BSc (Hons) Computer Science

University of Portsmouth

Second Year

Data Structures and Algorithms

M21270

Semester 1

Hugh Baldwin

hugh.baldwin@myport.ac.uk

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**

7 Lecture - Workshop 10 and 11 Async **18**

8 Lecture - Workshop 12 Async **23**

9 Lecture - Workshop 13 and 14 Async **25**

10 Lecture - Workshop 15 and 16 Async **29**

- Mid-Unit assessment (30%) on 8th November
- Exam (70%) in TB1 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 structre, 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 it's 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 it's 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 structre - 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 < ...
  - $O(1) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < ...$

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 $\mathrm{O}(\mathrm{n})$
  - Theoretically, the best case would be where the item is at the start of the array and so would be $\mathrm{O}(1)$
  - The average case is theoretically $\frac{\mathrm{n}}{2}$, but that is still a Big-O of $\mathrm{O}(\mathrm{n})$
  - 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 $1 + \log_2 \mathrm{n}$ operations, so $\mathrm{O}(\log_2 \mathrm{n})$
  - The best-case would still be $\mathrm{O}(1)$ if the item is at the mid-point in the list
  - The average would be $\mathrm{O}(\log_\mathrm{n} 2)$
  - 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 worse 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 unforseen 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 $\mathrm{O}(\mathrm{n}\log_2\mathrm{n})$
- 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 $\mathrm{O}(\mathrm{n}\log_2\mathrm{n})$, worst-case is $\mathrm{O}(\mathrm{n}^2)$. On average, the Big-O is approximately $\mathrm{O}(\mathrm{n}\log_2\mathrm{n})$

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, $\mathrm{O}(1)$ 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 $\mathrm{O}(\mathrm{n})$
  - 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 $\mathrm{O}(\log_2 \mathrm{n})$ 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 sucessors 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 it's 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 it's 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 it's height
- A rotation changes the structure of the tree, but without chaning 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 psuedocode 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 thr 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 sinle 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 $\mathrm{O}(1)$
- The worst case Big-O is still $\mathrm{O}(\mathrm{n})$
- However, over a series of operations, the average Big-O trends towards $\mathrm{O}(\log_2 \mathrm{n})$

Lecture - Workshop 10 and 11 Async

11:00 18/11/23 Dalin Zhou

Two-Three Trees

- A two-three tree is a perfectly balanced tree in which
  - Each node contains one or two keys (a 2-node and 3-node respectively)
  - Every internal node has either two (for a 2-node) or three (for a 3-node) children
  - All leaves are on the same level, hence the tree is always balanced
- For each individual node,
  - The keys of all children in the left subtree are less than the first key
  - The keys of all children in the centre subtree are greater than the first key
  - If a right tree exists (making it a 3-node), the keys of all children in the centre subtree are less than the second key
  - The keys of all children in the right subtree are greater than the second key

Searching

- A similar searching method is used to that of a BST
- Start at the root node, if the key it not in the root node, move to the only subtree it could be in and recurse
- You must compare with both keys of the node before moving to a subtree
- Since there can be 3 subtrees for each node, it is effectively a ternary search, rather than a binary search

Inserting

- A similar insertion method is used to that of a BST
- Unlike a BST insertion, a new node is not created
- Use the same searching strategy to find where the key should be inserted, and then depending upon if the node is a 2-node or a 3-node, a different method is needed
- If the leaf node contains one key, the new key is inserted into the leaf and the insertion is finished
- If the leaf node contains two keys, a new space must be created
  - The two existing keys and the new key are of equal importance
  - The existing 3-node is split into two nodes, in this case node1 and node2
  - The smallest key is placed in node1 and the largest in node2
  - The remaining key (the middle key) is promoted to be the parent node

  - If the existing parent node is a 2-node, then the promoted key is inserted into the existing parent, and the references are set to node1 and node2
  - If the existing parent node is a 3-node, then the split and promote process is repeated until a new place is found for the promoted key. This may result in the creation of a new root node

