree on the ADT's interface, so that they can agree at the start then work separately
  - When we define interfaces, we need to include information about the pre and post conditions (what conditions hold at the start and end of the function) → add them through comments

ADT steps:

1. Create a .h interface file that determines the interface of your ADT (all operations we can do with the ADT listed as function prototypes with short comments) → set.h
2. Create a .c implementation file that contains all the code for the functions specified in the .h interface file
3. Compile a 'user' program that uses the implemented functions

Three key principles of ADTs in C:

- The "Set" (or equivalent) is usually a pointer of some sort
- That pointer is usually the first argument in every ADT function, with the exception of the create function
- When we write .h files we use header guards to prevent re-definition:

```
#ifndef SET_H
#define SET_H

#include <stdio.h>
#include …

Typedef struct SetRep *Set;

// ADT function prototypes

#endif
```

The set data type is a collection of unique integer values (no duplicates)

- We first need to figure out what behaviour does this ADT have (interface) → what operations should users be able to do on the ADT
    - Create empty sets
    - Insert one item into the set
    - Remove one item from the set
    - Find an item in the set
    - Find the size of the set
    - Drop the entire set
    - Display the set
    - Find unions or intersections with another set
-

Set usage

- To work with a set we need to write our "main" file, and compile it with the set library (the code for all the functions we defined in the .h file) that the ADT developer has implemented → we compile with the set library using
  `#include "fileName"`
- When we compile, we include the implementation file (the .c file with the code for the functions in the .h file) after the main
  `gcc -Wall -Werror -o testSet testSet1.c set_array.c`

EXAMPLES:

- set.h → simple .h file for set ADTs (defines the operations)
- set_array.c → implementation of some operations (functions) in set.h where sets are implemented as unsorted arrays (doesn't know how it is used)
- set_array_sorted.c → implementation of some operations in set.h where sets are implemented as sorted arrays
- set_linked_list.c → implementation of some operations in set.h where sets are implemented as linked lists
- testSet1.c → main file (the user of the ADT) testing the implementation

Lec 3.2 – Binary Search Trees

Concepts

Binary tree data structures are a fundamental set of data structures and algorithms for efficient programs

- They are ADTs so we will need the .h (interface) file, .c (implementation) file and main user program again
- Tree data structures are connected graphs that have edges (represented as lines) and nodes (represented as circles), and no cycles (can't get caught in loops)
- Each node has a particular data value, and links up to k children nodes → each node has k pointers to other nodes
- Leaf nodes are those that link up to NULL only
- The root refers to the start of the tree
- The level of a node refers to the length of the path from the root to the node
- The height/depth of the tree is the maximum path length from a root to a leaf

Binary search trees (BSTs) are linked-list-like structures

- Like linked-lists, we can make big modifications without shuffling everything by simply swapping/adding pointers
- Unlike usual linked-lists, they can be binary searched
- BSTs are either NULL (empty) or consist of a node with two subtrees → all children node are also roots of BSTs in themselves → BSTs are recursive structures

BSTs are ordered trees

- All values in the left sub tree are less than the root, and all values in the right sub tree are greater than the root
- This applies for all nodes in the tree → left < parent node < right
- BSTs are usually used with set data types where no duplicates are allowed

4 ways to traverse a tree → diagram at 1:17:52 Hayden lec 3.1 → pseudocode at 1:40:34

- Preorder → current, left, right → find and do sth as soon as you find it
- Inorder → left, current, right → good for printing
- Postorder → left, right, current → find a node then do sth after (mainly freeing)
- Level order or BFS → root, left children, right children → MORE IN GRAPHS
- We can use these different methods to do sth to every node, whether it be printing or freeing

Key BST operations

- insert(Tree, item) → $O(h)$ time complexity (h is height of BST) → $O(h) = O(\log_2 n)$ for balanced BSTs
- search(Tree, item) → best case $O(1)$, worst case $O(h)$
- print(Tree) → always $O(n)$ (not just printing, but for anything where we want to explore the whole tree → e.g. counting num nodes)
- create(item)

- join(Tree, Tree) → diagram at 13:14 Hayden lec 3.1.2 → typically $O(m)$, where m is the height of the right subtree
    - Find the smallest node in the right subtree (t2)
    - Replace the smallest node with the subtree on its right (if not empty)
    - Elevate min node to be the new root of both trees
- delete(Tree, item) → typically $O(h)$ complexity
    - 4 cases to consider:
        - Empty tree → new tree is also empty
        - No subtrees → unlink node from parent
        - One subtree → replace with child
        - Two subtrees → store the pointer to the node to be deleted, join the two subtrees, replace the node to be deleted with the root of the joint subtrees and delete the node
        - join the two subtrees and replace with the root of the join
    - Alternatively, find the minimum element in the right subtree and make the left subtree the minimum element's left subtree

Examples:

- BSTree.h, BSTree.c, main.c

Code

Lec 4.1 – Balancing BSTs

Concepts

Costs

- A balanced BST has height log(n), and therefore a search cost of $O(\log n)$
- A completely unbalanced BST has height n, and therefore a search cost of $O(n)$
- Hence, maintaining balanced BSTs saves work in later operations that require searches (amortisation)

A BST is balanced if it is a tree that, for all nodes, the absolute difference between the number of nodes in the left and right subtrees is less than 2

- That is, the number of nodes the left and right subtrees of any given node in a tree can differ by 1 at most → THINK THIS IS A HEIGHT BALANCED TREE
- This should ensure that the height of all subtrees on the same level are the same

BSTs often trend towards imbalanced trees, so we implement operations to prevent that

