02105 Algorithms and Data Structures

Daniel Brasholt s214675

Saturday 14.05.22

Contents

Week 3: Analysis of Algorithms

Date: Thursday 17.02.22

Agenda

- Analysis of algorithms
  - Running time
  - Space
- Asymptotic notation
  - O, Θ, and Ω- notation
- Experimental analysis of algorithms

Notes

Analysis of algorithms

Goal: to determine and predict computational resources and correctness of algorithms.

- Does it work?
- How quickly?
- Can it scale?
- Will it run out of memory?
- Caching per query / performance?

Primarily:

- Correctness, running time, space usage
- Theoretical and experimental analysis

Running time Reading, writing, operations, comparisons are all steps

3 types:

- **Worst case**. Maximal running time
- **Best case**. Minimal running time
- **Average case**. Average running time

Time is almost always *worst case* unless otherwise specified.

Space Number of memory cells used by algorithm. Variables and pointers are 1 memory cell, array of length k = k memory cells.

Terminology: space, memory, storage, space complexity.

Asymptotic notation

Notation to bound the asymptotic growth of functions (larger input sizes).

O-notation E.g. $f(n) = O(n^2)$ if $f(n) \leq cn^2$ for large n. This could $5n^2$.

$5n^2 + 3$ is upper bounded by something like $6n^2$, so it is also $O(n^2)$. Basically only the most significant factor is considered.

Writing $f(n) = O(n)$ is more like $f(n) \in O(n)$, so $f(n) = O(n)$ is fine, $O(n) = f(n)$ is illegal.

Examples (true or false):

- $3n + 2n^3 - n^2 = O(n^2)$ - FALSE
- $3n^2 + \log n = O(n^3)$ - TRUE
- $5n^7 + 2^n = O(n^7)$ - FALSE
- $n \cdot \log^3 n = O(n^2 \cdot \log n)$ - TRUE
- $4n^2 + \log(n) = O(n^3)$ - TRUE
- $n(n+3)/1000 + 10000 \log^4 n = O(n^2)$ - TRUE

Ω-notation and Θ-notation Where O-notation is upper bounded, Ω-notation is lower bounded. Θ means that both Ω and O are true.

Examples (true or false):

- $n \log^3 n = O(n^2)$ - TRUE
- $2^n + 5n^7 = \Omega(n^3)$ - TRUE
- $n^2(n-5)/5 = \Theta(n^2)$ - FALSE (not upper bounded)
- $4n^{1/100} = \Omega(n)$ - FALSE
- $n^3/300 + 15 \log n = \Theta(n^3)$ - TRUE
- $2^{\log n} = O(n)$ - TRUE
- $\log^2 n + n + 7 = \Omega(\log n)$ - TRUE

Basic properties Exponentials grow faster than polynomials, polynomials faster than logarithms. Any and all logarithms are the same.

`for i=1 to n` is linear.

`for i=1 to n` and `for j=1 to n` is quadratic.

`for i=1 to n` and `for j=i to n` is quadratic as well.

Other typical running times are outlined in the notes - specifically recursive running times.

Experimental analysis of algorithms

The doubling technique is often used. This makes it easy to recognize every one of the common running times. E.g. when the input size doubles and the time is multiplied by 4, the algorithm probably runs in quadratic time.

Hand-in Exercise: Superheroes

- Not mandatory. Has nothing to do with grade; only for practice.
- Will be posted on Learn
- Handed in by preparing single PDF and uploading to Learn
- Algorithms must be written and described in natural language. Not needed to reinvent and re-explain algorithms used already in the course (e.g. binary search)
- Run time analysed in *Big-O*-notation
- Interested in simplicity (easy solutions for all problems)
- Clarity is important (must be concise)
- Pseudocode is not wanted / needed unless otherwise specified
- Not required to argue the choice of algorithm

Week 4: Introduction to Data Structures

Date: Thursday 24.02.22

Stacks

Maintains a dynamic sequence which support the operations:

- Push (push to top of stack)
- Pop (pop from top of stack)

Used for:

- Virtual Machines
- Parsing
- Function calls
- Backtracking

Is implemented by creating an array of capacity N. An index of the max element is then kept. `push()` and `pop()` must check for overflow. `isEmpty()` returns `true` if the top index is -1.

Analysis of stacks:

- Time
  - `push` in $\Theta(1)$ time
  - `pop` in $\Theta(1)$ time
  - `isEmpty` in $\Theta(1)$ time
- Space
  - $\Theta(N)$ space
- Limitations
  - Capacity must be known
  - Wasting space (space independent of actual stack size)

Queue

Dynamic queue supporting

- Enqueue (add x to Q)
- Dequeue (remove and return the ***first added*** element in Q
- isEmpty (returns true if empty)

Used for:

- Scheduling processes
- Buffering
- Breadth-first searching

Is implemented by an array with capacity N. Also indeces of head and tail counter. Enqueue and dequeue is done cyclically, thereby preventing overflow.

Analysis of queues:

- Time
  - `enqueue` in $\Theta(1)$ time
  - `dequeue` in $\Theta(1)$ time
  - `isEmpty` in $\Theta(1)$ time
- Space
  - $\Theta(N)$ space
- Limitations
  - Capacity must be known ahead of time
  - Wasting space

Linked Lists

Data structure to maintain a dynamic sequence of elements in linear space. The sequence order is determined by pointers / references called links. One element stores the index of the next element (and perhaps the previous element). The first and last elements contain a null byte instead of index. Index to next is called singly linked. Index to both is called doubly linked.

```
class Node {
    int key;
    Node next;
    Node prev;
}

Node head = new Node();
Node b = new Node();
Node c = new Node();
head.key = 7;
b.key = 42;
c.key = 18;

head.prev = null;
head.next = b;
b.prev = head;
b.next = c;
c.prev = b;
c.next = null;
```

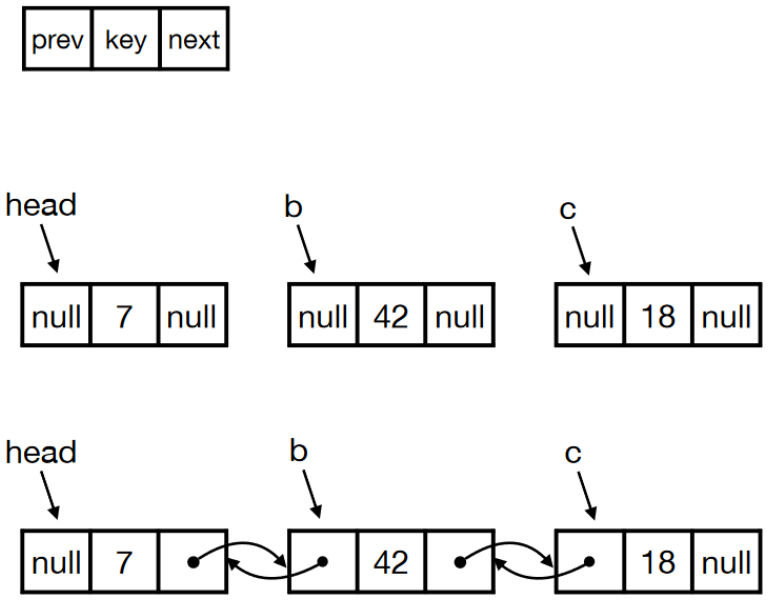


Figure 1: Linked List Java Implementation

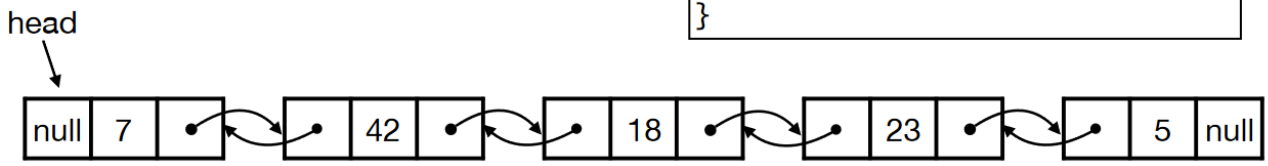
Simple operations:

- `Search(head, k)`: Returns node with key k. Return null if it does not exist.
- `Insert(head, x)`: Insert node x in front of list. Return new head.
- `Delete(head, x)`: Delete node x in list.

```
Node Search(Node head, int value) {
    Node x = head;
    while (x != null) {
        if (x.key == value) return x;
        x = x.next;
    }
    return null;
}
```

```
Node Insert(Node head, Node x) {
    x.prev = null;
    x.next = head;
    head.prev = x;
    return x;
}
```

```
Node Delete(Node head, Node x) {
    if (x.prev != null)
        x.prev.next = x.next;
    else head = x.next;
    if (x.next != null)
        x.next.prev = x.prev;
    return head;
}
```



- Ex. Let p be a new with key 10 and let q be node with key 23 in list. Trace execution of `Search(head,18)`, `Insert(head,p)` og `Delete(head,q)`.

