**Data Structures and Algorithms**

Linked List

free list

Skip List

**Binary Tree**

A binary tree is a hierarchical graph structure with nodes falling in layers called **level**s. A binary tree is composed of **nodes** and **arcs** (edges). An empty structure, which can be any empty graph, is also an empty tree. The following rules apply to define a binary tree.



Each node in a binary tree can be stored in an array, and call it arr. Let the root node be on level 1, located at arr[0]. That means level 2 has two (2) nodes at arr[1] and arr[2]. The following formulas are helpful in determining the index in arr for a specific node and count of nodes in a binary tree. Also assume the tree is a complete binary tree unless stated otherwise.



**Binary Tree Traversal Algorithms**

**Traverse By Breadth**

Starting at level 1, each level will line up in queue, and the nodes in each level will be visited 1-by-1. An array (or a similar container) will hold the visited nodes.

*Let arr be an array that holds visited nodes and will append a new node when visit is called.*

*Let Q be a queue that holds the same data type as each node.*

1. *Push the root node onto Q.*

*While Q is not empty:*

1. *Pop the front element of Q and call it F.*
2. *IF F’s left child exists:*

*push it onto Q.*

1. *IF F’s right child exists:*

*push it onto Q.*

1. *Visit F (append F to arr).*

*Once Q is empty:*

1. *Return arr.*

**Traverse by Depth**

A pointer or node placeholder starts at the root and keeps going to deeper levels while trying to be as leftward as possible at each fork. Anytime a dead end is reached, retraces back up until an unvisited path opens. This repeats until all paths from the root to each leaf has been traveled. Three (3) variations of traverse by depth exist, depending on the order of performing the following actions: go to left child, go to right child, visiting current node. All three (3) variations use a recursive function. The algorithm below only illustrates the LRV or postorder variation.

*Let arr be an array to hold visited nodes.*

*Let p be a placeholder for a node, and initially set it to root node.*

1. *while p has unvisited left child:*

*p = left child*

1. *while p has unvisited right child:*

*p = right child*

1. *visit p (append current node to arr)*
2. *IF p is NOT the root:*

*p = most recently visited node’s,*

*repeat Steps 1 – 4*

*ELSE*

*return arr*

The three variations in depth-first traversal have the following names.

1. preorder: VLR
2. inorder: LVR
3. postorder: LRV

**Insertion in Binary Tree in Level Order**

For a regular, unspecialized binary tree, insertion is easy. To keep the binary tree balanced after insertion, the new node will be appended at the at the first opening in the sequence from the root to the bottom rightmost node. A queue will be used to detect any openings (nodes lacking a left or right child).

*Let Q be a queue for nodes, initially empty. Let N be a new node holding any value.*

*IF empty tree:*

*Root = N.*

*Exit algorithm.*

1. *Push the root onto Q.*

*WHILE Q is NOT empty:*

*top = Q.pop.*

1. *IF top has no left child:*

*Make N its left child.*

*Exit algorithm.*

*ELSE:*

*Push top’s left child onto Q.*

1. *IF top has no right child:*

*Make N its right child.*

*Exit algorithm.*

*ELSE:*

*Push top’s right child onto Q.*

**Deletion in Binary Tree in Level Order**

**Inorder Traversal of Single-Threaded Binary Tree**

A threaded tree presents a more efficient way of traversal by depth without the need for a stack. This algorithm works for all three variations, but the following pseudocode exemplifies inorder binary tree traversion. An important term, the **inorder successor** of a node is the next node to be visited in sequence using inorder (LVR) traversal by depth. In a right-threaded tree, the inorder successor is the right child if it exists or another node linked by a thread. In a left-threaded tree, the inorder successor is the left child if it exists or another node linked by a thread. The binary tree in this algorithm is right-threaded, as the defacto used in binary tree traversal.

*Let arr be an array to hold visited nodes.*

*Let p be a placeholder for a node, and initially set it to root node.*

*IF empty tree*

*Return empty arr.*

*ELSE IF root has no children:*

*Return arr = {root}.*

*WHILE p has right child XOR p has a thread:*

1. *WHILE p has a left child:*

*p = p’s left child.*

1. *Visit p (append current node to arr).*
2. *IF p has a right child:*

*p = p’s right child.*

*ELSE IF p has a thread:*

*p = other node connected by thread.*

*Visit p (append current node to arr).*

*Repeat this step.*

**Convert Binary Tree to Single-Threaded Binary Tree**

The following is a recursive method for attaching threads to nodes in a binary tree.

