

Hugh Baldwin
up2157117

Data Structures and Algorithms

M21270

TB1

University of Portsmouth

BSc Computer Science

2nd Year

Contents

1	Lecture - Workshop 1 and 2 Async	3
2	Lecture - Workshop 3 and 4 Async	6
3	Lecture - Workshop 5 Async	9
4	Lecture - Workshop 6 Async	10
5	Lecture - Workshop 7 Async	12
6	Lecture - Workshop 8 and 9 Async	14

CONTENTS

- Mid-Unit assessment (30%) on 8th November
- Exam (70%) in main assessment period

Lecture - Workshop 1 and 2 Async

16:00

27/09/23

Dalin Zhou

Data Structures

- A data structure is a way to store and organise data
 - A collection of elements
 - A set of relations between the elements
- Classification of data structures
 - Linear
 - * Unique predecessor and successor
 - * e.g. Stack, Queue, etc
 - Hierarchical
 - * Unique predecessor, many successors
 - * e.g. Family tree, management structure, etc
 - Graph
 - * Many predecessors, many successors
 - * e.g. Railway or road map, social network
 - Set
 - * No predecessor or successor
 - * e.g. A class of students
- Static vs Dynamic
 - A static data structure has a fixed size, which cannot be exceeded and must be allocated in its entirety when created
 - A dynamic data structure has a dynamic size, which can change at runtime and only uses as much memory as is needed for its contents
- Most data structures follow CRUD(S) for their basic operations;
 - Create - Add a new element
 - Read - Read an existing element
 - Update - Modify an existing element
 - Destroy (or Delete) - Remove an existing element
 - (Search) - Search for an element matching certain search conditions
- Each program would need a different data structure - there is no one-size-fits-all

Abstract Data Types

- An ADT is a collection of data and associated methods
- The data in the ADT cannot be directly accessed, only by using the methods defined in the ADT
- Stacks
 - A stack is a collection of objects in which only the top-most element can be modified at any time
 - This means it is a LIFO (Last-in first-out) structure
 - A stack must implement the following methods:
 - * Push - Add an item to the top of the stack
 - * Pop - Remove the item from the top of the stack
 - * Peek - Examine the item at the top of the stack
 - * Empty - Determine if the stack is empty
 - * Full - Determine if the stack is full
 - * A stack is typically implemented using an array, which is hidden behind the methods of the ADT
- Queues
 - A queue is a collection of objects in which the object which has been in the queue for the longest time is removed first
 - This means it is a FIFO (First-in first-out) structure
 - A queue must implement the following methods:
 - * Enqueue - Add a new item to the tail of the queue
 - * Dequeue - Remove the item at the head of the queue
 - A queue is typically implemented using an array, which is hidden behind the methods of the ADT
 - There are two methods of implementing a queue
 - * Fixed head
 - The head of the queue is always at the 0th position in the array, and elements are shifted as they are dequeued
 - * Dynamic head
 - The head of the queue changes to the index of the current head of the queue, and elements are not shifted
 - Since this leaves empty spaces before the head, space may be wasted. To solve this, a circular array could be used

Algorithms

- An algorithm is a procedure that takes a value or set there of as input and produces a value or set there of as an output
- A sequence of computational steps that transforms the input into the output
- Ideally, we would always use the most efficient algorithm that is available
- Classification of algorithms
 - Brute-force
 - Divide and conquer
 - Backtracking
 - Greedy
 - etc

Big-O Notation

- To determine the Big-O of an algorithm
 - Count how many basic operations (assignment, addition, multiplication and division) there are for the worst-case scenario (e.g. 10 items in a stack of length 10)
 - Ignore the less dominant terms of this equation
 - Ignore any constant coefficients
 - The remaining terms become the Big-O complexity of the algorithm
- Example 1:
 - There are $2n + 1$ operations for an algorithm
 - We can ignore the $+1$ as this has less impact, leaving $2n$
 - We can ignore the constant coefficient 2, leaving n
 - So the Big-O notation would be $O(n)$
- Example 2:
 - There are $2n^2 + n + 1$ operations for an algorithm
 - We can ignore the $+1$, leaving $2n^2 + n$
 - We can ignore the $+n$, leaving $2n^2$
 - We can ignore the 2, leaving n^2
 - So the Big-O notation would be $O(n^2)$
- Order of dominance:
 - Constant < Linear < Logarithmic < Quadratic < Cubic < ...
 - $O(1) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < \dots$

