Algorithms and Data Structures Study Guide

May 8, 2020

1 Data Structures

Definition. *Contiguously-allocated structures* are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.

Definition. *Linked data structures* are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

1.1 Arrays

Arrays are structures of fixed-size data records such that each element can be efficiently located by its index or address. Advantages include:

- *Constant-time access given the index* - index maps directly to a particular memory address
- *Space efficiency* - consist purely of data, no space wasted with links
- *Memory locality* - physical continuity between successive data accesses helps exploit high-speed cache memory

The downside is that we cannot adjust their size in the middle of a program's execution. Can efficiently enlarge arrays as needed with dynamic arrays. This involves doubling the size of the array each time we run out of space. The $(n/2+1)$st through nth elements will move at most once and might never move at all such that the cost of doubling is only $\mathcal{O}(n)$.

1.2 Pointers and Linked Structures

Pointers represent the address of a location in memory. All linked data structures share certain properties:

- Each node in the structure contains one or more data fields that retain the data that we need to store.
- Each node contains a pointer field to at least one other node.
- Need a pointer to the head of the structure to know where to access it.

1.2.1 Lists

```
typedef struct list {
    item_type item;
    struct list *next;
}

list *search_list(list *l, item_type x) {
    if (l == NULL) return(NULL);

    if (l->item == x)
        return(l);
    else
        return(search_list(l->next, x));
}
```

Since we don't need to maintain the list in any particular order, insert in the simplest place, the beginning.

```
void insert_list(list **l, item_type x) {
    list *p;
    p = malloc(sizeof(list));
    p->item = x;
    p->next = *l;
    *l = p;
}
```

To delete an element, need to find it's predecessor, set predecessor's link to the deleted element's link.

```
delete_list(list **l, item_type x) {
    list *p;
    list *pred;
    p = search_list(*l, x);
    while (pred == NULL) {
        if (l->next->item == x)
            pred = *l;
        else
            l = l->next;
    }
    pred->next = pred->next->next;
    free(p);
}
```

The relative advantages of linked list over array include:

- Overflow can never occur
- Insertions and deletions are simpler

- With large records, moving pointers is easier and faster than moving the items

while the relative advantages of arrays include:

- Linked structures require extra space for storing pointers
- Linked list do not allow efficient random access to items
- Arrays allow better memory locality and cache performance

1.3 Stacks and Queues

Definition. *Container* denotes a data structure that permits storage and retrieval of data items independent of content.

Definition. *Stacks* support retrieval by last in, first out (LIFO) order. Right container to use when retrieval order doesn't matter at all. Includes *Push(x, s)* to insert item at the top of stack and *Pop(s)* to return and remove the top item of the stack.

Definition. *Queues* support retrieval in first in, first out (FIFO) order. Useful for holding jobs to be processed to minimize the maximum time spent waiting. The average waiting time will be the same for FIFO or LIFO. Includes *Enqueue(x, q)* to insert item x at the back of the queue and *Dequeue(q)* to return and remove the front item from the queue.

1.4 Dictionaries

The *dictionary* data type permits access to data items by content. The primary operations include:

- *Search(D, k)* given a search key return a pointer to the element in the dictionary if one exists.
- *Insert(D, x)* given a data item, add it to the set in the dictionary.
- *Delete(D, x)* remove a data item given a pointer to it from the dictionary.

Certain dictionary data structures also support other operations:

- *Max(D)* or *Min(D)* retrieve the item with the largest (smallest) key from the dictionary. Allows dictionary to be a priority queue.
- *Predecessor(D, k)* or *Successor(D, k)* retrieve the item whose key is immediately before (after) k in sorted order.

