Algorithms and Data Structures in C

Course Summary

B Pijnacker | s4106164

Contents

About this summary

This summary is based on the course *Algorithms and Data Structures in C* for BSc Computing Science Year 1 2019-2020. Most literal code implementations will not be included in this summary.

Chapter 1

Linear data structures

1.1 Stacks

A stack is a data structure where we can insert items with a function `push()` and remove items with `pop()`. You should visualize a stack literally (for example a stack of books), where items are placed on the top and removed from the top. No items can be taken out that are not on the top.

A `push` can be implemented by placing an item at the first free position in the array of the stack, and `pop` by retrieving the first nonempty position of an array. When pushing onto a full array, the array will have to be extended. This is most often done by multiplying the current size by 2.

1.2 Queues

Queues contain the functions `enqueue()` and `dequeue()`. A queue might be said to be the oposite of a stack, and a queue adds items to the tail and removes them from the head.

A queue will contain a pointer for the front and one for the back. Because we don't want to move all items any time we dequeue or enqueue, the queue structure may wrap around in an array. If we enqueue into a full array, we need to double the queue size just like we did with a stack. We need to make sure that we don't break the queue in case of a split configuration here.

1.2.1 Priority queues

A priority queue has a number that indicates the priority of an element in the queue. Upon dequeuing, the element with the highest priority is chosen.

1.3 Lists

A list is a data structure that stores items using nodes. Every node contains an item, and a reference (pointer) to the next node. Lists are dynamically allocated; we thus need a function to add new nodes to a list.

Freeing a list happens iteratively. We create a copy of the list starting at the second node, we free the current node, and use the copy to free the rest of the list.

1.3.1 Stacks and queues implemented with lists

A stack can be easily implemented with a list. We add items to the begin, and remove items from the begin. We can just add a new node at the begin of the list and give it a reference to the old first item. Removing an entry is as simple as freeing it and setting the pointer to the first item to the second item.

A queue can be implemented in almost the same way, except we need to have a pointer to the first and the last item. Dequeuing is equal to popping in a stack, but enqueuing is a little bit more involved. We want to add an item to the end of the list, so we need to create a new item, reference the previous last item to it, and populate the new item.

1.3.2 Traversing a list

This section is only code, so we just copy it here.

```
void visitList(List li) {
    while (li != NULL) {
        visit(li);
        li = li -> next;
    }
}
```

```
void visitListRec(List li) {
    if (li == NULL) return;
    visit(li);
    visitListRec(li -> next);
}
```

1.3.3 Operations on an arbitrary position in a list

An operation on an arbitrary position in a list can be done by traversing through the list until the item we need is found. An edge case that can occur here is that we reach the last node before we're at the node we're looking for. In this case, we should give an error and do nothing to the list.

Adding a node at a certain position requires us to go the node before, add a node, and change the pointers of this node and the new node to make the list correct again. Removing a ndoe does basically the same, because we need to change the node before the node we want to remove.

1.3.4 Ordered lists

An ordered list is a list with ordered items. We can implement a priority queue with this by inserting items in the right order. We can remove items at the front.

1.4 Application: recognize and evaluate arithmetical expressions

1.4.1 Grammars

Expressions are generated by a grammar that outlines the rules an expression needs to follow. Generally, a grammar is expressed in terms of other parts of itself, until we reach something that cannot be generalized. For example, an integer is 0, or a positive integer, or minus a positive integer. A positive integer is a positive digit with any other digits following. A digit is 0 or a positive digit, a positive digit is 1 through 9.

1.4.2 Arithmetical expressions

We can also create a grammar for arithmetical expressions. This is a grammar we can use for interpreting the expression.

1.4.3 The interpretation of expressions

We first of all want to recognize of a given string of characters is an expression or not. The first step for this is scanning, reading the string of characters into a tokenlist we can use. The second step is parsing. A parser checks if the tokenlist we have can be produced by the specified grammar. The third step is evaluation, where we compute the value of the expression.

Scanning

We want to create a tokenlist. This is a linked list, where every node has a tokentype, a token, and a pointer to the next node in the list. First of all, we read the input into a character string. We can then use this string to create our tokenlist.

We create functions `matchNumber`, `matchIdentifier`, and `matchCharacter`. We can use these functions to add the necessary token to the list. We use functions `isnumber` and `isalpha` to detect which one of these functions we need to call.

