

EECS 281: Data Structures and Algorithms

Summary Part 2

University of Michigan at Ann Arbor

Last Edit Date: 06/26/2023

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted (derived) from the slides and codes from *Professor Marcus Darden and David Paoletti*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

1. Hash Tables

1. Three Parts: **1) translation** – converts a search key into an integer; **2) compression** – limits an integer to a valid index; **3) collision resolution** – resolves search keys that hash to same table index. On average, insert - $O(1)$; search - $O(1)$; remove - $O(1)$.
2. Advantages & Disadvantages: **Good** at inserting, removing, and finding elements in a dataset with lots of distinct keys. **Bad** for sorting, $>$, and $<$ queries on numbers and when load factor $\alpha = N/M$ is high (N - number of keys, M - table size).

3. STL: unordered_set<key>, unordered_map<key, value>. Worst $O(n)$. **Unordered.**

2. Hash Collision Resolution

1. Two General Types: **1) Separate Chaining ($\alpha = N/M > 1$)** - Store a linked list in each bucket, append key value pair to the end; **2) Open Addressing ($\alpha = N/M \leq 1$)** - Use empty slots in the table. α is load factor, **do not** count deleted elements as N .
2. Separate Chaining: On average, insert - $O(1)$ if duplicate keys are allowed, $O(\alpha)$ otherwise; search - $O(\alpha)$; remove - $O(\alpha)$. When $M \approx N$, $O(\alpha) = O(1)$. If we do not use linked lists, instead **1) M vectors** - kept sorted, insert - $O(\alpha)$; search - $O(\log \alpha)$; remove - $O(\log \alpha)$. **2) M BST's** – insert & search & remove - $O(\log \alpha)$.

3. Open Addressing (must be $\alpha = N/M \leq 1$) larger α more fail in linear probing

Linear Probing	Quadratic Probing	Double Hashing
$t(k) \% M \Rightarrow \text{index}$ If index is full, then try: $t(k) = [t(k) + i] \% M$ for $i = 0, 1, 2, 3, 4, \dots$	$t(k) \% M \Rightarrow \text{index}$ If index is full, then try: $t(k) = [t(k) + i^2] \% M$ for $i = 0, 1, 2, 3, 4, \dots$	$t(k) = [t(k) + i \cdot t'(k)] \% M$ for $i = 0, 1, 2, 3, 4, \dots$ $t'(k) = q - (t(k) \% q)$ where $q < M$ and q is a prime number

Note: some prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, ...

3. Tree & Binary Search Tree Basics

1. Some Concepts: **root** - the topmost node; **parent** - immediate predecessor of a node; **child** - immediate successor of a node; **siblings** - children of the same parent; **complete tree** - all levels filled, except maybe the last, and all nodes in the last level are as far left as possible; **full / proper tree** - every node has either 2 or 0 children.

2. Tree properties:

1) Height - height(empty) = 0; height(node) = max(height(leftChild), height(rightChild)) + 1. Height balance property: $h \approx \log(\#nodes)$.

2) Size - size(empty) = 0; size(node) = size(leftChild) + size(rightChild) + 1.

3) Depth - depth(empty) = 0; depth(node) = depth(parent) + 1.

3. Tree Traversal Inorder predecessor Inorder successor

Pre Order	In Order	Post Order	Level Order
void preorder(Node *p) { if(!p) return; visit(p->key); preorder(p->left); preorder(p->right); }	void inorder(Node *p) { if(!p) return; inorder(p->left); visit(p->key); inorder(p->right); }	void postorder(Node *p) { if(!p) return; postorder(p->left); postorder(p->right); visit(p->key); }	- Breadth First Search - Use a queue
9, 4, 3, 6, 5, 7, 17, 22, 20	3, 4, 5, 6, 7, 9, 17, 20, 22	3, 5, 7, 6, 4, 20, 22, 17, 9	9, 4, 17, 3, 6, 22, 5, 7, 20

4. Binary Search Tree

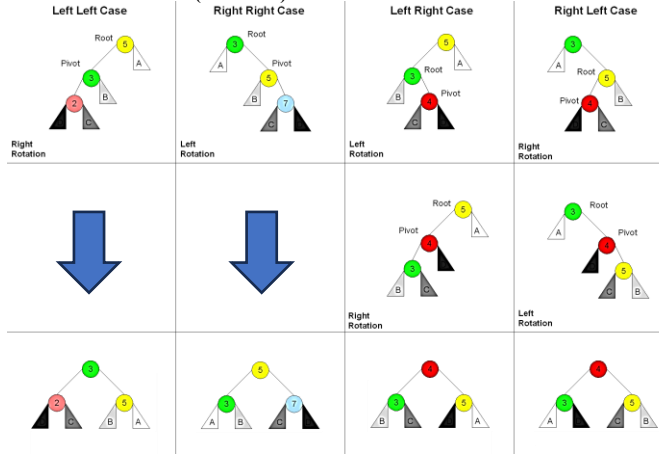
1) Important Property: left node $<$ parent & right node $\geq$ parent. Two children max.