- Tree rotations → only needs simple pointer rearrangement, so cost is $O(1)$
    - If rotation is applied from leaf to root along one branch, cost is $O(height)$ but tree height is reduced so search cost goes towards $O(\log n)$
    - Right rotation → moving the left child up to the node then rearranging links to retain order → 7:52 Hayden lec 3.2
    - Left rotation → push the parent node to the left and down one level, and bring the right child up to where the parent node was then rearranging links to retain order → 16:20 Hayden lec 3.2
- Insertions at root → 33:18 lec 3.2
    - Each new item is added at the root node
- Tree partitioning
    - Rearranging the tree around a specified node that becomes the root

Insertions at root → $O(height)$ complexity

- Works by inserting each new item as a leaf in the tree then using rotation (left rotate if new node is currently a right child, and right rotate if node is currently a left child) to bring the new node up to the root
- However, balance is not guaranteed, just high tendency to result in a balanced BST
- Insertion at root ensures newer nodes are closer to the root, such that there is a lower cost if recently added items are searched more often
- Disadvantageous in that it requires large scale rearrangement of tree

Tree partitioning rearranges the tree so that a specified node becomes the root

- The node specified is indexed i, and indexes are based on in-order traversal → 50:44 → this means the 0th indexed node is the smallest item, and the last indexed node is the largest in the BST (1:00:00) → the middle index will be the median element
- We then move the median to the root
- PSEUDOCODE AT 56:59, BUT HOW TF THIS SHIT WORK

- Do I need to implement a wrapper struct to keep track of nelems (but that won't work since I need nelems in every left subtree), then add an int index field to the node struct and a function to redo the indexes everytime I partition? Or is there a simple way
-

Code

In lec03 directory → BSTree.c

Lec 4.2 – AVL Trees

Concepts

AVL trees are trees that are less memory efficient, but easier to rebalance

- Rebalancing AVL trees have $O(\log n)$ cost in the worst case
- They fix imbalances as soon as they occur

Following the AVL tree method, a tree is unbalanced when
$|height(left) - height(right)| > 1$

- When this happens, the tree can be repaired by rotating right is the left subtree is too deep, and rotating left if the right subtree is too deep
- Storing information in each node about their height (not depth in the main tree, but the height of each subtree) allows this process to be done faster, but at a greater space cost since we have larger node structs → 3:38 lec 4.2 Hayden
- After rotating, we need to update the height stored in each node
- We rebalance locally → rebalance subtrees, not the whole tree → after rebalancing, every subtree should be height balanced

- Insert new item as leaf like normal, then check if the right and left trees have a height difference > 1 → if the left tree is taller than the right tree, rotate right, else if the right tree is taller than the left tree, rotate left
    - Additional checking and rotations before the rotation above

AVL tree performance

- Average/worst case search performance of $O(\log n)$
- Trees are always height balanced, but may not always be weight-balanced

EXAMPLES

- BSTree.c → TreeAVLInsert and TreeHeight functions
- NOTE: A real AVL Tree would never actually use TreeHeight since the node structs would just contain the height of each node

Code

Lec 4.3 – 2-3-4 Trees

Concepts

2-3-4 trees

- NOTE: 2-3-4 trees are B-Trees with a max degree of 4, so the USFCA visualisation of 2-3-4 trees is their B-Tree visualisation with max degree 4
- The branching factor of a tree is equal to the number of children per node
    - BSTs have a maximum path length of $\log_2 n$, and so their branching factor is 2, and the search cost is also $\log_2 n$ in the worst case
    - 2-3-4 trees have a maximum path length of $\log_4 n$, and so their branching factor is 4, and the search cost is also $\log_4 n$ in the worst case, which is faster (smaller) than $\log_2 n$
    - NOTE: in big-Oh notation, both still have complexity $O(\log n)$
- Nodes in 2-3-4 trees can have multiple items in them, and can have up to 4 children, thus there are 3 different kinds of nodes
    - 2-nodes → 1 item and 2 children (BST)
        - Left is < x, right is > x
    - 3-nodes → 2 items and 3 children
        - Left is < x, middle is >x but < y, and right is >y
    - 4-nodes → 3 items and 4 children
        - Outer left is < x, inner left is > x but < y, inner right is >y but < z, and outer right is > z
    - That is, children slot between the Items in each node based on where they belong, and the Items in the node are also sorted
- 2-3-4 tree nodes have the order of the node (if it is a 2-node, 3-node or 4-node), an array of the data with max size 3, and an array of 4 pointers to the children nodes
- A balanced 2-3-4 tree has all nodes the same distance from the root

2-3-4 tree insertion → 40:09 lec 4.3 → UFSCA B-Tree degree 4 visualisation

- Find the leaf node where the item belongs using normal leaf-insert, then
    - If the node of insert is not full (less than 3 Items), insert item into this node in the correct position
    - If the node is full (has 3 items), insert item into the correct position then split the node by → 45:17 lec 4.3
        1. Moving the middle item to the parent node (if the current node is the root of the whole tree, make the middle item be the start of a new node that is the root → 48:27)
        2. Move items right of the middle item to a new child node of the parent, leaving the original node with only the leftmost item
        3. Recursively apply this to the parent node (since we added an item to the parent node) until no more splitting is required
- Try think of 2-3-4 trees as trees that grow upwards in a triangle
- The worst case cost is still $O(h)$

2-3-4 Tree performance

- Searching has complexity $O(\log n)$, but in reality the log base is bigger so search is faster
- Inserting has worst case cost of $O(\log n)$, with most of the cost coming from the search

Code

THE PSEUDOCODE IS AT 1:05:50 of lec 4.3

- ATTEMPTED TO IMPLEMENT IN lec03 directory → 2-3-4_Trees, but idk wtf going on in the psuedocode

```
typedef struct node {
        int order;
        int data[3];
        struct node *child[4];
} node;
```

