University of Portsmouth
BSc (Hons) Computer Science
Second Year

Data Structures and Algorithms (DSALG)

M21270
September 2023 - January 2024
20 Credits

Thomas Boxall
up2108121@myport.ac.uk

Contents

Page 1

Async lecture - Introduction to Data Structures and ADT

2023-09-30

1.1 Data Structures

A *Data Structure* is a way to store and organise data in order to facilitate access and modification. There is no single data structure which is perfect for every application - we need to choose the best for whatever we are creating.

There are two parts to a data structure: a collection of elements, each of which is either a data type of another data structure; and a set of associations or relationships (the structure) involving the collection of elements.

1.1.1 Classification of Data Structures

Data structures can be classified based on their predecessor and successor.

Name	Predecessor	Successor	Examples
Linear	unique	unique	stack, queue
Hierarchical	unique	many	family tree, management structure
Graph	many	many	railway map, social network
Set Structure	no	no	DSALG class

Table 1.1: Classifications of Data Structures

1.1.2 Choosing the right Data Structure

When choosing a data structure, it is important to analyse the problem, determine the basic operations needed and select the most efficient data structure. Choosing the right data structure will make the operations simple & efficient and choosing the wrong data structure will make your operations cumbersome and inefficient.

1.1.3 CRUD

CRUD Operations: Create, Read, Update and Delete are the basic operations which all data structures must be able to do. It is common for a data structure to use a different name to refer to the operation, however.

1.2 Abstract Data Type

An Abstract Data Type (ADT) is a collection of data and associated methods stored as a single module. The data within an ADT cannot be accessed directly, it must be accessed indirectly through its methods. An ADT consists of: the data structure itself; methods to access the data structure; methods to modify the data structure; and internal methods (which are not accessible from outside the ADT).

1.3 Algorithms

An *Algorithm* is any well-defined computational procedure that takes some data, or set of data as input and produces some data or set of data as output. It is the sequence of computational steps which are gone through that transforms the input into the output which is the algorithm. The algorithm must process the data efficiently (both in terms of time and space).

1.3.1 Classifications of Algorithms

There are a number of different classifications of algorithms - four are shown below. Definitions are from *National Institute of Standards and Technology - Dictionary of Algorithms and Data Structures*

1.3.1.1 Brute-Force Algorithm

An algorithm that inefficiently solves a problem, often by trying every one of a wide range of possible solutions. E.g. exhaustive search.

1.3.1.2 Divide & Conquer Algorithm

An algorithm which solves a problem either directly because that instance is easy (typically, this would be because the instance is small) or by dividing it into two or more smaller instances. Each of these smaller instances is recursively solved, and the solutions are combined to produce a solution for the original instance.

1.3.1.3 Backtracking Algorithms

An algorithm that finds a solution by trying one of several choices. If the choice proves incorrect, computation backtracks or restarts at the point of choice and tries another choice. It is often convenient to maintain choice points and alternate choices using recursion.

1.3.1.4 Greedy Algorithms

An algorithm that always takes the best immediate, or local, solution while finding an overall answer. Greedy algorithms find the overall, or globally, optimal solution for some optimisation problems, but may find less-than optimal solutions for some instances of other problems.

1.4 Stack Abstract Data Type

A *stack* is a collection of objects where only the most recently inserted object (`top`) can be removed at any time. A stack is linear and operates in Last In First Out (LIFO).

Stacks must support the following operations.

`push` add an item to the top of the stack

`pop` remove an item from the top of the stack

`peek` examine item at the top of the stack

empty determine if the stack is empty

full determine if the stack is full

An application using a stack will expect the ADT to thrown an exception if: a push operation is requested on a full stack; or a push / pop operation is requested on an empty stack. The stack manages its own storage, therefore an application which uses a stack is not concerned with how the storage used by the stack is managed. In general, an ADT is not interested in the application using the ADT.

Stacks can be implemented using a static array and have a number of uses, including: matching brackets in arithmetic expressions; recursive algorithms; and evaluating arithmetic expressions.

1.5 Queue Abstract Data Type

A Queue is a collection of objects organised such that the first object to be stored in the queue is the first to be removed and so on. It is a linear data structure with elements - inserted at one end (the tail) and removed from the other (the head). A queue operates in First In First Out (FIFO).

Queues must support the following operations:

enqueue add item to the queue's tail

dequeue remove item from the queue's head

full check if queue is full (therefore total number of elements exceeds max capacity)

empty check if queue is empty

first check the first element (the head) in the queue

1.5.1 Implementations of a Queue

There are three different implementations of a queue, all of which can use a static array.

1.5.1.1 Fixed Head

The head of the queue is fixed, this means it will always be in index 0 of the static array. When an element is dequeued, the rest of the elements in the queue must be shuffled along so that the new head is in index 0. This is extremely time inefficient for large queues however quite space efficient as there won't be "dead space" at one end of the queue.

1.5.1.2 Mobile Head

The head of the queue is mobile, this means it can be in any index of the array. When an element is dequeued, the rest of the elements in the queue stay where they are and the head pointer is updated to represent the new head's index. This is more time efficient however not space efficient as you may end up with a lot of "dead space" where the head of the queue used to be.

1.5.1.3 Circular Queue

The queue is circular in a logical, not physical, way. This means the head and tail can be anywhere in the array. If the head is not at the start of the array and the space is needed, the tail will loop around to use the space at the start of the static array.

Page 2

Async lecture - Tools of the Trade I: Efficiency & BigO

2023-09-30

BigO Notation is used to define the efficiency of an algorithm quantitatively. This is a very helpful tool to have when designing algorithms as it allows us to compare multiple algorithms to and understand which is the best one to use.

Name	Notation	Description
Constant	$O(1)$	Algorithm always executes in the same amount of time regardless of the size of the dataset.
Logarithmic	$O(\log_n)$	Algorithm which halves the dataset with each pass, efficient with large datasets, increases execution time at a slower rate than that at which the dataset size increases.
Linear	$O(n)$	Algorithm whose performance declines as the data set grows, reduces efficiency with increasingly large dataset.
Loglinear	$O(N \log_n)$	Algorithm that divides a dataset but can be solved using concurrency on independent divided lists.
Polynomial	$O(N^2)$	Algorithm whose performance is proportional to the size of the dataset, efficiency significantly reduces with increasingly large datasets.
Exponential	$O(2^n)$	Algorithm that doubles with each addition to the dataset in each pass, very inefficient.

Table 2.1: BigO Notation complexity values, listed best to worse

BigO doesn't look at the exact number of operations, it looks at when the size of a problem approximates infinity therefore two very similar algorithms which are slightly different may have the same Big O despite one having double the number of operations.

2.1 Calculating The BigO Value

1. Determine the basic operations (including: assignment, multiplication, addition, subtraction, division, etc)
2. Count how many basic operations there are in the algorithm (some basic algebraic addition required here!)
3. Convert the total number of operations to BigO (done by: ignoring the less dominant terms; and ignoring the constant coefficient - i.e. $2n + 1$ becomes n).

Page 3

Async lecture - Iterative Algorithms and Efficiency

2023-10-05

3.1 Searching Algorithms

When searching a dataset to find the item you are looking for, the aim of an efficient searching algorithm is to exclude elements to reduce the searching space after each comparison.

3.1.1 Sorted vs Unsorted Data

Sorted data, when the data is in either ascending or descending order, makes it easier to locate the item you are searching for. Depending on the algorithm, when using sorted data, we can exclude more elements from the list to search.

Unsorted data, when the data is in a random order, makes it harder to locate the item you are searching for. This is because you have to examine (in the worst case) every element in the array you are searching to realise that the item you are searching for is not in the array.

3.1.2 Sequential (Linear) Search

1. Start at the beginning
2. Check every element of the array in turn until item located or the end of the array reached (therefore item not located)

This search excludes one data item at a time, which is not great. It works on sorted and unsorted data.

The best case BigO is $O(1)$. This is the case where the first item we examine is the item we want to find.

The worst case BigO is $O(n)$ when the searched value is the last element in the array or not in the array, n comparisons are required.

The average case BigO is $O(\frac{n}{2})$ because if the data is distributed randomly, each element has an equal chance to be the one searched for.

Overall, the BigO for a sequential search is $O(n)$.

3.1.3 Binary Search

1. Check the middle element of the array

2. If not found, work out which half of the array the item could be located in and exclude the half which it won't be included in
3. Repeat by searching the half array which may contain the required item by examining the middle element and eliminating half the array
4. Process repeated on the halved array until either find the item or determine that it doesn't exist.

A binary search only works on a sorted array.

The best case BigO is $O(1)$. This is achieved when the value we are searching for is the middle element, hence it is found on the first comparison.

The worst case BigO is $O(\log_2 n)$, which is achieved when the value we are searching for is not found in the array. We therefore need $1 + \log_2 n$ comparisons which gets converted into BigO.

The average case is $O(\log_2 n)$, if the data is distributed randomly - each element has an equal chance to be the one searched for.

The binary search algorithm has an additional cost - it requires a sorted sequence of data items, which incurs cost if the data is not already sorted.

3.2 Sorting Algorithms

A sorting algorithm aims to make comparisons between data items and swap them according to the desired order of the items. All sorting algorithms will involve two basic steps:

1. Compare items
2. Swap elements

3.2.1 Selection Sort

A selection sort works by sorting the array one item at a time. It divides the list into two parts. The sorted section (on the left) is initially empty and the unsorted section (on the right) is initially the complete unsorted list. The smallest (or largest, depending on if we want ascending or descending) is selected from the unsorted array and swapped with the left-most element from the unsorted section. The item is now in the correct final position within the ascending / descending order. The sorted part has increased in size by 1. The process is iterated on the unsorted part of the array until all items have been considered, the array will now be sorted.

The best case BigO is $O(n^2)$, when the array is already sorted hence each element only has to be compared to its direct neighbours to establish this.

The worst case is $O(n^2)$.

Selection sorts are unsuitable for large data sets.

3.2.2 Bubble Sort

A bubble sort works by repeatedly passing through the list, swapping adjacent items if they are in the wrong order. If sorting into ascending (or descending) order, the largest (or smallest) item will be bubbled to the end of the array, hence the name. The process is iterated on the whole list until all items have been considered, therefore the whole list will now be sorted.

Without using a flag, the best case BigO is $O(n^2)$. The worst case BigO is also $O(n^2)$.

When a flag (variable which denotes when the array is sorted, enabling sorting to stop as soon as array detected as sorted) is used, the best case BigO is $O(n)$ and the worst case BigO is $O(n^2)$.

With or without the flag, the algorithm is very slow for large datasets.

3.2.3 Insertion Sort

An insertion sort works by sorting the list one item at a time. THe list is split into a sorted and non-sorted part. With each iteration, the next element waiting to be sorted (in the unsorted part) is inserted in its correct location within the sorted list. The process iteration on whole list until all items have been considered; when completed, the list will now be sorted.

The best case BigO is $O(n)$, when the data is already sorted. The worst case BigO is $O(n^2)$.

