LINKED LISTS

* Sequential (no random access)
* Rearrangeable
* Self referent (list element links to another)

Using Malloc

Common Linked List Uses

*Reversing a Singly Linked List*

*Duplicating a List*

**

RECURSION

* Problems can be expressed simpler
* Develop a function that calls itself
* Must include
  + Base case
  + Recursive case: calls the function on a ‘smaller’ version of the problem
* Recursive lists not very useful on large lists

*Printing list in Reverse*

*Finding Maximum in array* ***(O(logN))*** *Each call is O(N)*

*Checks if all elements in array are even*

**In the worst case - that they are all even, all elements in the array must be checked so the time complexity ends up being O(n).**

FUNCTION POINTERS

* Refers to functions’ memory addresses
* Can be assigned/passed variables

Declaration

*typeOfReturnValue (\*functionname)(typeOfArguments)*

Using Function Pointers

*Used with recursion*

*Applies a function to all the nodes (e.g. printList)*

**

TESTING

* Cannot establish that a program is correct
  + This would require showing all possible inputs produces correct output
  + Impossible except in trivial cases
* Different types of parameters require different cases of testing
  + Numeric:
    - Positive
    - Negative
    - Zero
    - Large values
    - Boundary cases
  + String
    - String length
    - String empty
    - 1 element
    - Many elements
  + Properties
    - Increasing
    - Decreasing
    - Random

Blackbox Testing VS. Whitebox Testing

Blackbox Tests

* Testing code from the outside (only using .h functions)
  + Checks behaviour
  + Input = Correct output?
* Doesn’t know implementation:
  + If implementation changes, tests should still pass

Whitebox Tests

* Testing code from the inside (.c functions)
  + Checks code structure
  + Tests internal functions
* These tests rely on and can access implementation (i.e. arrays, linked lists)

Asserts shouldn’t be used in production code unless accompanied by a useful error message for clients

ABSTRACT DATA TYPES

* Abstraction is a way of ‘hiding’ complexity
* For consumer use, it should be enough to understand **what** (interface: .h files) it’s doing without knowing **how** (implementation: .c files)
* E.g. We operate a TV via its interface – remote control and buttons. We don’t need to open up the TV and see inside it

When designing a new library, it’s important to know..

* What are the abstract properties of the data types we want to provide?
* I.e. which operations do we need to:
  + **Create** (createList)
  + **Destroy** (freeList)
  + **Query** (numElements)
  + **Manipulate** (addNode)

A data type is…

* A set of values
* A collection of operations on these values
* E.g. linked lists, arrays

An abstract data type…

* Separates interface from implementation
* E.g. Stacks and queues
* Clients see only the interface
  + i.e. they don’t know if you used arrays, linked lists etc.
* **Allows implementation to change without breaking client code**

Indepth in regards to interface and implementation

* Interface:
  + .h file
  + typedef of ADT
  + Function prototypes fix function names and types
* Implementation
  + .c file/s
  + Structs – actual representation of data type e.g. linked list
  + Functions
  + Static Functions

Stacks  *Last in, first out*

* Two basic operations to **manipulate** a stack:
  + Push onto a stack
  + Pop from a stack
* Operation to **create** a stack
* Operation to **query** a stack
  + Is it empty?

Uses

* Backtracking search
* Evaluating expressions (balanced brackets)
* Function call stacks

Implementing Stacks

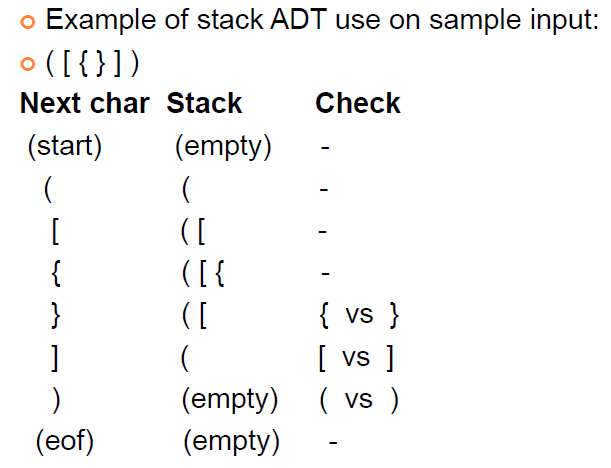
* As an array:
  + Allocate array with max elements
  + Fill items into a[0], a[1]…
  + Keep a counter of number of pushed items
* As a linked list:
  + Add new nodes to front of the list