2) Function and Time Complexity: $O(\log n)$ average (balanced), $O(n)$ worst (stick).

4. AVL Tree

1. Concepts: **1) AVL trees** are always balanced, |height(leftSubtree) - height(rightSubtree)| ≤ 1 for all nodes. **2) Insertion and deletions** are same as BST but requires checking balance. **3) Search, insert, remove** - $O(\log n)$; sort - $O(n \log n)$ build tree, $O(n)$ inorder traversal. **4) Rotation** (single & double) - $O(1)$.

2. Rotation Illustration (four cases)



5. Graph

1. Concepts: **1) A graph $G = (V, E)$** is a set of vertices $V = \{v_1, v_2, \dots\}$ together with a set of edges $E = \{e_1, e_2, \dots\}$ that connect pairs of vertices. Parallel edges are allowed; self-loops are allowed, and $E = \frac{V(V-1)}{2}$. **2) Simple Path:** sequence of edges leading from one vertex to another with no vertex appearing twice. **3) Connected Graph:** a simple path exists between any pair vertices. **4) Cycle:** simple path, except that first and last nodes are the same. **5) Complete Graph:** $|E| = \frac{|V||V-1|}{2}$. **6) Dense Graph:** $|E| \approx |V|^2$ (many edges). **7) Sparse Graph:** $|E| < |V|^2$ or $|E| \approx |V|$ (few edges).
2. Adjacency Matrix vs. Adjacency List



- $|V| \times |V|$ matrix representing graph
 - Directed: has to form adjacency matrix
 - Undirected: $V^2/2$ space adjacency matrix
 - Unweighted: 0 = no edge, 1 = edge
 - Weighted: ∞ = no edge, value = edge

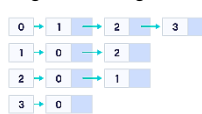
- E/V edges on each vertex list, random edge

- Access vertex list $O(1)$, find edge in list $O(E/V)$

- Average cost for individual vertex: $O(1 + E/V)$

- Directed contains each edge once, vice versa.

- Unweight: nothing = no edge, item = edge, vice versa.



Note: If sparse, use adjacency list. If dense, use adjacency matrix.

3. Depth First Search (DFS) $O(|V| + |E|)$: Uses **stack or recursion**. Visit the child nodes before visiting the sibling nodes; this allows you to traverse the depth of any particular path before exploring its breadth.

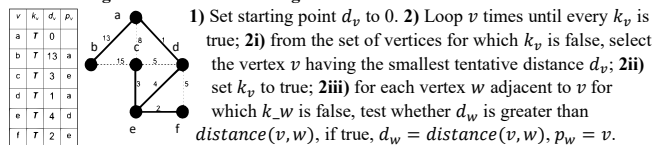
4. Breadth First Search (BFS) $O(|V| + |E|)$: Uses **queue**. Visit the sibling nodes before visiting the child nodes. When all edges have the same cost, BFS finds the shortest path between nodes.

Note: Both DFS and BFS, in adjacency list, called for each vertex at most once - $O(V)$; adjacency list for each vertex is visited at most once and set of edges is distributed over set of vertices - $O(1 + E/V)$; linear with number of vertices and edges - $O(V + E)$; in adjacency matrix, called for each vertex at most once - $O(V)$; adjacency matrix row for each vertex is visited at most once - $O(V)$; quadratic with number of vertices - $O(V^2)$.

6. Minimum Spanning Trees

1. Definition: The spanning tree of graph G that has lowest total cost. As a result, MST is a subgraph of G , contains all vertices in V , is connected, is acyclic, contains edges whose weights have a smaller sum than any other spanning tree.

2. Prim's Algorithm & Kruskal's Algorithm



When a graph is **dense** - use nested-loop Prim ($O(V^2)$) (heap Prim is $O(E \log V)$). When a graph is **sparse** - use heap Kruskal ($O(E \log V)$).

Note: Both Prim and Kruskal are greedy algorithms. Both only work when a graph is edge-weighted, connected, and undirected. Kruskal simply starts with the smallest weighted edge and finds the next smallest. Kruskal uses a disjoint-set data structure (Union-Find) to keep track of the vertices.

7. Algorithm Family

1. Algorithm Summary Table

Name	Notes	Example
Brute Force	Try every possible solution and see which one is the best solution (permutation). Simple to implement, guarantees optimal solution . Inefficient.	N/A
Greedy	Makes sequence of decisions, best at each point locally , never reconsider back. Faster than brute force. Optimal solution is not guaranteed .	Bubble sort, selection sort, insertion sort
Divide and Conquer	Divide a problem solution into multiple smaller problems preferably of equal size (top down). Often involves $\log n$. Efficient, but recursive calls make it expensive and may depend on initial state of subdomains.	Binary search, quick sort
Combine and Conquer	Start with smallest subdomain possible. Then combine increasingly larger subdomains until size = n . Similar to divide and conquer, but bottom up.	Merge sort
Backtracking	Constraint satisfaction problems (may not optimal)	N/A

Systematically consider all possible outcomes of each decision, but prune searches that do not satisfy constraint(s) (DFS with pruning). Search space is large. Eliminates exhaustive search. Not efficient if we prune a lot search results - $O(b^n)$.