Figure 2: Linked List Operations in Java

Analysis of linked lists:

- Time
  - Search in $\Theta(n)$ time
  - Insert and delete in $\Theta(1)$ time
- Space
  - $\Theta(n)$
- Unlike the other types, linked lists do not use more space than necessary

Implementing stacks and queues with linked lists

Instead of creating an array to store everything, use the `delete()` and `insert()` to add and remove from the stack/queue. The start of the queue would be the head of the linked list, and another index to the end of the queue can be kept to look up more efficiently. The top of the stack would be the last element of the linked list (also possible to reverse, since it would never be necessary to look up the start of a queue). By using linked lists, the space usage is more efficient and there is no longer a limit on the size of the stack/queue.

Dynamic Arrays

The goal is to implement a stack using arrays in $\Theta(n)$ space for n elements as fast as possible.

Solution 1

Start with array of size 1. To push, allocate a new array of size +1, move all elements, delete old array. This implementation is technically linear in insertion, but the time for n push-operations is $1 + 2 + 3 + ... + n = \Theta(n^2)$. The space is now $\Theta(n)$.

Solution 2

Idea: Only copy elements sometimes. Start with array of size 1. When the array is full, double the size of the array to only move sometimes. Then delete the old array. The time it takes for the 2^k'th `push` operation is $\Theta(2^k)$. All other push operations take $\Theta(1)$ time. The total time is therefore $1 + 2 + 4 + 8 + ... + 2^{\log n} + n = \Theta(n)$. The space is $\Theta(n)$.

Global rebuilding

Solution 2 is an example of ***global rebuilding***, a technique to make a static data structure dynamic.

Week 5: Introduction to Graphs

Date: Thursday 03.03.22

Undirected Graphs

Set of vertices pairwise joined by edges. Why?

- Models many natural problems from many different areas
- Hundreds of well-known algorithms

Graph	Vertices	Edges
communication	computers	cables
transport	intersections	roads
transport	airports	flight routes
games	position	valid move
neural network	neuron	synapses
financial network	stocks	transactions
circuit	logical gates	connections
food chain	species	predator-prey
molecule	atom	bindings

Figure 3: Applications of Graphs

Terminology

- **Undirected Graph**
  - V = set of vertices
  - E = set of edges
  - n = |V|, m = |E|
- **Path**: sequence of vertices connected by edges

- **Cycle**: path starting and ending at the same vertex
- **Degree**: $deg(v)$ = the number of neighbours of v (edges incident to v)
- **Connectivity**: pair of vertices are connected if there is a path between them

Lemma: $\Sigma_{v \in V} \deg(v) = 2m$

Representation

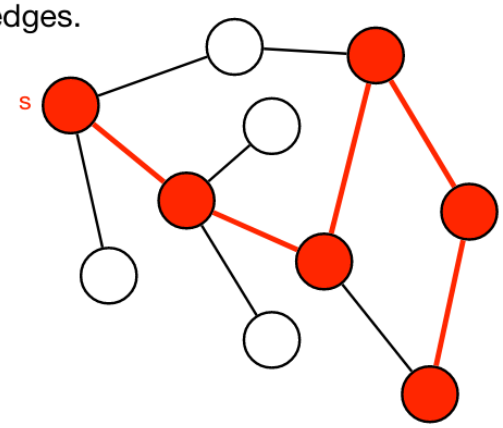
Need methods `adjacent`, `neighbours`, and `insert`.

Adjacency Matrix 2D array. A[i,j] is 1 if i and j are neighbours. $O(n^2)$ space complexity. `Adjacent` + `insert` is $O(1)$. `Neighbours` is $O(n)$.

Adjacency List. Array A[0..n-1]. Linked list of all neighbours of i. Space complexity $n + \Sigma_{v \in V} \deg(V) = O(n + m)$. Time of `adjacent`, `insert`, `neighbours` is $O(\deg(V))$.

Depth First Search

- Algorithm for systematically visiting all vertices and edges.
- Depth first search from vertex s.
  - Unmark all vertices and visit s.
  - Visit vertex v:
    - Mark v.
    - Visit all unmarked neighbours of v recursively.



- Intuition.
  - Explore from s in some direction, until we read dead end.
  - Backtrack to the last position with unexplored edges.
  - Repeat.

- Discovery time. First time a vertex is visited.
- Finish time. Last time a vertex is visited.

Figure 4: Depth-First search

Recursive algorithm on lecture slides.

Time:

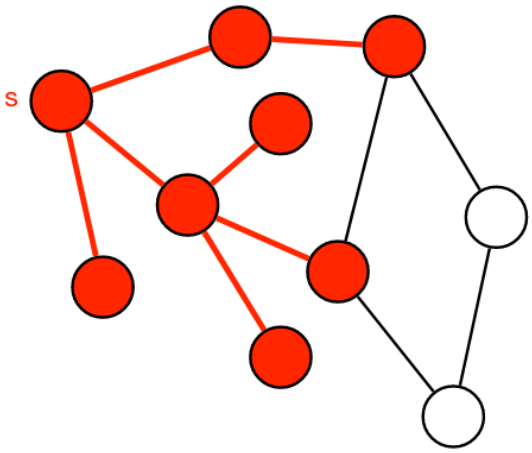
- One recursion pr. vertex
- $O(\deg(V))$ time spent on vertex v
- $\Rightarrow O(n + \Sigma_{v \in V} \deg(v)) = O(n + m)$ time
- Only visits vertices connected to s

Used for e.g. Flood Fill (painting program) & finding connected components.

Breadth First Search

Use queues to keep track of marked vertices.

- Breadth first search from s.
  - Unmark all vertices and initialize queue Q.
  - Mark s and Q.ENQUEUE(s).
  - While Q is not empty:
    - v = Q.DEQUEUE().
    - For each unmarked neighbor u of v
      - Mark u.
      - Q.ENQUEUE(u).



- Intuition.
  - Explore, starting from s, in all directions - in increasing distance from s.

- Shortest paths from s.
  - Distance to s in BFS tree = shortest distance to s in the original graph.

Figure 5: Breadth First Search

Substituting a queue for a stack makes the algorithm recursive = depth first

Bipartite Graphs

Every edge has 1 red and 1 blue endpoint. Can be partitioned into V_1 and V_2 such that all edges go between V_1 and V_2.