This algorithm is efficient for sorting nearly-sorted lists.

Page 4

Async lecture - Tools of the Trade II: Recursion

2023-10-05

4.1 Introduction

Computers are better at repeating themselves than humans are. Humans get bored, computers don't.

Iteration is explicit repetition of code. We often use for and while loops to repeat sections of code, both of which use control variables to control the repetitions.

Recursion is an alternative method of repeating code, where the code is repeated implicitly. Recursion occurs when a function or method calls itself.

4.2 Recursion

Recursion is a technique whereby a problem is expressed as sub-problems in a similar / same form to the original problem but smaller in scope. The sub-problems only differ in input or size. Shown below is an example of a recursive algorithm, that never ends.

```
recursive_print_example(i){
    print(i)
    recursive_print_example(i+1)
}
```

Recursion is applied to problems where: a solution is easy to specify for certain conditions (the stopping case, which is required or the program will continue indefinitely); and rules for proceeding to a new state which is either a stopping case or eventually leads to a stopping case (recursive steps) are identified.

Shown below is an actual recursive algorithm with its output:

```
cheers(int times){
    print("hip)
    if (times > 0){
        cheers(times - 1)
    }
    print("hooray")
}
//outputs: hip hip hip hooray hooray hooray
```

4.3 Pitfalls of Recursion

Recursion must always be used with care and understanding. It is possible to write compact and elegant recursive programs that fail spectacularly at runtime. The main pitfalls of recursive algorithms are as follows:

- Forgetting the stopping case
- Failure to reach a stopping case
- Excessive use of space
- Excessive repeated computations

There are some times, when you really shouldn't use recursions:

- When the algorithm / data structure is not naturally suited to recursion
- When the recursive solution is not shorter and understandable than the linear solution
- When the recursive solution doesn't run in acceptable time limits and / or space limits
- When the intermediate states of the algorithm don't pass the same data to / from them.

Recursion to be continued next week.

Page 5

Async lecture - Recursive Algorithms & Efficiency

2023-10-11

The BigO value of a recursive algorithm is generally way worse than that of a linear algorithm which is performing the same function. Both the time and space efficiency is worse - as it will involve more operations and take up considerably more memory due to the number of function calls made.

5.1 Merge Sort

The *Merge Sort* is an example of a divide and conquer algorithm. This means it works by dividing the bigger problem into smaller problems which can then be solved independently, making it easier to program. The smaller problem's solutions will then be combined back together to make the overall solution. The outline of how it works is shown below:

1. Divide the array of data to be sorted into two equal halves until we have one element in each group. Note: if there is only one element in the list, it must be sorted (this is the stopping case of the recursive algorithm).
2. Merge the smaller arrays in to one large sorted array, merging one array with another array at a time. Repeat this until the entire array is sorted.

5.2 QuickSort

This is another example of a divide and conquer algorithm. The outline of how it works is shown below.

1. Pick a pivot value (this will be an item in the array)
2. Put all values less than the pivot into one array and all those bigger than the pivot into another. Recursively repeat this until you end up with just one element in the arrays at the end (these will be the biggest and smallest elements in the array).
3. Merge back together as the pivot values will now be in the correct order.

The best case BigO is $O(n \log_2 n)$ which is achieved when the pivot splits into two equally sized parts each time.

The worst case BigO is $O(n^2)$ where the pivot is split into very unequal sizes.

The average case BigO is $O(n \log_2 n)$.

5.3 Backtracking

Backtracking is a type of algorithm which continually searches for a solution by constructing partial solutions, using the correct parts of the solution to be the starting point for the next solution attempt. It is a trial and error algorithm.

When the algorithm finds an incorrect partial solution, it returns "up" a layer to where it had a closer match to the correct solution and tries another branch. The algorithm is complete when either all possibilities have been exhausted or the item has been found.

Page 6

Async lecture - Linked Lists

2023-10-19

6.1 Linear Data Structures: a recap

A *linear data structure* is a collection of nodes (elements) where each node has a unique predecessor and unique successor. We saw in Workshop 1 that there are also other types of data structure where there can be many predecessors and many successors.

6.1.1 Example: Static Array

A static array is a static sized structure, which is different from a dynamic sized structure for example the ArrayList in Java. The advantage to using a Static Array is that there is faster access to each node providing as the node is known, with a BigO of $O(1)$ to visit `arr[i]`. The disadvantages of using a static array is that its fixed size means it cannot be easily extended (as there has not been enough space allocated in memory) and it cannot be reduced (as there would then be wasted space in memory); static arrays can also be expensive to maintain in terms of time, especially for a group of sorted elements.

CRUD Operation	Data type	Efficiency
Searching	Unsorted Data (linear search)	$O(n)$
	Sorted Data (binary search)	$O(\log_2 n)$
Insertion	Unsorted data	$O(1)$
	Sorted data	$O(n)$
Deletion	Unsorted data	$O(1)$
	Sorted data	$O(n)$

Table 6.1: Efficiency of CRUD operations for a static array

The increased BigO value for the sorted array is due to the fact that the current elements have to be shifted within the array to maintain the sorted status, which obviously increases the number of operations which need to be performed.

6.2 Linked List

A linked list is a collection of objects called *nodes*. Every node has 2 components - information to be stored (the data) and the reference to the next node in the list (the link). A linked list is s dynamic data structure, which means the number of nodes in the list is not fixed and it can grow or shrink on demand.

6.2.1 Singly Linked List

This is the most common type of linked list whereby each node is only linked to one predecessor and one successor - exceptions being the head which is only linked to one successor and the tail which is only linked to the predecessor.

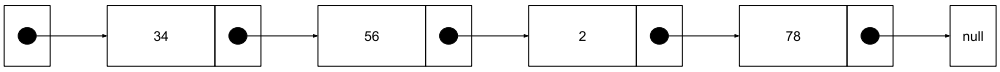


Figure 6.1: Singly Linked List

Advantages of using the Singly Linked List is that it is easily extendable or reduced to fit the data needing to be stored in it. Once the item is located - it has efficiency $O(1)$ to insert or delete the item.

The significant drawback of a SLL is that it does not allow direct access to individual items; to find an individual item - you have to start at the head and follow references until you find the item you're looking for, which has the efficiency $O(n)$. Another disadvantage is that it uses more memory compared to a static array as the reference to the subsequent node also has to be stored.

6.3 Other Types of Linked Lists

Alongside the SLL, there are other types of Linked Lists which we may come across.

6.3.1 Singly Linked List with Dummy Nodes

A significant issue with the SLL is that the head item has to point to something. The idea of using a dummy node is that this is the first node in the list which then means the head element can always point to something, even if it is an empty node which makes the code more efficient.

6.3.2 Circular Linked List

In this linked list, the tail node points to the first node after the head (which is the first node to contain data). This is useful for a list which undergoes a large number of traversals.

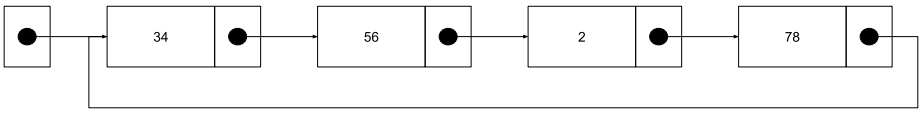


Figure 6.2: Circular Singly Linked List

6.3.3 Doubly Linked List

In this linked list, each node is linked to both its predecessor and successor in a way such that the list can be traversed in either direction.

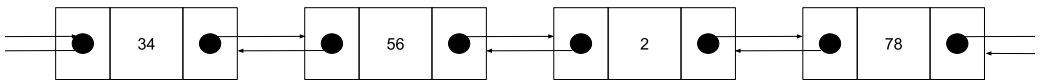


Figure 6.3: Doubly Linked List

6.3.4 SkipList

A SkipList is an extension of an ordered SLL with a number of additional forward links added in a randomised way. The data in the list does have to be ordered, however, as during searching the elements in the array - bits of the list can be skipped to reduce the searching space quickly (like in a binary search). This leads to having an efficiency on all CRUD operations of $O(\log_2 n)$, which is logarithmic random time.

The ideal SkipList would have half of the nodes having one reference to a subsequent node; $\frac{1}{4}$ would have 2 references to subsequent nodes; $\frac{1}{8}$ will have 3 references to subsequent nodes; and so on. The distance between nodes on on each 'level' would be equal.

SkipLists aren't guaranteed to give good performance. Their performance depends on how many nodes are skipped at each level. For searching, the best case will be $O(1)$ and the worst case will be $O(n)$.

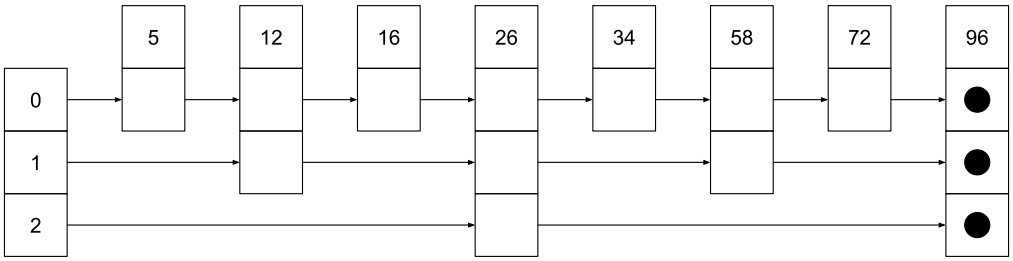


Figure 6.4: SkipList

Page 7

Async lecture - Herarchical Data Structures

2023-11-02

7.1 Why Another Data Structure?

The highest priority of any programmer when thinking about handling data is to ensure that the data is structured in a way to ensure that all the required operations to be performed on the data are efficient.

We have already learnt about a number of data structures: Static arrays which are fixed size; linked list which are dynamic sized; and SkipList which is also dynamic. The Static Array and SkipLists can be searched using a binary search permitting they are sorted however the Linked List can only be searched in a sequential fashion due to it's nature.

7.2 Hierarchical Data Structures

A hierarchical data structure (commonly a tree) consists of a collection of nodes where each node has a unique predecessor and many successors.

There is some key terminology to be aware of when discussing Hierarchical Data Structures, this is outlined in the table below and makes use of the following example of a tree.