Branch and Bound	Optimization problems, ideas extend from backtracking. A partial solution is pruned if its cost $\geq$ cost of best-known complete solution. If the cost of a partial solution is big, drop the partial solution.	N/A
Dynamic Programming	Save answers to overlapping subproblems to optimize time complexity.	N/A

2. Backtracking & Branch and Bound Detail

Backtracking	Branch and Bound
Algorithm checknode(node v) if (promising(v)) if (solution(v)) write solution // can exit else for each node u adjacent to v checknode(u)	Algorithm checknode(node v, Best cur) node u if (promising(v, cur)) if (solution(v)) update(cur) // can exit else for each child u of v checknode(u, cur) return cur
- solution(): check "depth" of solution (constraint satisfaction). - promising(): different for each application. - checknode(): called only if partial solution is both promising and not a solution.	- solution(): check "depth" of solution (constraint satisfaction). - update(): if the solution is better than current solution, then update (optimization). - checknode(): called if promising and not solution. - promising(): different for each application.

Note: When using branch and bound, there are two types of problems. Minimization: Start with a ∞ upper bound, if promising (new bound $<$ current best), then update current best. **Maximization:** Start with a 0 lower bound, if promising (new bound $>$ current best), then update current best.

3. Generate Permutation (Application: optimal TSP)

```
void genPerms(vector<T> &path, size_t permLength) {
    if (permLength == path.size()) {
        // Do something with the path
        return;
    }
    if (path.size() == permLength) {
        if (!promising(path, permLength))
            return;
    }
    if (!path.empty()) {
        for (size_t i = permLength; i < path.size(); ++i) {
            swap(path[permLength], path[i]);
            genPerms(path, permLength + 1);
            swap(path[permLength], path[i]);
        }
        // for ..unpermuted elements
    }
    genPerms(path, permLength + 1);
}
```

Time Complexity: $O(n!)$

4. Dijkstra's Algorithm Shortest path algorithm between a "source" node and all other nodes in a graph. A greedy algorithm. Uses **weight** of an edge to minimize the total distance. **Difference from Prim:** 1) Finds shortest path; 2) can work on both direct and undirected graphs; 3) cannot handle negative weights.

5. Dynamic Programming Detail

1) Concepts: Use when you have smaller shared / common subproblems. **Basic idea:** save subproblem results in extra memory, use extra memory to speed up. **Memoization (top-down DP):** Write a recursive function. At the start, check if we have already done this one. Before returning, store the result. **Bottom-up DP:** Store the results in an array / map. Build up the array / map starting from a base case. Stop when you get to the input you are trying to compute.

2) Steps of using Dynamic Programming:

Step 1: come up with a "subproblem": **1)** focus on how to solve a subproblem, not where it is used later; **2)** which other subproblems can you use? What is the final step to combine the results of these subproblems? **Step 2:** write a base case for that subproblem. **Step 3:** write a recurrence for that subproblem. **Step 4:** implement it bottom-up or top-down.

8. Knapsack Approaches Summary

- **Brute Force:** $O(n2^n)$ optimal solution
 - **Greedy:** $O(n)$ cannot guarantee optimal solution
 - **Dynamic Programming:** **1) $O(n)$** optimal solution; **2)** knap(int n, int cap) = max(knap(n - 1, cap), knap(n - 1, cap - cost[n]) + value[n])
 - **Branch and Bound:** **1)** Initial estimate of best solution is lower bound on knapsack value; **2)** need to estimate upper bound on remaining value of partial solution.

9. Strings and Sequences

C++ strings support begin(), end(). C++ strings use stl::string, STL. $O(1)$ for $\text{size}()$.
 template<class InputIt, class OutputIt>
bool equal(InputIt first1, InputIt last1, InputIt first2, OutputIt last2) {
for(; first1 != last1; ++first1, ++first2) {
if (*first1 != *first2) return false;
return true;
}
}
while((first1 != last1) && (first2 != last2)) {
if (*first1 < *first2) return true;
if (*first2 < *first1) return false;
++first1, ++first2;
}
return (first1 == last1) && (first2 == last2);
}

string::find(): Brute force approach: worst $O(\text{lenNeedle} * \text{lenHaystack})$, average & best $O(\text{lenHaystack})$. **String Fingerprint approach:** when fingerprints differ, strings must differ; when fingerprints match, string rarely differ. Works like hash tables. **Sliding Rabin Fingerprint:** **1)** Calculate fingerprint f_p for first m characters. **2)** Remove a character on the left $O(1)$. **3)** Add a character on the right $O(1)$. **4)** Initial calculation of f_p takes $O(m)$ time. **5)** Each of $n - m$ slides takes $O(1)$ time. **Rabin-Karp String Search:** **1)** Compute f_p of needle. **2)** Check if the haystack starts with the needle. **3)** Incrementally update the f_p of window. **4)** if fingerprint and length matches, done, or go **3)**.