Lecture - Workshop 3 and 4 Async

15:00

05/10/23

Dalin Zhou

Searching Algorithms

- It is much easier to search for an item if they are already sorted
- Linear Search
 - In a linear search, you start at index 0 and keep looking through the items in the array until the item is found
 - Since the worst-case would be checking every item in the list, the worst-case Big-O would be $O(n)$
 - Theoretically, the best case would be where the item is at the start of the array and so would be $O(1)$
 - The average case is theoretically $\frac{n}{2}$, but that is still a Big-O of $O(n)$
 - This works for sorted and unsorted lists
- Binary Search
 - In a binary search, you start by checking the middle of the list
 - If the item is larger than the value in the middle, you check the top half of the list. If it's smaller, check the bottom half
 - Repeat this process, splitting the list in half, until the item is found
 - In the case of an list with an even number of items, look at the middle item rounded down (e.g. in a list of 8 items, start with the item at index 3)
 - The worst-case would be $1 + \log_2 n$ operations, so $O(\log_2 n)$
 - The best-case would still be $O(1)$ if the item is at the mid-point in the list
 - The average would be $O(\log_n 2)$
 - This only works for pre-sorted lists, so if the list is not sorted, there is an efficiency tradeoff to sort the list before it can be searched

Sorting Algorithms

- Selection Sort
 - This works by sorting the list one item at a time
 - It does this by dividing the list into two sections: the sorted part on the left and the unsorted part on the right
 - The smallest or largest item is selected from the list (depending if sorting ascending or descending) and swapped into the first position in the sorted part
 - This process is repeated until the entire list is sorted

- Each selection requires $n - 1$ comparisons, and the selection process must be performed $n - 1$ times
- Therefore both the best and worst-case would be $(n - 1)(n - 1) = n^2 - 2n + 1$ and therefore a Big-O of $O(n^2)$
- Bubble sort
 - This works by repeatedly looping through the list, swapping adjacent items that are in the wrong order
 - This causes either the largest or smallest item to 'bubble up' to the top of the list
 - The process is repeated until all items have been rearranged into the correct order
 - Each iteration requires $n - 1$ comparisons, and must be repeated n times
 - Therefore the best and worst case would be $(n)(n - 1) = n^2 - n$ and therefore a Big-O of $O(n^2)$
 - There is a variant of bubble sort which uses a flag to determine when the list is sorted, and so if the list starts out sorted, it would only need n operations and therefore the best-case Big-O becomes $O(n)$
- Insertion Sort
 - This works by sorting the list one item at a time
 - The list is once again divided into a sorted and unsorted section
 - Start by iterating over the items in the array until the first unsorted item is found
 - Then shift it down a place and check again. If it is sorted now, it has been inserted into the correct place. If not, continue shifting and comparing items until it is in the "relatively" correct place
 - This may not be the correct position, but it is now sorted correctly relative to all the items it has been compared against
 - The insertion process should be repeated until the entire list is iterated over without finding an item out-of-order
 - The best-case is that the list is already sorted, and so n comparisons are made to determine that it is sorted, resulting in a best-case Big-O of $O(n)$
 - The worst-case is that the list is entirely unordered, and so n comparisons need to be made n times and therefore a Big-O of $O(n^2)$

Recursion

- With an iterative algorithm, the code is explicitly repeated using a for or while loop
- With a recursive algorithm, the code repeats implicitly, depending upon how many times the function is called within itself
- All recursive algorithms use a divide-and-conquer strategy to split the problem into smaller problems
- However, there are some significant issues with recursive algorithms:
 - If a recursive function does not have a **stopping** or **base** case, it will infinitely recurse
 - There is also system overhead when calling functions, so a recursive function is often less time efficient than an iterative function
 - There is a limit to how many times you can recurse before you get a stack overflow error
 - They can also end up using an excessive amount of memory due to placing everything on the stack