(newnode->next = head);

* + Pop nodes from front of list



Examples



ABSTRACT DATA TYPES 2

Infix, Prefix and Postfix Expressions

* Infix:
  + 2 + 3
* Prefix:
  + + 2 3
* Postfix:
  + 2 3 +

Stack ADT Postfix Expression Exercise

Pseudocode:

1. Encounter a number, push it
2. Encounter an operator, pop two topmost numbers, apply operator to them then push result

HASH TABLES

* Used to map a universal set into a smaller one
* Always choose a tablesize that is prime

*Collision avoiding*

* Chaining – an array of linked lists
* Linear probing – check for next available empty space
* Double hashing – using another hashing algorithm to find next available space – second alg should be prime to table size

*Better to avoid collisions altogether*

* Grow table dynamically

*Hashing strings*



Queues *First in, first out*

* Two basic operations to **manipulate** a queue:
  + Put onto a queue
  + Get from a queue
* Operation to **create** a queue
* Operation to **query** a queue
  + Is it empty?

Implementing Queues

* As an array:
  + Allocate array with max elements
  + Fill items into a[0], a[1]…
  + Maintain counter for beginning and end of queue
  + Roll over when reaching the end of the array



* As a linked list:
  + Add node to end of the list when putting
  + Take node from front of the list when getting



WORK COMPLEXITY

*Problem:* A problem to be solved

*Algorithm:* Well defined instructions to solve problem

*Program:* Implementation of algorithm in a programming language

*Process:* Instance of program whilst executing on a machine

What makes software **good?**

* Returns expected result for valid inputs
* Behaves sensibly for invalid inputs
* Clear code
* Interface is clear and consistent
* Returns results quickly (efficient)
* Could be influenced by:
  + Memory/disk space, network traffic, disk IO

Algorithmic Complexity

Complexity Classes

* O(1)… constant functions e.g. g(n) = 1, 2, 3…
* O(n)… linear functions e.g. n/2, n, 5n+100…
* O(logn)… log functions e.g. log(n), log2(n)…
* O(n2)

General approach to analysing

* Go through code line by line and add up all the analysis
* This analysis is represented by big-O notation
* Big-O notation don’t have constants
  + E.g. for a problem of size n, cost of worst case is:

1.5n2 + 3n + 10 OR O(n2)

Searching in a Sorted Array (Linear **O(N)**)

Steps required to search an array of N elements:

* Best case: O(1)
* Worst case: O(N)
* Average: O(N/2)

Binary Search **O(log n)**

* Start in the middle of the array
* If a[N/2] == element, element found
* If a[N/2] < element

, search from a[0] to a[N/2 – 1] and vice versa

*Iterative Implementation:*

Analysing Algorithms

O(1): Constant, instructions in the program are executed a fixed number of times

O(log n): Logarithmic, divide & conquer algorithms with trivial splitting and combining operations

e.g Binary Search, BSTs (avg)

O(n): Linear, every element of the input has to be processed

O(n log n): Divide & conquer algorithms where splitting or combining operation is proportional to the input