- This is the struct for the node of a 2-3-4 tree that stores integers
- It has the order of the node → if order == 2, it is a 2-node, if order == 3 it is a 3-node, and if order == 4 it is a 4-node
- child[0] is the pointer to the node with values less than data[0], child[1] is to values greater than data[0] but less than data[1], child[2] is to values greater than data[1] but less than data[2], and child[3] is for values greater than data[2]
- 2-nodes will have two uninitialized slots in the data and child arrays

Lec 4.3 – Graph ADTs

Concepts

A graph G = (V, E) is a general data structure that consists of a set V of vertices (collections of items) and a set E of edges (relationships between items)

- Graphs may contain cycles, and there is no implicit order of items
- Vertices are collections of items, and therefore nodes
- Edges are relationships between items, and therefore pointers (links) between nodes (bit of an oversimplification, but just for the idea)
- |V| and |E| stand for the cardinality → the number of vertices and edges of a graph G = (V, E)
    - A graph with |V| vertices has at most |V| * (|V| - 1) / 2 edges
    - If |E| is closer to |V|^2 than |V|, the graph is dense, else if |E| is closer to |V| than |V|^2 the graph is sparse

Graph terminology

- Two vertices are adjacent if they have an edge between them
- An edge is incident on two vertices if it connects them
- The degree of a vertex is the number of edges it is incident with
- A path is a sequence of vertices where each vertex has an edge to its predecessor (you can 'walk' from the first vertex in the sequence to the next vertex and so on until you arrive at the last vertex)
- A cycle is a path where the last vertex in the path is the same as the first vertex
- The length of a path is the number of edges in it
- A connected graph is a graph where there is a path from each vertex to every other vertex → graphs that are not connected are disjoint and have multiple subgraphs
    - Subgraphs are graphs created from a graph by taking only some and not all vertices and edges
- Complete graphs are graphs where there is an edge from each vertex to every other vertex → all vertices are adjacent to all vertices
    - Cliques are complete subgraphs

Trees are connected graphs (or subgraphs) with no cycles

- Spanning trees are graphs (or subgraphs) that have a path from all vertices to all vertices without having a cycle → a tree that contains all vertices of a graph, but maybe not all edges to avoid cycles
- A spanning forest of a non connected graph G = (V, E) is a subgraph of G containing all of V, and is a set of trees with one tree for each connected component → 28:53 lec 5.1 → if you can make a spanning tree in each of the disjoint subgraphs of G, you have a spanning forest

Directed graphs are those where edges have arrows, and you can only move in the direction of the arrow

- Linked lists and most data structures that use pointers are directed graphs since the pointer only goes one way

Weighted graphs are graphs where the edges have a ‘weight’ attributed to them

Graph ADTs

- Core operations
  - Creating
  - Destroying
  - Inserting vertex/edge
  - Deleting vertex/edge
  - Searching for vertex/edge

We have 3 ways of representing Graphs in code → summary of comparisons at 56:33 of lec 5.2

- An array of edges (vertex pairs)
  - Space efficient, and mildly complex adding/deleting and lookup
  - An array of struct edges
  - For directed graphs, the ordering of vertices within an edge pair denote direction (x, y) means x→y only, not y→x
- An adjacency matrix
  - A |V| by |V| matrix with a 1 signifying an edge and 0 signifying no edges
  - Stores whether or not there is an edge between two vertices in a 2D array
  - Easy to implement
  - Not memory efficient for sparse graphs
- An adjacency list
  - Edges are defined by entries in an array of V lists

Array-of-edges cost

- Takes $O(E)$ storage
- Initialisation is very fast with $O(1)$ complexity
- Inserting and deleting has complexity $O(E)$
- If we maintained the edges in order, we could have a search complexity of $O(\log E)$ using binary search

Adjacency Matrix cost

- Takes $O(V^2)$ to initialise (the nested for loop to initialise the adj matrix)
- ONLY takes $O(1)$ to insert or delete edges

Adjacency List cost

- Takes $O(V + E)$ storage → much more efficient than adj matrices for sparse graphs
- Initialization, insert and delete are all $O(V)$
- There is no benefit to sorting the vertex lists, giving us less to do

EXAMPLES:

- Graph.h + Graph.c + GraphMain.c

- Graphs.c has 3 versions of the basic operations, one version for each method of representing graphs

Code

```
typedef struct GraphRep {
	Edge *edges;
	int nV;
	int nE;
	int n;
} GraphRep;
```

- The struct we use to represent a graph using the array of edges method
- Edge *edges is a dynamic array we store the edges in
- nV is the number of vertices (vertices are numbered 0 to nV – 1)
- nE is the number edges (edges are numbered 0 to nE – 1)
- n is the size of the edge array

Lec 5.1 – Graph Traversal

Concepts

Traverse for…

- item searching → using an algorithm to move between nodes to look for a value
    - Boolean visited array
- Path finding → using an algorithm to look for a value, but keeping track of how to get there
    - Predecessor array
- Traverse fully → path finding but not looking for a value, just exploring the entire graph
    - Use a predecessor array and loop until queue is empty
    - OR use a Boolean visited array and loop until all visited

Graph traversal is the systematic exploration of a graph via the edges

- Focus is often to explore paths between a starting and finishing vertex
- Solutions/methods can be iterative or recursive
- Sometimes traversal will require us to store the path we explore as we explore it, and sometimes even the nodes

Two main traversal methods → 12:02and 14:30 lec 5.3:

- Depth-first search (go deep) → recursive and iterative solutions
    - Goes all the way down one edge, then moves to the next edge and repeats
    - Can go to smallest vertex first or any order, as long as it is consistent and has some order
- Breadth-first search (go wide) → iterative solutions only
    - Visits nodes one edge away from the starting vertex, then two, then three and so on
- Both methods ignore some edges by remembering previously visited vertices so as to avoid cycles, and thereby infinite loops/recursion
    - Often done using arrays or linked lists
    - Predecessor arrays/linked lists are those that keep track of where the indexed vertex (the vertex number and index in the array are the same) came from → 17:59
        - Typically initialised to -1 or NULL or 9999