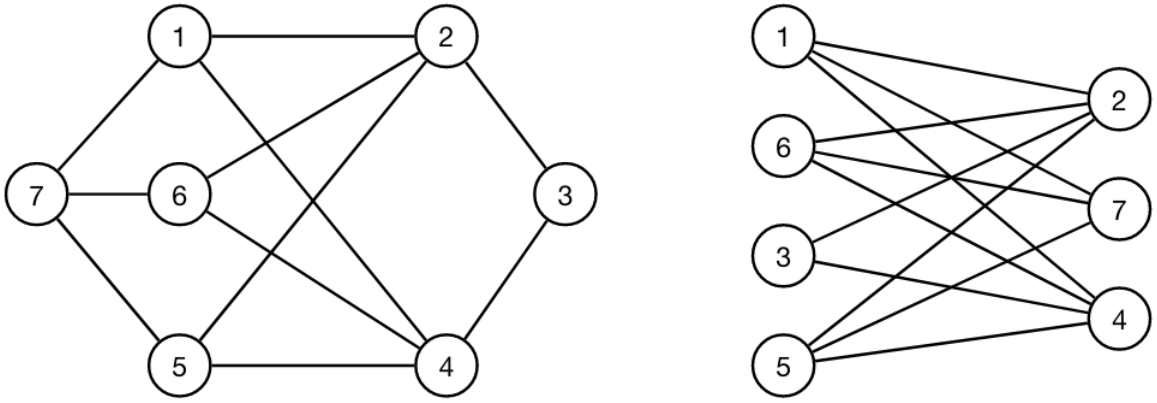


Figure 6: Bipartite Graph

Graph is bipartite if and only if all cycles in G have even length. Also means there is no edge between vertices of same BFS layer. Layers can be coloured in alternating colours (red and blue).

Run BFS on G. For each edge, check if its endpoints are in the same layer. Complexity $O(n+m)$.

Algorithm	Time	Space
Depth first search	O(n + m)	O(n + m)
Breadth first search	O(n + m)	O(n + m)
Connected components	O(n + m)	O(n + m)
Bipartite	O(n + m)	O(n + m)

Figure 7: Graph Algorithms

Week 6: Directed Graphs

Date: Thursday 10.03.22

Notes

Directed Graphs

Graphs where an edge is directional. The degree of a node is split in two - input and output degrees. E.g. a road network, garbage collection in memory, web crawling or dependencies.

$$\sum_{v \in V} = \deg^{-}(v) = \sum_{v \in V} \deg^{+}(v) = m$$

Algorithmic problems

- Path
- Shortest Path
- Directed acyclic graph (cycle in graph)
- Topological sorting (order vertices such that all edges are in same direction)
- Strongly connected components (path between all pairs of vertices)
- Transitive closure (for which vertices is there a path from v to w?)

Representation

- Following operations:
  - PointsTo(v,u): determine if v points to u
  - Neighbours(v): return all vertices that v points to
  - Insert(v,u): add edge unless there

An adjacency matrix, like with the undirected graph, can be made. No longer symmetrical. Adjacency lists can also be made.

Searching

The exact same as in undirected graphs. Only change is that we only look at vertices that the current one points to. Can implement DFS and BFS.

Topological Sorting

Sorting such that all edges go in the same direction. Find v with in-degree 0. Then recurse on graph with that vertex removed. If `G` is empty, it must be possible to sort topologically.

Solution 1 Construct reverse graph G^R.

- Search in adjacency list representation of G^R to find vertex v with
- in-degree 0. Remove v and edges out of v.
- Put v leftmost
- Repeat

Total time $O(n^2)$ (see lecture slides).

Solution 2 Maintain in-degree of every vertex + linked list of all vertices with in-degree 0.

- Find vertex v with in-degree 0
- Remove v and edges out of v
- Put v leftmost
- Repeat

This way, you do not have to search for in-degree 0 each time the algorithm has run; it is being tracked in the linked list.

Total time $O(n + m)$.

Directed Acyclic Graphs

G is a DAG if it contains no (directed) cycles. DAG and topological sorting is equivalent, so finding out is merely running the algorithm.

Strongly Connected Components

v and u are strongly connected if there is a path from v to u and u to v.

A **Strongly Connected Component** is a maximal subset of strongly connected vertices.

This can be computed in $O(n + m)$ time.

Implicit Graphs

Undirected/directed graph with implicit representation. This means:

Start vertex s + algorithm to generate neighbours of a vertex. This implicitly generates a graph.

Week 7: Priority Queues and Heaps

Date: Thursday 17.03.22

Priority Queues

Maintain dynamic set S (no duplicates) supporting operations:

- `Max` - Return element with largest key
- `ExtractMax` - Return and remove element with largest key
- `IncreaseKey` - Set key to higher value
- `Insert` - Insert element into set

Can be used for scheduling, shortest path, minimum spanning trees, compression (Huffman's Algorithm).

Solution 1: Linked Lists

S is maintained as a doubly linked list. Use linear search for `max` and `extractmax`.

Time: $n = |S|$, max in $O(n)$ time, `IncreaseKey` in $O(1)$.

Solution 2: Sorted Linked List

Every time an element is added, it needs to be added in the right spot. Further complicates some elements; however `max` and `ExtractMax` become faster.

Data structure	MAX	EXTRACTMAX	INCREASEKEY	INSERT	Space
linked list	O(n)	O(n)	O(1)	O(1)	O(n)
sorted linked list	O(1)	O(1)	O(n)	O(n)	O(n)
heap	O(1)	O(log n)	O(log n)	O(log n)	O(n)

Figure 8: Priority Queue Implementations

Trees

Rooted trees

- Nodes connected with edges
- Connected and acyclic
- Designated root node
- Special type of graph

Terminology: children, parent, descendant, ancestor, leaves, internal nodes, path

Descendant = recursive children

Leaves = nodes with no children

Depth and height:

- Depth of v = length of path from v to root
- Height of v = length of path from v to descendant leaf
- Depth of T = height of T = length of longest path from root to a leaf

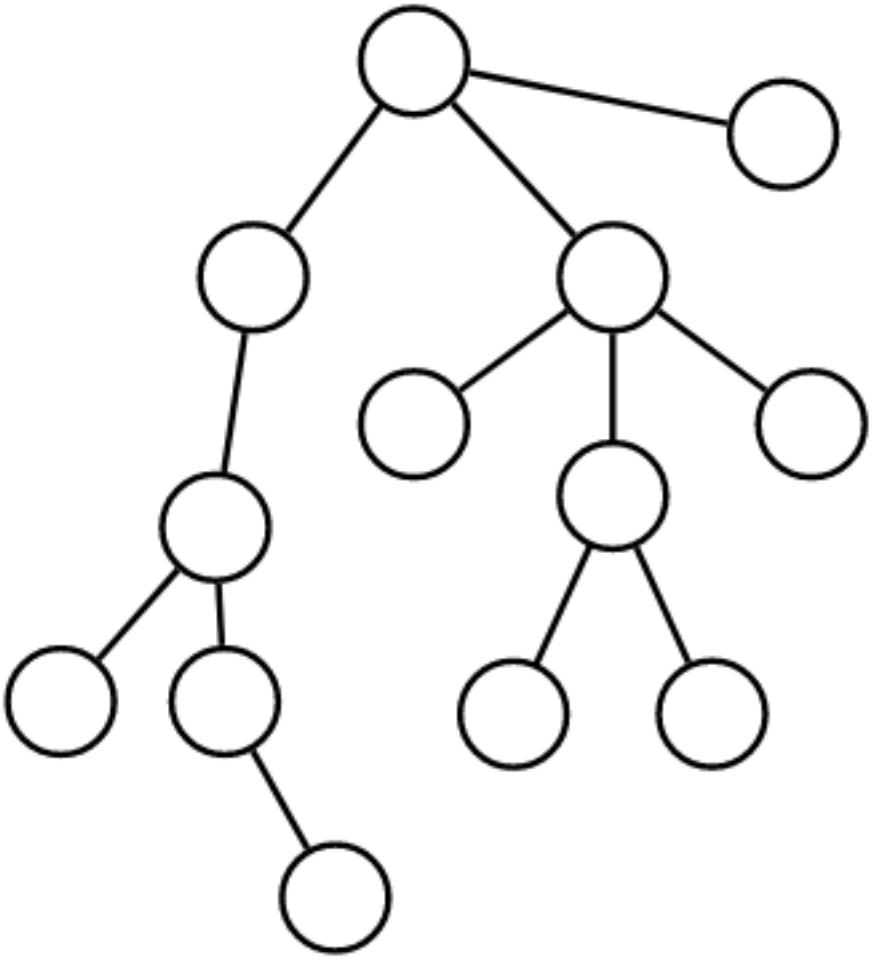


Figure 9: Example of Tree

Binary trees

- Rooted tree
- Each node has at most two children called the *left child* and *right child*

Complete binary tree: Binary tree where all levels of tree are full.