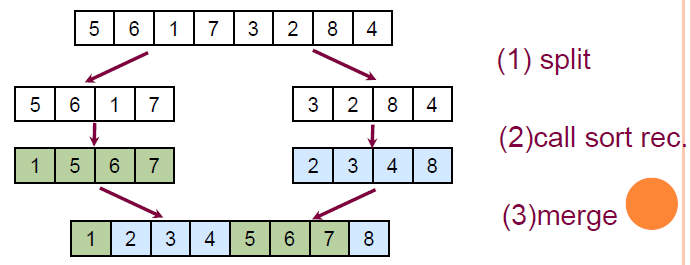
e.g. Merge Sort, Quick Sort

O(n2): Quadratic, algorithms that have to compare each input value with every other input value

e.g. Insertion, Selection, Bubble Sort

O(n3) and Exponentials: Unfeasible

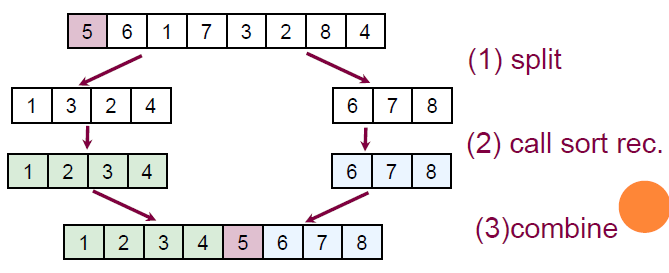
Merge Sort **O(nlogn), Stable, ~~In Place~~, ~~Adaptive~~**



*Recursive Implementation:*

Quick Sort (faster in practice)

**O(nlogn)-O(n2), ~Unstable, In Place, ~~Adaptive~~**



* O(nlogn) best case: both parts have same size
* O(n2) worst case: one part contains all items
* Ordered data makes it worse
* Can be made stable
* Not good for small partitions

*Recursive Implementation:*

****Median of 3 Partitioning Quick Sort

* Makes worst case scenario less likely
* Linked lists: Picking the pivot via randomising or median of 3 is O(n) instead of O(1) – picking first or last

*Non-Recursive Quick Sort Implementation*

**

Comparing Sorting Algorithms

* Worst case time complexity
* How much memory is used (In place)
* Stability (Duplicates preserved)
* Number of comparisons between items
* Number of times items are swapped

Insertion > Selection > Bubble unless ordered data in which case:

Insertion > Bubble > Selection

Bubble Sort **O(n2), Stable, In-Place, ~~Adaptive~~**

1. Compare first node with second
2. If first > second, swap
3. Repeat until no swaps required in a traversal

Bubble Sort Early Exit **O(n2) – O(n), Stable, In-Place**

* O(n) for ordered data

*Array Implementation:*

****Selection Sort **O(n2), ~~Stable~~, In-Place, ~~Adaptive~~**

1. Find largest node from original list
2. Insert into head of new list
3. Delete from old list and repeat

*Array Implementation:*



Insertion Sort **O(n2), Stable, In-Place, Adaptive**

* O(n) for ordered data

1. Take first node and insert into new list
2. Take next node and scan through the new list to insert in order

Shell Sort **O(h-sequence), ~~Stable~~, ~~In-Place~~, Adaptive**

* Improving Insertion Sort
* Take every h’th element
* H-Sort the sequence with smaller values of h until h = 1
* Knuth’s proposition: 1 4 13 40 121…
  + Easy to compute, results in efficient sort
* Better suited to arrays

*Array Implementation:*

**Key Indexed Counting Sort **O(n), Stable, ~~In-Place~~**

* Non-comparison based sorting
* Not very efficient for large keys; 1, 2, 536
* Use an array, count number of times each key appears
* Place items in new array based on index

E.g. sorting a[0] = 2, a[1] = 2 🡪 0, 0, 1, 1

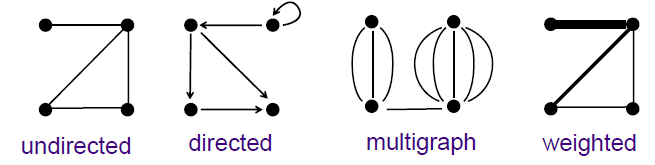
GRAPHS

* Applications require a collection/set of items
  + Maps: Items are cities, connections are roads
* Allow general, arbitrary connections

Answers that a graph should tell you

* Is there a way to get from item A to item B?
* What’s the best way?
* Which items are connected?