*Let p be a placeholder for a node, and initially set it to root node.*

*IF empty tree OR root has zero (0):*

*Exit algorithm.*

*ELSE:*

1. *IF p has left child:*

*p\_2 = Steps 1 – 3 with p initially set to this.p’s left child*

*p\_2 adds thread connecting to p\_2’s parent.*

*\*any boolean flag or indicator of a thread is turned on.*

1. *IF p has right child:*

*Return Steps 1 – 3 with p initially set to this.p’s right child.*

1. *Return p.*

**Insertion into Threaded Binary Tree**

Suppose a right-threaded binary tree needs to append a new node somewhere in the structure. The algorithm below decides whether to add a thread and to which node a thread should be linked to.

*Let S be a stack to hold nodes.*

*Let p be a placeholder for a node, and initially set it to null.*

*Insert new node anywhere in binary tree using any desired algorithm, and let new node be N.*

*IF N == root node:*

*Exit algorithm.*

*WHILE traversing by depth using postorder (LRV):*

1. *IF encountered node has a left child:*

*Push node onto S.*

1. *IF encountered node == N:*

*p = N.*

*Break (Exit WHILE loop).*

*After WHILE loop exits:*

1. *top = S.pop.*
2. *Append new thread from p (holding N) to top.*

*IF p’s parent has a thread to top:*

*Remove that thread.*

**Verification for Perfect Tree**

A **perfect tree** is a binary tree where each non-leaf node must have two (2) children and all the leaves fall on the same layer. Therefore, a perfect tree is also complete and balanced. The algorithm below returns a Boolean indicating whether a certain binary tree fits the criteria to be perfect.

*Before starting the algorithm, find the height of any path in the binary tree.*

*Let P be a placeholder for a node and is initially set to root node.*

*FUNCTION findAnyPathLength(P){*

*COUNTER = 0.*

*WHILE(){*

*IF()*

*P = P’s left child.*

*ELSE IF()*

*P = P’s right child.*

*++COUNTER.*

*}*

*RETURN COUNTER.*

*}*

The following function is a recursive function meant to verify whether a binary tree is perfect or not. If any intermediate return value is false, the final return value will automatically be false.

Let LEVEL be the indexed row of T that P is inside. For example, T.root is on level 0.

HEIGHT = findAnyPathLength(). //supposed height if T is perfect

*FUNCTION VerifyPerfectTree(P, LEVEL, HEIGHT){*

*IF()*

*RETURN false.*

*RETURN VerifyPerfectTree( AND*

*VerifyPerfectTree().*

*}*

The actual algorithm to determine whether a binary tree is perfect or not is below.

*Let T denote the tree.*

*IF(*

*)*

*RETURN false.*

*ELSE IF(T is empty OR P has no children)*

*RETURN true.*

*ELSE*

*VerifyPerfectTree(T.root).*

**Verification of Full Binary Tree**

A full binary tree is a binary tree in which each node that is not a leaf must have two (2) children. This is one rule less strict than a perfect tree; a full tree does not necessarily need to fill each level completely, and the height of an subtree does not matter.

Let T denote the tree.

*Let Q be a queue to hold nodes, initially empty.*

*IF()*

*RETURN*

*ELSE:*

1. *Push root onto Q.*

*WHILE Q is NOT empty:*

1. *top = Q.pop.*

*IF top has 2 children:*

*Push both children onto Q.*

*ELSE IF top has only 1 child:*

*RETURN false.*

*RETURN true.*

**Binary Search Tree Algorithm**

A BST is a special binary tree holding numerical values such that the left child is lower than the parent and the right child is higher than the parent when comparing their values. The algorithm to search for a specific value is below.

*Let p be a placeholder for a node and is initially set to root node.*

*WHILE p is not null:*

1. *IF p == val:*

*Return address of val.*

1. *IF p is smaller than val:*

*p = the right child of current node.*

*Continue (Skip Step 3).*

1. *IF p is larger than val:*

*p = the left child of current node.*

*IF p is never found (p == null):*

1. *Return null.*

**Insertion in Binary Search Tree**

The following algorithm inserts a new node into a BST. This assumes that each node has a unique, unrepeated value in the tree.