Almost complete binary tree: Complete binary tree with 0 or mor rightmost leaves deleted.

The height of an (almost) complete binary tree with n nodes is $\Theta(\log n)$.

Heaps

Almost complete binary tree. All nodes store one element and the tree satisfies *heap order*.

Heap order:

- For all nodes v:
  - All keys in left subtree and right subtree are $\leq v.key$

Max heap is where this is true. Min heap is where all elements are $\geq v.key$ instead.

Representation

Solution 1: Linked representation

- Time: $O(1)$
- Space: $O(n)$

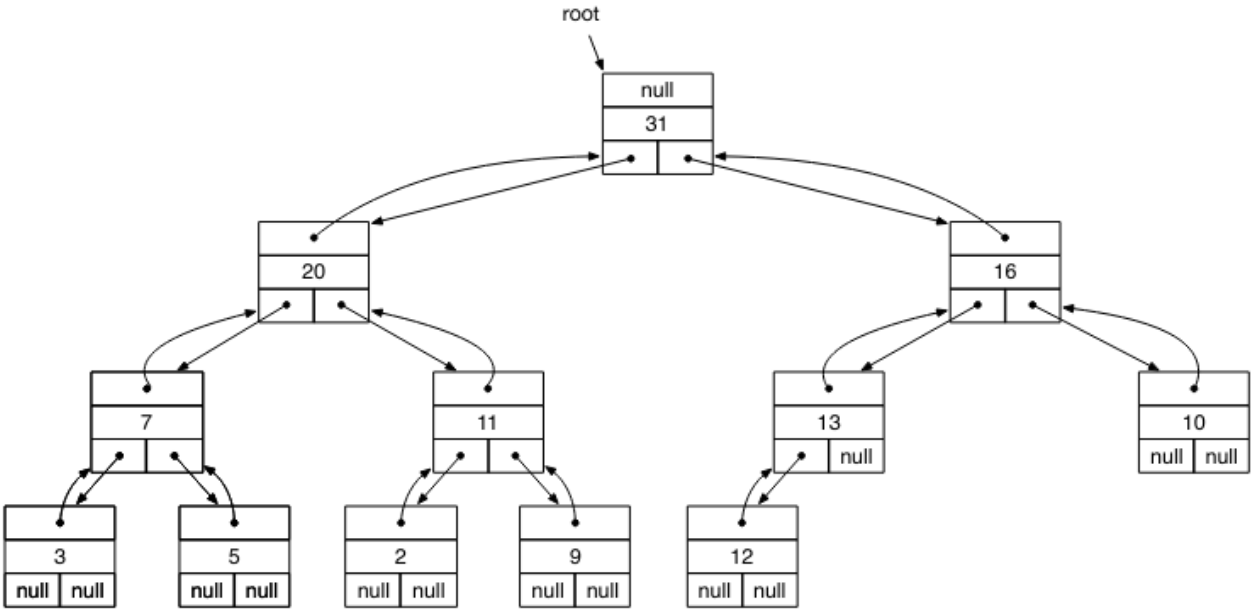


Figure 10: Heap in a linked representation

Solution 2: Array representation

Store in array of size $n + 1$. `H[0]` is unused, `H[1]` is root.

- `Parent(x)`: Return $x/2$
- `Left(x)`: Return $2x$
- `Right(x)`: Return $2x + 1$
- Time: $O(1)$
- Space: $O(n)$

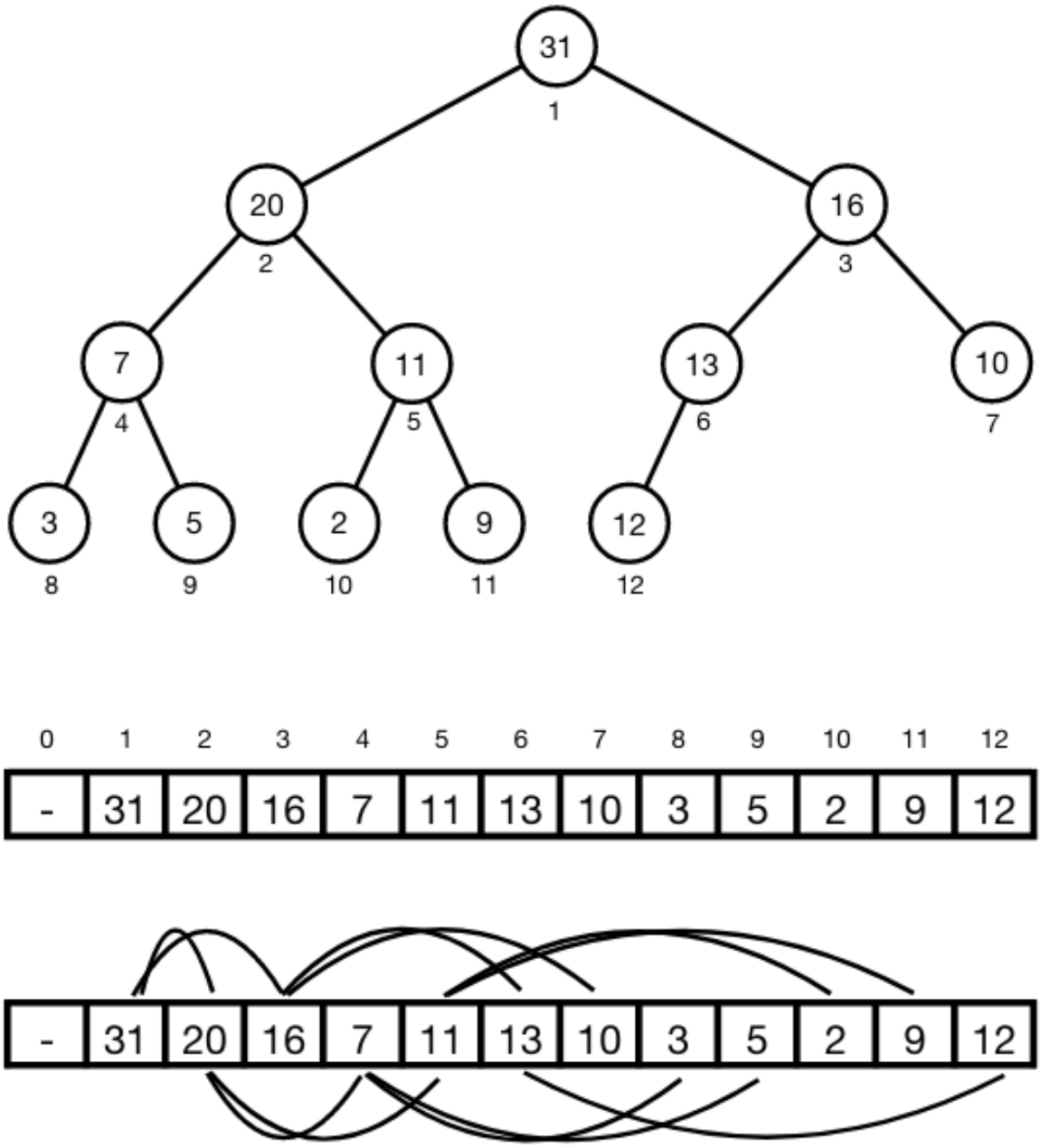


Figure 11: Heap with array representation

Algorithms

- `BubbleUp()`:
  - If heap order is violated at node x because key is larger than key at parent:
  - Swap x and parent
  - Repeat with parent until heap order is satisfied
- `BubbleDown()`:
  - If heap order is violated at node x because key is smaller than key at left or right child:
  - Swap x and child c with largest key
  - Repeat with child until heap order is satisfied

Both of these are done in $O(\log n)$ time.