Different characteristics of graphs:



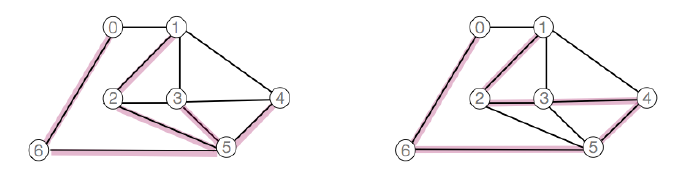
Vertices and Edges

Edges can be **<= V\*(V-1)/2**, ie. This formula gives the maximum amount of edges that can exist for a graph with V vertices

* Sparse Graph:
  + Adjacency list ideal (requires constant space)
  + Number of edges is close to minimal number of edges
* Dense Graph:
  + Adjacency matrix ideal
  + Number of edges is close to maximum number of edges

Terminology

* *Degree*: Number of edges from a vertex
* *Complete Graph:* Every vertex is connected to every vertex
  + Degree of every vertex is N-1
* *Adjacent:* Two vertices A and B are adjacent if there is an edge between them
* *Incident:* An edge is incident to a vertex A and vertex B
* *Subgraph:* Subset of vertices and their edges
* *Path:* Sequence of vertices connected to their predecessor
* *A graph is a tree:* when there is only one path between each pair of vertices
* *A path is simple:* when it doesn’t have any repeating vertices
* *A path is a cycle:* if it is simple apart from its first and last vertex
* *Connected graph:* if there is a path from every vertex to every other vertex in the graph
* *Disconnected graph:* consists of a set of connected components
* *Spanning tree:* Subgraph that contains all the vertices and is a tree



* *Spanning Forest:* Subgraph containing all the vertices and is a set of trees
* *Cliques:* Complete connected subgraphs

Path Terminology

* Hamilton Path (**NP complete**)
  + Connects 2 vertices while visiting every other vertex in the graph ONLY once
  + Hamilton Tour if it connects back to itself
* Euler Path **O(E+V) adj.list ~ O(V2) adj.matrix**
  + Connects two vertices while travelling along every edge exactly ONCE
    - Only if it is a connected graph and exactly 2 vertices are of odd degrees
  + Euler Tour if it connects back to itself
    - Only if it is a connected graph and all vertices are of even degrees

Directed Graphs

* Can have at most V2 edges
* Can have self loops
* Is a tree if one vertex is connected to all other vertices and there is at most one path between any two vertices

Weighted Graphs

* Each edge has an associated value e.g. distance

Multi Graphs

* Allows multiple edges between two vertices
* E.g. different modes of transport to the same place

ADT Interface

Adjacency Matrix Representation “Array of an Array”

* Advantages
  + Easily implemented as 2D array
  + Can represent graphs, digraphs and weighted graphs
* Disadvantages
  + Memory inefficient if there are very few edges

*Implementation:*

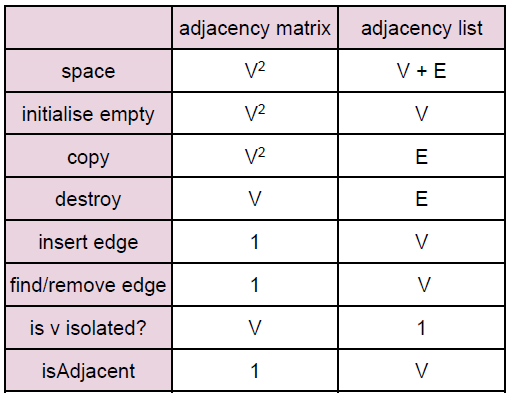
Storage Optimisation:

* If undirected, store only top-right part

Adjacency List Representation

* Advantages
  + Easily implemented in C
  + Can represent graphs and digraphs
  + Memory efficient if graph is small
* Disadvantages
  + One graph can have many representations unless ordered by some criteria – e.g. ascending