Deleting

- A similar deletion method is used to that of a BST
- The key to be deleted is either
  - In a leaf node
    - Which is removed, as for a normal BST
    - This leaves either a 2-node leaf or a hole in the tree
  - In an internal node
    - Which is replaced by it's In Order successor or predecessor, which will be in a leaf node, which is once again normal for a BST
    - This leaves either a 2-node leaf or a hole in the tree
- In either case, we end up with a 2-node leaf or a hole in the tree
  - A 2-node leaf can be left as is
  - A hole in the tree needs to be removed
- To remove the hole, you
  - Traverse the 2-3 tree upwards towards the root
  - The hole is propagated through the tree until it can be eliminated
- The hole is eliminated by either
  - Being absorbed into the tree
  - Being propagated to the root of the tree, which results in the node being removed and the height of the tree being reduced by one

B Trees

- A B tree is a multi-way search tree which is optimised for reading directly from a disk
  - Each node represents a block or page of secondary storage
  - Accessing a node means reading the keys from secondary storage, which is expensive and slow. This means that we want to reduce the number of nodes as much as possible
- A B tree of order n has the following properties
  - The root node has either no children, or between 2 and n children
  - All internal nodes have between $\frac{n}{2}$ and n children
  - All leaves are on the same level
  - Each node can store 1 less key than it has children
- This also means that a 2-3 tree is a B tree of order 3, and a BST a B tree of order 2
- For any given tree of order n and height h, we can store $n^h - 1$ keys
  - e.g. a tree of order 10 and height 3 can store $10^3 - 1 = 1000 - 1 = 999$ keys

Searching

- Start at the root node, and perform a binary search on the keys in the root noed
- If the key is found, we are finished
- If the node we are currently in is a leaft node, and the key is not found, then the key is not in the tree
- Otherwise, follow the required branch to the next node an repeat the process

Inserting

- The tree is searched to find the leaf node in which the key should be inserted
- If the node is not full
  - Insert the key into the node in it's correct order
- If the node is full
  - The node is split in two and the middle key is promoted upwards to the parent
  - If the parent node is full, then the split and promote process is repeated until space is found, or a new root node is created

Deleting

- Similar to the deletion method for a 2-3 tree
- If the key to be deleted is not in a leaf node, then the immediate predecessor or successor will be in a leaf node, which will replace the key to be deleted
- Therefore, you need to check if there will be enough items in the leaf node
- If more than the minimum number of keys will be left, you can delete the key
- If not, you cannot delete the item directly, and so must
  - Check adjacent leaf nodes and if possible move items between nodes
  - If there are no spare items in adjacent nodes, then nodes must be combined, possibly reducing the height of the tree

B* Trees

- In a B* tree, all nodes except the root must be at least two thirds full, as opposed to a B tree where all other nodes must be half full
- The frequency of splitting a node is reduced by delaying a split
- When a node overflows
  - A split is delayed by redistributing the keys between the node and it's sibling
  - When a split is made, two nodes are split into three
- Items are redistributed by
  - Moving the median key into the root
  - Moving the remaining keys into the children of the root, half in each
  - This leaves room in both nodes for at least 1 additional key

- If both the node and it's sibling is full, a split occurs and a new node is created
- keys from the split node, it's sibling and the parent are distributed evenly between the now three nodes
- The two separating keys are put into the parent node so that it can have enough children
- This always leaves the child nodes two thirds full

B** & Bn Trees

- These forms of B trees allow the "fill factor" to be increased
- Some DBMSes will allow the user to select a fill factor anywhere between 0.5 and 1.0
- A B tree with a fill factor of 75
- In general, a Bn tree is a B tree whose nodes are required to be $\frac{n+1}{n+2}$ full
- B* trees are much less used than B+ trees