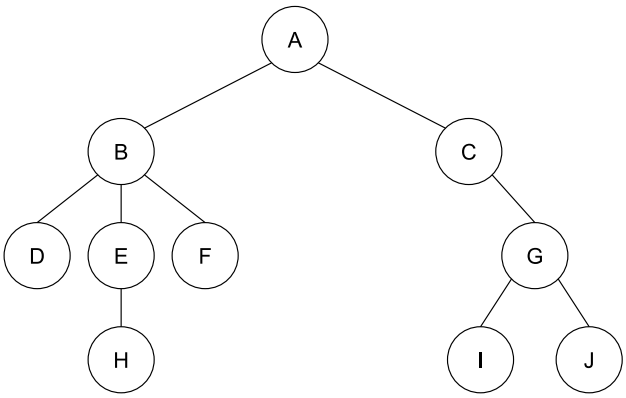


Figure 7.1: Example of a Hierarchical Data Structure

Terminology	Definition	Example
Tree	A set of interconnected nodes with no closed circuits or loops	
Subtree	A tree rooted an internal node of the tree	Tree rooted at node B is a subtree of the given tree
Leaf (terminal node)	Any node of the tree which has no subtrees	The nodes containing D, H, F, I and J
Root node	The first node of the tree	The root node of the above tree is A
Branch	A link between two nodes within the tree	
Degree of a node	Number of subtrees of that node	The degree of the node containing A is 2; and B is 3
Level of a node	Number of branches on the path from the root to the node	The level of nodes B and C is 1; H, I and J is 3; A is 0
Height / Depth of a tree	Number of nodes on the longest path from the root node to a leaf node	The above tree has height 4; height of a tree containing one node is 1.

Table 7.1: Efficiency of CRUD operations for a static array

7.3 Binary Trees and Binary Search Trees

Binary Trees (BT) are a special case of tree where every node is of degree two or less. A Binary Search Tree (BST) is a special case of Binary tree where: the keys in the left subtree of the root precede the key in the root; the key in the root node precedes the keys in the right subtree; and the left and right subtrees of the root are also BSTs.

7.3.1 Searching

The algorithm to search a BST is based on the normal Binary Search algorithm (that we've used for a static array).

The first item to be evaluated is the root node, if this is the required item the we can stop our search. If the required item comes before the data item in the root node then we search the left subtree; if the required item comes after the data item stored in the root node then we search the right subtree. This process continues until either we find the required item or we reach a leaf node (hence the item is not in the BST).

7.3.2 Constructing a Binary Search Tree

When constructing a BST from scratch, you take the first data item you've been given and use that as the root node. From there add each data item as a child node, ensuring to maintain order.

7.3.3 Special Names for cases of Binary Trees

A binary tree is full if every node other than the leaf nodes has two children.

A binary tree is complete if all levels, except possibly the last level, are completely full and the last level has its node on the left side.

A balanced tree is one in which all path from the root node to the leaf nodes are of the same length.

7.3.4 Deletion of a node from a BST

In the case that the node we want to delete is a leaf node, we simply delete it.

In the case that the node has a single subtree then we simply replace the node to be deleted with the root node of the subtree.

In the case that the node to be deleted has two subtrees, we do something. Dalin hasn't actually told us what yet as that would be far too simple.

7.3.5 Traversal of Binary Trees

To traverse a data structure requires us to be able to access each node of the data structure once and only once in a predetermined sequence. Traversal of a tree is considerably more complex than that of a Static Array or even a Singly Linked List. There are two distinct approaches which can be taken to traverse a binary tree: DFT and BFT. Within each approach, there are sub-approaches - each of these will be explored below.

All examples shown below will make use of the following tree

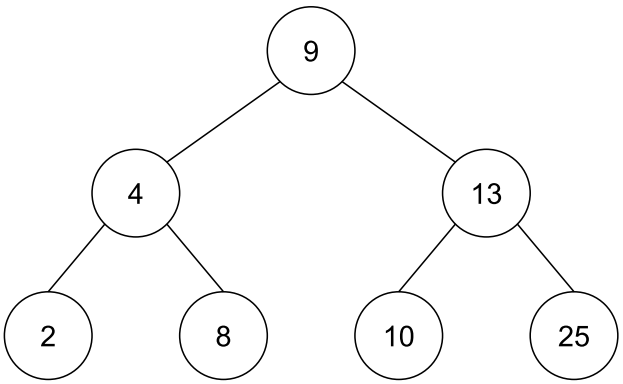


Figure 7.2: Binary Search Tree used in traversal examples

7.3.5.1 Depth First Traversal

Depth First Travel (DFT) proceeds along a path from the root to the most distant descendent of the first child passed through before processing the second child. A stack is used to implement this.

The Depth First Traversal operates on the basis that we need to do three things: Visit the node (V), traverse the Left subtree (L), and traverse the Right subtree (R). There are three orders we do this in: VLR (PreOrder Traversal), LVR (InOrder Traversal) and LRV (PostOrder Traversal).

PreOrder Traversal In PreOrder Traversal, the binary tree is traversed as follows:

1. Visit the root node
2. PreOrder traversal of the left subtree
3. PreOrder traversal of the right subtree

Using the above example tree, a PreOrder Traversal would result in the following route: 9, 4, 2, 8, 13, 10, 25

InOrder Traversal In InOrder Traversal, the binary tree is traversed as follows:

1. InOrder Traversal of the left subtree
2. Visit the root node
3. InOrder traversal of the right subtree

Using the above example tree, an InOrder Traversal would result in the following route: 2, 4, 8, 9, 10, 13, 25

PostOrder Traversal In PostOrder Traversal, the binary tree is traversed as follows:

1. PostOrder Traversal of the left subtree
2. PostOrder Traversal of the right subtree
3. Visit the root node

Using the above example tree, a PostOrder Traversal would result in the following route: 2, 8, 4, 10, 25, 13, 9

7.3.5.2 Breadth First Traversal

Breadth First Traversal (BFT) proceeds horizontally from the root to all of its children then to its children's children and so on until all nodes have been processed. To implement this traversal, a queue is used.

Using the above example tree, a Breadth First Traversal would result in the following route: 9, 4, 13, 2, 8, 10, 25.

Page 8

Async lecture - Self Balancing Trees

2023-11-09

8.1 Complexity of Binary Search Trees

When discussing the ideal case in a binary search tree - the tree will be balanced, or nearly balanced, as this minimises the height of the tree. This is important as the height f the tree relates to the number of comparisons we have to make whereby we make one comparison on each level. The worst case refers to the situation where there is only one node per level.

Operations	**Ideal Case**	**Worst Case**
Searching	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$

Table 8.1: Complexity of a Binary Search Tree

8.2 Self Balancing Tree

A *elf balancing tree* automatically keeps its height as small as possible at all times. It does this by performing "rotations" which reduce the height. These rotations change the structure of the tree without changing the order of elements therefore it remains a binary search tree.

8.2.1 Tree Rotation

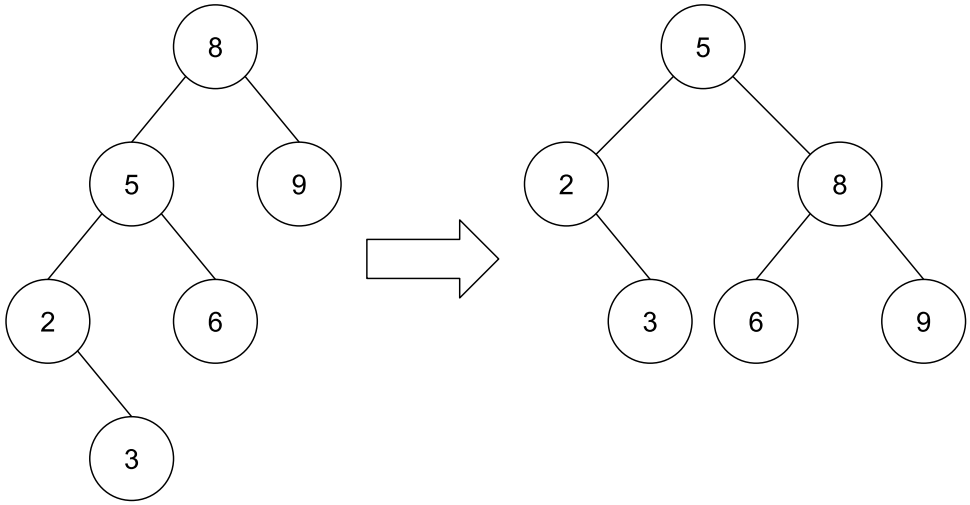


Figure 8.1: Example of a rotation of a tree

Tree rotation is used to change the structure of the binary search tree without chainging the order of the elements, thus after a rotation the tree is still a valid BST.

There are two popular approaches to implementing a self balancing tree - AVL Tree and Red-Black Tree (the later is not covered in the scope of this module).

8.3 AVL Trees

The *AVL Tree* is a type of Binary Search Tree which is named after it's inventors (Adelson, Velskii, and Landis). A BST is a valid AVL tree where:

1. the heights of the Left Subtree (LST) and Right Subtree (RST) of the root node differ by at most 1; and
2. LST and RST are AVL trees (this means we define it recursively)

8.3.1 Balance Factors

Each node of an AVL Tree contains and additional variable called the balance factor. The balance fator of a node can be calculated as follows

$$balanceFactor = height_{LST} - height_{RST}$$

We can use the balance factor to determine if a node is *balanced* or *unbalanced*. If the balanced factor is -1, 0 or $+1$ then the node is balanced. Whereas if the node has a balance factor of -2 or $+2$ then the node is unbalanced; this will also require rotation to be carried out so that the tree can become balanced.

8.3.2 Creating an AVL Tree

Much the same as a BST or a BT, an AVL tree is created as the data items are inserted as nodes. Initially the tree is empty, this means it satisfies the rules for an AVL tree. As data items are inserted, the balance factor of each node will change (nodes with balance factor ± 1 may be changed to ± 2). Where the balance factor is ± 2 then the tree is unbalanced and as such a rotation must take place. Rotation will take place around the unbalanced node which is closest to the most recently inserted node.

8.3.2.1 Rotations of an AVL Tree

There are only four possible rotations which can take place within an AVL tree, two single and two double.

Single Rotations:

- Left-Left Rotation (LL)
- Right-Right Rotation (RR)

Double Rotations:

- Left-Right Rotation (LR)
- Right-Left Rotation (RL)

These rotations take place around the unbalanced node with a balance factor of ± 2.

The types of rotations are demonstrated below through use of an example. In the diagrams, the left hand tree shows the pre-rotation and the right hand tree shows the post-rotation layout.

8.3.2.2 Left-Left Rotation

The unbalanced node and the left subtree of the unbalanced node are both left heavy. The balance of the unbalanced node is +2 and the balance of the left child of the unbalanced node is +1. The tree is rebalanced by completing a single rotation to the right around the unbalanced node.