*Let the new node hold a numerical value of val.*

*Let p be a placeholder for a node and is initially set to root node.*

*IF BST is an empty tree:*

1. *Make new root set to val.*

*WHILE p is NOT NULL:*

1. *IF p’s value is smaller than val:*

*p = p’s right child*

1. *IF p’s value is larger than val:*

*p = p’s left child*

*IF p’s parent has a value smaller than val:*

1. *P’s previous node creates new right child with value val.*

*ELSE*

1. *P’s previous node creates new left child with value val.*

**Deletion in Binary Search Tree**

.

*\*Note that complexity above does not include searching for the node first, just removing the node. BST is assumed to be a graph, not an array of elements.*

*Let N be the node to be deleted.*

*IF N is a leaf:*

*Remove N.*

*ELSE IF N has only 1 child:*

*Replace N with its child along with the rest of the child’s offspring.*

*ELSE:*

*Replace N with a copy of its inorder successor.*

*Run this algorithm on the actual inorder successor node.*

**Deletion in BST By Merging**

This presents an alternative way to delete a node in a binary search tree if the deleted node has two (2) children. Simply put, this is the 3rd scenario in the algorithm above but replacing the deleted node with the rightmost node in the left subtree rather than the inorder successor.

*Let N be the node to be deleted, and assume that N has two (2) children.*

*Let p be a placeholder for a node and is initially set to N.*

1. *R = p’s left child.*

*WHILE R has a right child:*

*R = R’s right child.*

1. *Change p to a copy of R.*
2. *Remove actual R using the deletion algorithm above this one.*

**Convert to BST to Single-Threaded BST using Stack**

This algorithm utilizes the fact that numbers are sorted at each split in a BST to add right threads.

*Let p be a placeholder for a node and is initially set to root node.*

*Let S be a stack to hold nodes.*

*IF empty tree OR root has no children:*

*Exit algorithm.*

*ELSE*

*Push root node onto S.*

*WHILE S is NOT empty:*

1. *WHILE p has left child:*

*p = p’s left child.*

*Push p onto S.*

1. *IF p has right child:*

*p = p’s right child.*

*Push p onto S.*

*Go back to Step 1.*

1. *top = S.pop.*

*other\_node = S.pop.*

*WHILE other\_node.value < top.value:*

*IF S is empty:*

*Exit algorithm.*

*other\_node = S.pop.*

1. *Create new thread between top AND other\_node.*
2. *p == other\_node (located in BST, not S).*
3. *IF p has no right child AND p is NOT root node:*

*Push p onto S.*

*Repeat Steps 3 – 5.*

1. *Go back to Step 2.*

**Balancing a Binary Tree**

A binary tree is classified as **height-balanced** or simply **balanced** if the difference in height between the two (2) subtrees of any node is no more than one (1). A **perfectly balanced** tree is a balanced tree with leaves on levels differing no more than one (1). The following algorithm tests whether a binary tree is balanced.

*As a prerequisite, the following algorithm will be used to determine the height of a tree or subtree. Denote it as “findHeight”.*

*Let p be a placeholder for a node and is initially set to root node.*

*IF empty tree:*

*Return 0.*

1. *IF p has any children:*

*Return 1 + max(Steps 1 but with p = p’s left child, Steps 1 but with p = p’s right child).*

*ELSE:*

*Return 1.*

*The algorithm to test for a balanced tree is below.*

*Let p be a placeholder for a node and is initially set to root node.*

1. *IF empty tree OR root has no children:*

*Return true.*

1. *IF abs(findHeight(p’s left child) – findHeight(p’s right child)) <= 1:*

*Return (Steps 1 – 2 with p = p’s left child) AND (Steps 1 – 2 with p = p’s right child).*

*ELSE:*

*Return false.*

*Exit algorithm.*

**AVL Tree**

**Red Black Tree**

2-3 Tree

**More Application-Based Trees**

B Tree

B+ Tree

Cartesian Tree

R Tree

This type of tree is used for partitioning a group of objects in some plane (if 2D) or some finite space (if 3D) into rectangular regions. A partition can be made by a **bounding box**, which is just a rectangular field, that completely covers 1 or more objects. For a specific partition, the smallest possible rectangle that completely contains all elements inside the partition is a **minimum bounding box (MBB)**. A simply test whether an arbitrary bounding box X is a member inside another bounding box Y is given below. Both conditions below must be met.