B+ Trees

- What if we need all of the data in a B tree to be presented in ascending order?
  - We can use an In Order traversal, but this requires accessing each page of data to be accessed
  - Or we can use a B+ tree
- In a B+ tree, only the leaf nodes contain the data associated with each key or index
- Leaf nodes
  - Contain a set of keys and their associated data
  - Are linked together in sequence
  - Stored in secondary storage
- Internal nodes and the root are stored in main memory, and
  - Contain only keys
  - Are used as an index to the leaf nodes
  - Are known as an index set
  - Are implemented as a B tree
- B+ trees are often used as an alternative to indexed-sequential files

Inserting

- A new record is inserted into a leaf node following the same rules as a regular B tree
- Scan the index set to located the relevant leaf node
- If the leaf node has space, the record is inserted and the index set is unchanged
- If the leaf node is full
  - The leaf is split
  - The index set is updated to show the new leaf node

  - Records are distributed evenly between the old and new leaves
  - The first key of the second node is copied to the parent node, without it's data
  - If the parent node is not full, the key is inserted as usual
  - If the parent node is full, then the splitting process is performed as usual for a B tree

Deleting

- If the deletion of the record does not cause an underflow, then delete the record
  - No changes are made to the index set, even if the record is also in the index set, as the key is still required to search through the B+ tree
- If the deletion of the record causes an underflow, delete the record and then either
  - Redistribute the records in the leaf and it's sibling(s), and update the index set
  - Delete the leaf and move the remaining records to it's sibling(s), and update the index set

B+ Trees vs Indexed-Sequential Files

- Advantages of B+ trees
  - Automatically reorganises itself during insertions and deletions
  - Only small, local updates are required
  - The entire file does not need to be reorganised to maintain performance
- Disadvantages of B+ trees
  - Extra overhead for insertion and deletion operations
  - Space overhead for the index set
- Disadvantages of Indexed-Sequential files
  - Performance degrades as the file grows, as overflow blocks are required
  - Periodic reorganisation of the entire file is required to maintain performance

Lecture - Workshop 12 Async

11:00 05/12/23 Dalin Zhou

Hash Tables

- Hashing involves using an array for the efficient storage and retrieval of data
- Ideally, both the insertion and retrieval of data would have an efficiency of $\mathrm{O}(\mathrm{n})$
- A hash table allows this, as far as possible with a finite array
- A perfect hash table has an infinite length, and has a hashing function which uniquely maps keys to the data they relate to
- It is not possible to do either of these things, and so some tradeoffs must be made

Hashing Functions

- A good hashing function must
  - Be quick to compute
  - Minimise collisions
  - Evenly distribute keys
- There are a few types of hashing function
- Truncation
  - Ignore parts of the value and use the remaining parts as the hash value
  - This is very fast but does nothing to distribute keys evenly, or avoid collisions
- Folding
  - Split the value into several parts and then combine them in a different way to obtain the hash value
  - Usually results in better distribution than truncation
- Modulo Arithmetic
  - Divide the value by the table size and use the remainder as the hash value
  - To achieve good distribution, the size of the table should be a prime number

Collisions

- A collision occurs when more than one data value results in the same key value
- In this situation, it must be decided how to handle the collision and there are multiple strategies
- Chaining
  - Singly linked lists (SLLs) are used to resolve collisions
  - Elements which collide are stored in an SLL with it's first element stored in the hash table
  - This is an easy method but is not fast, as SLLs have an efficiency of $\mathrm{O(n)}$
- Linear probing
  - If an element collides, search the hash table linearly from the initial hash position and store the value in the first available slot
  - This makes both insertion and retrieval slower as there is a chance that the entire table must be searched linearly to check the item is not there
- Quadratic probing
  - Rather than linearly checking the next slot, check them quadratically
  - i.e. If a collision occurs, check $\mathrm{h + i^2}$, so $\mathrm{h + 1}$, $\mathrm{h + 4}$, $\mathrm{h + 9}$, etc
  - This works well if the size of the table is prime
- Double hashing
  - Two hashing functions, can either be the same or different functions, are used
  - The hash value of the first function is added to that of the second function repeatedly until a slot is found
  - i.e. If a collision occurs, check $\mathrm{h + g}$, $\mathrm{h + 2g}$, $\mathrm{h + 3g}$, etc
  - This works well if the size of the table is prime
- Random rehashing
  - A pesudo-random number generator is used to generate the offsets
  - Since a seed is used for the random number generator, it is possible to recreate the sequence of random numbers when attempting to retrieve data
  - This method is excellent at avoiding collisions but much slower than the other available methods