Recognition

We can now determine if our generated tokenlist can be generated by our grammar. We have a function `acceptNumber`, `acceptIdentifier`, and `acceptCharacter`. With these functions, we can create new functions that check if our tokenlist is correct for our given grammar.

We also have a function `recognizeExpressions` that repeatedly asks for an expression, generates a tokenlist, and checks if the expression is correct.

Evaluation

We can evaluate functions by going up the grammar. We start by evaluating the easier things, and then go up the grammar to evaluate the more complex parts. This happens in basically the same way as the recognition. For example, if we find a number, we check what comes after it. We then determine the right side of this as well and perform the computation that is needed. Doing this for the entire expression gives us the answer.

We have a function `evaulateEpxressions` that does the same thing as `recognizeExpression`, but also evaluates it.

Chapter 2

Trees

A tree is a datastructure that consists of nodes and edges. A tree starts at the top, with a root. The height of a tree is the maximum depth of a node of the tree. The number of edges in a tree equals the amount of nodes -1.

2.1 Binary trees

A binary tree is a tree where every node has at most two children; a left child and a right child.

2.1.1 Relation between height and number of nodes

A tree where every node has the same depth, and every node (except leaves) have two children is called a 'fat' binary tree. A fat binary tree with height h has $2^{h+1} - 1$ nodes. We may also call this a perfect binary tree with height h.

In a perfect binary tree, we can store $\mathcal{O}(2^h)$ items with every item accessible in h steps.

2.1.2 Numbering the node positions

We can number the node positions using 2 easy rules:

1. the root has position 1
2. a node with position n has a left child with position $2n$ and a right child with position $2n + 1$

2.1.3 Two representations of binary trees

Pointer representation

This representation is very close to that of linked list, except that instead of a `next` pointer, we have a `leftChild` and a `rightChild` pointer.

Array representation

We have defined how to number node positions in a tree, we can use this indexing of nodes as array indeces so we can represent our tree as an array.

2.1.4 Traversing a binary tree

We have three different ways to traverse a tree:

Preorder traversal

```
void preOrder(Tree t) {
    if (t == NULL) return;
    visit(t);
    preOrder(t -> leftChild);
    preOrder(t -> rightChild);
}
```

Inorder traversal

```
void inOrder(Tree t) {
    if (t == NULL) return;
    preOrder(t -> leftChild);
    visit(t);
    preOrder(t -> rightChild);
}
```

Postorder traversal

```
void potOrder(Tree t) {
    if (t == NULL) return;
    preOrder(t -> leftChild);
    preOrder(t -> rightChild);
    visit(t);
}
```

2.2 Search trees

A search tree is a binary tree where every node contains a value from a linearly ordered set of values, and which satisfies the *search tree property*:

All nodes k with a value x satisy:

1. $\forall q \in k_{left} : q < x$
2. $\forall q \in k_{right} : q > x$

Or translated, all values in the left subtree of k are smaller than x and all values in the right subtree of k are greater than x.

A search tree guarantees that an inorder traversal returns a sorted list.

2.3 Heaps

A heap, like a search tree, is a binary tree where every node contains a value from a linearly ordered set. Moreover, a heap is always a complete binary tree. In a heap, any node has a value lesser then or equal to its parent. A heap implements a priority queue.

We first describe how to enqueue and dequeue, then we describe algorithms upheap and downheap to restore the 'heapness'. Enqueue is easy:

1. Add a new node v to the heap so that it remains a complete tree
2. Put value n in v
3. Upheap(v)

Removing the largest value is done as follows:

1. Save the value of the root to return
2. Set the value of the root equal to the last node of the heap
3. Remove the last node
4. Downheap($root$)

For the algorithms of upheap and downheap I direct you to the reader but these are important to study.

2.4 Tries

2.4.1 Standard tries

A standard trie T for the collection W of words is a tree with the following properties:

- the root of T is empty, and every other node contains a letter;
- the children of a node of T contain different letters and are in alphabetical order;
- the branches in T from the root correspond exactly with the words in W.