1.5 Binary Search Trees

The above offer fast search or flexible update but not both. A *rooted binary tree* is recursively defined as either empty or consisting of a node called the root together with two rooted binary trees called the left and right subtrees. The order for subtrees matters so left is different from right. A binary search tree lavels each node in a binary tree with a single key such that for any node labeled x, all nodes in the left subtree of x have keys $< x$ while all nodes in the right subtree of x have keys $> x$.

```
typedef struct tree {
    item_type item;
    struct tree *parent;
    struct tree *left;
    struct tree *right;
} tree;
```

Search runs in $\mathcal{O}(h)$ time, where h denotes the height of the tree.

```
tree *search_tree(tree *l, item_type x) {
    if (l == NULL) return(NULL);
    if (l->item == x) return(l);
    if (x < l->item)
        return(search_tree(l->left, x));
    else
        return(search_tree(l->right, x));
}
```

Finding minimum (maximum) in a BST means retrieving the leftmost (rightmost) subtree.

```
tree *find_minumum(tree *t) {
    tree *min;
    if (t == NULL) return(NULL);
    min = t;
    while (min->left != NULL)
        min = min->left;
    return(min);
}
```

Traversing the tree. Where we process the node, affects the order they are processed. To process in sorted order, *in-order* traversal of the search tree:

```
void traverse_tree(tree *l) {
    if (l != NULL) {
        traverse_tree(l->left);
        process_item(l->item);
        traverse_tree(l->right);
    }
}
```

To make a copy of a tree, use *pre-order* traversal:

```
void traverse_tree(tree *l) {
    if (l != NULL) {
        process_item(l->item);
        traverse_tree(l->left);
        traverse_tree(l->right);
    }
}
```

To delete nodes from a tree, use *post-order* traversal:

```
void traverse_tree(tree *l) {
    if (l != NULL) {
        traverse_tree(l->left);
        traverse_tree(l->right);
        process_item(l->item);
    }
}
```

Inserting is a constant-time operation after the search is done in $\mathcal{O}(h)$.

```
insert_tree(tree **l, item_type x, tree *parent) {
    tree *p;
    if (*l == NULL) {
        p = malloc(sizeof(tree));
        p->item = x;
        p->left = p->right = NULL;
        p->parent = parent;
        *l = p;
        return;
    }
    if (x < (*l)->item)
        insert_tree(&((*l)->left), x, *l);
    else
        insert_tree(&((*l)->right), x, *l);
}
```

Deletion depends on the node to be deleted has children nodes. If no children, can just delete. If it has one child, can replace the parent's pointer to the deleted to the child. If it has two chldren, relabel this node with the key of its immediate successor. If the tree is perfectly balanced, the tree will have height $\mathcal{O}(\log n)$. On average there is high probability the tree will have that height.

1.6 Priority Queues

Priority queues provide more flexibility than simple sorting, because they allow new elements to enter at arbitrary intervals. A basic implementation uses a heap. Basic operations include:

- *Insert(Q, x)* given an item with key, insert it into the priority queue.
- *Find-Minimum(Q)* return a pointer to the item whose key value is smaller (larger) than any other key.
- *Delete-Minimum(Q)* remove the item from the queue whose key is minimum (maximum).

1.7 Heaps

Maintains a partial order on the elements which is weaker than sorted but stronger than random. A *heap-labeled tree* is a binary tree such that the key labeling of each node dominates the key of each of its children. A node in a min-heap dominates its children by having a smaller key, a node in a max-heap dominates its children by being bigger. Instead of using a tree and incurring the cost of pointers, a heap stores the data as an array, using the position to implicitly be the pointers. The root is in the first position, it's children in the second and third, respectively. The 2^l keys of the lth level will be in positions 2^{l-1} to $2^l - 1$. Inserting into a heap is as easy inserting new elements at the end of the heap. This ensures the tree is balanced but not the dominance ordering the of the keys. The solution is to swap the inserted element with its parent. Each insert takes at most $\mathcal{O}(n \log n)$.

```
pq_insert(priority_queue *q, item_type x) {
    q->n = (q->n) + 1;
    q->q[ q->n ] = x;
    bubble_up(q, q->n);
}

bubble_up(priority_queue *q, int p) {
    if (pq_parent(p) == -1) return;

    if (q->q[pq_parent(p)] > q->q[p]) {
        pq_swap(q, p, pq_parent(p));
        bubble_up(q, pq_parent(p));
    }
}
```