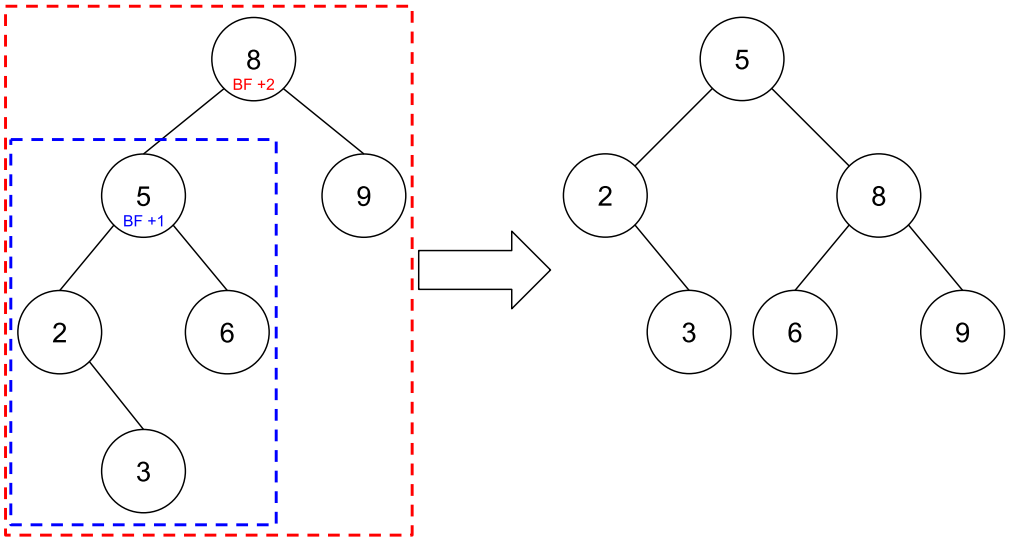


Figure 8.2: AVL tree rotation (Left-Left)

In the above example, the node containing 3 has just been inserted into the AVL tree - causing the node containing 8 to be unbalanced, therefore we rotate to the right about 8.

The pseudocode for the left-left rotation is as follows:

```
temp = root.left
root.left = root.left.right
temp.right = root
root = temp
```

8.3.2.3 Right-Right Rotation

The unbalanced node and the right subtree of the unbalanced node are both right-heavy. The balance of the unbalanced node is -2 and the balance of the right child of the unbalanced node is -1. The tree is rebalanced by completing a single rotation to the left around the unbalanced node.

The Right-Right rotation is the mirror operation to the Left-Left rotation.

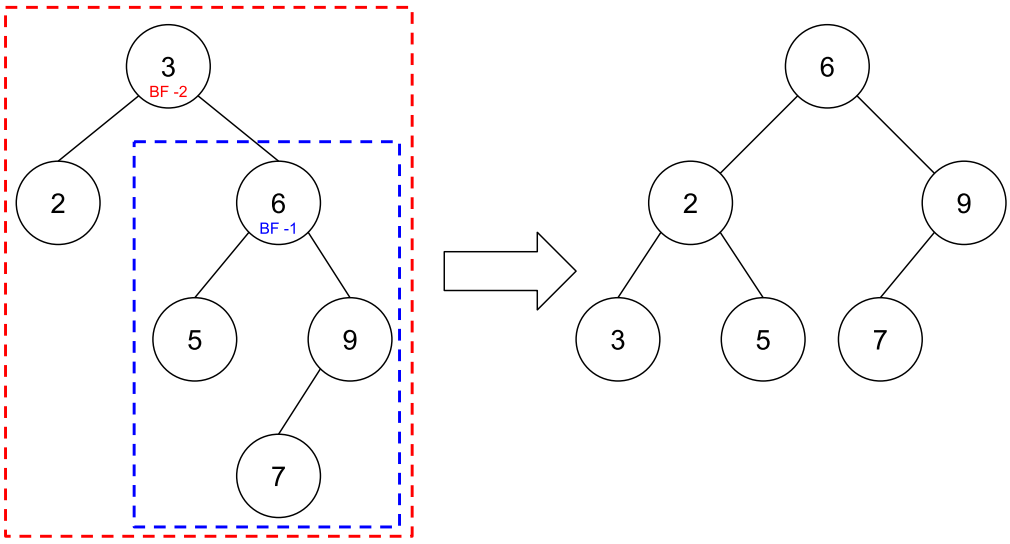


Figure 8.3: AVL tree rotation (Right-Right)

In the above example, the node containing 7 has just been inserted into the AVL tree causing the node containing 3 to become unbalanced. Therefore we rotate to the left about 3.

The pseudocode for the right-right rotation is as follows:

```
temp = root.right
root.right = temp.left
temp.left = root
root = temp
```

8.3.3 Right-Left Rotation

The unbalanced node is right heavy and the right subtree of the unbalanced node is left heavy. The balance of the unbalanced node is -2 and the balance of the right child of the unbalanced nodes is $+1$. This tree cannot be rebalanced by a single rotation - two rotations are required: a right rotation around the child followed by a left rotation around the unbalanced node.

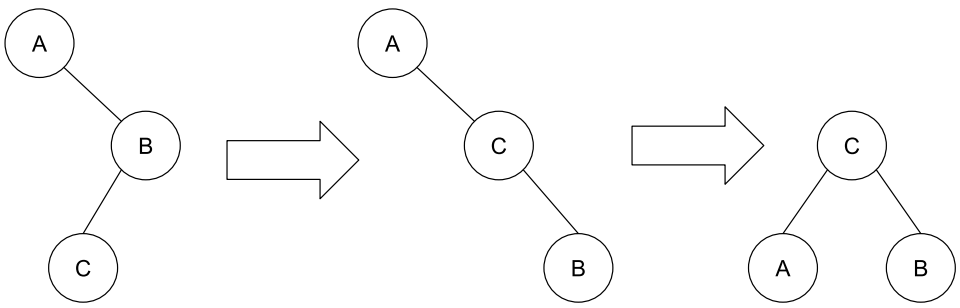


Figure 8.4: AVL tree rotation (Right-Left)

The first rotation rotates node C right around node B, this results in the middle diagram. The second rotation rotates node A around node C, this results in the final state of the tree.

The pseudocode for the right-left rotation is shown below:

```
// setup
t1 = root.right
t2 = t1.left
// rotation 1: right
t1.left = t2.right
t2.right = t1
// rotation 2: left
root.right = t2.left
t2.left = root
root = t2
```

8.3.4 Left-Right Rotation

The unbalanced node is left-heavy and the left subtree of the unbalanced node is right-heavy. The balance factor of the unbalanced node is +2 and the balance factor of the left child of the unbalanced node is -1. This tree cannot be rebalanced by a single rotation - two rotations are required: a left around the child, followed by a right rotation around the unbalanced node.

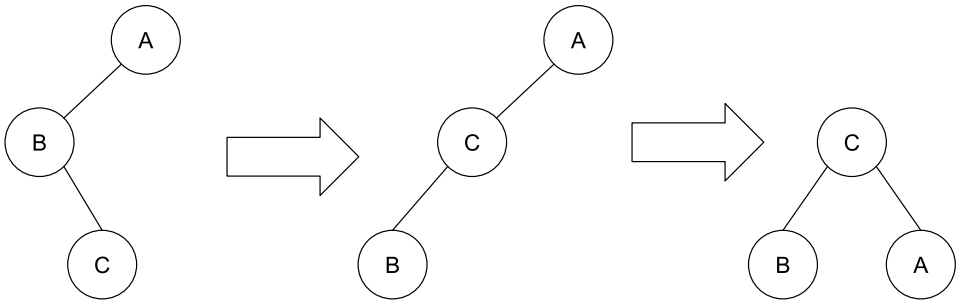


Figure 8.5: AVL tree rotation (Left-Right)

The first rotation rotates node C left around node B, resulting in the middle diagram. The second rotation rotates node A right around node C, resulting in the right diagram.

Page 9

Async Lecture - Self-Organising Trees

2023-11-09

Another type of tree is a *splay tree*. These are self-organising binary trees which organise themself such that recently accessed elements are always quick to access again.

9.1 Basic Operations

All basic operations (insertion, search, removal) are performed in $O(log_2 n)$ time and they all include an extra step called *splaying*.

Searching:

1. Locate node
2. Splay the node to the root

Insertion:

1. Insert node
2. Splay inserted node to the root

Deletion

1. Locate the node to be deleted
2. Replace node to be deleted with its in-order predecessor or its in-order successor (same process as deletion in a BST)
3. Splay the parent of the removed node to the root

9.2 Splaying

Splaying is the movement of a node to the root, which is achieved through a series of tree rotations which are similar to those used in AVL trees. Splaying works from the bottom to the top (root) of the tree, which is often called bottom-up splaying. Splaying involves a series of double rotations, until the accessed node reaches either the root or the child of the root when a single rotation is performed.

No single access operation on the Splay tree is guaranteed to be efficient, this is by design. The worst case will be $O(n)$, and the average across a series of operations tends to be $O(log_2 n)$.

9.3 Rotations

9.3.1 Single Rotations

Single rotations are used when the accessed node `S` is the child of the root node `P`. The effect of this rotation is to move the node up one level in the tree.

9.3.1.1 Zig Rotation

There are two variations of the *Zig* rotation.

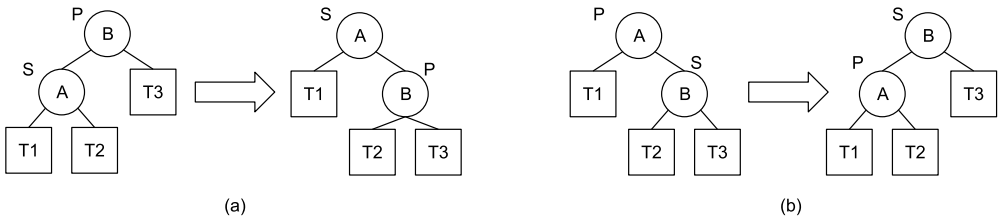


Figure 9.1: Splay tree rotations (Zig)

Rotation *(a)* is the equivalent of the LL rotation used with AVL trees; rotation *(b)* is the equivalent of the RR rotation used with AVL trees.

9.3.2 Double Rotations

Double rotations involve the accessed node `S`, the parent `P` and the grandparent `G`. The effect of these rotations is to move the node up two levels in the tree.

9.3.2.1 ZigZag Rotation

There are two variations of the *zigzag* rotation.

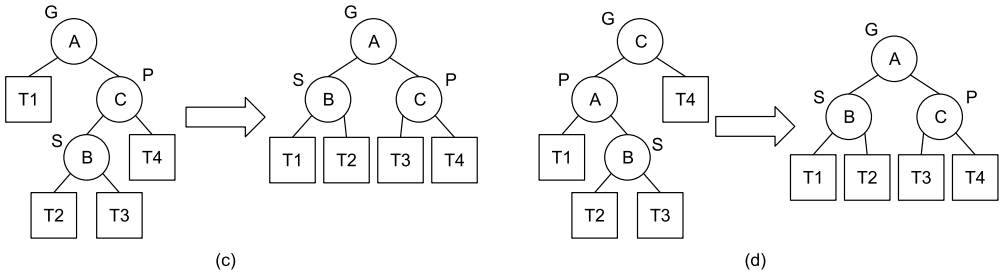


Figure 9.2: Splay tree rotations (ZigZag)

Rotation *(c)* is the equivalent of the RL rotation used with AVL trees; rotation *(d)* is the equivalent of LR rotation used with AVL trees. The ZigZag rotations tend to make the trees more balanced, and they will reduce the height of the tree by 1.

9.3.2.2 ZigZig Rotation

There are two variants of the *ZigZig* rotation.

