Data Structures & Algorithms[*]

1 Generics

- Generics add stability by making more bugs more detectable during compile-time.
- Eliminates casting. This reduces code redundancy by eliminating unnecessary method overloading.
- In generic code, the wildcard, a question mark (`?`), represents an unknown type and helps control type safety. See documentation for complete implementation details.

```java
// Example method utilizing generics.
public static <T> void print_array(T [] list1){
  for(T element: list1)
    System.out.print(element + " ");
}
```

2 Mathematical Preliminaries

2.1 Induction

Algorithm 1 Proof by induction

```
1: procedure INDUCTION($P(x), c$)        ▷ Prove $P(x)$ for all integers $x \geq c$.
2:     if $P(c)$ then
3:         $k \geq c$
4:         assume $P(k)$
5:         show $P(k) \rightarrow P(k+1)$        ▷ Q.E.D.
```

Also see: Strong induction, structural induction.

3 Asymptotic Analysis

Algorithms can be analyzed by describing two key properties: run-time and space complexity. Run-time complexity describes the rate at which run-time of an algorithm grows with varying input size. Space complexity describes the memory requirements of an algorithm with variation in input size. We can characterize these with the following three functions:

- Big $\mathcal{O}$: Bounds from above, used for worst case. We say a function g is big $\mathcal{O}$ of a function f if there are positive constants c and n such that $|g(x)| \leq c|f(x)|$ for all $x \geq n$. Denoted $g = \mathcal{O}(f)$.
- Big Ω: Bounds from below, used for best case. We say a function g is big Ω of a function f if there are positive constants c and n such that $|g(x)| \geq c|f(x)|$ for all $x \geq n$. Denoted $g = \Omega(f)$.
- Big Θ: Bounds from above and below, describes average case. A function g is big Θ of f if and only if g is big $\mathcal{O}$ of f and big Θ of f. Denoted $g = \Theta(f)$, note f is also big Θ of g.

General Hierarchy: $C \leq \log N \leq \log^2 N \leq N \leq N \log N \leq N^2 \leq N^3 \leq 2^N \leq N! \leq N^N$. Each function is big $\mathcal{O}$ of any function to its right.

[*]Summarized from Shaffer's Data Structures & Algorithm Analysis in Java, 3rd ed.

3.1 Measuring Run-Time

First, you need some measure of the size of the input, such as the size of a number or an array, or the length of an array, etc. Then, programs can be analyzed using two categories: basis cases which are generally constant time such as: assignment, break, return, printing; and induction which involves loops, branching (if-else statements), and functions. For loops, multiply the big-O of the body by the number of times the loop runs. For branching, use the maximum iteration of all the branches. For functions, calculate it as if it were a separate program.

Example. Consider the following code:

```
int sum = 0;
for(int i = 0; i < n; i++){
   for(int j = 0; j < n; j++)
      sum++;
}
```

The declaration and assignment of `sum` is $\mathcal{O}(1)$. The inner for loop executes the increment n times, thus it is $\mathcal{O}(n)$. Likewise, the outer for loop executes the inner for loop n times and is $\mathcal{O}(n)$. Therefore, the program is $\mathcal{O}(1) + \mathcal{O}(n)\mathcal{O}(n)$ which is equivalent to $\mathcal{O}(n^2)$.

4 Linked Lists

It's easier to insert and delete data from a linked list since it doesn't require data to be moved around like in an array.

```
public class LinkedList{
   private Node head;
    //Other methods...
}

class Node{
   private Object data; //the data
   private Node next; //the link
}
```

Observe that the node type is self-referential. Infinite recursion doesn't occur because `next` is default `null`. A linked list simply consists of a single instance variable `head` which points to the first `Node` in the list. Thus the linked list has-a reference to `Node`.

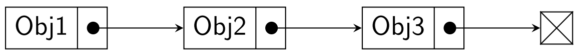


Figure 1: Example of a linked list.

Documentation.

4.1 Doubly Linked Lists

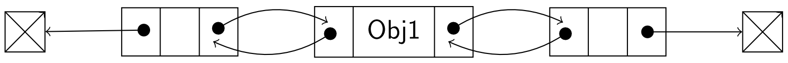


Figure 2: Example of a doubly linked list.