- The following are reasons recursive algorithms may not be ideal:
 - Some algorithms or data structures are not well suited to be iterated recursively
 - The recursive solution may be significantly harder to understand or program than its iterative counterpart
 - There may be unforeseen space or time consequences to using a recursive algorithm

Lecture - Workshop 5 Async

15:00

12/10/23

Dalin Zhou

Sorting Algorithms

- Merge Sort
 - Recursively divide the list into two equal halves until there is only one element in each group
 - Merge the smaller lists into several larger ones, sorting as you go
 - * Start by checking the first item of each list, move the smaller one to the combined list
 - * Continue comparing the first item from each list until all items have been moved into the combined list
 - Repeat the merging process until you are left with only one sorted list
 - On average, the Big-O is $O(n \log_2 n)$
- Quick Sort
 - Select an item from the array to be the pivot (this can be selected arbitrarily, or randomly)
 - Any elements which are smaller than the pivot are placed to it's left, and all larger elements to it's right
 - The pivot will always be placed in it's correct, final position
 - Then, recursively perform a quicksort on the two halves of the array (everything to the left and everything to the right of the pivot)
 - The best-case Big-O is $O(n \log_2 n)$, worst-case is $O(n^2)$. On average, the Big-O is approximately $O(n \log_2 n)$

Backtracking Algorithms

- A backtracking algorithm is one which attempts to search for a solution by constructing partial solutions, and checking if they're consistent with the requirements and limitations of the problem
- The algorithm takes partial solutions one step at a time, and if said step violates the requirements, it backtracks a step and tries again
- If nothing it can do from there works, it backtracks again until either it can find a suitable solution, or it has tested every option and deems the problem unsolvable
- Backtracking algorithms work especially well for problems which have a large solution space (lots of possible solutions) and very strict requirements, as many possible solutions can be eliminated quickly

Lecture - Workshop 6 Async

15:00

19/10/23

Dalin Zhou

Linked Lists

- A linked list is a collection of items, known as nodes, which have 2 components each
 - Information - The actual data
 - Reference - A pointer or reference to the next node in the list (sometimes called a link)
- This means that it is a type of linear data structure
- Unlike an array, a linked list is a dynamic data structure. This means that the list can increase or decrease in size
- The first element in the linked list is just a pointer to the first node, known as the head
- The last element in the linked list is a node with a null reference, known as the tail
- Advantages over a static data structure:
 - Easy to increase or decrease in size to fit the required number of elements
 - Very efficient, $O(1)$ insertion and deletion operation *once located*
- Disadvantages
 - Does not allow direct access to individual items - you must start at the head and follow the path of references to find the item you want
 - This means an access operation has a Big-O of $O(n)$
 - More memory is needed as compared to a static data structure, as each node needs to store the data as well as a reference to the next node
- There are several types of linked list:
 - Singly linked lists - Each node stores one data value and one reference to another node
 - Singly linked list with dummy node - This adds a dummy node at the start which contains no data, but links to the next node. This is useful to reduce programming errors when deleting items from a linked list
 - Circular singly linked list - The last node's reference is set to the first node in the list, rather than a null reference. This can be useful if the list needs to be traversed several times for one algorithm
 - Doubly linked list - Includes a reference to the previous node. This allows easier reverse traversal at the cost of more memory use. Can also be circular
 - SkipList - This is an extension of a singly linked list, to include randomised forward links which allow skipping part of the list (hence the name). This is especially useful as it allows for more efficient searching, and on average all operations are performed with $O(\log_2 n)$ efficiency

- With a skiplist, you can include as many levels of references, but this comes with the tradeoff of using more storage for each level
- Since there are multiple levels in a skiplist, it's possible that by starting with the highest level, we can exclude a large number of items with only one comparison
- This comes with the tradeoff that if the item we're looking for is only on the lowest level, we have to make multiple comparisons as we move down the levels
- This means that there's a chance of using either more or less comparisons than a linear search, and so on average it could be faster, depending upon how many levels we have
- One method of determining which level to insert an item at is to randomly select it. This way, there is a random distribution of items in each level
- An ideal skiplist has half as many references for each new level that's added (all on the lowest level, half on the next, a quarter on the next, etc)
- There is no guarantee of better performance than a standard linked list, so it is best to weigh up the tradeoffs depending upon your specific use case