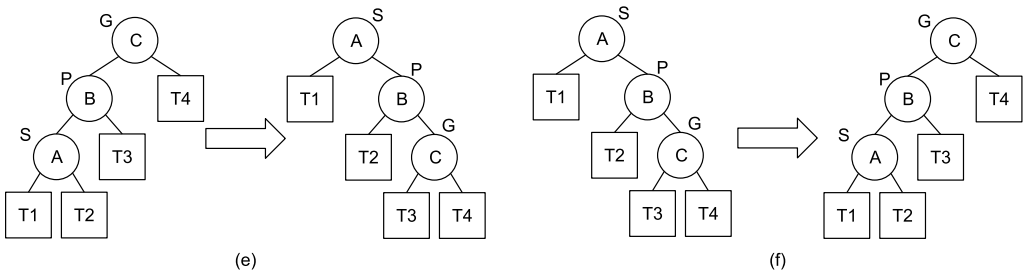


Figure 9.3: Splay tree rotations (ZigZig)

Pseudocode to the effect of rotation *(e)* is shown below:

```
G.left = P.right
P.right = G

P.left = S.right
S.right = P
```

Pseudocode to the effect of rotation *(f)* is shown below:

```
G.right = P.left
P.left = G

P.right = S.left
S.left = P
```

Page 10

Async Lecture - Multi-Branch Trees (2-3 Trees)

2023-11-17

10.1 Introduction

A *multi-way search tree* is a tree where:

- Each node contains one or two data items
- Every internal node has either two children or three children
- All leaves are on the same level therefore a 2-3 tree is always balanced

10.2 Nodes

There are two types of node, a 2-node and 3-node. The 2-nodes contain one item of data, and have either 2 children or no children; they are equivalent to nodes in a binary tree. A 3-node contains two items of data, and has either 3 children or no children.

The order of items behaves much the same as it does for a Binary Search Tree, where the values in the Left-Subtree must be less than the first item in the the node. The right and centre subtrees rules differ depending on what is present. If there is a center subtree, the values of all descendants in the center subtree are less than the value of the second data item and values of all the descendants in the right subtree are greater than the value of the second data item. If there is not a centre subtree (we only have a left and right therefore it is a 2-node), the values in the right subtree are greater than the value of the first data item.

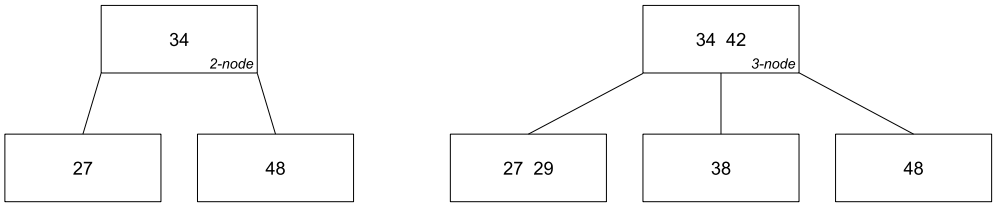


Figure 10.1: The 2-Node and 3-Node

10.3 Algorithms

10.3.1 Searching

Searching a 2-3 tree is similar to searching a BST.

10.3.2 Insertion

Insertion of a new value into a 2-3 tree is similar to that of inserting a new value into a BST as the new value gets inserted into a leaf node; however, unlike a BST - a new child node is not created.

Insertion is carried out by locating the leaf node which should contain the new value: then if the leaf node contains one data item - the new item is inserted into this leaf node and the insertion is completed; however, if the leaf node contains two data items - space has to be created, which is achieved by splitting nodes.

10.3.3 Splitting Nodes

The two data items in the node to be split and the data item to be inserted are treated the same.

1. Node is split into two nodes, `L1` and `L2`
2. The smallest of the 3 data items is placed into `L1`
3. The largest of the 3 data items is placed into `L2`
4. The remaining data item (the middle one) is promoted to the parent node

If the parent node is a 2-node then the promoted node is inserted into the parent node along with references to the two child nodes (`L1` & `L2`) - this turns the parent node into a 3-node. However if the parent node is a 3-node, then the splitting and promoting process is repeated until a place for the promoted node is located; which may result in the creation of a new root node.

10.3.4 Deleting Nodes

Deleting a node in a 2-3 tree is similar to the process of deleting a node in a BST. If the item to be deleted is in a leaf node (either a 3-node or 2-node), then it's removed which will either leave a 2-node leaf or a hole. If the item to be deleted is in an internal node (either a 2-node or 3-node) then it's replaced by its InOrder successor or Predecessor which will be in a leaf node (as is usual for a BST deletion), this leaves either a 2-node or a hole. In the event we have a 2-node leaf, this is fine and we can carry on; however if we have a hole in the tree - this needs to be removed.

To remove the hole, we traverse the 203 tree upwards towards the root. The hole is propagated upwards through the tree until it can be eliminated; which is done by either being absorbed into the tree or being propagated to the root of the tree where it can be removed, which reduces the height of the tree by one (the only time when the height of the tree is reduced).

10.4 Comparison of a BST and a 2-3 tree

Property	**BST**	**2-3 Tree**
Nodes	2-node only (containing 1 element and at most 2 children)	2-node and 3-node (containing 1 or 2 elements and at most 2 or 3 children)
Height	$\lceil log_2(n+1) \rceil$ and n	between $\lceil log3(n + 1)/2 \rceil$ and $\lceil log2(n+1)/2 \rceil$
Constraint	LST precedes the root and root precedes RST	Similar to BST; all internal nodes must have 2 or 3 children; leaf nodes are at the same level.
Search	Follow the path (either LST or RST) until the element is located or leaf node reached.	Similar to BST but with more potential paths (LST, CST, RST).
Insertion	Knowing the location, element is inserted as a leaf node	Knowing the location, element is inserted as a leaf node; then the node is either kept or split (middle element promoted)
Deletion	The node containing the element is deleted after being located; the hole is replaced by the successor or predecessor.	The element is deleted after being located; the node remains with 1 element or becomes a hole (which is replaced by its successor / predecessor then propagated up the tree and absorbed).

Table 10.1: Classifications of Data Structures

Page 11

Async lecture - Multi-Branch Trees (B & B+ Trees)

2023-11-17

11.1 B Trees

A B Tree of order m is a *self-balancing multi-way search tree* or order m which has the following properties:

- Root node has either no children or between 2 and m children;
- Internal nodes have between $\lceil \frac{m}{2} \rceil$ and m children;
- All leaves are on the same level

Property	2-3 Tree	B Tree
Nodes	2-node and 3-node (containing 1 or 2 elements and at most 2 or 3 children)	Can have different kinds of nodes from $\lceil \frac{m}{2} \rceil - node$ to $m - node$
Number of Data Items	For a tree of height h: between $2^h - 1$ and $3^h - 1$	For a tree of height h: between $(\lceil \frac{m}{2} \rceil) - 1$ and $m^h - 1$
Constraint	All leaves are on the same level	All leaves are on the same level

Table 11.1: Comparison of the 2-3 Tree and B Tree

Each node in a B Tree represents a block (or page) of secondary storage. Accessing a node means accessing the secondary storage, which is expensive when compared to accessing a node in main memory - therefore the fewer nodes created, the better.

11.1.1 Algorithms

11.1.1.1 Searching

Searching a B Tree starts at the root node where you perform a Binary Search on the keys. If the data is found, then the task is completed. If it is not found, follow the required branch to the next node and repeat the process. If the node reached is a leaf node and the data item is not found then the search is unsuccessful.

11.1.1.2 Insertion

This algorithm is based on the insertion for 2-3 trees. The tree is searched to locate where to insert the new data item (in a leaf node). If the node isn't full then the data item is inserted into the node and the insertion is complete. However, if the node is full, then the node is split into two and the middle key promoted upwards into the parent node (which if full, also gets split and the middle promoted, until space is found or a new root node is created).

11.1.2 Deletion

If the item to be deleted is not in a leaf node then its immediate predecessor or successor must be in a leaf node; which will replace the deleted item. At this stage - it must be checked that the number of data items left in the leaf node are not less than the permitted minimum number. If the leaf node contains at least the minimum number of items - deletion completed. If the leaf node now contains less than the minimum number of data items, an additional item needs to be found. This can be from an adjacent leaf node if it has spare capacity, which can have one of its items moved into the node missing an item; or if no adjacent leaf nodes have spare capacity then other nodes must be combined.

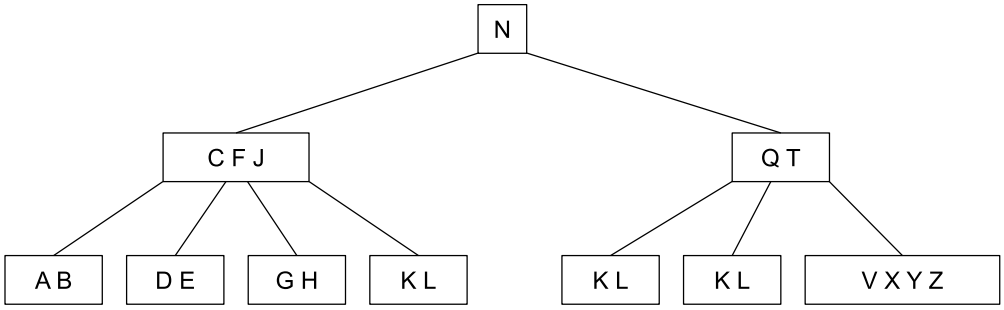


Figure 11.1: B-Tree

11.2 B* Trees

In a B* Tree, all nodes except the root must be at least two-thirds full. The frequency of splitting a node is decreased bt delaying a split. When a node overflows, a split is delayed by redistributing the keys between the node and its sibling; and when a split is made - two nodes are split into three.

11.2.1 B** Trees and B^n Trees

The increase in the "fill factor" of a tree can be done in a variety of ways. This can be seen where a B Tree is used in a database system and where it allows for a user to define the fill factor as a value between 0.5 and 1.0. A B Tree whose nodes are required to be at least 75% full is called a B** Tree; which can be generalised as a B^n Tree is a tree whose nodes are required to be $\frac{n+1}{n+2}$ full.

11.3 B+ Trees

In a B+ Tree, the indexing of data items and their storage is separated. Only the leaf nodes contain the data associated with each key / index, with the internal nodes just containing the keys. This reduces the amount of data movement required to traverse the tree, which makes them more efficient. The keys in the internal nodes are used to point a search in the direction of the leaf node which contains the required data.

When implemented - the leaves often contain a key and a reference to the record of data, which allows data files to exist separately from the B+ Tree, functioning as an *index* giving an ordering to the data

in the file - therefore making it more efficient to search!

The B+ Tree is used as a dynamic indexing method in relational database management systems.