Deletion

- When deleting an item from a hash table, you cannot leave that space blank
- This is because it would break the following keys in the group which hashed to the same key
- Instead, a special value or flag can be used to mark that the space is free for insertion, but searches should continue looking and not stop there
- When inserting, we can check if a space is free, or has this special value in it
- When searching, we can still check if there is an item in the space, as it is taken up by the flag

Lecture - Workshop 13 and 14 Async

11:00 12/11/23 Dalin Zhou

Heaps

- A max-heap is a binary tree with the following characteristics
  - It is a complete binary tree - all levels other than the last are full and the last level has all it's leaves on the left side
  - The key in the root is larger than both child nodes
  - Both subtrees satisfy these requirements
- A heap can be implemented using a linear datastructure, such as a list or array
  - The root of the tree is stored at element 0
  - Then it's children are stored from left to right, and so on down the tree
  - We can get the index of the current node's children or parent using the following formulae, assuming the indexes start at 0
    - $p = \frac{c-1}{2}$ where p = parent index, c = current index
    - $l = 2c + 1$ where l = left index, c = current index
    - $r = 2c + 2$ where r = right index, c = current index
- A heap can be used to implement
  - Djikstra's Algotithm
  - Kruskal's Algotithm
  - Huffman's Algorithm
  - Priority Queues

Insertion

- When an item is inserted, it is always placed as the next leaf on the lowest level
- If the lowest level is full, it is placed as the left-most node on a new, lower level
- Insertion will probably break the heap properties of the tree, and so must be corrected
  - The inserted item is moved up the tree until it either ends up as the root node, or finds a parent which restores the heap properties

Deletion

- Any item in the tree can be deleted at any time, but this may leave two detached subtrees
- To restore the tree, move the right-most item on the lowest level into the root, then move it down the tree until the heap properties are restored

Priority Queues

- A priority queue is an abstract data type with the following operations
  - Insert an element with an associated priority
  - Remove the element with the highest priority
- This can be implemented with a static, unsorted array
  - Each entry contains the value of the element, and it's priority
  - Inserting an element is efficient as it can be inserted into the first empty space, and so has a Big-O of $\mathrm{O}(1)$
  - Removing an element however, is inefficient as the entire array must be searched to find the element with the highest priority, and so has a Big-O of $\mathrm{O}(\mathrm{n})$
- It can alternatively be implemented using a heap
  - The highest priority item will always be sorted to the root of the tree
  - This gives a Big-O of $\mathrm{O}(\log_2 \mathrm{n})$ for both insertion and removal

Heap Sort

- A heap can be used as a form of sorting algorithm
- Build a heap form the elements to be sorted
- Retrieve the root of the tree as the largest or smallest item for a max- and min-heap respectively
- Swap the root with the rightmost leaf on the biottom level
- Rebuild the heap without the previous root
- Recurse this algorithm until the last node is found

Data Compression

- Since files such as videos can be very large when stored at original quality, compression is needed to reduce the size and bandwith needed to store and transfer files
- We only talk about text compression as this is the most simple to implement, but most of the concepts carry over to other types of data
- Both the sender and reciever of the data must know and understand the encoding scheme used to represent the text
- There are two types of data compression
- Lossless Compression
  - All of the original data can be reproduced using only the compressed data
  - Used in text compression, as al of the original text must be preserved
  - Compression ratios of between 2:1 and 8:1 depending upon the implementation
- Lossy Compression
  - The decompressed data is slightly different to the original data, and the original data cannot be reproduced
  - A small loss of accuracy is traded for a massive increase in compression ratio
  - Used for the compression of sound, images and videos where a slight loss of data is acceptable given the greatly reduced size