In an R tree, nodes are often called pages. Each **page** is an array containing all MBBs immediately contained by the common parent; that is, if any child node has even smaller MBBs and/or objects, these are not part of the page array that makes up the child node. From a simple summary, each R tree has 1 **root page**, any intermediate nodes called **branch pages**, and pages with no further inner MBBs called **leaf pages**.

**Search for Object**

Algorithm searches for a specific object, starting the search at a given node N.

FUNCTION searchForObject(){

IF()

FOR\_EACH()

IF(){

RETURN searchForObject().

BREAK.

}

ELSE

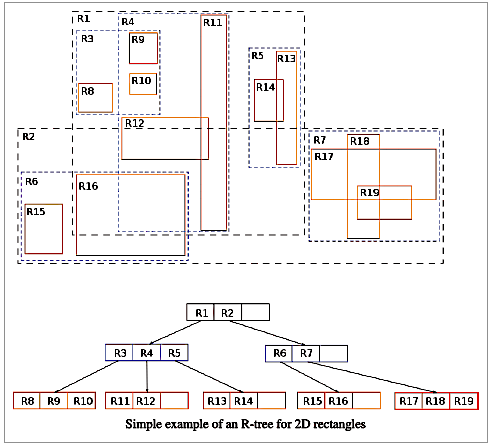
FOR()

IF()

RETURN reference to object.

}

Insertion



Quad Tree

Wavelet Tree

**Heaps**

**Binary Heap**

A commonly applicable heap, binary heaps are further classified into two (2) subgroups. A **max heap** is a binary tree where any node holds a larger value than any children. A **min heap** is just the opposite; any node holds a smaller value than any children. On each level for both types, sibling nodes do not have to be in any particular order.

**Heapify a Non-heap into Binary Max Heap**

The following algorithm swaps nodes in any binary tree and converts to a binary max heap. The sub-algorithm below swaps the top node of a subtree with its largest child if that child holds a larger value. This repeats with a new subtree descending from the previous root and continues until that node’s value is larger than either of its children’s values. Let the process be “heapify”.

*Let p be a placeholder for a node and is initially set to root node.*

*FUNCTION HeapifyLocal(){*

*L = largest node out of {p, p’s left child, p’s right child}.*

*Swap p with L.*

*IF(p changed position within tree){*

*p = parent of current p.*

*HeapifyLocal(p).*

*}*

*ELSE*

*RETURN. //exits algorithm*

*}*

The actual algorithm to turn the binary tree into a max heap is below.

*Let H be the index of the furthest node; , where n is the no. of nodes in the binary tree, implying n-1 is the index of the furthest node.*

*FOR (i = H; i >= 0; i--):*

HeapifyLocal().

**Heap Sort**

This algorithm sorts an array of numbers by using a heap. The array is split into two (2) partitions: an unsorted left partition and a sorted right partition initially empty. Repetitively, a max heap is created out of the unsorted partition to yield the max number in the unsorted partition (top node). This value is then transported to the front of the sorted partition.

*Let arr1 be an expandable array of numbers. Let arr2 be an empty and expandable array. These arrays represent the unsorted and sorted partitions. Let T=binary tree constructed from arr1.*

*WHILE arr1.size > 0:*

1. *Convert T to max heap.*
2. *Append top node to front of arr2.*
3. *Remove the same node from arr1.*

**Treaps**

A **treap** is a cross between a binary search tree and a heap, usually a binary max heap. In a treap, each node holds two (2) values, a key and a priority. The nodes are sorted into a binary search tree according to the key and also heapified according to the priority without violating the BST rule. A left child must have a smaller key than the parent, and the right child must have a larger key than the parent.

**Priority Queue**

A queue that has its elements ordered according to ranking is called a **priority queue**. Each element is associated with a rank or priority. A higher priority element is located closer towards the front than a lower priority element. That is, the highest priority element will be dequeued first. If two or more elements tie in priority, the first of them to be dequeued depends on the group’s relative order.

**Other Sorting Algorithms**

**Shell Sort**

Shell sort uses several iterations of comparisons by comparing elements located a certain apart. During each iteration, each element located at an index is swapped with its partner element, located at , if the earlier element is larger. The algorithm is below.