11.3.1 Algorithms

11.3.1.1 Insertion

Insertion for a B+ tree is much the same as insertion for a normal B tree, with the new record (key and data) being inserted into to a leaf node. This is carried out by scanning the index to locate the leaf node, which if it has space - the record is inserted. However, if it doesn't have space then the leaf node is split, this means that the new leaf node is included in the sequence and the records are distributed evenly between the old and new leaves. Then, the index of the first record in the second node is copied to the parent node as an index (without data, it exists here for referencing only); if the parent isn't full then the key is inserted into it's correct position however if the parent is full then the splitting process is performed as in a standard B Tree.

11.3.1.2 Deletion

The record to be deleted must be in a leaf node. Before deleting the record, it must be tested to see if through deleting it, we will cause an underflow.

If it doesn't cause an underflow, then delete the record. This doesn't involve changing the index set. This case stays the same if the key of the record is to be deleted but is also in the index set, as the key is still required to guide a search through the B+ Tree.

If the deletion causes an underflow then, delete the record and either: records in the leaf & its siblings are redistributed and the index updated; or the lef is deleted & remaining records included in its sibling and the index updated.

11.3.2 B+ Tree Index Files

B+ Trees are used as an alternative to indexed-sequential files.

There are some disadvantages to using indexed-sequential files:

- Performance degrades as file grows - many overflow blocks created
- Periodic reorganisation of the entire file is required

Advantages of the B+ Files

- Automatically reorganises itself with small, local changes (during insertions and deletions)
- Reorganisation of the entire file is not required to maintain performance

Disadvantages of B+ Trees:

- Extra insertion and deletion overhead
- Space overhead (need to store the index plus the data)

Advantages of B+ Trees outweigh disadvantages:

- Used extensively.

11.4 Comparison of B Tree Types

B Tree	**B+ Tree**
Keys (indices) and data are not repeated	Stores redundant keys and data
Data stored in all nodes	Data only stored in leaf nodes
Searching takes more time as indices are not separated from data that may be found in an internal node or in a leaf-node	Searching for data is quick and easy as indices are separate from the data, which can be found in the leaf nodes only
Deletion of non-leaf nodes is very complicated and time-consuming	Deletion is simple because data will be in the leaf node only
Difficult and time-consuming to output a sequential list of data	Quick and easy to output a sequential list of data
The structure and operations are complicated	The structure and operations are simple

Table 11.2: Comparison of the B Tree and B+ Tree

Page 12

Async lecture - Hash Tables

2023-11-24

12.1 Hashing

Hashing involves using an array for efficient storage and retrieval of information. As we are retrieving information from an array - it has the efficiency $O(1)$. This works by mapping the input key (data item) to an output index of the array, which will not keep a sorted array of keys. However - due to the fact the hashed input key is mapped to an index, we can efficiently retrieve that data item, in a constant average time of $O(1)$.

12.1.1 The Ideal Hash Table

The *hash table* is the table which stores the mapping from the input key to the output. This can be represented in a number of ways, however it's simplest is as a 1-Dimensional array.

If we take the scenario where there is a 1-D array of m entries with keys $k_i (i = 0 \dots m-1)$ and there exists a hash function $H(k_i)$ that uniquely maps keys k_i onto the indices of the array (integers 0 through $m-1$ (one-to-one mapping of keys)). Then we will see that the data item with key k_i will be stored in the slot of the table, at the known position $h = H(k_i)$. Therefore, the data item can be accessed with one probe ($O(1)$ time).

Continuing with the above example - what happens if our hashing function isn't as good. This would result in multiple keys being hashed to result in the same position. This is called a *collision* and they are bad. Later in this lecture, methods will be explored on what to do when this happens. In reality - the majority of hash functions will have collisions.

12.2 Hash Functions

A *hash function* is any function that can be used to support the following operations:

- Map an input key / value to a fixed-size (consistent format) output value
- The output value can be used as an index to locate data in storage
- All valid input keys / values have valid output indices after mapping
- Consistency - equal input keys must produce the same output hash value

A good hash function must be quick and easy to compute, minimise collisions and achieve an even distribution of keys. The aim is for an efficiency of $O(1)$ for storage and retrieval of data.

12.2.1 Hash Function Families

12.2.1.1 Truncation

Truncation is where part of the key is ignored, with the remaining parts being used as the hash value (the index which is used to store the data in the HT). This is fast however it can fail to distribute the keys evenly.

12.2.1.2 Folding

Folding is where the key is partitioned into several parts, then these parts get combined (by addition or multiplication) to obtain the hash value.

12.2.1.3 Modulo Arithmetic

Modulo Arithmetic is where the key gets divided by the table size and using the remainder as the hash value. Where a good table size has been chosen (prime numbers generally work well), the spread of hash values will be good.

12.3 Collision Reduction

As good as any hashing function can be, we will still run into collisions. Where there are collisions, we are unable to achieve the ideal $O(1)$ access time, as we will need to perform a secondary operation to get to the data we are looking for. A good hashing function will reduce the number of collisions, however we still need to be prepared to handle them. There are four methods we need to know about on how to handle collisions, in reality there are many more.

12.3.1 Chaining

Chaining is a technique whereby a linked list is pointed to from the hash table. The linked list is used to store the collided data items. The linked list behaves as any other linked list should, following all its CRUD operations. A popular implementation choice of this is to store the head of the linked list in the hash table's slot. After getting the index of the data item through hashing, the data item is retrieved by traversing the singly linked list.

```
i = h(x) // h is the hash function, x is the input key, i is the index
head = t[i] // t is the hash table
// t[i] maintains all data mapped to bucket/slot i
if the list is not empty
    traverse the singly linked list until the target key is found
    operations on the found data
end
```

The advantages / disadvantages of using chaining are shown below:

- The linked list in a slot only contains the keys of the same hashing index - easy to retrieve the whole list of data items to match customised criteria.
- The dynamic nature of linked lists allows the storage of more data items than the table size and can always increment - the hash table never needs to be resized.
- Singly linked lists require a sequential search - traversal is inevitable.
- Insertion to a singly linked list (normally unsorted) can be done either at the beginning or the end of the list - efficient to conduct.
- Deletion in a singly linked list can be done by simply reassigning the links / pointers - efficient to conduct.

12.3.1.1 Searching

1. Hash the key into the table index
2. Search in the singly linked list maintained in the indexed slot

The average time consumption for searching a Hash Table which uses Chaining, is $O(1)$.

12.3.1.2 Insertion

1. Hash the key into the table index
2. Insert the key into the Singly Linked List maintained in the indexed slot

The average time consumption is $O(1)$.

12.3.1.3 Deletion

1. Hash the key into the table index
2. Delete the key from the Singly Linked List maintained in the indexed slot

The average time consumption is $O(1)$.

12.3.2 Linear Probing

Linear Probing is part of the *Open Addressing* family of Collision Reduction Techniques; which handles collisions by allocating the item which caused a collision to a different slot.

Linear Probing works by storing the collided data in the next available slot; this is completed by the table index increasing by 1 until an available slot is found.

```
// h is the hash function, x is the input key, i is the index
i = h(x)
j = h(x)
k = 1 // k is the increment
while there is collision at index j // j is the target index
    j = (i + k) mod n // the index increments by k or we say, linearly
    k++ // n is the size of the table
end
operations on t[j] // t is the table
```

A significant issue with Linear Probing is that when most collisions happen at the same index, items will begin to cluster. This is called *Primary Clustering* and complicates the searching, insertion and deletion.

The advantages / disadvantages of Linear Probing can be seen below:

- Easy to compute.
- No extra consumption of keeping the pointers/links.
- Space can be fully used when the increment is set as 1.
- Table resizing may be required when the size of the problem increases.
- Primary clustering dependent on the quality of hash functions.

12.3.2.1 Searching

1. Hash the key into the table index
2. Repeat incrementing the table index by 1 until the key or an available slot is found

The average time consumption will be $O(1)$.

12.3.2.2 Insertion

1. Hash the key into the table index
2. Repeat incrementing the index by 1 until an available slot is found

The average time consumption is $O(1)$.

12.3.2.3 Deletion

1. Hash the key into the table index
2. Repeat incrementing the table by 1 until the key is found and deleted

The average time consumption is $O(1)$.

12.3.3 Quadratic Probing

Quadratic Probing is another member of the Open Addressing family. It works by addressing the primary clustering issues caused by Linear Probing by chaining how the index changes when a collision occurs.

Quadratic Probing works by incrementing the index by a quadratic increment rather than a linear one. For example - the collided index 1 keeps incrementing as $1 + 1^2$, $1 + 2^2$, $1 + 3^2$, ..., $1 + x^2$ until an available slot is found.

```
// h is the hash function, x is the input key, i is the index
i = h(x)
j = h(x)
k = 1 // k is the input argument for the quadratic function
while there is collision at index j // j is the target index
    j = (i + k^2) mod n
    // the index increments by k^2 or we say, quadraticly
    k++ // n is the size of the table
end
operations on t[j] // t is the table
```

Quadratic Probing removes the primary clustering which occurs with Linear Probing; however it isn't perfect as it creates its own type of clustering - *Secondary Clustering*.

12.4 Double Hashing

Double Hashing uses another hashing value to increment the collided index. This avoids both primary and secondary hashing.

```
i = h(x) // h is the hash function, x is the input key, i is the index
j = g(x) // g is another hash function, j is the index increment
while there is collision at index i
    i = (i + j) mod n   // the index increments by j,
                        // another hashing value for each iteration
end
operations on t[i] // t is the table
```

Double hashing keeps incrementing the collided index by a hashing value until the target or an available slot is found.

Page 13

Async lecture - The Heap

2023-12-01

13.1 Priority Queue

A *priority queue* is an Abstract Data Type which supports the two following operations:

- Insert an element into the queue with an associated priority
- Remove the element from the queue which has the highest priority

They generally get implemented using a static array, where they aren't sorted by priority. Each index in the array will contain the element and its corresponding priority. Insertion into the priority queue is efficient, as we are just inserting into the first empty space; hence it has an efficiency $O(1)$. However, removing an element from the array isn't as simple; we have to search sequentially through the entire list to find the element with the highest priority, therefore it has an efficiency $O(n)$.

13.2 Heap Data Structure

The *heap* data structure further develops on a priority queue's inadequacies.

A heap is a binary tree with the following characteristics:

- It is a complete binary tree (see *Hierarchical Data Structures*)
- The key in the root is larger than the key in either child node and both sub-trees satisfy the heap property.

In reality, the highest priority element would be stored at the root. This is also called a *max-heap*. There is also a *min-heap*, which as the name suggests is the inverse of a max-heap with the lowest priority at the root.