If given a suitable pointer, insertion and deletion at an arbitrary point in the list is $\Theta(n)$ and operations to access data within the list will run in $\Theta(n)$, since they require a traversal of the elements. But, insertion/deletion to the front or end are $\Theta(1)$. Using an array-based implementation insertion and deletion are $\Theta(n)$, and accessing via indices is $\Theta(1)$.

5 Stacks

A stack is a list-like structure in which elements can only be added or removed from one end. The accessible element is called the top. Inserted elements are said to be pushed onto the stack, and those to be removed are popped off. Stacks can be more efficient when applications only require a limited form of insertion and deletion, relative to a generic list. It's a LIFO list, or "last-in, first-out".

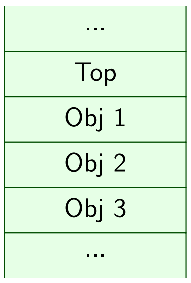


Figure 3: Example of a stack.

Insertion and deletion run in $\Theta(1)$.

```
// Stack ADT
public interface Stack<E>{

   public void clear();

    //Add element to top of stack.
    public void push(E it);

    //Remove and return element on top of stack.
    public E pop();

    //Return front element.
    public E peek();

    public int length();
}
```

6 Queues

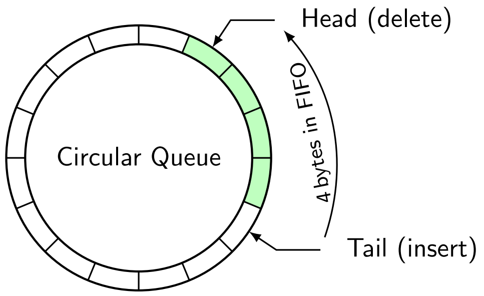
A queue is similar to a stack, however, element may only be inserted at the back (called an enqueue operation) and removed from the front (called a dequeue operation). It's a FIFO list, or "first-in, first-out".
Insertion and deletion run in $\Theta(1)$.

```
// Queue ADT
public interface Queue<E>{

   public void clear();

    //Add element to rear of queue.
    public void enqueue(E it);

    //Remove and return element at front of queue.
```



```
    public E dequeue();

    //Return front element.
    public E frontValue();

    public int length();
}
```

7 Binary Trees

A **binary tree** is made up of a finite set of elements called **nodes**. This set is either empty or has a **root** node, from which stem two other binary trees, called **subtrees**. The roots of each subtree are **children** of the root. An **edge** connects a node to its children, and the node is said to be the **parent** of its children.

If $n_1, n_2, \ldots, n_k$ is a sequence of nodes in a tree such that n_i is the parent of n_{i+1} for all $1 \leqslant i < k$, then this sequence is a **path** from n_1 to n_k. The path has length $k - 1$. If the path leads from node R to M then R is an **ancestor** of its **descendant** M. The **depth** of a node is the length of a path from the root to that node. The **height** of a tree is 1 more than its deepest node.

All nodes a depth of d are at **level** d in the tree. A **leaf** node is any node with two empty children. An **internal** node has at least one non-empty child.

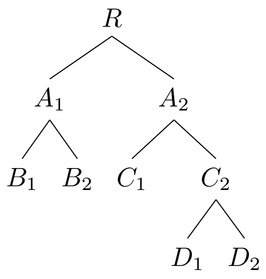


Figure 4: R is the root. A_1, A_2 are R's children, and R is their parent. A_2 and R are C_1's ancestors. B_1, B_2 are leaf nodes. Level 2 is composed of nodes B_1, B_2, C_1, C_2. The node D_1 has depth 3. The height of the tree is 4.

A binary tree is **full** if each node is either a leaf node, or an internal node with two non-empty children. In a **complete** binary tree of height d, all levels except possible level $d - 1$ are full. The bottom level fills left to right.

Full Binary Tree Theorem: The number of leaves in a non-empty full binary tree is one more than the number on internal nodes.

Corollary. The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.