Let n denote the length of the original array or list of numbers, arr.

*FOR():*

*FOR():*

*temp = arr[i].*

*FOR():*

*IF():*

*SWAP().*

There are many variations of gap decrements. The most common one, as used above, is the original by Donald Shell. Some other types of increment sequences for gap, especially **Hibbard’s increments**, make the whole sorting process much more efficient. They are given below along with time complexity.

**Divide-and-Conquer Paradigm**

This is not a specific sorting algorithm but a general strategy that some algorithm can adopt. Especially for sorting, the general steps below lay out a foundation for the algorithm.

**Merge Sort**

This sorting algorithm uses the divide-and-conquer paradigm to sort a list of elements in ascending order. Obviously, recursion will be needed. The algorithm differs somewhat in certain steps when using an array versus a list of elements. The steps below illustrate merge sort using a list. First, a function called MERGE used in merge sort is defined.

*MERGE(arr1, arr2, arr):*

*Let arr1 and arr2 be 2 lists of numerical values, and let arr be an empty list.*

*i1 = 0, i2 = 0.*

*WHILE(both S1 and S2 has elements not in S):*

*IF():*

*Append arr1[i1++] to arr.*

*ELSE:*

*Append arr2[i2++] to arr.*

*Load the rest of the larger list between arr1 and arr2 into arr.*

*RETURN arr.*

*MERGE\_SORT(arr)*

*IF():*

*RETURN arr.*

*mid = .*

*arr1 = MERGE\_SORT(partition of SO containing indices [0, mid]).*

*arr2 = MERGE\_SORT(partition of SO containing indices [mid + 1, arr.length - 1]).*

*RETURN MERGE(arr1, arr2, some empty list).*

**Quick Sort**

*QUICK\_SORT(): //start and end denote the inclusive interval of a list or array to sort, if the whole list is not desired to be sorted*

*Pick a pivot or boundary element, and denote it with P.*

*lowerBound = start + 1, upperBound = last.*

*SWAP().*

*bound = arr[first].*

*WHILE():*

*WHILE():*

*++lowerBound.*

*WHILE():*

*++upperBound.*

*IF():*

*SWAP().*

*SWAP().*

*IF():*

*QUICK\_SORT().*

*IF():*

*QUICK\_SORT().*

**Bucket Sort**

**Bucket sort**, or **bin sort**, is especially useful for a roughly uniform distribution of numbers such as floating point values. Essentially, a collection of buckets are ordered and spaced equally apart. Each bucket will append elements from the to-be-sorted array only for elements that fit the bucket’s ranking criteria. For example, suppose a bucket allows numbers in the range . Then, the next adjacent buckets, in ascending order, will allow numbers , etc. Sometimes, a constant will be multiplied to all elements to the array before entering into any buckets. Even though 0.00012425, 0.00045464, 0.00056789, and the like appear to be in close proximity to one another, they are far apart when looking from a microscopic context. Hving will allow each number to be filed into buckets based on the 1s digit.

**Complexity Analysis**

Time complexity depends on the sorting algorithm used on each bucket if multiple elements share common buckets. When an element is placed into a non-empty bucket, the elements within said bucket need to be sorted. Using binary sort per bucket will make the whole bucket sort optimum. The initial setup of buckets also affects the total time complexity. At the expense of space, if there are way more buckets than elements from the array, each non-empty bucket will likely have only 1 element. This is simply linear time complexity as each element from the array is assigned a unique bucket. The table below summarizes specific time complexity based on several scenarios.

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Comments** | **Average Time Complexity** |
| Uneven distribution, elements spaced far apart | Occupied buckets only end up with 1 element, while most buckets are empty |  |
| Uneven distribution, elements relatively close, using binary sort per bucket | Some buckets have >1 elements, some buckets have 1 element, and a few empty |  |
| Perfectly uniform distribution, using binary sort per bucket | Each bucket holds approximately equal number of elements, save -1 or +1 differences between a few |  |
| Perfectly uniform distribution, using sorting algorithm per bucket | Each bucket holds approximately equal number of elements, save -1 or +1 differences between a few |  |

*BUCKET\_SORT(arr):*

*Let B denote an array of empty buckets (Each bucket is a source for a single list).*

*Let be some constant such that .*

*FOR():*

*IF():*

*B[] = arr[i].*

*ELSE:*

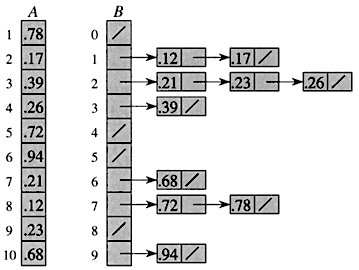
*INSERTION\_SORT().*

*index = 0.*

*FOR():*

*FOR():*

*arr[index] = B[i][j].*



In the picture above, a constant is chosen such that is a floating point number with a zero or maybe nonzero tenth digit but always nothing left of the decimal point.