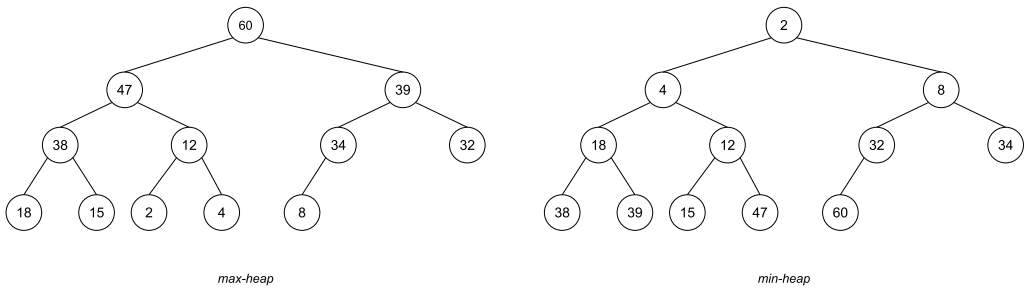


Figure 13.1: The Heaps

13.3 Heap Algorithms

13.3.1 Insertion

As with all other types of Trees, the new item is inserted at the bottom level. It is inserted as the next leaf on that level; which if full - it gets inserted as the first node on the next level on the left hand side. Adding a new item to the heap will destroy the second property of heaps. This must be rectified which is done so via moving the item up the tree until it ends up at the root or until it finds parents & children which restore the heap property. This takes $O(log_2 n)$ exchanges

13.3.2 Removal of a Node

The item at the root can easily be removed - this leaves us with two sub-trees from which we must re-create a single tree that satisfies the heap properties. To re-create the tree, move the furthest right leaf node to the root, then move nodes around internally to ensure that the heap properties are still satisfied.

13.4 Heap Implementation

A heap can be stored using a static array. Due to one of the rules of a heap being it must be a complete binary tree, this means a 'row' will always be full other than the final row. Therefore, the indexes working down row by row and left to right on each row can just be stored in the array. The indexes of parents and children can be calculated as seen below:

```
Index_LeftChild = Index_Parent * 2 + 1
Index_RightChild = Index_Parent * 2 + 2
Index_Parent = (Index_Child - 1) / 2
```

Heaps get used in a number of ways:

- Dijkstra's Algorithm - find the shortest path tree in a graph
- Kruskal's Algorithm - to find the Minimum Spanning Tree of a graph
- Huffman's algorithm - text compression
- In a heap sort

Page 14

Async lecture - Huffman Coding

2023-12-01

14.1 Data Compression

Data Compression aims to reduce the size of the data as much as possible, whilst representing the information as accurately as possible. This is important because some uncompressed data can be extremely large and through compression we are able to reduce its size therefore improve performance when transmitting the data. Images, Text, Sound and Video can all be compressed; however this workshop will only focus on the compression of Text data.

When transmitting compressed data - the sender and receiver of the information must understand the encoding scheme used. To become compressed, the original data is passed through an encoder; and at the other end - to obtain the decompressed data from the compressed data, we pass the compressed data through a decoder. A formula can be used to calculate the compression ratio, which describes the difference between the original data and a compressed version of itself:

$$\text{compression ratio} = \frac{\text{original data size}}{\text{compressed data size}} : 1$$

14.1.1 Lossless Data Compression

Lossless Data Compression is a technique where the original data can always be reproduced exactly from the compressed data - no data is lost in the compression process. Lossless data algorithms are used in text compression where we must be able to recreate the original data. Generally, Lossless compression will achieve a compression rate of between 2:1 and 8:1.

14.1.2 Lossy Data Compression

Lossy Data Compression os a technique where the original data cannot be reproduced from the compressed - the decompressed data differs to the uncompressed data we started with. This is fine for many applications including audio, video and images, where the human eye / ear cannot perceive minute details which can be lost; or where average people don't have the required hardware to appreciate the details. Audio can be compressed at a compression rate of 10:1, video can be compressed at a compression rate of 100:1 and images can be compressed at a compression rate of about 10:1. These won't make obvious changes to the media however on very close inspection - changes may be visible. However in some cases, lossy data compression may not imply a loss of quality and could lead to an improvement in quality through reducing the amount of random noise in images, or removing background noise in music.

14.2 Fixed Length Coding

Fixed Length Coding is a compression technique whereby the length of the codeword for each character is the same - using either ASCII code or Unicode.

If we take a string of 1000 characters, consisting of the characters `a`, `u`, `x` and `z`. Without compression, this string requires 8000 bits to be stored in (derived from 1000 characters multiplied by 8-bits per character using ASCII code). However - if we apply a compression algorithm to this, we can encode the characters differently:

```
00 - a
01 - u
10 - x
11 - z
```

This means we now need 2000 characters to store the string (1000 characters multiplied by 2-bits per character); *and* 48-bits to store the table (8-bits for the table size, 32-bits for the original letters, 8-bits for the encoded letters (as 2-bit codes)). Therefore, the overall size of this string is 2048 bits, giving us a compression ratio of 3.9 ($\frac{8000}{2048}$)

14.3 Variable Length Coding

Variable Length Coding is a compression technique which is similar to Fixed Length Coding, except it uses variable-length code words to represent the characters. This means the code-words used to represent the characters differ in length, with shorter code-words being used to represent frequently occurring characters and longer code-words used to represent less frequently occurring characters.

If we take the string `aaxuaxz`, the frequencies of the letters are shown below:

```
a - 3
x - 2
u - 1
z - 1
```

If we now establish codes for each of these letters, the encoded string will be `0010110010111` and the codes are shown below:

```
  0 - a
 10 - x
110 - u
111 - z
```

Through using variable length coding, we have achieved a 1-bit size reduction as compared to the same string being encoded using 2-bit codes in FLC. This difference scales up massively: if we give the characters the frequencies 996, 2, 1, 1; then using FLC would give us a encoded version which is 2000 bits long and VLC would result in 1006-bits.

Decoding a VLC string is where it gets more complex as we do not know immediately how many bits to pick off the string. We start from the left hand side and work left to right. We examine the current and the next bit(s) and using the table of bits-to-character mappings we are able to determine what bits correspond to a character. This works for this VLC example as all the codes start with `1` apart from `a`'s which begins with `0` and they all finish with `0` apart from `z`'s which finishes with `1`. This means we are easily able to identify a chunk of bits which correspond to a character. As there are only two edge cases to the rule, we are able to work around them.

14.4 Huffman Coding

Huffman coding is much like VLC, in that it assigns short code-words to characters with high frequencies and longer code-words to characters with lower frequencies. The Huffman encoder takes a block of characters with fixed length as an input then creates a block of output bits of variable length as an output; the Huffman encoder is a fixed-to-variable length coder.

14.4.1 Generating Huffman Codes

The first stage to generating Huffman Codes is to create a Huffman Tree. The Huffman codes are then determined by following the path from the root to the leaf nodes.

14.4.1.1 Creating a Huffman Tree

A Huffman tree is a weighted binary tree where:

- The leaves of the Huffman tree are the characters to be encoded
- The branches of the Huffman tree are labelled with either `0` or `1` (this is used to determine their code-word).

Initially, we have a forest of leaf nodes containing the characters and their frequencies. The Huffman tree is created by:

1. Sorting the forest of leaf nodes by frequency
2. Combining the two nodes with lowest frequency
3. Creating a binary tree with parent node containing the frequencies of its children
4. Sorting nodes and repeating process until one node left (this is the root node)

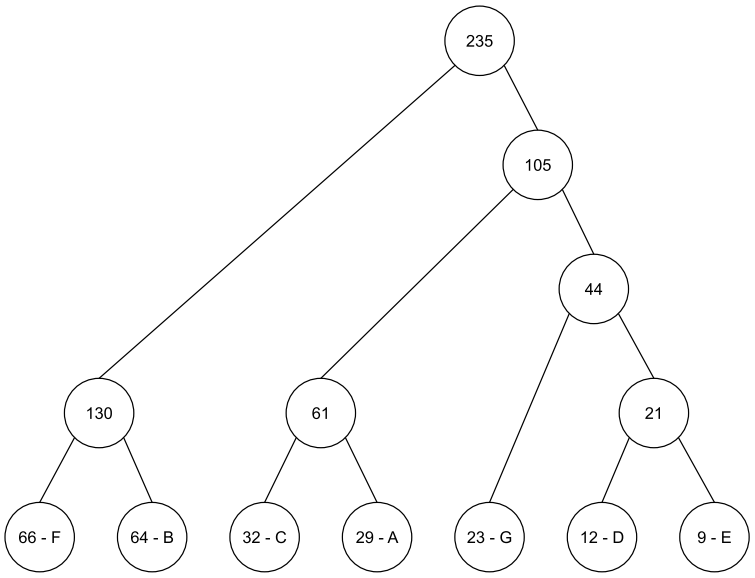


Figure 14.1: Huffman Tree

The figure above shows the first stage of Huffman coding, generating the Huffman tree. The figure below shows the next stage where we assign each branch a value (either 0 or 1, with 1 being assigned to the right). From here we are able to work from the root node to the node we are interested in and generate the Huffman Codes, as seen below.

```
F - 00
B - 01
C - 100
A - 101
G - 110
D - 1110
E - 1111
```

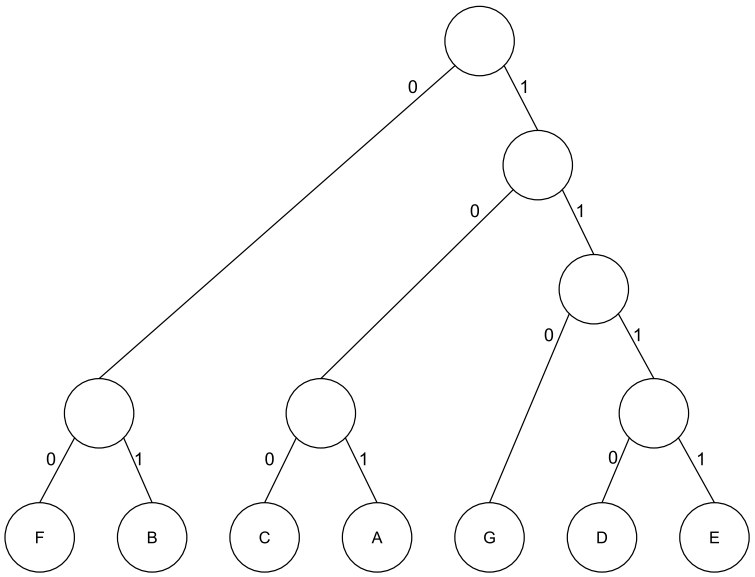


Figure 14.2: Huffman Tree showing Code-Words

Page 15

Async lecture - Graphical Data Structures

2023-12-08

15.1 Graphical Data Structures

The *Graphical Data Structure* (or *Graph*, for those who aren't pretentious) consists of nodes where each node has many predecessors and many successors. The need for graphs arise through the numerous varieties of data structures we have explored thus far in this module not having the ability to have many predecessors and many successors, which is required for applications such as a social network, or transport map.

Graphs are used mostly when a linear and tree structure isn't applicable.