A process that visits all the nodes in some order is a **traversal**. There are three primary traversal methods:

- Preorder: Visit any given node, before visiting its children.
- Postorder: Visit each node only after we visit its children, and their subtrees.
- Inorder: Visits left child, and its subtree, before visiting the node, and finally visiting the right child and its subtree.

```
// Binary tree node ADT
public interface BinNode<E>{

    //Get and set element value.
    public E element();
    public void setElement(E v);

    //Return left/right child.
    public BinNode<E> left();
    public BinNode<E> right();

    //Return true if node is leaf.
    public boolean isLeaf();
}
```

7.1 Binary Search Tree

BST Property: All nodes stored in the left subtree of a node whose key value is K have key values less than K. All nodes in the right subtree have key values greater than or equal to K.

Insertion and locating functions have fairly straightforward recursive implementations. Deleting from a BST can be complicated. The cases where the node has zero or one child are easy. But when the nodes to be removed has two children, typically one finds the smallest key value in the right subtree, say in node R, that is greater than the key value to be removed, in node N, and removes node R and stores the value in N.

Printed in inorder traversal and costs $\Theta(n)$. The cost of insertion and finding methods is the depth of the node to be found or inserted. Where as deletion's cost is the depth of the node to be removed. Thus, there costs depend on how well balanced the BST is. Optimally, these operations will cost $\Theta(\log n)$.

7.2 Heaps and Priority Queues

When a collection of object is organized by importance or priority we call this a **priority queue**. The **heap** data structure is used to efficiently implement the priority queue.

The heap is a complete binary tree, generally implemented using an array based representation. Additionally, the values in the heap are partially ordered meaning there is a relationship between the value stored at a node and the values of its children.

A **max-heap** has the property that every node stores a value greater than or equal to the value of either of its children. A **min-heap** is simply the opposite. Insertion and deletion $\Theta(\log n)$. Heapify (turning an arbitrary array into a heap) is $\Theta(n)$.

Heap sort: Repeatedly delete the maximum element from a heap, for $\Theta(n \log n)$ runtime.

7.3 Huffman Coding Trees

Huffman coding assigns variable length codes to characters dependent on their relative frequency of use. They're can be used for basic compression. Each leaf of a Huffman tree corresponds to a letter, and the weight of the leaf is the weight of its associated letter. The goal is to build a tree with minimum external path weight. The **weighted path length** of a leaf is the weight of that leaf times its depth in the tree.

8 General Trees

A node's **out degree** is the number of children it has. A **forest** is a collection of trees. A K-**ary tree** is a tree whose internal nodes all have exactly K children.

Note: Only pre- and post-order traversals work on general trees.

```
//General Tree ADT
interface GTNode<E>{
   public GTNode<E> parent();
   public GTNode<E> leftmostChild();
   public GTNode<E> rightSibling();
}

interface genTree<E>{
   public GTNode<E> root();
}
```

8.0.1 Parent Pointer

In a parent pointer implementation, each node only stores a pointer to its parent. In general, this isn't general purpose, since its inadequate for finding the leftmost child or the right sibling. Yet it's useful in determining if two nodes are in the same tree, since they'll have the same root.

The parent pointer implementation is often used to maintain a collection of disjoint sets, and supports two operations:

- Union: Merge two sets together.
- Find: Determine if two objects are in the same set.

Path compression can be used when finding the root, R of a given node, X, by resetting the parent of every node on the path from X to R, to point to R.

8.0.2 List of Children

Stores with each internal node, a linked list of its children. While determining leftmost child is efficient, finding the right sibling to a given node is not.

8.0.3 Left-Child, Right Sibling

Each node stores the values of its parent, leftmost child, and right sibling. Makes tradeoff of additional space, for better runtime.

8.0.4 Sequential Tree

This implementation saves space, as no pointers are stored, accessing any node requires that all nodes before it in the list be processed first (tradeoff greater runtime for less space). Nodes are typically stored as they'd be enumerated by a pre-order traversal, and additional information is provided to describe the shape of the tree.

Serialization is the process of storing an object as a series of bytes, so it can be transmitted.

8.1 AVL Trees

AVL Tree Property: For every node, the height of the left and right children can differ by at most ± 1. To insert into an AVL tree, we can insert like in a BST, then perform 'rotations' on the lowest node in the tree violating AVL, as in Figures 5 or 6. Proceed until the root is processed.