Breadth first search involves starting at a node then expanding out equally from there → 33:18 walkthrough

- You visit all the neighbours of the current vertex, then all the neighbours of those neighbours → REMINDER neighbours are vertices one edge away from a vertex
    - NOTE: the order we visit neighbours doesn't matter, as long as it is consistent
- Pseudocode at 19:41
- We often store a queue Q (some other ADT on the stack) of vertices not yet visited, and a visited/predecessor listed of vertices visited
    1. Insert the index of the starting vertex into Q

2. Mark it as visited by itself in the predecessor array PA, and remove the starting vertex from Q
3. Add all unvisited neighbours of the starting vertex into Q using listAppend, and mark them as visited by the starting vertex in PA
4. Remove the head of Q from Q and append all its unvisited neighbours to Q then mark those off as visited by the removed head
5. Repeat step 4 until either Q is empty or the end/finishing vertex has been visited

- The predecessor list lets us track the path we took to find the end vertex
    - We go the index of the end vertex then go the index of the value there and so on
    - In the 37:07 example, the end vertex is 9 so we go to arr[9] = 7, so we go to arr[7] = 5 then arr[5] = 0 and arr[0] = [0] so it is our starting vertex and thus we have the path 9 <- 7 <- 5 <- 0
    - THIS WILL GENERATE THE SHORTEST PATH IN AN UNWEIGHTED GRAPH
- BFS with an adjacency list for unweighted graphs has complexity O(V + E)

Queue NOTE:

- A Queue is an ADT that utilises the stack
- Like pushing and popping in 1521, the most recently added Item is on the top of the stack, and is therefore the first thing popped → first in first out
- Its implementation sometimes involves a struct that has 2 pointers to the top of two stacks
-

Depth first search involves going as deep as possible until you reach a dead end then unwinding back 'up' until there is another path to take, and going as deep as possible on that other path

- Recursive tree searching is a depth first search
- Pseudocode at 48:23, iterative walkthrough at 50:30
- Instead of using a queue Q, we use the stack in recursive calls
    1. Add the start vertex to the stack, then mark it as visited by 0 and remove it from the stack
    2. Add neighbours of the start vertex to the stack starting with the largest and mark them as visited by the start vertex
    3. Remove the most recently added value on the stack and add its unvisited neighbours, marking them as visited by the smallest
    4. Repeat 3 until the stack is empty or the end has been found
- NOTE: THERE MAY BE AN INCONSISTENCY IN THE LEC METHOD WITH THE LEC DIAGRAM, AS SOME ONLY MARK THE VISITED ONCE THE VERTEX IS POPPED OFF THE STACK → THE LEC METHOD AT 50:30 MAY HAVE BEEN ITERATIVE → just watch 23T3 lec
- DFS with an adjacency list for unweighted graphs is also O(V + E)
- DFS does not always return the shortest path/solution

Recursive DFS → pseudocodes at 1:06:46

- Full traversal using DFS is really simple recursively → elegant solution

NOTE: all full traversals produce spanning trees

Examples:

- Lec04 → Graph.c → GraphFindPath
- Labs07 → solveBfs and solveDfs both require us to find a path

Code

Lec 5.2 – Graph Algorithms and Problems

NOT AS IMPORTANTS AS KNOWING BFS AND DFS

Concepts

Other graph algorithms include:

- Cycle checking
- Connected components
- Hamiltonian path and circuit
- Euler path and circuit
- WE ARE GIVEN THE CODE AND NOT EXPECTED TO KNOW???

A graph has a cycle if, at any point in the graph, it has a path of length > 2 where the start and end vertex are the same, and the path does not use any edge more than once

- Graphs can have multiple cycles
- Cycle checking involves checking if there is a cycle between two vertices and looping over all possible vertex combinations
    - One function to check if a vertex is part of a cycle, which calls the second function
    - A second function to loop over all possible vertex combinations
- Cycle checking typically uses DFS

To find out how many connected subgraphs there are in a graph

- We can create a componentof[] array, where each index refers to a vertex and the value at that index refers to which subgraph it is in → 9:05 lec 5.4, note component means subgraph
- Pseudocode and walkthrough at 9:47
    1. Initialise componentof[] to -1, and set the CompID, which tells us which subgraph, to 0
    2. For all vertices (from v = 0 to g->nV → v < g->nV), if componentof[v] == -1, call another function dfsComponent which takes in the graph G, current vertex v and CompID, then increment CompID by 1
    3. The dfsComponent function should set componentof[v] = CompID, then call itself for all of v's neighbours that have componentof == -1

Caching information

- In situations where connectivity is critical, we can't afford to run components each time, so we cache some information by adding information to our graph implementation struct
    - We add the nC field which is the number of connected components
    - We also add a *cc field which is an array showing the component each vertex is part of → the componentof[] array
    - Adding and removing edges may increase or decrease nC and will change the cc array
- When searching, we can first check that the start and end vertices are in the same component so as to not waste time running impossible searches/finding paths

Hamiltonian

- Hamiltonian paths are paths that connect two vertices in a graph G such that the path includes every vertex exactly once → can't backtrack
    - A Hamiltonian path search generates simple paths using DFS, keeping count of how many vertices visited in the current path, until either a path containing nV vertices is generated or all possible simple paths have been generated
    - Pseudocode at 21:42
    - Worst case O((V – 1)!) complexity → there are (V – 1)! Possible paths
    - This 'problem' has no simpler algorithm than the one shown in the pseudocode, so it is 'NP-hard'
- Hamiltonian circuits are Hamiltonian paths where the starting and ending vertex is the same