15.1.1 Graph Theory

Graphs and their applications are based on *Graph Theory*:

- Shortest Path Problem: Graph traversal and path search with tradeoff between space and time
- Ramsey Theory: For any six people, either at least three of them are mutual strangers or at least three of them are mutual acquaintances
- Graph colouring: No more than four colours are required to colour the region of the map so that no two regions have the same colour

15.2 Graph Terminology

Term	Definition
Graph	A collection of nodes (called 'vertices') and line segments connecting the vertices (called 'edges').
Undirected Graph	A graph where each edge in the graph has no direction
Directed Graph (di-graph)	A graph where each edge in the graph has a direction to it's successor
Acyclic Graph	A graph with no cycles
Directed Acyclic Graph	A directed graph with no cycles. Abbreviated to 'DAG'

continued on next page

Term	Definition
Adjacency	Two vertices are adjacent if they are connected by a single edge
Path	Sequence of edges
Cycle	Path containing at least three vertices that starts and finishes with the same edge
Degree	The degree of a vertex is the sum of the adjacent vertices. In a digraph: the out-degree of a vertex is the number of edges leaving the vertex and the in-degree is the number of edges entering the vertex
Sparse Graph	A graph is said to be 'sparse' if there are only a few edges between nodes
Dense Graph	A graph is said to be dense if most of the edges between vertices are present
Connected Graph	A graph is connected if there is a path from any vertex to any other vertex
Strongly Connected Graph	A directed graph is strongly connected if there is a path from any vertex to any other vertex
Weakly Connected Graph	A directed graph is weakly connected if, on suppressing the direction of the edges - the resulting undirected graph is connected
Disconnected Graph or Disjoint Graph	This is a graph which is not connected
Weighted Graph	These are graphs where the edges are assigned a weight; they can be either directed or undirected; the weight can be used to represent anything

Table 15.1: Graph Terminology

15.3 Graph Representation

There are two different methods as to which Graphs can be stored by. They differ in the way the nodes and edges are maintained internally. If the graph is sparse then the adjacency list representation will be more space-efficient than the adjacency matrix representation.

Property	Matrix	List
Space	$O(v^2)$	$O(v+e)$
Vertex Insertion	$O(v^2)$	$O(1)$
Vertex Deletion	$O(v^2)$	$O(v+e)$
Edge Insertion	$O(1)$	$O(1)$
Edge Deletion	$O(1)$	$O(e)$

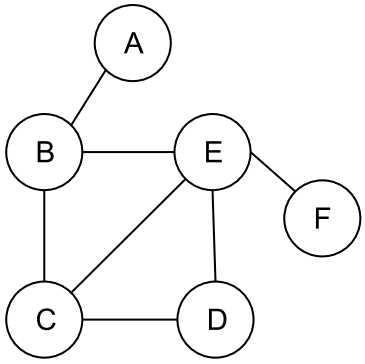
continued on next page

Property	Matrix	List
Adjacency Check	$O(1)$	$O(e)$

Table 15.2: Comparison of Matrix and List representations for Graphs (v vertices, e edges)

15.3.1 Adjacency Matrix Representation

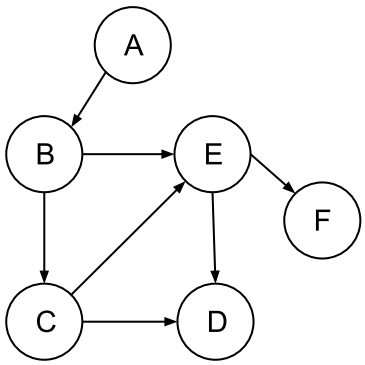
This uses a n-by-n boolean matrix with one row and one column for each of the n vertices in the graph.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Figure 15.1: Undirected graph represented in an Adjacency Matrix

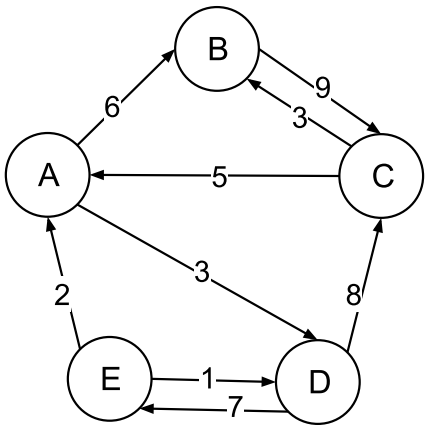
If the element in the i-th row and j-th column is equal to 1, then there is an edge from the i-th vertex to the j-th vertex. Therefore if the element in the i-th row and j-th column is equal to 0, then there is not an edge from the i-th vertex to the j-th vertex.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Figure 15.2: Directed graph represented in an Adjacency Matrix

For weighted graphs, the element is either the weight of the graph or infinity (∞)



	A	B	C	D	E
A	∞	6	∞	3	∞
B	∞	∞	9	∞	∞
C	5	3	∞	∞	∞
D	∞	∞	8	∞	7
E	2	∞	∞	1	∞

Figure 15.3: Weighted graph represented in an Adjacency Matrix

A *vector* (1-D array) is used to store the vertices and a *matrix* (2-D array) is used to store the edges.

15.3.1.1 Observations

- The size of the graph (number of vertices) needs to be known in advance
- You cannot store duplicate edges, only one edge can be stored between vertices
- It takes $O(1)$ time to determine if there is an edge between vertex `u` and `v`
- If the graph is sparse then a significant part of the adjacency matrix will be empty
- Undirected graphs are symmetrical around the diagonal therefore half the graph will contain repeated information
- Inserting an edge into a directed graph takes $O(1)$ time. Undirected graphs take double the time as you have to insert two entries
- Deleting an edge from a directed graph takes $O(1)$ time. Again, undirected graphs take double the time as you have to clear 2 entries
- To determine the degree of a vertex, count all the non-zero entries in the corresponding row of the adjacency matrix

15.3.2 Adjacency list Representation

The *Adjacency List Representation* of a Graph uses a set of Singly Linked Lists, with one for each vertex. Each SLL contains all the vertices that are adjacent to the vertex. A SLL or array is used to store the vertices in a 'Vertex List'.

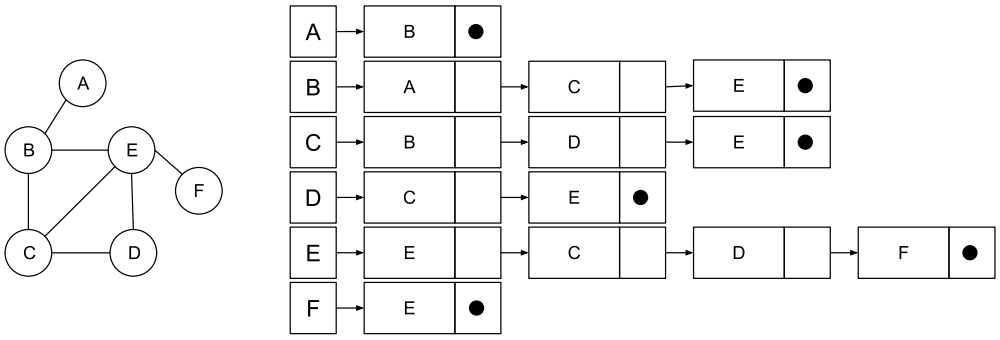


Figure 15.4: Undirected graph represented in Adjacency List

For weighted graphs, the nodes of the SLL will contain the name of the vertex and the weight of the corresponding edge.

15.3.2.1 Observations

- Flexible to use - it is easy to insert / delete vertices
- Allows for duplicated edges
- For undirected graphs - each edge is stored twice
- Space-efficient representation of a sparse graph
- To determine if there is an edge from vertex `u` to vertex `v`, requires `u`'s linked list to be searched. For dense graphs, there may be many vertices in the linked list therefore this completes in $O(n)$ time.
- Inserting an edge to a directed graph takes $O(1)$ time as it gets inserted at the head of the list. Undirected graphs take double the time as two new nodes have to be added to two linked lists.
- Deleting an edge from a directed graph takes $O(e)$ time as the program will need to traverse the list to locate the corresponding vertex. This will take double the time for an undirected graph as two deletions are needed.
- Determining the degree of a vertex requires the length of the corresponding linked list to be found.

15.4 Searching A Graph

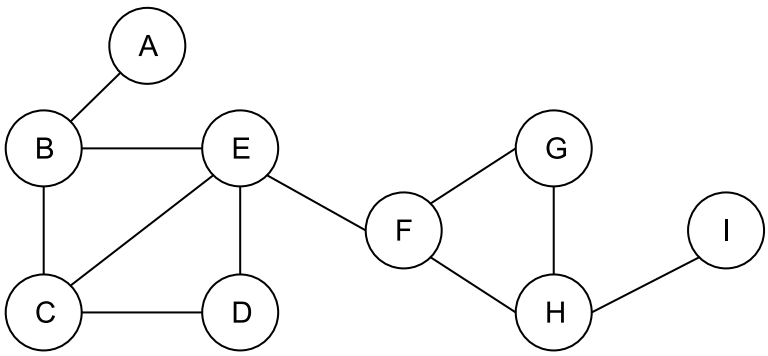


Figure 15.5: Generic Graph Searching Examples Graph

15.4.1 Depth First Search

A *Depth First Search* (DFS) for a Graph data structure works much the same as a DFS does for a Tree data structure with the pre-order traversal. For a Graph DFS, it will visit (process) all of a vertex's descendants before moving onto an adjacent vertex.

1. Assume all nodes are marked as "not visited".
2. 'Visit' the start vertex (this can be any vertex)
3. Select an unvisited vertex which is adjacent & connected to the current vertex and visit that one
4. Repeat step 3 until a dead end is reached (this will be where a a vertex has no adjacent vertices left)
5. Backtrack to find another unvisited vertex and repeat steps 3 & 4 until all vertices have been searched

The DFS uses a stack to store the unvisited vertices. An Iterative implementation for the DFS is below:

```
Assume each vertex marked as 'not-visited' push first vertex onto stack
mark as visited
while stack not empty loop
    pop vertex off stack
    process the vertex
    for each adjacent unvisited vertex
            push vertex onto stack
            mark as visited
    end
end
```

A recursive implementation for the DFS is shown below:

```
Assume each vertex marked as 'not-visited' process first vertex
mark as visited
for all nodes adjacent to the vertex
    if not visited then
        perform Depth First Search
    end
end
```

Using the example graph in the figure above, the DFS search would return the order: A. B, E, D, F, G, H, J, C.

15.4.2 Breadth First Search

A *Breadth First Search* (BFS) for a Graph works much the same as a BFS (level-by-level search) of a BST. In a graph, a BFS will start at a node and work out layer-by-layer through each connected vertices. It works through the vertices adding un-visited adjacencies to it's queue then visiting each in turn and repeating the visiting & enqueueing at each, until every vertex has been visited.

The basic BFS algorithm for a graph is shown below:

```
Assume each vertex marked as "not-visited" enqueue vertex onto queue
mark as visited
while queue not empty loop
    dequeue vertex off queue
```

```
        process the vertex
        for each adjacent unvisited vertex
            add vertex to queue
            mark as visited
        end
    end
```

Using the example graph in the figure above, the BFS search would return the order: A, B, C, E, D, F, H, G, J.