When extracting the minimum from a heap it leaves an empty space at the top of the array. This can be filled with the last element and bubbling down until dominance is restored. This percolate-down operation is also called *heapify* because it merge two heaps.

```
item_type extract_min(priority_queue *q) {
    int min = -1;
    if (q->n <= 0) {print "empty queue");
    else {
        min = q->q[1];
```

```
        q->q[1] = q->q[q->n];
        q->n = q->n - 1;
        bubble_down(q, 1);
    }
    return(min);
}
```

1.8 Hashing

A hash function is a mathematical function that maps keys to integers. We use the value of our hash function as an index into an array and store our item at that position.

1.8.1 Collision Resolution

Chaining has the hash table represented as an array of linked lists. If there is a collision, search through the list for the desired element. *Open addressing* maintains the hash table as an array of elements. On insertion, if the desired position is not empty, insert the item in the next open spot in the table.

1.9 Strings

1.9.1 Substring Pattern Matching

The *Rabin-Karp* algorithm is linear expected-time for string matching. Compare the hash of the match pattern to hashes of the substrings of the text. The trick is that the incremental hash is a function of the previous hash and the next character.

1.10 Union-Find

2 Algorithms

Three desirable properties for a good algorithm: correct, efficient, and easy to implement. Problem specifications has two parts: the set of allowed input instances and the required properties of the algorithm's output. Best way to prove an algorithm is incorrect is to produce an instance that yields an incorrect answer. Need to be able to verify which can be done with simple counter examples. Think small, think exhaustively, hunt for the weakness, go for a tie, seek extremes.

2.1 Summations

2.1.1 Arithmetic Progressions

$\sum_i^n i = n(n+1)/2$, note the sum is quadratic. Generalized: $\sum_i^n i^p = \Theta(n^{p+1})$, noting that for $p \geq 1$, the sum of square is cubic, the sum of cubes is quartic,

etc. If $p \leq -1$ the sum always converges to a constant, even as $n- > \inf$.

2.1.2 Geometric series

$$G(n,a) = \sum_{i=0}^{n} a^i = a(a^{n+1} - 1)/(a-1)$$

When $a < 1$, this converges to a constant even as $n- > \inf$, meaning that the sum of a linear number of things can be constant. When $a > 1$, the sum grows rapidly with each new term.

2.2 Modeling the problem

2.2.1 Combinatorial Objects

- Permutations - arrangements, or orderings, of items: arrangement, tour, ordering, or sequence
- Subsets - selections from a set of items where order does not matter: cluster, collection, committee, group, packaging, or selection
- Trees - hierarchical relationships between items: hierarchy, dominance, relationship, ancestor/descendant relationship, taxonomy
- Graphs - relationships between arbitrary pairs of objects: network, circuit, web, relationship
- Points - locations in some geometric space: sites, positions, data records, locations
- Polygons - regions in some geometric space: shapes, regions, configurations, boundaries
- Strings - sequences of characters or patterns: text, characters, patterns, labels

2.2.2 Recursive Objects

Learning to think recursively is learning to look for big things that are made from smaller things of exactly the same type as the big thing. Recursive descriptions of objects require both decomposition rules and basis cases, the specification of the smallest and simplest objects where the decomposition stops.