Euler

- Euler paths are paths that connect two vertices in a graph G such that the path includes each edge exactly once → can revisit vertices, but still no backtracking
    - A graph has a Euler path (non-circuit) iff it is connected and exactly two vertices have odd degree
- Euler circuits are Euler paths where the starting and ending vertex is the same
    - A graph has a Euler circuit iff it is connected and all vertices have even degree
        - Check that num edges is the same as nE
        - Every edge is connected to the next edge
        - All edges are unique
- Pseudocode at 35:33
    - If we assume connectivity is already checked, we have O(V) if we have O(1) lookup time for a vertex's degree → we need to check the degree for all V vertices, so one loop with V iterations
        - Check that either all vertices have even degree or exactly two have odd degree
    - If we need to calculate a vertex's degree, we have O(V^2) complexity since we need a nested loop iterating over every vertex to calculate degree

Examples:

- Cycle checking code at 3:58 of lec 5.4

P, NP and NP-Complete

- Generalising different categories of algorithms allows us to understand how feasible they are to solve a given problem →
- Big-Oh only really measures time complexity, but most algorithms are either reasonably solvable or → solvable refers to how easy it is to get an answer (e.g. a solvable sorting algorithm easily gives you a sorted array/ll)
- When it comes to problems we must solve in computing, there are two key ways we can look at the difficult of the problem
    - Solving refers to how hard it is to find or guess the correct answer
    - Verifying refers to how hard it is to verify if a given answer is correct
    - Table at 6:34 of 7.1

Hard/NP problems to solve usually take long amounts of time to solve → 22:37

- Most of these problems have O(n!) or O(k^n) or worse time complexities (factorial or exponential time complexities) → the amount of time taken with every step up in input size increases massively
- They generally cannot be solved by large data sets by standard computers
- We describe the algorithms required to solve hard problems as non-deterministic polynomial (NP) time problem → e.g. algorithm to find Hamiltonian path O((V – 1)!) is NP-hard
- To solve NP problems in a reasonable amount of time, we would need some theoretical (non-existent) supercomputer that can simulate many-to-all possibilities at the same time, or guess the correct answer perfectly instantly

Easy/P problems to solve still take large amounts of time to solve for large input sizes

- Most of these problems have O(log n), O(n), or some O(n^k) time complexities → log(n) or some polynomial
- We describe the algorithms required to solve easy problems as polynomial (P) time problems
- Easy problems can be solved, even with large input sizes, can still be feasibly solved using certain external strategies (paralleling processing, servers, etc.) → though they would still take a while, they are not ridiculous

Summary:

- P: Polynomial
    - Can be solved in polynomial time
    - Can be verified in polynomial time
- NP: Non-deterministic Polynomial
    - 'Can' be solved in non-deterministic polynomial time
    - Can be verified in polynomial time
- NP-Complete:
    - 'Can' **only** be solved in non-deterministic polynomial time
    - Can be verified in polynomial time
    - NP-Hard category includes this?

Code

Lec 7.1 – Directed/Weighted Graphs

Concepts

In the real world, the edges of graphs often have a sense of direction and cost to traverse

- Directed → sense of direction
- Weighted → sense of cost to traverse

Directed graphs (digraphs) are graphs whose edges have a sense of direction, such that an edge from v to w does not imply an edge from w to v (one directional)

- Directed paths and cycles exist
- Degree
    - Outdegree → deg(v) = number of edges in the form (v, _) → edges from v to somewhere
    - Indegree → $\deg^{-1}(v)$ = number of edges in the form (_, v) → edges from somewhere to v
- Reachability → w is reachable from v if there is a directed path from v to w
- Strong connectivity → every vertex is reachable from every other vertex
- Directed acyclic graphs are digraphs that contain no directed cycles
- 8:44 lec 7.2 → time complexity of different digraph implementations
- Traversing is the same as for undirected graphs, as the way we store edges is the only thing that has changed

Adj matrix

- 1 if there is an edge from the row vertex to the column vertex, 0 otherwise → look at (0, 2) and (2, 0)at 9:37
- No more symmetry

Adj list

- Store only the (v, _) edges for vertex v → linked list for v will have x, y, z if (v, x), (v, y), (v, z)

Weighted graphs are graphs whose edges have a sense of cost/weight

- This means some edges will be 'cheaper' to explore than others
- Edges are now represented as (s, t, w) instead of (s, t), where w is the weight of the edge the source s to target t
- Main problems to solve with weighted graphs:
    - Cheapest way to connect all vertices for undirected weighted graphs → minimal spanning tree problem (unlike Hamiltonian paths, minimal spanning trees can have branches)
    - Cheapest way to get from A to B for directed weighted graphs → shortest path problem

Adj matrix

- We store the weight of the edge instead of a 1 if there is an edge between two vertices, and a 0 otherwise → the adj matrix must be an int or double, NOT bool

- 27:00 lec 7.2 → for undirected weighted graphs
-

Adj list

- Our Link/Edge nodes need an additional weight field → struct node will contain an int (the index of the vertex adjacent), another int or a double (the weight of the edge), and a pointer to the next node

Edge array

- Struct edges needs an additional weight field → Vertex v, Vertex w, int or double weight

Code

Lec 7.2 – Minimum Spanning Trees

Concepts

Whenever we have a complex network of things and want to find an efficient way of connecting every vertex, we need to find a minimum spanning tree

- Spanning Trees (STs) are connected subgraphs that contains all vertices with no cycles
- Minimum Spanning Trees (MSTs) are STs whose sum of edge weights is no larger than any other ST → the ST with the minimum sum of edge wights
- We could just use brute force and generate all STs then find the one with the minimum weight, but that is expensive and inefficient
- Kruskal and Prim's algorithms are efficient 'methods' of generating minimum spanning trees from a graph