Lecture - Workshop 7 Async

19:00

02/11/23

Dalin Zhou

Hierarchical Data Structures

- A collection of elements, where each element has a unique predecessor and multiple successors
- Basic terminology
 - Tree - A set of interconnected nodes without a loop
 - Root node - The first node which all nodes are related to
 - Branch - The link between two nodes
 - Subtree - A tree whos root node is within another tree
 - Leaf - A node with no sucessors
 - Degree of the node - How many successors it has
 - Level of the node - How many nodes are between this node and the root
 - Height or Depth - How many levels are there in the tree

Binary Search Tree

- A tree in which every node has two or fewer successors
- A binary search tree is a binary tree in which the nodes in the left subtree precede the root and the nodes in the right follow the root
- All subtrees are also binary search trees
- Since the nodes on the left are smaller, and nodes on the right larger than the root, we can perform a recursive binary search on a binary search tree
 - Check if the root is the item
 - If the item is smaller, recurse on the left subtree
 - If the item is larger, recurse on the right subtree
 - Once you get to a node with no successors, the item is not in the tree
- To create a binary search tree, you start by inserting the initial item as the root
- To insert an item, you must search through the tree by checking if the item is larger or smaller than the node at each level, until you find any empty spot to place it
- To delete an item, you must first check how many successors it has
 - If the node is a leaf, it can just be removed
 - If the node has only one successor, the successor can replace the node

- If the node has two successors, you take either the right-most item in the left subtree or the left-most item in the right subtree. Since these nodes may have successors, you have to recurse over the subtrees to ensure they are still BSTs
- Since there are two options available, the deletion of a node with two successors usually has two correct answers so the implementation will need to pick between them
- A balanced binary search tree is one in which all leaf nodes are of the same degree
 - A balanced tree is important as it allows half of the items in the tree to be disregarded for each comparison

Traversal of BSTs

- Depth First Traversal
 - Traverses the tree from the root to the most distant child before moving on to the second most distant child, etc
 - Usually implemented using a stack
 - Processes leaf by leaf
 - There are 3 orders in which this can be performed
 - * Pre-Order - Read the node, then traverse the left subtree, then the right
 - * In-Order - Traverse the left subtree, read the node, then traverse the right
 - * Post-Order - Traverse the left subtree, then the right, then read the node
- Breadth First Traversal
 - Traverses the tree horizontally from the root to all of its children, then to their children, etc
 - Usually implemented using a queue
 - Processes level by level
 - Usually outputs each level from left to right

Lecture - Workshop 8 and 9 Async

11:00

12/11/23

Dalin Zhou

Self-Balancing Trees

- A self-balancing tree is one which automatically reduces its height as much as possible
- This improves the searching performance within the tree, and since insertion and deletion also require searching, the entire tree becomes more efficient
- Rotations are performed on the tree, which reduces its height
- A rotation changes the structure of the tree, but without changing the order, and as such it remains a BST
- There are two popular methods for creating a self-balancing tree
 - AVL trees
 - Red-Black trees

Tree Rotation

- A tree rotation is used to change the structure of the tree, without changing the order of the elements
- This is useful for reducing the height of a BST, while keeping it a BST
- A pseudocode implementation would look like this
- ```
temp = root.left
root.left = root.left.right
temp.right = root
root = temp
```
- This would cause a right rotation, and could be used on a left-sided tree. The same operation could be performed on a right-sided tree simply by flipping left and right in the example

## AVL Trees

- The heights of the left and right tree differ by no more than 1
- Left and right subtrees are also AVLs in and of themselves
- Each node of an AVL tree includes an additional variable known as the balance factor
  - The balance factor is the height of the left subtree minus the height of the right subtree
  - A node with a balance factor anywhere from -1 to 1 is considered balanced

- A balance factor of -2 or 2 is considered unbalanced, and any other value should be impossible as the tree should've been rebalanced before it was possible for it to become so unbalanced
- To rebalance the tree, a rotation is performed on either the entire tree, or a subtree depending upon which node is unbalanced