Implementation of Priority Queues

```
MAX()
    return H[1]
```

```
INSERT(x)
    n = n + 1
    H[n] = x
    BUBBLEUP(n)
```

```
EXTRACTMAX()
    r = H[1]
    H[1] = H[n]
    n = n - 1
    BUBBLEDOWN(1)
    return r
```

```
INCREASEKEY(x,k)
    H[x] = k
    BUBBLEUP(x)
```

Figure 12: Algorithms on Priority Queues

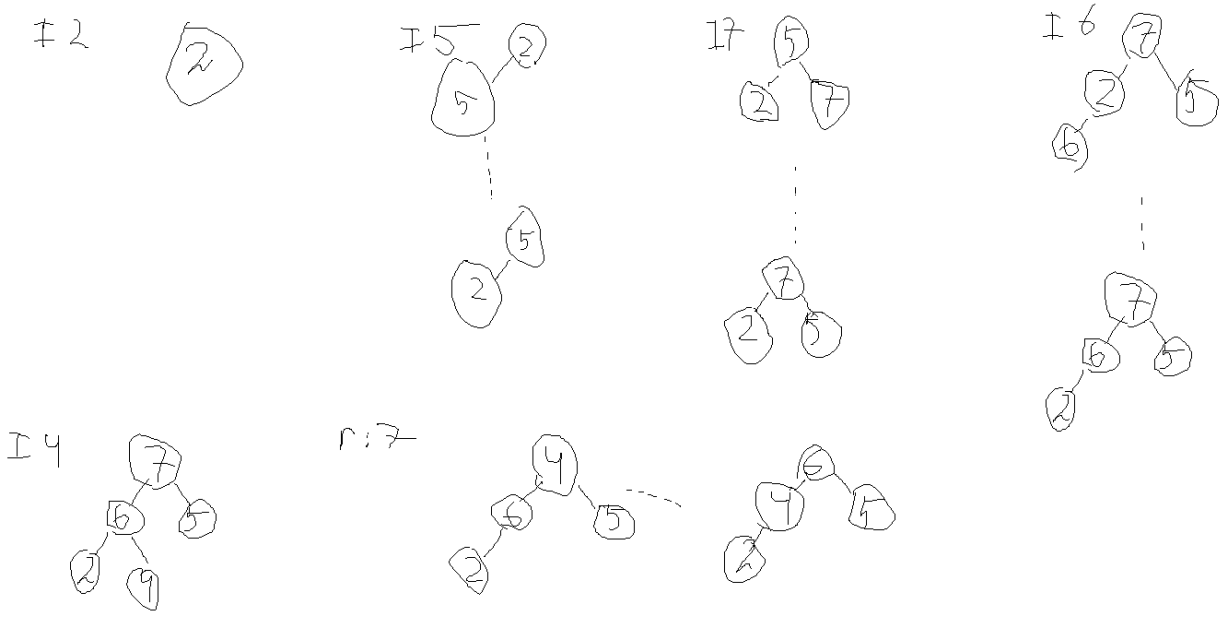


Figure 13: Example run of algorithms

Building a Heap

Problem: Given integers in array, convert to heap that satisfies ordering.

Solution 1: Top-down construction

For all nodes in increasing level order, apply `BubbleUp()`. Total time is $O(n \log n)$. Explanation on lecture slides.

Solution 2: Bottom-down construction

For all nodes in decreasing level order, apply `BubbleDown()`. Total time is $O(n)$ since all leaves are fast and root is slow; with top-down, the root is fast and all leaves are slow.

Heapsort

- Build a heap for H
- Apply n `ExtractMax()`
  - Insert results in the end of array
- Return H
- Heap construction in $O(n)$ time
- n `ExtractMax()` in $O(n \log n)$ time
  - $\Rightarrow O(n \log n)$ time

Uses only $O(1)$ time $\Rightarrow$ in-place sorting algorithm. Can be used to sort an array in $O(n \log n)$ time with no extra space complexity (unlike merge-sort).

Week 8: Union Find, Dynamic Connected Components

Date: Thursday 24.03.22

Union Find

Maintain a dynamic family of sets supporting the following operations:

- Init(n): constructs sets {0}, {1},...,{n-1}
- Union(i,j): forms the union of the two sets that contain i and j. If i and j are in the same set, nothing happens
- Find(i): return a representative for the set that contains i

Applications

- Dynamic connectivity
- Minimum spanning tree
- Unification in logic and compilers
- Nearest common ancestors in trees
- Illustrations of techniques in data structure design

Quick Find

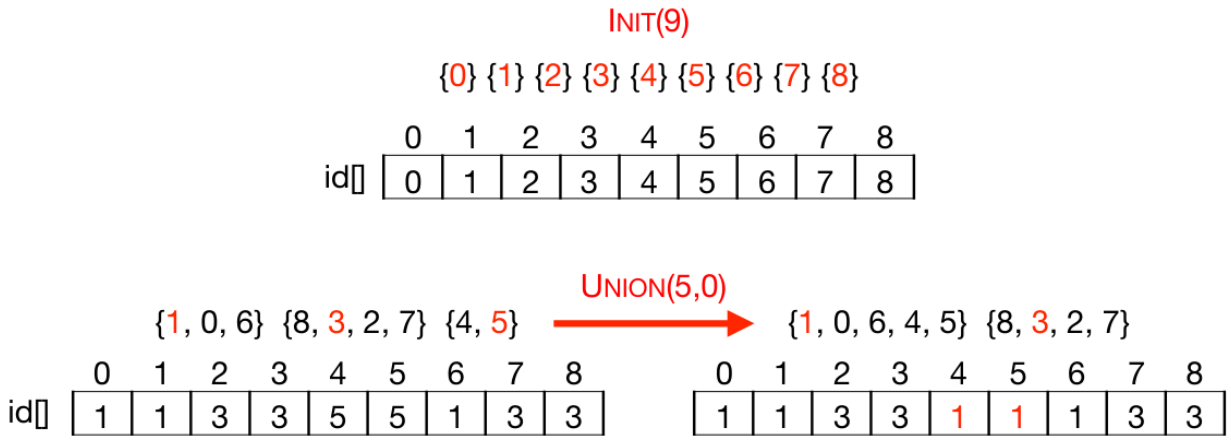


Figure 14: Quick Find Demonstration

Pseudocode for the operations can be found on the lecture slides.

Quick Union

Maintain each sets as a rooted tree. Store trees as arrays such that p[i] is the parent of i and p[root] = root. Representative is the root of the tree.

To find: follow path to root and return root.

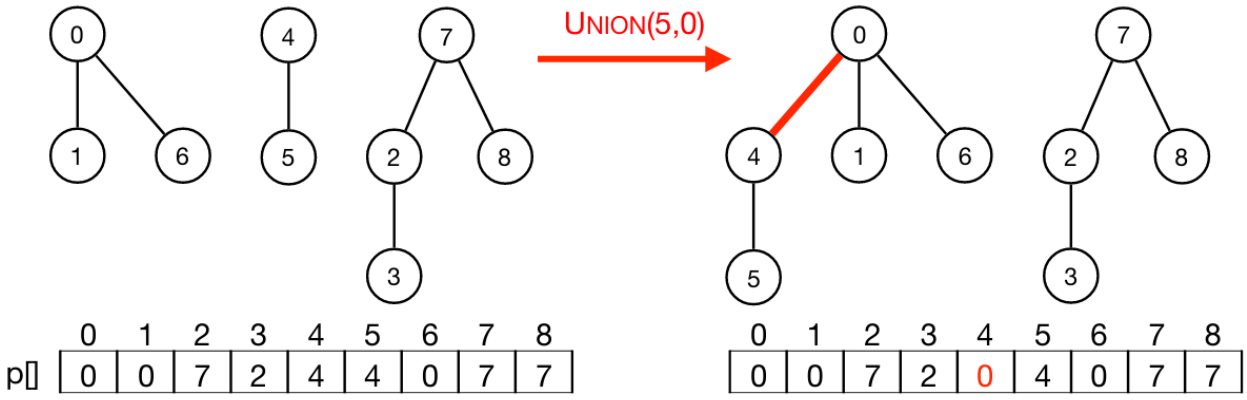


Figure 15: Quick Union Demonstration

Again, code snippets available on lecture slides.