*Implementation:*



Graph Algorithms

Both of the below algorithms avoid cycles and some edges as well as use global variables ‘count’, pre[] array (preorder) and st[] array (spanning tree)

* Depth First Search **(Stack)**
  + Visit and mark current index before recursively visiting each neighbour
  + Each time we visit a vertex we record the previous vertex – if a graph is connected, this forms a spanning tree
* Breadth First Search **(Queue)**
  + Visit all the vertices adjacent to V before visiting all vertices adjacent to those visited in step 1
  + Can do BFS for every node as root and store shortest paths in a VxV matrix

Depth First Search

* Get the pre array, whenever you reach a vertex that doesn’t have any new valid vertices to go to, go back to the prev (st)
* **O(V2) for adjacency matrix**
* **O(V + E) for adjacency list**

*Stack Implementation*

*Recursive Implementation*

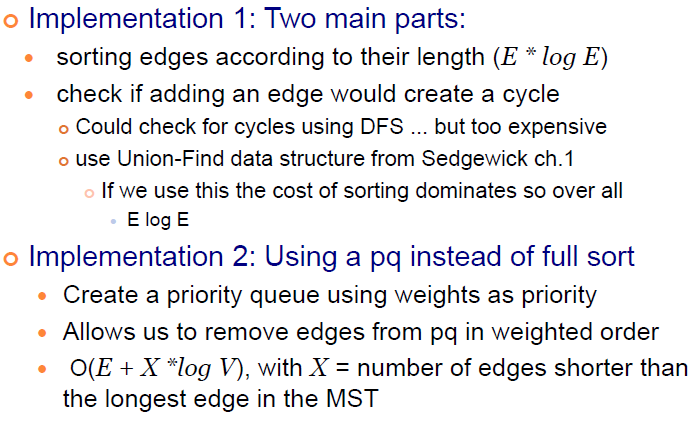
**

Breadth First Search

* **O(V2) for adjacency matrix**
* **O(V + E) for adjacency list**

*Queue Implementation*

*Kruskal’s Minimum Spanning Tree*

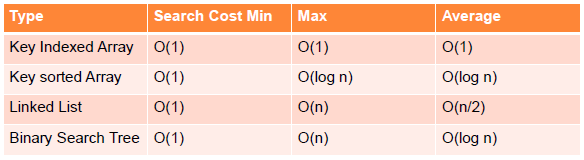


COMPUTABILITY

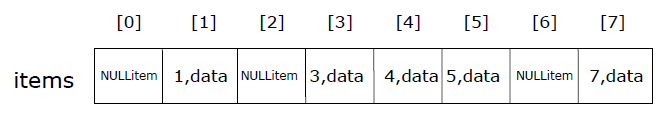
* Polynomial
  + A problem is in *class P* if its answer can be computed in polynomial time
* Non-Deterministic Polynomial
  + If the correctness of its answer can be checked in polynomial time
* Non-Deterministic Polynomial Complete
  + If it is in *NP* and is a difficult sumbitch
  + E.g. Hamilton Path, Longest path

SYMBOL TABLES

* Data structure of items that supports at least two basic operations: inserting and returning



*Key Indexed Array Implementation*



*Binary Search Tree Implementation*



* How to ‘select’ the kth smallest element of a search tree?

Binary Search Trees **Average: O(logn), Worst: O(n)**

* Branched data structures consisting of nodes and links with no cycles
* Binary trees: Each one has links to <= 2 nodes
* Perfectly balanced tree:
  + Weight balanced:
    - Size(leftsubtree) – size(rightsubtree) < 2
  + Height balanced:
    - Height(leftsubtree) – height(rightsubtree) < 2
* Visiting orders:
  + Depth first traversals
    - **Prefix (root 🡪 left 🡪 right)**
    - **Infix (left 🡪 root 🡪 right)**
    - **Postfix (left 🡪 right 🡪 root)**
  + Breadth first traversal aka Level Order
    - **Root 🡪 children 🡪 children’s children**
* Deletion
  + Case 1: Just delete the leaf
  + Case 2: Delete and replace with child
  + Case 3: Delete and replace with leftmost node from right subtree
  + OR