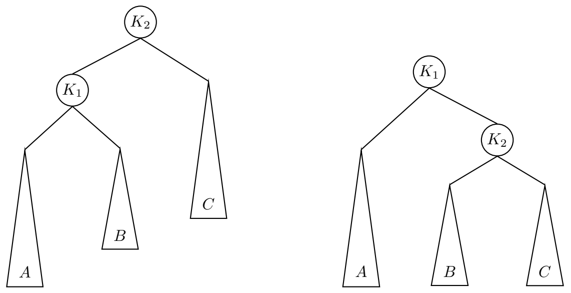


Figure 5: Single Rotation.

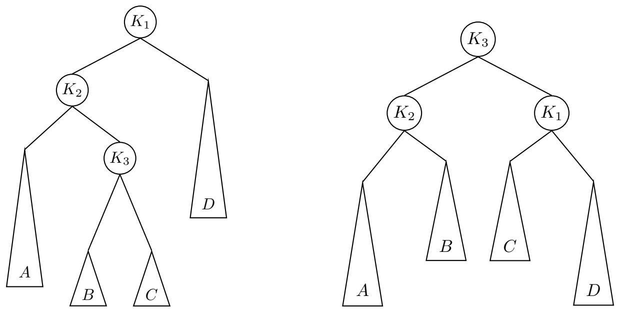


Figure 6: Double Rotation. Apply a single rotation about K_2, then another about K_1.

9 Sorting

9.1 Insertion Sort

Iterate through a list of records, and insert each record in turn at the correct position within a sorted list composed of the records already processed.

The best case, where the records are already sorted is $\Theta(n)$, whereas the worse case, where the records are in reverse order is $\Theta(n^2)$, the average case performing no better.

9.2 Bubble Sort

Works by comparing adjacent pairs, swapping if necessary. Smallest values bubble up to the correct position. After first pass the smallest value is at the front of the array.

$\Theta(n^2)$ in the best, worst, and average cases.

9.3 Selection Sort

Scan array to find the minimum element, swap with first and increment. Then repeat until completely sorted. Requires fewer swaps that bubble sort. Costs $\Theta(n^2)$ in the best, worst, and average cases.

These so called exchange sorts, which only compare adjacent elements all cost at least $n(n-1)/4 = \Omega(n^2)$ in the average case because for any list and its inverse there will be a total of $n(n-1)/2$ inversions between the two.

9.4 Shellsort

Makes comparisons/swap between non-adjacent elements. Attempts to make the list mostly sorted before utilizing insertion sort's best-case performance.

First, it breaks the array into "virtual" sublists, whose elements are a fixed distance apart, which each get sorted by insertion sort. So, if n elements are to be sorted we may use our initial gap to be $n/2$, thus we form $n/2$ sublists of 2 elements which will individually be sorted. A second pass will use a smaller gap to make larger lists, perhaps, $n/4$ lists of size 4 with distance $n/4$. Eventually, this culminates with a normal pass of insertion sort on the entire array. Thus, as along as the final increment is 1, the array will end up sorted.

Shellsort's runtime is dependent on the gap increment. An increment of 2 is rather inefficient, however, one of 3 typically yield a runtime of $\mathcal{O}(n^{1.5})$.

9.5 Merge Sort

Mergesort works by recursively halving up the array, sorting the small individual pieces, then merging them back together. Merge works by examining the front elements of each sublist, removing the smallest to append to the output list and continuing until all elements have been added. Can efficiently merge linked lists, but requires special alternating division of the list elements in order to break up the initial input list.

Splitting the array in half requires $\log n$ recursion depth and at each step $\Theta(n)$ steps are required to merge the array, requiring $\Theta(n \log n)$ runtime overall for the best, worst, and average cases.

9.6 Quicksort

Quicksort first selects a pivot, which acts like a root node's value in a BST. Then, the k values less than the pivot are placed in the leftmost k positions in the array, the values $\geq$ pivot are in the rightmost $n-k$ positions, with the pivot itself in position k. Note, the values in the "partitioned" array aren't necessarily sorted. Then, quicksort is performed again one the subarrays to the left and right of the pivot.

The pivot can be selected randomly from anywhere in the array, but is typically the middle index. The partition function works by swapping the pivot to the end of the array. Then, using two incrementing variables, one at the beginning of the array, one at the index one from the end, it compares values to the pivot, swapping as necessary, until the variables meet in the middle. Then places the pivot back into position right before the larger half of the array.