2.4.2 The compressed trie

A compressed trie is obtained by compressing any non-branching parts into a single node. In this way, a node can contain mutiple letters. It has the following properties:

- the root is empty, and every other node contains a nonempty string;
- the children of a node contain strings with different initial letters and are ordered alphabetically on the initial letter of the string;
- there are no nodes with branching degree 1 (if W contains at least two words);
- the branches from the root correspond exactly with the words in W.

2.4.3 The compact trie

The compact trie is obtained from the compressed trie by replacing the strings in the nodes by their coordinates. For define an array A that represents the collection of words. The compact trie has the following properties:

- every node except the root contains two numbers referring to a string;
- the children of a node are ordered alphabetically on initial letter;
- there are no nodes with branching degree 1;
- the brances from the root correspond exactly with the words in W.

We have reduced the size of the nodes in the tree to a fixed value, reducing memory use of the trie.

2.4.4 Suffix tries

We might try to create a compact trie that contains all the substrings of a given text T. There are n substrings with length 1, $n - 1$ with length 2, ..., and 1 substring with length n. In total, this is $n(n + 1)/2$. This is a lot. Luckily it suffices to work only with the n suffixes of T.

2.5 Application: expression trees

2.5.1 Expression trees

An expression tree is a binary tree that contains the explicit structure of an arithmetical expression. The root node contains an operator, and the left and right children contain the left and right segments that belong to that operator.

2.5.2 Prefix expressions

Prefix expressions is a way to write down expressions that is different from 'normal'. We first have the operator, and then the two operands.

Example: $3 + 3 \Rightarrow +3\ 3$; $(3 \times 4) + 7 \Rightarrow + \times 3\ 4\ 7$

Chapter 3

Graphs

A graph $G = (V, E)$ consists of a collection V of nodes and a collection E of edges that connect nodes. Two edges are parallel when they connect the same nodes, and two nodes are neighbors when they have an edge between them.

3.1 The start of graph theory

An Euler path is a path where every edge occurs exactly once. An Euler cycle is an Euler path with an identical begin and end node. A connected graph has en Euler cycle iff all nodes have an even degree. A connected graph has an Euler path iff at most two nodes have an odd degree.

3.2 More notions related to graphs

A connected graph without simple cycles is called a tree. A tree in graph theory may not always have a root. A tree with an indicated root is called a rooted tree.

A subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. If $V' = V$ then G' is a spanning subgraph. A special case of this is a minimal spanning subgraph that is also a tree.

A weighted graph is a graph where all edges contain a number; the weight of that edge.

3.4 Searching in a graph

Graph traversal is somewhat more difficult than tree traversal because a graph has less structure than a tree. Two common traversal stratagies are depth-first search (DFS) and breadth-first search (BFS). These algorithms are restricted to searching only the nodes that are connected to the starting node.

3.5 Depth-First Search

The algorithm traverses down edges as long as possible, until it finds a node that is has seen before. It then backtracks until an edge is available that has not been visited before, and goes down that path. In this way, we visit all the nodes.

3.6 Breadth-First Search

BFS is not as greedy ad DFS is. It first treats all the nodes that are one step away, then two steps away, and so forth. This is realized with a queue. Newly discovered nodes are enqueued, and when a node is dequeued, all its incident and unexplored edges are explored. Because of this property, BFS always finds a minimal path between two nodes.

3.7 Dijkstra's Shortest path algorithm

Dijkstra's algorithm is an algorithm that calculates the shortest path from one node to all the others. It does this by calculating the distance to our begin node v from all nodes reachable in one step by nodes of which the distance is currently known. The shortest distance that is here found is assigned to the node it belongs to, as this is the shortest distance to this node. This increases the size of our known set. Eventually, we will have our known set be filled with all the nodes that are somehow connected to v, and the distance from v to those nodes. It is possible to have Dijkstra's algorithm also construct the paths taken. You can do this by saving the node that was used to get to a new node for every new node added to our known set. We can then, when we are done, reconstruct our path by going through this list.

3.8 A variant: A* algorithm

The A* algorithm is a variant of Dijksta's algorithm. It doesn't find the shortest path to *all* nodes, but only to one. We can only do this is we have a lower bound for for the distance from v to w.

When choosing which node to add to our known set, we don't choose the node with the shortest distance to v, but we choose the smallest value of d (the distance) $+h$ (where $h : \text{nodes}(G) \to \mathbb{N}$ and $h[u]$ indicates the lower bound).