Kruskal's Algorithm → walkthrough at 11:22 lec 7.3 → pseudocode at 16:14

1. Start with an empty MST
2. Consider edges in increasing weight order
    - Array of edges → sort array by weight
    - Adjacency list → create a sorted list of edges?
    - Adjacency matrix → create a sorted list of edges?
3. Add the next edge if it does not form a cycle
    - Cycle checking could be done using DFS that returns true (hasCycle) if it tries to visit something that has already been visited → uses a visited list not predecessor since we don't care about the path → normal DFS already has cycle checking that stops it from going into an infinite loop/recursion
4. Repeat until V – 1 edges are checked/added

Kruskal's Algorithm time complexity → very unsure

- Sorting the edge list is O(E logE) (using an O(n logn) sorting algorithm), and there are at least V iterations to loop through the edges
- On each loop, we use O(1) to get the lowest cost edge, and use O(V^2) to check whether adding it forms a cycle or not
- Overall time complexity is O(E logE) = O(E logV) IDFK Y, IT JUST IS

Prim's Algorithm → walkthrough at 41:00, pseudocode at 43:58

1. Start from any vertex v and an empty MST
2. Choose an edge not already in the MST to add
    - Chosen edge must be the minimum weighted edge that connects one vertex in the MST to one not in the MST → connecting one in to one not in avoids cycles
3. Repeat until MST covers all vertices → V – 1 edges added

Prim's Algorithm time complexity

- V iterations of outer loop
- In each iteration, we find the min-weighted edge
    - Using a set of edges, the cost of this is O(E) → overall O(V * E)

    - Using a priority Queue, the cost of this is O(log E) → Overall O(V * log E)
- NOTE:
    - Using a priority Queue allows for non-recursive traversal
    - MORE ON PRIORITY QUEUES LATER

Kruskal’s v Prim’s

- Prim’s is better for dense graphs
- Kruskal’s is better for sparse graphs

Code

Lec 7.3 – Shortest Path

- Shortest path
- Edge relaxation
- Dijkstra's Algorithm

Concepts

The shortest path between two vertices is the path with the minimum weight

- This is on weighted graphs with non-negative weights

We have something similar to a BFS, but with weights → 5:18 walkthrough lec 7.4 → 16:50 walkthrough

- To find the shortest path from s to all other vertices, we have
    - dist[] → a V-indexed array of the cost of the shortest path from s to the indexed vertex
        - We put -1 or infinity or whatever if there is no path from s
    - pred[] → a V-indexed array of predecessors in the shortest path from s → stores the vertex we just came from → gives us the actual path

1. Begin at the vertex u want to find the shortest paths from
2. Inspect all the vertices it has an edge to, and update the dist[] and pred[] of those vertices if necessary (edge relaxation) → the if code down a bit
3. Explore the vertex connected by the edge with the smallest weight, and repeat step 2

Edge relaxation occurs while we explore the graph for the shortest path → 11:55 walkthrough

- This is due to the fact that dits[] and pred[] show the shortest path found so far
- If dist[v] and dist[w] are the shortest known paths from s to v and w, and an edge(v,w,weight), edge relaxation updates data for w if we find a shorter path from s to w

    -
      ```
      If (dist[v] + weight < dist[w]) {
              dist[w] = dist[v] + weight;
              pred[w] = v
      }
      ```
- We typically explore along the shortest route first when doing the BFS to find all shortest paths, since it makes sense that following the shortest route is more likely to lead to the shortest paths

Dijkstra's Algorithm Time Complexity

- Each edge needs to be considered one → O(E)
- Outer loop has O(V) iterations
- Finding the adjacent vertex with the minimum dist[]
    - By trying all s in vSet, the cost is O(V) → Overall O(E + V^2) = O(V^2)
    - Using a priority queue to implement extracting minimum → Overall O(E + V*logV)

Quick Note:

- E < V^2 always since E =

Code

13:15 lec 7.4 → Dijkstra’s Algorithm

Lec 8.1 – Heaps and Priority Queues

Concepts

Heaps can be thought of as tree-like structures used for things like priority queues

- NOT TO BE CONFUSED WITH HEAP MEMORY
- HEAPS ARE NOT ACTUALLY TREES, THEY ARE ARRAYS THAT CAN BE CONCEPTUALISED AS TREES
-

Heaps can be conceptualised as dense tree structures where: → 7:35 of lec 9.1

- The tree maintains a general order with higher (larger) elements up top → parents are larger than children
- Items are inserted into the tree in level order → they are added in the lower-most, left-most empty leaf then drift up to the appropriate level in the tree → insert in the level's leaves from left to right → the first unfilled slot in the array
- For nodes on the same level, Items on the left are larger and Items on the right are smaller
- Items are deleted by removing the root → the top priority node
- Since heaps are 'dense trees', the depth is $floor(\log(2N) + 1$

Heaps are often implemented as arrays

- The implementation often assumes we know the max size
- Simple index calculations allow tree-like navigation:
    - Left child of item at index i is located at 2i
    - Right child of item at index i is located at 2i + 1
    - Parent of Item at index i is located at i/2
- The 0th element/indexed slot is always empty → The index calculations get fucked if there is an element indexed 0

Insertion into a heap is a 2 step process: → walkthrough at 15:00

1. Add the new element at the next available position on the bottom row
2. Reorganise values along path to root to restore the property of heaps that parents are larger than children
    - NOT AS COMPLEX AS TREE REBALANCING → just swap item up to its correct position

- Adding new item to end of array is O(1), moving the item up is O(log (2n)), so overall time complexity is O(log n)

Deletion is a three step process: → walkthrough at 23:24

1. Replace the root value by the bottom-most right-most value
2. Remove the bottom-most right-most value
3. Reorganise values along path from root to bottom-most right-most value to restore heap