Partitioning takes $\Theta(n)$ for an array of size n, so in the worst case quicksort is $\Theta(n^2)$, but in the best and average cases, takes $\Theta(n \log n)$. Further improvement can be made by stopping quicksort when the partitions get smaller than 9 and just using the best case for insertion sort.

9.7 Heapsort

Heapsort converts an array into a max-heap structure, then incrementally removes the root (largest) elements and stores in the final array before restoring the heap property. Since heap building takes $\Theta(n)$ time and each deletion takes $\Theta(\log n)$ time, the entire operation is $\Theta(n \log n)$.

9.8 Binsort & Radix Sort

Binsort efficiently sorts integers. The basic idea is that it creates an array of "bins", whose position in the array corresponds to the values it'll hold. Duplicate entries can be handled using an array of linked lists. If we know the value of our maximum key, say K, then to sort an array of n integers, this merely requires $\Theta(n)$ work. However, this assumes K is small compared to n. If however, $K \approx n^2$ then the cost becomes $\Theta(n^2)$. Furthermore, a large key range may often require an unacceptably large array.

An alternative approach is bucket sort, which first sorts the the records into bins which may contain a range of key values. Radix sort is an extension of this principle (the term radix means the base of the key values). As an illustration, imagine we wish to sort two digit integers. First, we can perform a bucket sort to organize the numbers according to their rightmost digit, then copy them back to the original array. Then, in that order we sort them according to their second digit, before recopying once again. Since the numbers are base 10, we only require 10 bins (one for each digit $0-9$) to sort or in general, r bins to sort integers in base r.

A typical implementation uses an array to keep track of which indices in the bucket array contain which sets of keys to avoid the overhead of linked lists. So suppose we wish to sort n numbers in base r with at most k digits. If we assume all records are distinct and r is a constant, then $k \geq \log_r n$. Therefore, radix sort is at best $\Omega(nk) = \Omega(n \log n)$. Note however that increasing r can greatly reduce runtime at the cost of additional space.

9.9 Lower Bounds on Sorting

Sorting algorithms determine which unique permutation of the input corresponds to a sorted list. Since the decision tree must contain one leaf for each possible permutation, it has $n!$ leaves. Thus, the depth of the tree is $\Omega(\log n!) = \Omega(n \log n)$. So in the worst case any comparison based sorting algorithm requires $\Omega(n \log n)$ comparisons. Since we know sorting algorithms with $\mathcal{O}(n \log n)$ runtime, the problem of sorting is $\Theta(n \log n)$.

10 File Processing & External Sorting

Primary or **main** memory refers to **Random Access Memory** (RAM), while **secondary** refers to alternative storages devices such as hard drives, SSDs, removable USBs, CDs, etc. The advantages of these secondary media are cheapness, persistency (as opposed to the volatility of RAM), and transportability; however, they sacrifice access time ($\sim 10^6$ times greater than than of RAM).

When designing efficient disk-based applications the general rule is to *minimize* the number of disk accesses. This can be achieved with a well-organized file structure or by selectively saving previously accessed information, i.e. **caching**.

A logical file is a contiguous series of bytes which may possibly combine to form a data record. However, the physical file actually stored on the record is usually not contiguous, but spread out in pieces over the disk. It's a **file manager**'s job to take requests for a logical file and map them to the physical location on a disk. Modern-day disk drives are direct access meaning it takes approximately the same time to access any record in the file.

10.1 Buffers

Nearly all disk drives read/write an entire sector of information when the disk is accessed. The information read is then stored in main memory in case information from the same sector also needs to be read. This is known as **caching** or **buffering** the information. **Double buffering** occurs when multiple sectors of a file must be read. Once the first sector is read, the CPU, acting independently can begin processing the information while the disk driver (in parallel) reads a new sector.

11 Searching

- Sequential Search: Iterate through the list in order, comparing each element to the desired key ($\mathcal{O}(n)$).
- Jump Search: Using a sorted array, jump using increments of size j until a range of indices bounds the desired key. Follow with a sequential search ($\mathcal{O}(\sqrt{n})$).