Find and union are $O(d)$ where d is the depth of the tree. This, in the worst case, is $O(n)$.

Weighted Quick Union

- Extension of quick union. Difference: we choose which way the union is carried out.
- In union: make the root of the *smaller* tree the child of the root of the *larger* tree.
- This also balances the trees.
- Maintain extra array sz[o..n-1] such sz[i] = the size of the subtree rooted at i.

Path Compression

Compress path on `find`. Make all nodes on the path children of the root. Does not change running time of single `find`, but makes subsequent operations faster. Time complexity defined on slides as theorems.

Dynamic Connectivity

Maintain a dynamic graph supporting the following operations:

- `init(n)`: create graph G with n vertices and no edges
- `connected(u,v)`: determine if u and v are connected
- `insert(u,v)` add edge (u,v). We assume (u,v) does not already exist.

With union find:

- Init: initialise a union find data structure with n elements.
- Connected: find(u) == find(v)
- Insert: Union of the inputs.

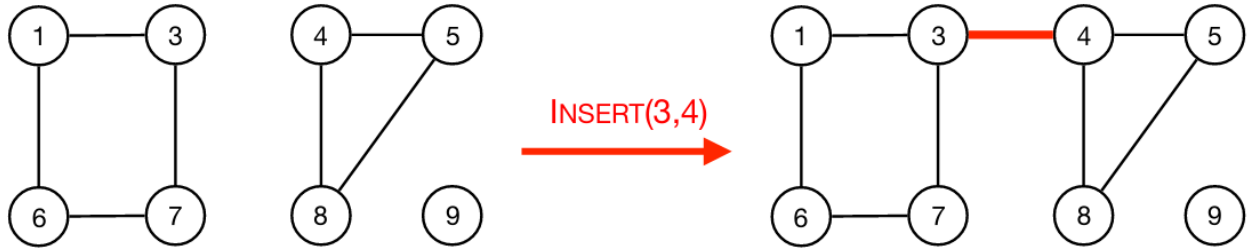


Figure 16: Dynamic Connectivity Example

Week 9: Minimum Spanning Trees

Date: Thursday 31.03.22

Minimum Spanning Trees

Weight on each edge in graph G. The spanning tree is a subgraph T of G over all vertices that is connected and acyclic - all nodes must be accessible with no cycle. The minimum spanning tree is then a spanning tree of minimum total weight.

Applications:

- Network design
  - Computer, road, telephone, electrical circuit
- Approximation algorithms
  - Travelling salesperson problem, Steiner trees
- Meteorology, cosmology, biomedical analysis and so forth

Representation of Weighted Graphs

With an adjacency matrix, instead of "1", write the weight between connections. Similar with adjacency list, where we store 2 values for each entry instead of 1.

Properties of Minimum Spanning Trees

For simplicity: all edge weights are distinct, G is connected $\Rightarrow$ MST exists and is unique.

Cut Property

- Definition:
  - A cut is a partition of the vertices into two non-empty sets
  - A cut edge is an edge crossing the cut
- Cut property:
  - For any cut, the lightest cut edge is in the MST

Cycle Property

For any cycle, the heaviest edge is not in the MST.

Prim's Algorithm

- Grow a tree T from some vertex s
- In each step, add lightest edge with one endpoint in T
- Stop when T has n-1 edges

Execution example

Starting from 0:

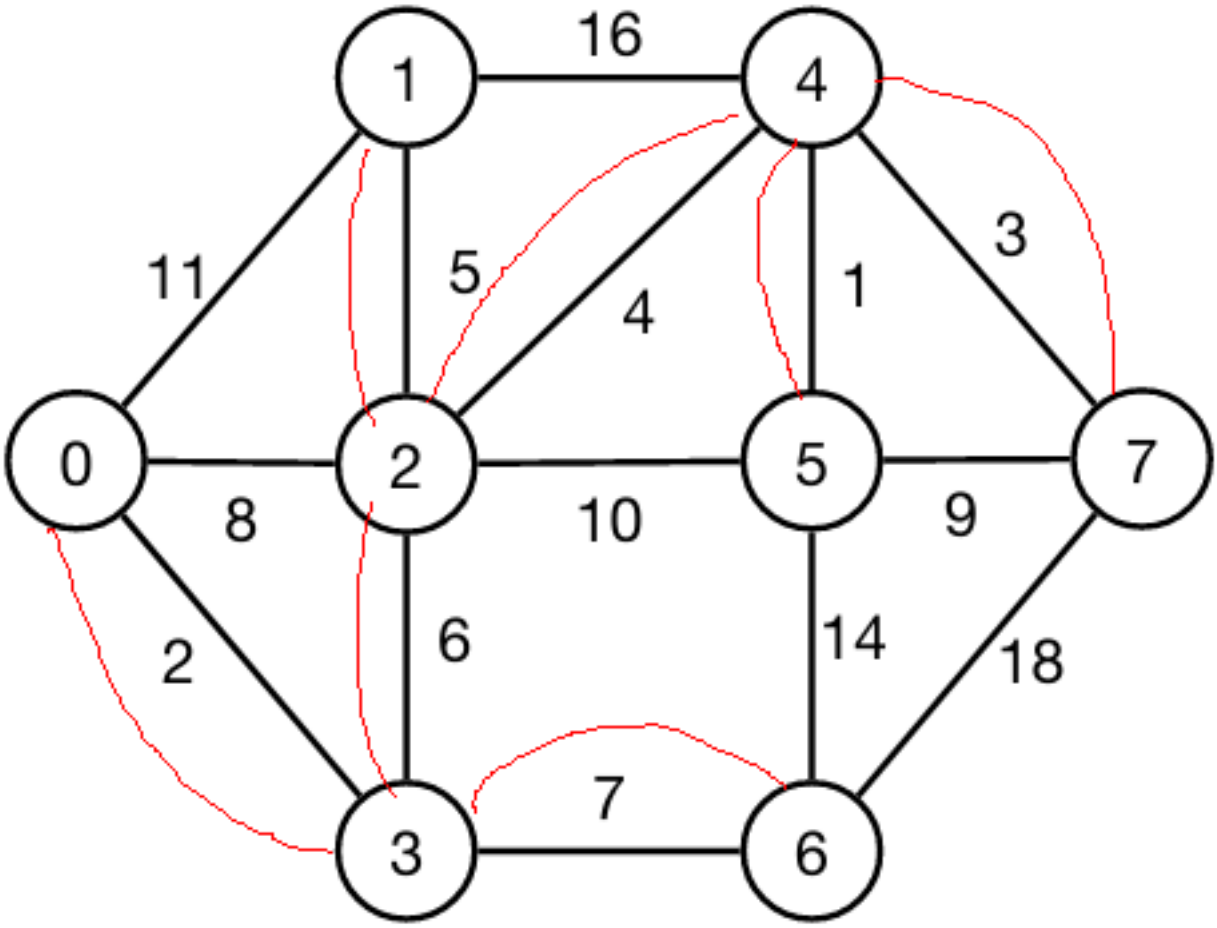


Figure 17: Prim's Algorithm Execution Example

Challenge: how do we find the lightest cut edge?

By using priority queue: find lightest edge, EXTRACT-MIN, then update weight of neighbours of new vertex with DECREASE-KEY.

```
PRIM(G, s)
    for all vertices v∈V
        v.key = ∞
        v.π = null
        INSERT(P,v)
    DECREASE-KEY(P,s,0)
    while (P ≠ ∅)
        u = EXTRACT-MIN(P)
        for all neighbors v of u
            if (v ∈ P and w(u,v)<key[v])
                DECREASE-KEY(P,v,w(u,v))
                v.π = u
```

Figure 18: Prim's Algorithm

The algorithm is ***greedy***: the best local option leads to the best global solution.

Kruskal's Algorithm

- Consider edges from lightest to heaviest
- In each step, add edge to T if it does not create a cycle
- Stop when T has $n-1$ edges

Challenge is to find cycles. This can be done with ***union find***.