**Counting Sort**

This type of sorting can be done on any type of distinct objects, such as integers, characters, and enumerable objects. The time complexity is the same for both worst-case and average-case analyses. Therefore, this algorithm is stable in nature.

**Radix Sort**

This sorting algorithm, useful for sorting both integers and strings, consists of multiple rounds of sorting, each dedicated to comparing the digit (or character) for . The constant is the number of digits (or characters) of the longest key in the array. A **key** is simply a synonym for an key in the to-be-sorted array. Note that this sort is stable; during any intermediate sorting, relative ordering of any set of keys not being relocated stays the same. There are 2 variations of radix sorting, outlined below. For demonstration purposes, an array of integers will be sorted.

**Radix Sort (LSB)**

This algorithm works well with integers more than 1 digit long; otherwise, a bucket sort will be the result if all keys are single digits. This version starts comparing the very last digit (LSB) of each key first. Iteratively, each digit closer to the most significant digit (MSB) is compared until the longest key’s first digit has been compared and sorted accordingly. During the round of comparisons, shorter keys that do not have a nonzero MSB for comparison will use 0. A list of bucket will be used for each round of sorting. The algorithm is below.

*shift = 1.*

*FOR(){*

*FOR(){*

*bucketIndex = .*

*B[bucketIndex].append(arr[i]).*

*}*

*arr = combine buckets in order.*

*Clear all buckets.*

*shift \*= 10.*

*}*

**Radix Sort (MSB)**

Sorting from the leftmost digit (or character) first to the last digit (or character) of the longest key is the only major distinguishing difference between radix sort using MSB vs LSB. For demonstration purposes, an integer array will be sorted. For the first iteration, each key is dropped into a specific bucket matching the most significant digit (MSB). Then, each bucket will act as an independent new array and be resorted into a new list of buckets, this time based on the 2nd digit. This continues until the subarray with the longest key(s) from the original array finds a bucket for the LSB. Every iteration is done stably in respect to the previous iteration; that is, no relative positions are changed between any one key from a specific subarray in the current iteration and any other key from a different subarray. At the end, each buckets holds only 1 key unless there were duplicates.

*FUNCTION radixSortMSB(){*

*IF()*

*RETURN arr.*

*FOR\_EACH(){*

*.*

*B[bucketIndex].append(arr[i]).*

*}*

*Remove all empty buckets from B.*

*listOfArrays = [].*

*FOR\_EACH()*

*listOfArrays.APPEND(radixSortMSB(AS\_ARRAY())).*

*RETURN listOFArrays.*

*}*

**Comparing Radix Sort using MSB vs LSB**

|  |  |  |
| --- | --- | --- |
| **Comparison** | **MSB** | **LSB** |
| Worst Case Time Complexity |  |  |
| Total time used when keys have differing lengths | Count of digits for entire array  (faster) | Number of keys \* Longest key length  (slower) |
| Memory Usage | Relatively much | Relatively little |
| Popularity of Usage | Relatively little | Relatively much |

**Cycle Sort**

This is an unstable, comparison-based sorting algorithm that modifies an array in place. The algorithm minimizes the number of memory writes done to the array. Each element in the array is either modified once to correct ordering, or if already in correct position, is not touched at all. In summary, there is an index that compares its element to every element after itself. The number of times a smaller element is passed indicates the correct index for past in a sorted array; in short, . If , then is already the correct location by coincidence for the element . If not, then elements at indices are swapped places. The cycle will be repeated at the same index until the correct in the sorted version of is swapped into this spot. Then and only then will start move 1 forward.

FOR(){

\*{location = start.

FOR(){

IF(){

++location.

}

IF()

continue.

WHILE()

++location.

SWAP().

}

WHILE()

REPEAT().