- We usually want to return the root value we delete
- Replacing root by item at end of array is O(1), moving the new root down into the correct position is O(log (2n)), so overall time complexity is O(log n)

Heap behaviour is the same as the behaviour required for a Priority Queue

- 30:37
- The join or enqueue function adds an Item to the end of the Queue, and ensures the highest priority item is at the front of the Queue
- The leave or dequeue function removes the highest priority item in the Queue and returns it, whilst also ensuring that the highest priority item is at the front of the Queue
- To create a PQ, we can just typedef a heap as a PQ, then the join or enqueue function is just HeapInsert, and leave or dequeue function is just returning HeapDelete

There are many ways of implementing PQs

- Sorted and unsorted arrays and linked lists, and heaps
- Time complexity comparison at 34:19 lec 9.1

Heap Sort → O(N logN)

1. Build a heap from the array
    - Iterates through the array N times to insert each element into the heap
    - Inserting needs fixup O(log N), and we do this N times so O(N logN)
2. Use the heap to build a sorted array
    - Iterates through the array N times from the end to the start, inserting the root of the heap into the array and deleting the root of the heap each time
    - Deleting the root requires fixDown O(logN), and we do this N times so overall O(N logN)

- Space expensive since we have to create the heap
- Not stable, not adaptive

Examples:

- lecs→lec08→ Heap.h and Heap.c

Code

Heap struct at 12:17 lec 9.1

- newHeap function as well
- Heap struct has an array of items, and two integers keeping track of the number of items in the array (nitems) and the max size (nslots) of the array (how many elements it can store)

Lec 8.2 – Hashing

Concepts

Ordered/sequential containers are those that have a sense of order between elements

- Typically, we can assign a meaningful index to each element to denote order → the structures are indexable → we can extract some order from them
- Linked lists, arrays and trees are ordered containers
    - Linked list have the head then the next and the next next and so on for order

Associative containers are those that map "keys" to various values

- Items in associative containers have no sense of order, and the keys are typically strings
    - Think of it as a bunch of keys that point to some values
- So we could do something like a["abc"] and assign that a value instead of using an index → allows us to label things by name instead of number

Hashing is how we map "keys" to various values

- We convert the key to an index then access the index in some array or structure to map to the value
- We take the given key, and our hash function converts it into an index that we use to map to some value → the sets of indexes and values is called the hash table

To use arbitrary values as keys, we need:

- A set of keys (strings) and values dom(key) → each key defines one Item
- An array of size N to store Items → the hash table?
- A hash function h() of type dom(key) → [0.. N – 1]
    - This requires that if x = y, then h(x) = h(y), and that h(x) always returns the same value for the given x
- PROBLEM: array is size N, but dom(Key) > N in nearly all cases, so we will have collisions → situations where two keys point convert to the same index and collide in the hash table → WILL GET TO THIS LATER

The Hash Table ADT has some basic operations

- HashTable newHashTable(int) → makes a new empty table of size N
- void HashInsert(HashTable, Item) → adds an Item into the collection of Items
- Item *HashGet(HashTable, Key) → finds the Item using the key
- void HashDelete(HashTable, Key) → finds and deletes the Item using the key
- void dropHashTable(HashTable) → frees all memory associated with the HashTable

The actual hash function takes in a key and the size of the hash table (N)

- We convert the key into a 32-bit int, then return that int % N → ensures we get a valid index → this is where collisions can happen
- If keys are ints, we just use the identity function
- If keys are strings, well fuck its complex as shit
    - We may have different strings whose sum of ascii values are the same

    - Different strings composed of the same chars
- 31:05 → the universal hashing function, and a hashing function for a database system

HashTables cost analysis

- O(1) search (lookup), insertion and deletion time in ideal scenarios
    - This will change if we have a lot of collisions
- BUT not very space efficient because we need a big array (the hashtable)

Problems with hashing

- The hash function relies on the size of the array, so the size must be kept constant or key hashing starts fucking up → hash tables are not dynamic
    - If we want to change the array size, we need to create a new one then re-insert all Items
- Items are stored in random order
- If size(KeySpace) >= size(IndexSpace), then collisions are inevitable → if the size of all possible combinations of keys is >= the size of all possible indexes, collisions are inevitable
    - $k \neq j$ BUT hash(k, N) = hash(j, N)
- If nitems > nslots, collisions are inevitable

Examples:

- lecs→lec08→ HashTable.h and HashTable.c

Code

Lec 8.3 – Hashing contd

Concepts

Three ways of dealing with collisions:

- Separate chaining → allows multiple items at a single array location
    - Could use something like an array of linked lists, but that will cause a worst case time complexity of O(N) from traversing the linked list
- Linear probing → systematically computes new indexes until a free slot is found
    - Needs strategies to compute new indexes
- Double hashing → increases the size of the array?
    - Kind of a different way of doing linear probing???
    - Needs a method to “adjust” hash()

Separate chaining solves collisions by having multiple items per array entry

- Each element in the array is a pointer to the start of a linked list of items
- All items in a given list have the same hash value
- Items are added using listAppend
- Modifications at 8:06 lec 9.2
    - Needs a list ADT

Separate chaining cost analysis:

- If we have N array entries (slots) and M stored items
    - The best case (good hash) is all lists are same length L = M/N
    - The worst case (Bad hash) is one list of length M
- Searching within a list of length n has 1 at best, n at worst and n/2 average so O(n)
- If good hash and M <= N, cost is O(1)
- If good hash and M > N cost is (M/N) / 2 → O(M/N/2)
    - Worst case time complexity for a good hash is O(M/N)
- Worst case time complexity for a bad hash is O(M)
- The Ratio of items to slots is called $load\ \alpha = \frac{M}{N}$
    - NOTE: separate chaining is the only method that can handle a load > 1 without remallocing

QUESTION: ARE KEYS STORED IN ITEM → IS ITEM A STRUCT WITH THE VALUE AND THE KEY?????????? → HOW DO I KNOW WHICH NODE IN THE LL TO TAKE OUT OR WHICH SLOT IN THE ARRAY AFTER LINEAR PROBING???