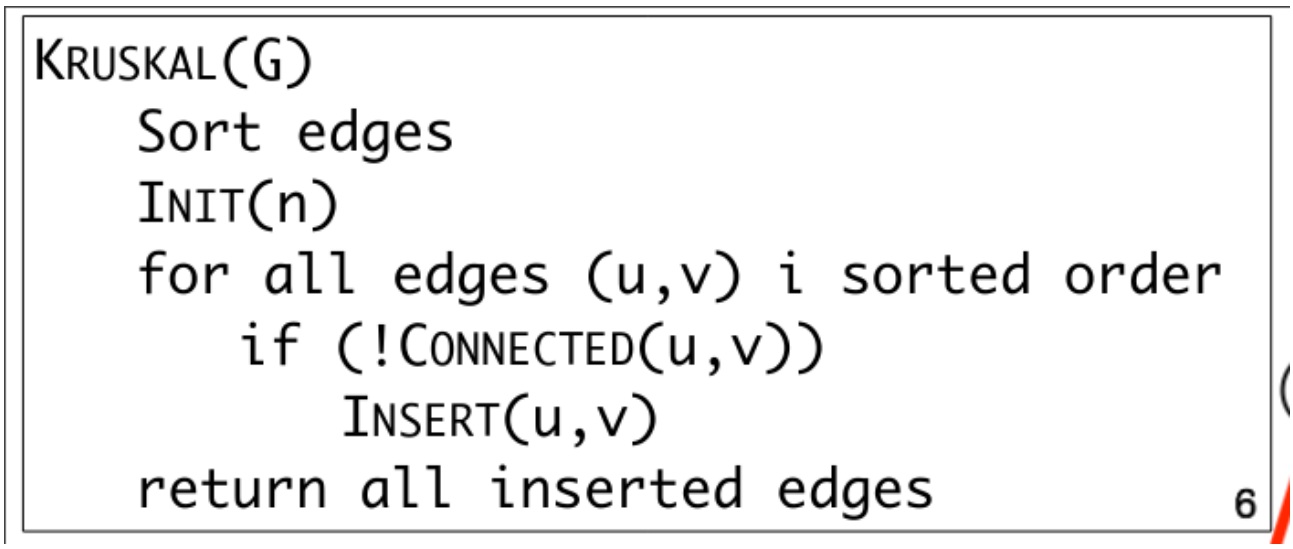


Figure 19: Kruskal's Algorithm Pseudocode

Kruskal's is also a ***greedy*** algorithm.

Week 10: Shortest Paths

Date: Thursday 07.04.22

Shortest Path

Properties of Shortest Path

For simplicity: all vertices are reachable from s $\Rightarrow$ shortest path to each vertex always exists.

Subpath Property: any subpath of a shortest path is a shortest path

Dijkstra's Algorithm

Goal

Given a directed, weighted graph with non-negative weights, and a vertex s, compute shortest paths from s to all vertices.

Algorithm

- Maintains distance estimate v.d for each vertex v = length of shortest *known* path from s to v
- Updates distance estimates by *relaxing* edges
  - If distance from u to v is less than the current estimate, relax the distance
- Initialise s.d=0 and v.d = ∞ for all vertices v$\in V \backslash \{s\}$
- Grow tree T from s
- In each step, add vertex with smallest distance estimate to T
- Relax all outgoing edges of v

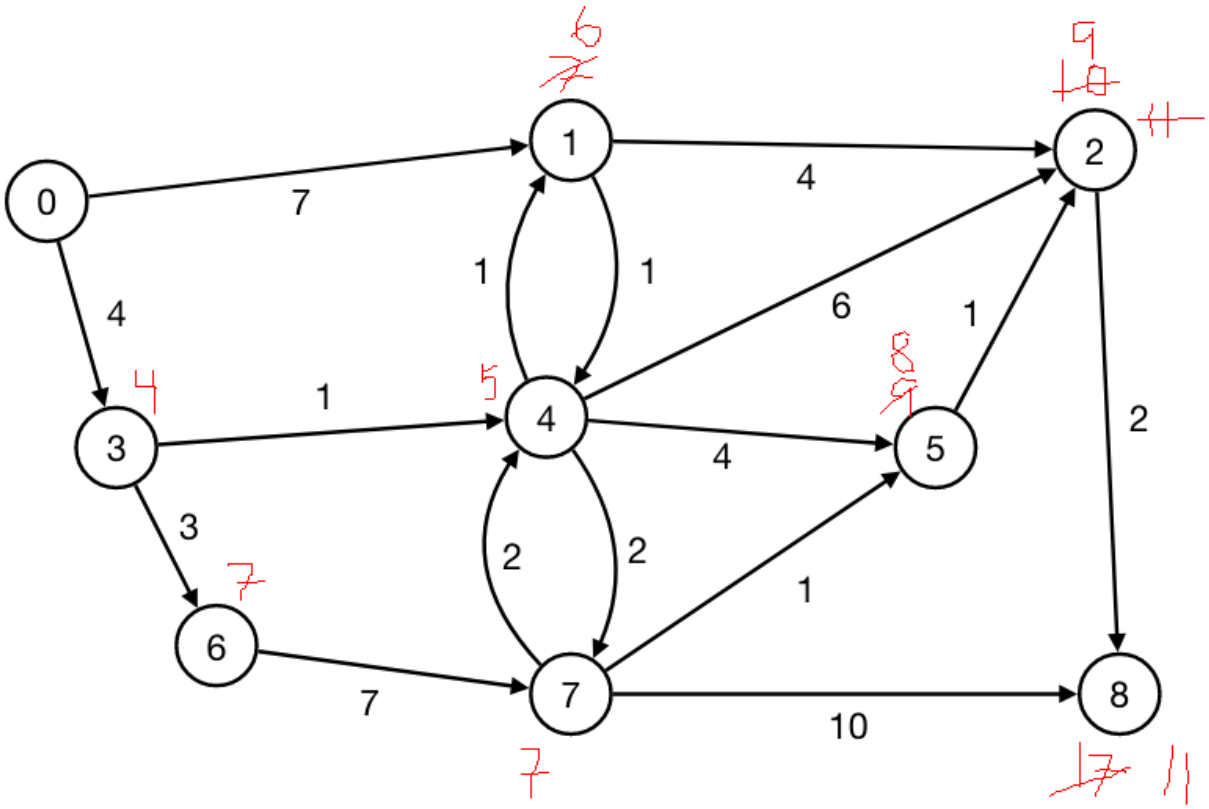


Figure 20: Example of Dijkstra's Algorithm

Implementation

Maintain vertices outside T in priority queue.

```
DIJKSTRA(G, s)
    for all vertices v∈V
        v.d = ∞
        v.π = null
        INSERT(P,v)
    DECREASE-KEY(P,s,0)
    while (P ≠ ∅)
        u = EXTRACT-MIN(P)
        for all v that u point to
            RELAX(u,v)
```

```
RELAX(u,v)
    if (v.d > u.d + w(u,v))
        v.d = u.d + w(u,v)
        DECREASE-KEY(P,v,v.d)
        v.π = u
```

Figure 21: Pseudocode for Dijkstra's Implementation

Shortest Path on DAG

For each vertex, relax all outgoing vertices. Also works with negative weights.

Implementation: Sort all nodes in topological order. Relax all outgoing vertices from each vertex. Running time $O(m+n)$.

Week 11: Hashing

Date: Thursday 21.04.22

Dictionaries

Maintain dynamic set S of elements spporting `search(k)`, `insert(x)`, and `delete(x)`. Each element x has a key `x.key` from a universe U and satellite data `x.data`.

Used as a basic data structure for representing a set & in numerous algorithms.

Solution 1 - Linked Lists

`Search` in $O(n)$ time, `insert` and `delete` in $O(1)$. Space in $O(n)$.

Solution 2 - Direct Addressing

The set of keys is just a subset of an array from 0 to the biggest key.

Universe will be much bigger than actual set of keys. `Search`, `insert`, and `delete` in $O(1)$. Space in $O(|U|)$.