- Permutations - delete the first element of a permutation of $1, ..., n$ things and you get a permutation of the remaining n-1 things
- Subsets - every subset of the elements $1, ..., n$ contains a subset of $1, ..., n-1$ made visible by deleting element n if it is present
- Trees - delete the root of a tree and get a collection of smaller trees

- Graphs - delete any vertex from a graph and you get a smaller graph.
- Points - Take a cloud of points, separate them into two groups by drawing a line, now you have two smaller clouds of points

2.3 Complexity

- Constant functions: $f(n) = 1$
- Logarithmic functions: $f(n) = \log n$, shows up in binary search
- Linear functions: $f(n) = n$, cost of looking at each item once in an n-element array
- Superlinear functions: $f(n) = n \log n$, shows up in quicksort, mergesort
- Quadratic functions: $f(n) = n^2$, cost of looking at most or all pairs of items in an n-element universe
- Cubic functions: $f(n) = n^3$, enumerate through all triples of items in an n-element universe
- Exponential functions: $f(n) = c^n$, functions like 2^n arise when enumerating all subsets of n items
- Factorial functions: $f(n) = n!$, functions like n! arise when generating all permutations or orderings of n items

2.4 Scheduling

Earliest job first might block us from taking many other jobs if it is long. Shortest job first may preclude us from taking two other jobs that overlap the shorter one. Optimal is to choose the job withe earliest completion date.

2.5 Linear Search $\mathcal{O}(n)$

Loop through each element until find element.

2.6 Binary Search $\mathcal{O}(\log n)$

Compare the middle element of a sorted array, then search the lower or upper half of the array if the value is less than or greater.

2.6.1 Counting Occurrences

Want to find the number of times an element occurs in an array. Could do binary search to find an instance of the element and search to the left and right. Can instead remove the equality test in binary search which will allow the search to run till the boundary.

2.6.2 One-Sided Binary Search

If we have a sorted array of 0's and 1's and want to find the transition point. Repeatedly test at larger intervals ($A[1], A[2], A[4], A[8], ...$) until we find a nonzero value. Then do binary search on the interval that contains it.

2.7 Sorting

Some examples of how sorting can be used to solve problems:

- *Searching* - binary search tests whether an item is in sequence in $\mathcal{O}(\log n)$ if the items are sorted
- *Closest Pair* - find the pair of numbers that have the smallest difference between them $\mathcal{O}(n \log n)$
- *Element Uniqueness* - find any duplicates, special case of closest pair (ie. any pair separated by a gap of zero)
- *Frequency Distribution* - find which element occurs the most, find how often an element occurs by look it up using binary then walking left and right, or by doing binary search on the bounds
- *Selection* - kth largest item

Definition. *Stable* - If two items are equal, a sorting algorithm is considered stable if it leaves in the same relative order.

2.7.1 Bubblesort $\mathcal{O}(n^2)$

For each element in the array compare and swap adjacent elements.

2.7.2 Insertionsort $\mathcal{O}(n^2)$

Loop through each element, if it is less than its predecessor, swap, and compare with the next predecessor until in the right location. Works well if the data is almost sorted. This is the simplest example of the incremental insertion technique where you build up a complicated structure on n items by first building it on $n-1$ items and then making the necessary changes to add the last item.

```
def insertionsort(seq):
    for i in range(1, len(seq)):
        j = i
        while j > 0 and seq[j] < seq[j-1]:
            swap(seq[j], seq[j-1])
            j = j - 1
```

2.7.3 Selectionsort $\mathcal{O}(n^2)$

Looop through each element, loop through successors to find the minimum, swap the current element and min.

```
def selectionsort(seq):
    for i in range(len(seq)):
        min = i
        for j in range(i + 1, len(seq)):
            if seq[j] < seq[min]:
                min = j
        swap(s[i], s[min])
```

2.7.4 Mergesort $\mathcal{O}(n \log n)$

Recursively split array until at a single element then merge back together. The merge works by interleaving the two sub-arrays. This is a divide and conquer algorithm. It is a great for sorting linked lists because it does not rely on random access to elements. Its main disadvantage is its need for an auxilliary buffer to store the result of the merged arrays.

```
def mergesort(A, low, high):
    if(low < high):
        middle = int(low+high)/2
        mergesort(A, low, middle)
        mergesort(A, middle+1, high)
        merge(s, low, middle, high)
```