Just mark a node as deleted and ignore this node in future traversals

Balancing BSTs

*Rotations*

* Move nodes up to the root using rotations
* Left rotation: makes original root the left sub-child of the new root
* Right rotation: makes original root the right sub-child of the new root

*Partitioning*

* Similarly to select, moves the kth element of a tree up to the root

Global Rebalancing(Too expensive to do every time)

1. Insert nodes normally
2. Rebalance whole tree using size/2th node



Local Rebalancing

* Incremental rebalances to improve balance of overall tree
* May not end up perfectly balanced
* Approaches:
  + Randomisation (Random decision making)
  + Amortisation (Do more work now and less later) e.g. splay trees
  + Optimisation (Maintain structural info to provide performance guarantees)

*Normal leaf insertion + Random root insertion* **O(logn)**

* Recently inserted items are close to the root
* Randomised BST performance is the same as standard but no penalty if sequence is already ordered

*Randomised Tree Deletion*

* Can make a randomised decision to replace nodes as well

*Splay Trees* **Worst case could still be O(n)**

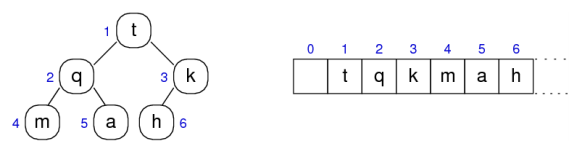
* Whenever you go down the tree, improve the balance of the tree by using root insertion with a twist: grandparent-parent-child
* Different implementation:
  + Node of most recently searched for item becomes new root
  + Makes search more expensive
* Helps with worst case degenerate trees

PRIORITY QUEUES

* Doesn’t sort by order of entry
* Leave: Removes item with highest priority key
* Uses a heap implementation so that both insert and delete is **O(logn)** rather than **O(n) insert** and **O(1)** **delete** when working with ordered array/list and vice versa with unordered

*Heap Ordered Trees* **O(logn)**

* They are ‘complete trees’: every level is filled in left to right before adding a node to the next level
* This ‘complete’ property makes it suitable for an array implementation rather than linked like BSTs
* The root/sub root contains largest value
* Index calculations allow navigation through the tree:
  + **Left child of node at i is located at 2i**
  + **Right child of node at index i is located at 2i + 1**
  + **Parent of node at i is located at i/2**

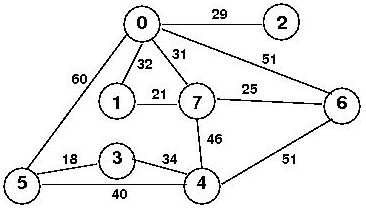


* Insert and move up if bigger than parent

*Deletion* – swaps with smallest item, then calls below



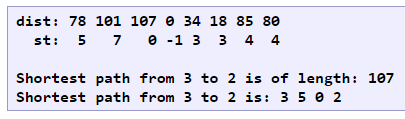
WEIGHTED GRAPHS



Dijkstra’s Shortest Path Algorithm

*Starting Node: 3*

* Use a priority queue table |node||weight| and visit each node, putting each adjacent node and added weight onto end of the queue



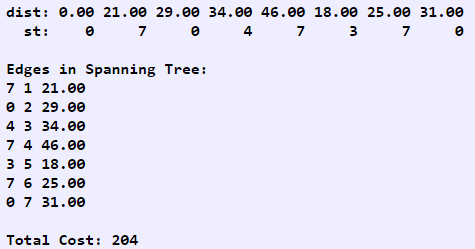
Minimal Spanning Trees

Kruskal’s Algorithm

* List out edge weights ascending order
* Go down list and draw graph as long as there are no cycles
* Cross out any that do have cycles

Prim’s Algorithm

* Visit nodes according to weight order (like cheapest least visited)
* Put in nodes accordingly into st array



*Priority Queues*

**