## Creating an AVL tree

- Data items are inserted using the normal BST insertion rules
- The balance factor of each node from the root to the newly inserted node may change, and so need to be recalculated
- If the balance factor of any node is too high, the tree becomes unbalanced
- This means that the tree needs to be rebalanced using a rotation
- The rotation will take place about the unbalanced node closest to the newly inserted node
- To reduce the complexity of rotations, there are only 4 possible rotations, split into two groups
- Single Rotations
  - Left-Left Rotation (LL)
    - \* The unbalanced node and it's left subtree are left-heavy
    - \* The balance factor of the unbalanced node is +2 and it's left subtree has a balance factor of +1
    - \* The tree is rebalanced by completing a single rotation to the right
  - Right-Right Rotation (RR)
    - \* The unbalanced node and it's right subtree are right-heavy
    - \* The balance factor of the unbalanced node is -2 and it's right subtree has a balance factor of -1
    - \* The tree is rebalanced by completing a single rotation to the left
- Double Rotations
  - Left-Right Rotation (LR)
    - \* The unbalanced node is left-heavy and it's left subtree is right-heavy
    - \* The balance factor of the unbalanced node is +2 and it's left subtree has a balance factor of -1
    - \* The tree cannot be rebalanced by a single rotation, so a left rotation around the child followed by a right rotation around the unbalanced node is required
  - Right-Left Rotation (RL)
    - \* The unbalanced node is right-heavy and it's right subtree is left-heavy
    - \* The balance factor of the unbalanced node is -2 and it's left subtree has a balance factor of +1
    - \* The tree cannot be rebalanced by a single rotation, so a right rotation around the child followed by a left rotation around the unbalanced node is required

## Self-Organising Trees

- A Splay Tree is a type of self-organising tree which ensures that recently accessed nodes are always quick to access again
- This is based on the idea that you are likely to need to access the same data again soon
- It keeps the most commonly used data near the top of the tree where it can be accessed quickly



## Splay Trees

- A Splay Tree adjusts itself after every search, insertion or deletion operation
- It has excellent performance in cases where some data is accessed more frequently than others
- Does not have the minimum height available
- Still follows all of the normal rules for a BST

## Creating a Splay Tree

- Data items are inserted using the normal BST insertion rules
- The inserted item is then moved to the root of the tree using a series of rotations, in a process known as splaying
- This means that the most recently inserted item is at the root of the tree, and as such will be the fastest item to access
- Splaying always moves a lower item to the top
- It uses a series of double rotations until the node is either
  - The root of the tree
  - A child of the root node, in which case a single rotation is used
- There are more possible rotations with a Splay tree, but 4 of them are the same as those used in an AVL tree
  - A Zig rotation is a single rotation and is equivalent to an LL or RR rotation
  - A ZigZag rotation is a double rotation, equivalent to an LR or RL rotation
- The additional type of rotation is known as a ZigZig rotation
  - A ZigZig rotation does not reduce the height of the splay tree, but instead essentially reverses the left and right subtree
  - (If all subtrees are on the right, they are re-ordered and placed on the left of the previously lowest leaf node)

## Searching a Splay Tree

- Use the normal searching rules for a BST
- If the item is successfully found, splay the found item into the root of the tree
- If the item is not found, splay the last checked item into the root of the tree
- This makes it faster to locate an existing item
- It also makes it faster to determine that the item is not in the tree

## Deleting from a Splay Tree

- Search for the item as usual
- If the item is located, either
  - Splay the item to be deleted into the root of the tree
  - Remove the item and replace it in the normal way for a BST (rightmost item in left subtree, or rightmost item in left subtree)
- or
  - Remove the item and replace it in the normal way for a BST
  - Splay the parent of the deleted node into the root of the tree
- If the item was not located, splay the last checked item into the root of the tree

## Performance

- No operation is guaranteed to be more efficient than a BST
- The best case Big-O is still  $O(1)$
- The worst case Big-O is still  $O(n)$
- However, over a series of operations, the average Big-O trends towards  $O(\log_2 n)$