2.7.5 Quicksort $\mathcal{O}(n \log n)$

Pick a random pivot point and move all elements smaller to the left and all elements greater to the right. Repeat for the left/right sections.

```
def quicksort(A, lo, hi):
    if lo < hi:
        p = partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

def partition(A, lo, hi):
    pivot = A[hi]
    i = lo - 1
    for
```

2.7.6 Heapsort $\mathcal{O}(n \log n)$

Construct a heap and repeatedly call heapify. It is an in-place sort so uses no extra memory over the array containing the elements to be sorted.

2.7.7 Distribution Sorting

Sorting via bucketing elements by the first letter. Then bucket by the second letter, etc until buckets contain only one element. Bucketing is very effective when the distribution of the data will be roughly uniform.

3 Graphs

A graph $G = (V, E)$ is defined on a set of vertices V, and contains a set of edges E of ordered or unordered pairs of vertices from V Fundamental properties:

- *Undirected vs. Directed* - undirected if edge $(x, y) \in E$ implies that (y, x) is also in E, otherwise it is directed.
- *Weighted vs Unweighted* - each edge (or vertex) in a weighted graph is assigned a numerical value, unweighted graphs have cost distinction.
- *Simple vs Non-simple* - A self-loop is an edge (x, x) involving only one vertex, an edge (x, y) is a multiedge if it occurs more than once in the graph. both require special care, if a graph that avoids them is simple.
- *Sparse vs. Dense* - graphs are sparse when only a small fraction of the possible pairs have edges between them.
- *Cyclic vs. Acyclic* - acyclic graph does not contain any cycles
- *Embedded vs Topological* - embedded if the vertices and edges are assigned geometric positions
- *Implicit vs Explicit* - certain graphs are not explicitly constructed and then traversed, but built as we use them
- *Labeled vs Unlabeled* - each vertex is assigned a unique identifier in a labeled graph to distinguish it from all other vertices

3.1 Data Structures

- *Adjacency Matrix* - Represent G using an $n \times n$ matrix M where element $M[i, j] = 1$ if (i, j) is an edge of G, 0 otherwise.
- *Adjacency Lists* - Use linked lists to store the neighbors adjacent to each vertex.

```
typedef struct {
    int y;
    int weight;
    struct edgenode *next;
} edgenode;
```

```
typedef struct {
    edgenode *edges[MAXV+1];
    int degree[MAXV+1];
    int nvertices;
    int nedges;
    bool directed;
} graph;

insert_edge(graph *g, int x, int y, bool directed) {
    edgenode *p;
    p = malloc(sizeof(edgenode));
    p->weight = NULL;
    p->y = y;
    p->next = g->edges[x];
    g->edges[x] = p;
    g->degree[x] ++;
    if (directed == False)
        insert_edge(g, y, x, True);
    else
        g->nedges ++;
}
```

3.2 Breadth-First Search

Breadth-first search processes each node in the adjacency of the start node first before moving onto others. Breadth first search returns the shortest path. Once the node is found, can follow its parents back up to the root.

```
bfs(graph *g, int start) {
    queue q;
    int v;
    int y;
    edgenode *p;
    enqueue(&q, start);
    discovered[start] = TRUE;
    while (is_empty(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = TRUE;
        p = g->edges[v];
        while (p != NULL) {
            y = p->y;
            if ((processed[y] == FALSE) || g->directed)
                process_edge(v, y);
            if (discovered[y] == FALSE) {
                enqueue(&q, y);
```

```
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

The vertex-coloring problem seeks to assign a label to each vertex of graph such that no edge links any two vertices of the same color. A graph is bipartite if it can be colored without conflicts while using only two colors.

3.3 Depth-First Search

The difference between BFS and DFS results is in the order in which they explore vertices. This order depends completely upon the container data structure used to store the discovered but not processed vertices. Using a queue, we explore the oldest unprocessed vertices first. Using a stack, we explore the vertices along a path, visiting a new neighbor if one is available.