- Dictionary Search: A modified form of binary search, used when key values are in an expected uniform distribution ($\mathcal{O}(\log n)$). Search position p for a value K is given by

$$p = n\frac{K - L[1]}{L[n] - L[1]}.$$

- Quadratic Binary Search: Initially examines p as in dictionary search. Then, sequentially probes the correct side of the array using steps of size $\sqrt{n}$, until the sublist bounding the key is found. This process is then repeated recursively on the found sublist. Cost of $\mathcal{O}(\log\log n)$, albeit a larger constant term than simple binary search.

11.1 Self-Organizing Lists

Self-organizing lists, wherein the key k_i with the greatest probability p_i of being accessed is placed first in the list, can yield vastly improved runtime, depending on the distribution of the p_i. For instance, a geometric distribution ($p_i = 1/a^i$) has a constant expected cost. The are a few heuristics for organizing the list:

- Use a counter. Keys with the highest count are stored closer to the front. Doesn't account for changes in frequency.
- Move most frequently accessed record to the front. Only efficient for linked lists.
- Swap record accessed with the one immediately preceding it.

11.2 Hashing

Hashing is the process of finding a record using a computation to map its key value to a position in an array. Finding a record with key value K in a hashing based database requires:

1. Computing the table location via the **hashing function**, $h(K)$.
2. Starting at slot $h(K)$ in the **hash table**, locate the record containing key K using, if necessary, a collision resolution policy.

12 Indexing

2-3 Trees. Its shape obeys the following rules:

- A node contains one or two keys.
- Every internal node has either two children (if it contains 1 key) or three (if it contains 2 keys).
- All leaves are at the same level in the tree (height balance).

All keys are organized in a similar fashion to a binary search tree.

B-Trees. A B-tree (generalized 2-3 tree) of order m is said to have the following properties:

- Root is either a leaf or has 2 children.
- Each internal node, except for the root, has between $\lceil m/2 \rceil$ and m children.
- All leaves are at the same level in the tree, so its always height balanced.

Searching is done via a binary search on the keys within each node, and then checking the proper child node if it isn't found.

13 Graphs

A graph $G = (V, E)$ is a set of vertices V are edges E. A graph with few edges is sparse, while one with many is dense. A graph containing all possible edges is said to be complete. Associated with an edge (U, V) between vertices U and V may be a cost or weight, such graphs are called weighted.

A sequence of vertices $v_1, \ldots, v_n$ form a path of length $n-1$ if there exist edges from v_i to v_{i+1} for all $1 \leqslant i < n$. A path is simple if it only visits distinct vertices. A cycle is a path of length 3 or more the connects a vertex to itself.

A subgraph S of a graph G is formed by selecting $V_S \subset V$ and $E_S \subset E$, such that for every edge in E_S both of its edges are in V_S. An undirected graph is connected is there is at least 1 path from any vertex to any other. A graph without cycles is called acyclic. A free tree is a connected graph with $|V| - 1$ edges.

Adjacency Matrix. Suppose there are n vertices in a graph $G(V, E)$. Then, an adjacency matrix for G is an $n \times n$ matrix, such that for every the i, j entry of the adjacency matrix is 1, if there is an edge from vertex i to vertex j in E.

Adjacency List. An adjacency list of a graph $G(V, E)$, is a linked list of the n vertices of G. Each node in the list stores a pointer to a linked list of the vertices it is adjacent to.

13.1 Traversals

Depth-First Search. Whenever a vertex V is visited during the search, a DFS will recursively visit all of V's unmarked neighbors.

Breadth-First Search. Similar to depth-first, except it examines all vertices connected to V, before proceeding further.

Topological Search. Used for processing nodes with prerequisites (direct acyclic graphs).

13.2 Shortest-Path Problem

The general problem is to find the minimal length of a path connecting two given vertices. A example of a single-source shortest path problem is given a vertex S in a graph G, find the shortest paths to every other vertex in G starting from S. For unweighted graphs, a BFS will solve this problem. The weighted problem is more difficult.

Dijkstra's Algorithm. Suppose we process in order of distance from S to the first $i-1$ vertices that are closest to S and call this set $\mathbf{S}$. Suppose the ith closest vertex is X. Then the shortest path from S to X must have its next to last vertex in $\mathbf{S}$. Hence

$$d(S, X) = \min_{U \in \mathbf{S}} (d(S, U) + w(U, X)).$$

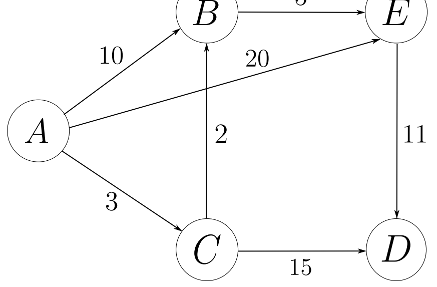


Figure 7: Weighted graph.