Fixed-Length Coding

- Each of the codes used to represent a chunk of data are the same length
- This is efficient when only needing to represent a small number of codes
- e.g. If only 4 different codes were needed, they could be represented by a 2-bit number
  - This would allow 4 different values of arbitrary length to be encoded using only 2 bits each
  - For a message made up with only 4 letters, the length could be reduced by 4-fold by reducing the number of bits for each letter from 8 to 2

Variable-Length Coding

- Each of the codes used to represent a chunk of data can be any length
- Shorter codes are used for more frequently occuring characters
- Longer codes are used for less frequently occuring characters
- Using variable length coding can reduce the size of an encoded string
- It is possible for the size to be increased, if the codebook is implemented improperly or if the frequency of characters is relatively similar
- Decoding is also slower as it hard to determine how many bits in the encoded string correspond to which codes

Huffman Coding

- Huffman coding uses a statistical method to determine each variable-length code
- The most frequently occuring characters are converted to short codes and the least frequent characesr are converted to longer codes
- A huffman tree is a weighted binary tree where
  - Each leaf node represents a character to be encoded
  - Each branch is labelled with a 1 or 0
- Huffman codes are determined by following the path from the root to the leaf node in question, adding a 1 or 0 to the code based upon the branches

Creating a Huffman Tree

- Initially, you have a list of leaf nodes containing the characters and their frequencies
  - The list of leaves is then sorted by frequency
  - Combine the two nodes with the lowest frequencies
  - Create a binary tree with the parent node containing the sum of frequencies of it's children
  - Sort the nodes and repeat until there's only one node left
- This requires a priority queue, and therefore can use a min-heap to represent the queue
  - Remove the two nodes with the lowest frequencies from the heap
  - Create a new internal node with said nodes as its children and the sum of their frequencies as its value
  - Insert to the new node back into the min-heap

- The final remaining node is the root of the huffman tree
- To get the huffman code for a symbol, you traverse the tree from the root to the symbol
  - For each left-branch you take, add a 0
  - For each right-branch, add a 1

Lecture - Workshop 15 and 16 Async

14:00 | 12/01/24 (Redone) | Dalin Zhou

Graphical Data Structures

- A graph consists of nodes which have many predecessors and successors
- Graphs are used for a wide variety of applications, such as storing a map of a rail network, social media connections, etc

Terminology

- Nodes/Vertices
  - The actual items, could be locations, people, or items
- Edges
  - Connections between nodes/vertices
- Undirected Graph
  - Each edge is just a link, with no particular direction
- Directed Graph
  - Each edge has a direction (explicit predecessor and successor) and can only be traversed in that direction
- Adjacent Vertices
  - Vertices connected by a single edge
- Path
  - A sequence of edges which can be used to traverse from one vertex to another
- Degree
  - The number of edges or adjacent vertices of a vertex
- In/Out-Degree
  - Sum of entering/leaving edges respectively
- Weighted Graph
  - Each edge has an assigned weight
  - The weight can mean anything (distance, capacity, etc)

Graph Representation

- Adjacency Matrix
  - An adjacency matrix is a matrix with a position for each possible connection between two vertices
  - It is an n-by-n matrix, where n is the number of vertices in the graph
  - Each position stores a boolean value, representing whether there is a connection between the two vertices
  - For a weighted graph, the value is either the weight of the edge, or infinity
  - For a directed graph, the value remains 0 or 1, but is a 1 for the row->column
  - The size of the graph must be known in advance, and duplicate edges cannot be stored
  - If a graph is sparse, a significant part of the matrix will be empty
  - $O(1)$ efficiency for all edge operations, $O(n^2)$ to insert or delete a new vertex
- Adjacency List
  - A set of SLLs, one for each vertex
  - Each SLL contains all vertices which are adjacent to the vertex, stored at the head
  - One SLL for each vertex in an undirected graph, or 2 for a directed graph
  - The size of the graph can dynamically change, allows for duplicated edges
  - For an undirected graph, each edge is stored twice
  - Space efficient for a sparse graph, less so for a dense graph
  - $O(n)$ efficieny to search for or delete an edge, $O(1)$ to insert an edge at the head, $O(1)$ to insert a new vertex, $O(n)$ to delete a vertex