```
dfs(graph *g, int v) {
    edgenode *p;
    int y;
    if(finished) return;
    discovered[v] = TRUE;
    time = time + 1;
    entry_time[v] = time;
    process_vertex_early(v);
    p = g->edges[v];
    while (p != NULL) {
        y = p->y;
        if (discovered[y] == FALSE) {
            parent[y] = v;
            process_edge(v, y);
            dfs(g, y);
        }
        else if ((!processed[y]) || (g->directed))
            process_edge(v, y);
        if (finished) return;
        p = p->next;
    }
    procecss_vertex_late(v);
    time = time + 1;
    exit_time[v] = time;
    processed[v] = TRUE;
}
```

It is easy to identify if an edge (y, x) in an undirected graph when processing (x, y) if y is undiscovered or has not been completely processed. If y is an ancestor of x and in a discovered state this is the first transversal unless y is the immediate ancestor x. Can find a cycle by finding a back edge:

```
process_edge(int x, int y) {
    if (parent[x] != y) {
        print("Cycle found");
        finishe = TRUE;
    }
}
```

A vertex is called an articulation vertex or cut-node if its deletion disconnects a connected component of the graph. The connectivity of a graph is the smallest number of vertices whose deletion will disconnect the graph. The connectivity is one if the graph has an articulation vertex. If v is visited for the first time as we traverse the edge (u, v), then the edge is a tree edge. Else, if v has already been visited if v is an ancestor of u, then edge (u, v) is a back edge else if v is a descendant of u, then edge (u, v) is a forward edge else if v is neither an ancestor or descendant of u, then edge (u, v) is a cross edge. Edge classification in directed graph:

```
int edge_classification(int x, int y) {
    if (parent[y] == x) return(TREE);
    if (discovered[y] && !processed[y]) return(BACK);
    if (processed[y] && (entry_time[y] > entry_time[x])) return(FORWARD);
    if (processed[y] && (entry_time[y] < entry_time[x])) return(CROSS);
    print("Unclassified")
}
```

3.4 Topological Sorting

Topological sorting order the vertices on a line such that all directed edge go from left to right. Such an ordering cannot exist if the graph contains a directed cycle. Each DAG has at least one topological sort. It gives us an ordering to process each vertex before any of its successors. This can be done using depth-first searching. A directed graph is a DAG iff no back edges are encountered. We push each vertex on a stack as soon as we have evaluated all outgoing edges.

```
process_vertex_late(int v) {
    push(&sorted, v);
}

process_edge(int x, int y) {
    class = edge_classification(x, y);
    if (class == BACK)
        print("Not a DAG");
}
```

```
topsort(graph *g) {
    init_stack(&sorted);
    for (i=1; i<=g->nvertices; i++)
        if (!discovered[i])
            dfs(g, i);
    print_stack(&sorted);
}
```

A graph is strong connected if there is a directed path between any two vertices.

4 Weighted Graphs

4.1 Minimum Spanning Trees

A spanning tree of a graph is a subset of the edges that form a tree connecting all vertices. The minimum spanning tree is the spanning tree whose sum of edge weights is as small as possible.

4.1.1 Prim's Algorithm

Prim's is a greedy algorithm which chooses the smallest weight edge that will enlarge the number of vertices in the tree. Start by picking an arbitrary node. The next node to visit is the node with the smallest weight edge from the starting node. Then continue by choosing the smallest weight edge available from the visited nodes, skipping any visited nodes.

4.1.2 Kruskal's Algorithm

Kruskal's is also a greedy algorithm. It starts by putting all the edges in a priority queue ordered by weight. Then loops through the edges adding the edge to the tree if adding it connects two previosly disconnected components.