	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	∞	20
Process C	0	5	3	18	20
Process B	0	5	3	18	10
Process D	0	5	3	18	10
Process E	0	5	3	18	10

Figure 8: Dijkstra's algorithm results.

13.3 Minimum Cost Spanning Trees

A minimum cost spanning tree of an undirected, weighted graph G, is a graph containing the vertex of G and a subset of G's edges, such that (1) the sum of the weights of the edges is minimal and (2) all the vertices are connected.

Prim's Algorithm. Starts at some vertex N an picks the least-cost edges connecting N to another vertex, say M. Then, picks the least-cost edge connecting *either* N or M to another vertex not already in the MST. This process repeats until all vertices are connected.

Kruskal's Algorithm. First, partitions the vertices each into their own equivalence class. Then, process the edges in order of weight, typically with a min-heap. Finally, an edge is added the MST, if it connects two different equivalence classes, until only 1 equivalence class remains.

14 Miscellanea

Master's Theorem. For any recurrence relation of the form $T(n) = aT(n/b) + cn^k$, $T(1) = c$, the following relationships hold.

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

Strassen's Algorithm. Let A and B be two $n \times n$ matrices and A_{ij}, B_{ij} refer to $n/2 \times n/2$ submatrices of A and B. Then, Strassen's algorithm calculates the product AB via

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix} \tag{1}$$

where

$$\begin{aligned} s_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\ s_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ s_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\ s_4 &= (A_{11} + A_{12}) \cdot B_{22} \\ s_5 &= A_{11} \cdot (B_{12} - B_{22}) \\ s_6 &= A_{22} \cdot (B_{21} - B_{11}) \\ s_7 &= (A_{21} + A_{22}) \cdot B_{11} \end{aligned} \tag{2}$$

This improves upon the naive implementation of matrix multiplication which is $\Theta(n^3)$, to $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Skip List. A skip list is a probabilistic data structure. It is basically a sorted linked list, except the number of pointers for each node is determined probabilistically, such that the probability of having $n \geq 1$ pointers is $P(n) = \frac{1}{2^n}$.

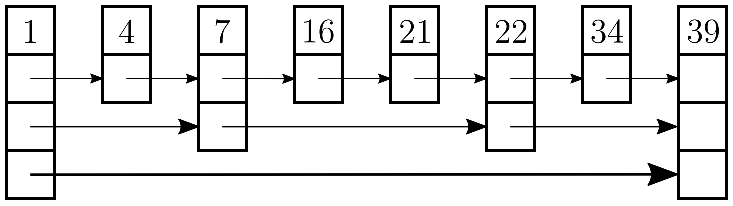


Figure 9: Basic skip list structure.

Define nodes with n pointers to be in level $n-1$. A search of a skip list starts at the outermost level and proceeds incrementally inwards. For example, a search for the key, 16, starts by processing 1, which is too small. Next, it processes 39, then 7, then 22, and finally it finds 16. Insertion and search of a skip list can approach $\mathcal{O}(n)$; however, such runtime is unlikely and it typically runs in $\mathcal{O}(\log n)$.

Computational Limits.

The idea of checking all possible solutions to a problem in parallel to determine which is correct is called **non-determinism** and an algorithm that works in this manner is a non-deterministic algorithm. Any problem that works on a non-deterministic machine in polynomial time is called an $\mathcal{NP}$ problem.

For example, the Towers of Hanoi problem is *not* an $\mathcal{NP}$ problem because it must print out 2^n moves for n disks in order to verify correctness.

A problem is $\mathcal{NP}$-hard if any problem in $\mathcal{NP}$ can be reduced to X in polynomial time. Hence X is as hard as any problem in $\mathcal{NP}$. A problem X is $\mathcal{NP}$-complete if

- X is in $\mathcal{NP}$
- X is $\mathcal{NP}$-hard.