Traversal

- Depth First Search (DFS)
  - Traverse as deep as possible until a dead-end is found
  - Backtrack to find another dead-end
  - Repeat until all paths are found
  - Pesudocode implementation (each vertex starts as being marked ”unvisited”)

```
push vertex onto stack
mark as visited
while stack not empty {
  pop vertex off stack
  process the vertex
  for each adjacent unvisited vertex {
    push vertex onto stack
    mark as visited
  }
}
```

- Breadth First Search (BFS)
  - Find all adjacent vertices (fan out as much as possible)
  - Fan out on all adjacent vertices until every vertex is visited
  - Pesudocode implementation (each vertex starts as being marked ”unvisited”)

```
enqueue vertex onto queue
mark as visited
while queue not empty {
  dequeue vertex off queue
  process the vertex
  for each adjacent unvisited vertex {
    enqueue vertex onto queue
    mark as visited
  }
}
```

Applications of Graphical Data Structures

Greedy Algorithms

- A decision is made at each stage which seems good at the time, with no consideration for it's consequences
- Once this decision is made, it is never reconsidered
- It is hoped that a series of relative good decisions leads to globally optimal solution
- Greedy algorithms do not consistently find the optimal solution, as they are unable to operate on all of the data, only that which they considered when making local decisions

Shortest Path Trees

- A shortest path tree is a tree (subgraph) that contains all of the vertices and a subset of edges, where the path from the root to any other vertex has the shortest distance
- One vertex is chosen as the root, which could be strategic, or arbitrary
- This is useful for route planning algorithms for road maps, airlines, etc

Dijkstra's Algorithm

- Finds all of the shortest paths from a given vertex of a graph to all other vertices at once
- Sometimes known as the 'single-source shortest paths algorithm'
- The method goes as follows
  - Place the source vertex, S at the root of the shortest path tree
  - For each vertex, V_i that has a single edge from S, the distance is stored and V_i is added to a priority queue
  - All other vertices are given a distance of infinity
  - Perform this loop until the priority queue is empty
    - The vertex in the priority queue with the shortest distance from S, V_s is selected and added to the shortest path tree
    - For each vertex V_i adjacent to V_s that has a single edge from V_s to V_i, the distance from S to V_i is calculated
    - If this new value is less than the value already stored, update the value and add the vertex V_i to the priority queue

Spanning Trees

- A graph can contain multiple paths between vertices
- A spanning tree is a tree (subgraph) which contains all of the vertices with only one path between each pair of vertices
- Any one graph can have many spanning trees
  - BFS of a graph gives a breadth-first spanning tree
  - DFS of a graph gives a depth-first spanning tree
- This is useful for planning routes to use the minimum resources, such as a telephone network between towns, connections between terminals on a circuit board, etc

Minimum Spanning Tree

- There can be several spanning trees for any given graph, but in a weighted graph you can have more and less efficient spanning trees
- The minimum spanning tree is a specific spanning tree which has the lowest total weight
- To find the minimum spanning tree, every other spanning tree must also be known and compared
- This is useful for planning a new network, with the weight of each edge representing it's cost

Prim's Algorithm

- Builds upon a single partial minimum spanning tree
- At each step, add an edge connecting the nearest vertex which is not already in the spanning tree
- Pseudocode implementation

```
while not all vertices in tree {
  examine all vertices in the graph with an endpoint in the tree and the other no
  find the shortest edge and add to the tree
}
```

Kruskal's Algorithm

- Combines sets of partial spanning trees
- At each step, add the shortest edge in the graph whose vertices are in different partial minimum spanning trees
- Pseudocode implementation

```
while tree is incomplete and there are edges to add {
  find the shortest edge which hasn't been considered yet
  if it's endpoints are in different trees {
    join the trees
  }
}
```