NOTE: Keys and Key values are stored as a pair (struct?) in the hash table

- This is how we know which Node in the LL to take out, or which slot in the linear probed array
- I THINK THIS IS HOW IT WORKS AT LEAST
- EXPLANATION AT 42:00 → SO Item IS A STRUCT???

Linear Probing solves collisions by finding a new location for the Item

- Hash indicates slot i which is already used

- We try the next slot and the next next until we find a free slot, whereupon we insert Item
- Since the value at every index stores the original hash value, it is very easy to tell when we have found it
- CODE AT 25:43 LEC 9.2 (Part 2) → CBF MODIFYING HashTable.c AGAIN
    - The for loops circled in red check if the slot is empty, then if it is the correct slot for the input key → it breaks out of that loop if it is, bc that means it found the correct slot for the key
    - HashDelete code at 32:33 → walkthrough at 35:00
        - The lines after the for loop cleans up the 'probe path' → the path from the hashed index to the actual slot → ensures all slots after the hashed indexed have the same hash value → will remove and reinsert all Items after the one we delete

Linear Probing cost analysis:

- Cost to reach first Item is O(1), subsequent costs depends on how much we need to check?
- Performance is affected by load α → the closer it is to 1, the worst it gets, and it cannot handle $\alpha > 1$
- MORE SHIT AT 42:43 BC FUCK → AIGHT ALL WE GOTTA KNOW IS THAT LINEAR PROBING GETS VERY EXPENSIVE AS α APPROACHES 1
- Average cost for successful search with good hash and reasonably uniform data is 0.5 * (1 + 1/(1 – α)), and 0.5 * (1 + 1/(1 – α)^2) for an unsuccessful search

Double Hashing improves on linear probing

- It does so by using an increment that is based on a secondary hash of the key and ensures all elements are visited
- Tends to eliminate clusters, creating shorter probe paths → eliminates clusters by spreading more NULLs throughout the array that will end search loops and stuff?
- CODE AT 47:34 Lec 9.2 (part 2)
    - nhash2 is a hash mod value we calculate when we create the hash table

Double hashing cost analysis → 51:28

- Cost to reach first item is O(1), subsequent cost depends on how much we need to scan
- Affected by load α → more expensive as α approaches 1
- Average cost for successful search is $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$, and $\frac{1}{1-\alpha}$ for unsuccessful searches
    - Compared to linear probing, successful searches are a little quicker, and unsuccessful searches are a lot quicker

Summary:

- Chaining is easy to implement and allows for $\alpha > 1$, but performs poorly when $\alpha > 1$
- Linear probing is fast is $\alpha < 1$, but complex deletion
- Double hashing is faster than linear probing, especially as α approaches 1
- For arrays, once M exceeds the initial choice of N, we need to expand the size of the array, which impacts the hash function so we many need to rebuild the entire table

Lec 9.1 – Tries

Concepts

Tries are data structures that store large sets of strings efficiently

- They have O(L) lookup and insertion → L is the length of a string
- We could use hash tables, but hash tables are expensive for large sets of input and when collisions are likely
- Diagram at 3:31 of lec 9.3
- They have a tree-like structure, and each word is a path down one branch
- The depth of a trie is the length of the longest word it stores

Tries structure

- The root node is empty → it does not contain a char
- Each node
    - Contains one part of a key (typically 1 char)
    - May have up to 26 children (alphabet) → an array of 26 pointers to the child nodes
    - May be tagged/marked as a "finishing" node → the letter is the last letter of a word
        - Finishing nodes can still have children
        - Can contain other data for application, such as the frequency of the word
    - PROBLEM: each pointer is 8 bytes, and we have 26 so the array alone is 208 bytes → total 214 bytes, which is a really big node → tries are not very space/memory efficient, but are much faster for operations

Extra note??:

- We can easily index into the array and check if there is a char we want in the Trie branch by doing keychar – a and using the result as the index into the children array
- keychar – a will give keychar's position in the alphabet → we would have to also insert every element into its correct position in the children array
- This would avoid the need to loop every time, and make the lookup cost of the children array O(1)

Tries cost analysis

- Space complexity of O(n) where n is the sum of all strings' lengths in the worst case
- Search and insertion complexity of O(m) where m is the length of the key string we want to find/insert

Trie Insertion → pseudocode at 16:24

- Go down the path for the given key, inserting a new node for each char not already there in the Trie branch

BST-like tries → diagram at 24:52

- The normal Trie representation is very space inefficient

- We could reduce the branching factor by reducing the alphabet:
    - Break each 8-bit char into two 4-bit “nybbles”
    - Each branching factor is then 16 even for a full ascii set of chars
    - But each branch will be twice as long
- I think nodes have a child pointer and a sibling pointer
    - Sibling points to another letter not in the word, but one that a parent may form a different word with
    - Child points to the next letter the current and parent nodes form a word with
- Go to next sibling if curr not next letter in word, else go to child
- The sibling nodes are ordered alphabetically
- DON’T THINK WE NEED THIS SHIT

Compressed tries → diagram at 32:44

- consecutive common nodes with <= 1 children become ‘compressed’ into a single node with the characters in those nodes as a string
- This saves space, but can be difficult to restructure when adding a new string that splits a compressed node

Code

Lec x.y

Concepts

Code

Lec x.y

Concepts

Code

Lec x.y

Concepts

Code

Lec x.y

Concepts

Code