Chained Hashing

Define a hash function that spreads keys from S approximately evenly over {0,...,m-1}. Maintain array A[0..m-1] of linked lists. Store element x in linked list at A[h(x.key)]. A collision is if h(x.key) = h(y.key).

- `search(k)`: linear search in A[h(k)] for key k
- `insert(x)`: insert x in front of list A[h(x.key)]
- `delete(x)`: remove x form list A[h(x.key)]

Basically a method of compressing solution 2 to a realistic amount of space.

- Time:
  - `search` in $O(lengthoflist)$
  - `insert` and delete in $O(1)$
  - `length` of list depends on hash function
- Space:
  - $O(m+n) = O(n)$

Simple Uniform Hashing

See slides, expected length of searching $O(1)$. This makes all operations $O(1)$.

Linear Probing

Maintain S in array A of size m. Element x stored in A[h(x.key)] or in cluster to the right of A[h(x.key)]. A cluster is a consecutive sequence of non-empty entries.

- `search(k)`: linear search from A[h(k)] in cluster to the right of A[h(k)]
- `insert(x)`: insert x on A[h(x.key)]. If non-empty, insert on next empty entry to the right of x (cyclically)
- `delete(x)`: remove x form A[h(x.key)]. Re-insert all elements to the right of x in the cluster.

Very cache-efficient - practical in computer science.

Assuming simple uniform hashing, constant time for all operations and $O(n)$ space.

Hash Functions

- Simple hash:
  - h(k) = k mod m - typically with prime m
- Others:
  - Tabulation hashing, MurmurHash, SHA-xxx, FNV

Week 12: Binary Search Trees

Date: Thursday 28.04.22

Nearest Neighbour

Maintain dynamic set S supporting:

- `Predecessor`: return element with largest key $\leq$ k
- `Successor`: return element with smallest key geq k
- `Insert`: add x to S
- `Delete`: remove x from S

Can be used for searching for similar data & routing in the internet.

Solution 1

Doubly linked list which contains S. Finding predecessor and successor takes $O(n)$ time where $n = |S|$. Insert and delete in $O(1)$.

Solution 2

Maintain S in a sorted array. Predecessor, successor take $O(logn)$ time. Insert, delete take $O(n)$ time, since new arrays have to be built.

Data structure	PREDECESSOR	SUCCESSOR	INSERT	DELETE	Space
linked list	O(n)	O(n)	O(1)	O(1)	O(n)
sorted array	O(log n)	O(log n)	O(n)	O(n)	O(n)
binary search tree	O(h)	O(h)	O(h)	O(h)	O(n)
balanced binary search tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)

Figure 22: Nearest Neighbour Methods

Binary Search Trees

Rooted tree where each internal vertex has a left child and/or a right child. Binary search trees satisfy the search tree property:

- Each vertex stores an element
- For each vertex v:
  - all vertices in left subtree are $\leq$ v.key
  - all vertices in right subtree are $\geq$ v.key

Representation requires each vertex to store: x.key, x.left, x.right, x.parent, (x.data). Space is $O(n)$.

Insertion

- `insert(x)`: start in root. At vertex v:
  - if x.key $\leq$ v.key go left
  - if x.key $>$ v.key go right
  - if null, insert x

```
insert(x,v):
    if v is None: return x
    if x.key <= v.key:
        v.left = insert(x, v.left)
    if x.key > v.key:
        v.right = insert(x, v.right)
```

Predecessor and Successor

```
predecessor(v, k):
    if v == null: return null
    if v.key == k: return v
    if k < v.key:
        return predecessor(v.left, k)
    t = predecessor(v.right, k)
    if t is not None: return t
    return v
```

Time in $O(h)$ where h is the height of the tree.

Deletion

- `delete(x)`:
  - 0 children: remove x
  - 1 child: splice x
  - 2 children: find y = vertex with smallest key > x.key. Splice y and replace x by y

Algorithms on Trees

Size

Compute size of tree rooted at v recursively:

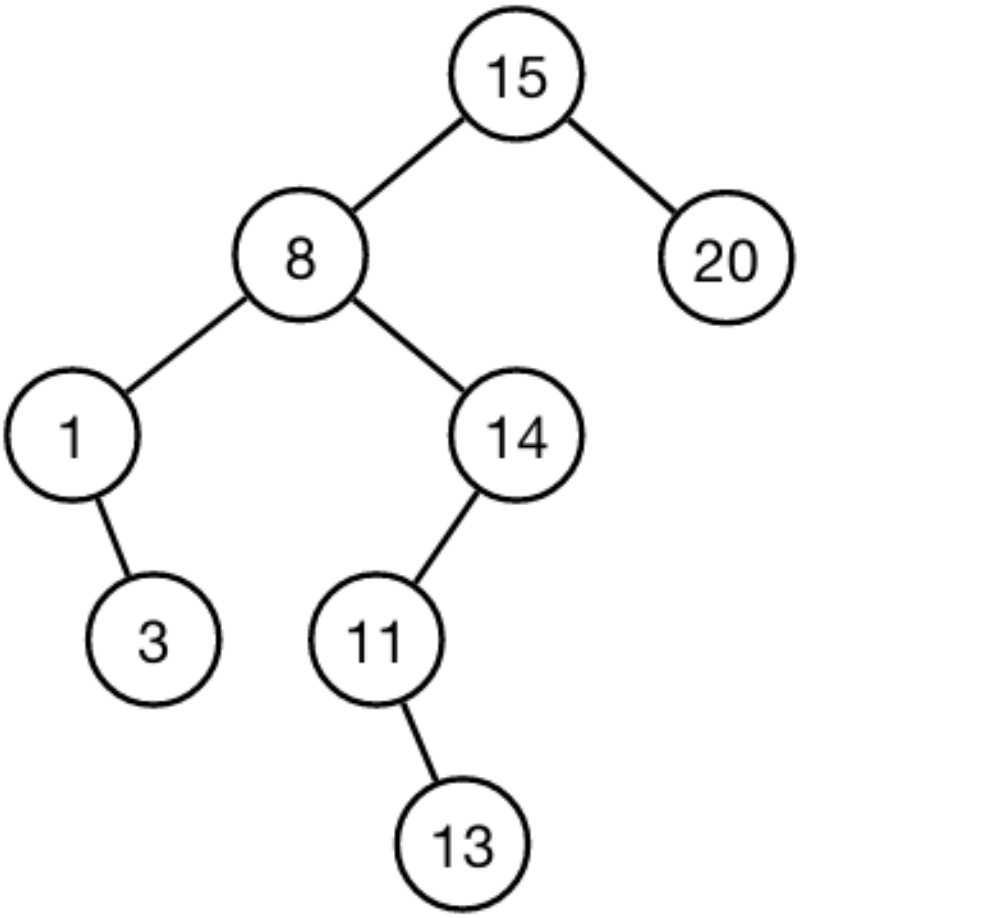
```
def size(v):
    if v is None: return 0
    return size(v.left) + size(v.right) + 1
```

Time $O(size(v))$

Traversal

- Inorder traversal:
  - Visit left subtree recursively
  - Visit vertex
  - Visit right subtree recursively
- Prints out the vertices in a binary search tree in sorted order
- Preorder traversal
  - Visit vertex
  - Visit left subtree resursively
  - Visit right subtree recursively
- Postorder Traversal
  - Visit left subtree recursively

- Visit right subtree recursively
- Visit vertex



Inorder: 1, 3, 8, 11, 13, 14, 15, 20

Preorder: 15, 8, 1, 3, 14, 11, 13, 20

Postorder: 3, 1, 13, 11, 14, 8, 20, 15

Figure 23: Tree Traversal Types

```
def inorder(v):
    if v is None: return
    inorder(v.left)
    print(v.key)
    inorder(v.right)
def preorder(v):
```

```
    if v is None: return
    print(v.key)
    inorder(v.left)
    inorder(v.right)
def postorder(v):
    if v is None: return
    inorder(v.left)
    inorder(v.right)
    print(v.key)
```