}

Bitonic Sort

Cocktail Shaker Sort

*swapped = false.*

*start = 0.*

*end = n – 1.*

*DO\_WHILE(){*

*FOR()*

*IF(){*

*SWAP().*

*swapped = true.*

*}*

*IF()*

*RETURN.*

*swapped = false.*

*--end.*

*FOR()*

*IF(){*

*SWAP().*

*swapped = true.*

*}*

*}*

Gnome Sort

*index = 0.*

*WHILE(){*

*IF()*

*++index.*

*IF()*

*++index.*

*ELSE{*

*SWAP().*

*--index.*

*}*

*}*

Stooge Sort

Tim Sort

**Search Algorithms**

**Binary Search**

This search uses the same concept as finding a specific node in a binary search tree (BST) but with a list or array. 2 bounds, L and H, are initialized to the first and last element indices. A middle index P is the average of L and H. With arr divided into 2 subarrays each iteration, the new outer bounds are set to the first and last element indices of either one subarray that must contain x, the target searched value.

*Let arr be a sorted array of n uniformly distributed values, and x is the searched value.*

*L = 0.*

*H = n – 1.*

*P = 0.*

*IF()*

*RETURN H.*

*WHILE(){*

*P = .*

*IF()*

*RETURN P.*

*IF()*

*L = P.*

*ELSE*

*H = P.*

*}*

*RETURN -1.*

**Interpolation Search**

This search works best if all elements combine to form a perfectly uniform distribution across an array size n. Basically, the distance from a lower bound index is calculated based on the value of x, the element to be searched. The larger x is compared to the lower bound index, the farther ahead index will be. If arr is a perfectly even uniform distribution, then every unit increase in index will correspond to a constant increase in element value. This is basically a slope.

*Let the probe position formula be defined below.*

*Let arr be a sorted array of n uniformly distributed values, and x is the searched value.*

*L = 0.*

*H = n – 1.*

*P = 0. //just an initialization*

*WHILE(){*

*IF(){*

*IF()*

*RETURN L.*

*RETURN -1.*

*}*

*P = POS(x, H, L).*

*IF()*

*L = P + 1.*

*ELSE*

*H = P – 1.*

*}*

*RETURN -1. //in case WHILE loop breaks before confirming absence of x in the 1st IF statement*

**Complexity Analysis**

If arr is equivalent to a uniform distribution of element values, the time complexity is . Otherwise, the worst case time complexity for any randomized sorted array is .

Jump Search

*STEP = .*

*PREV = 0.*

*WHILE(){*

*PREV = STEP.*

*STEP += .*

*IF()*

*RETURN -1.*

*}*

*FOR()*

*IF()*

*RETURN i.*

*RETURN -1.*

**Graph Algorithms**

\*Please refer to Graph Theory reference sheet for structure and properties of different graphs

**Dijkstra’s Shortest Path**

This is a greedy algorithm that generates a spanning tree to include the shortest path from a source vertex to every other node in graph G. The term “distance” from vertices u and v is the sum of weights encountered in the path, NOT the number of edges traversed. The algorithm is below.

*Let S be a tree, initially empty.*

*IF empty graph:*

*Exit algorithm.*

1. *Choose the desired source vertex and add it to S, and s = source vertex.*
2. *Out of all vertices not in S, choose the vertex v with the minimum distance to s, and that to S with the edge part of the minimum distance path between s and v.*

*Note that like in Prim’s Algorithm, v must be a vertex not in S but is one edge away from any vertex in S.*

1. *Repeat Step 2 until all vertices in G are included in S.*

*\*To find the shortest path from source vertex s to a particular chosen vertex t in G, stop the algorithm when t is added to S.*

**Kruskal’s Minimum Spanning Tree**

Given a simple weighted undirected graph G, there are numerous possible spanning trees. A **spanning tre**e of G is a graph with no cycles that contains all the vertices of G. The **minimum spanning tree** is the spanning tree with the smallest sum of edge weights compared to all other possible spanning trees.

*Let S be the minimum spanning tree, initially empty.*

*IF empty graph:*

*Exit algorithm.*

1. *Sort E, the list of edges, in place by ascending order by edge weight.*
2. *e = the smallest edge (front element) in E.*

*IF the e DOES NOT complete a cycle when added to S:*

*Add e to S and include each endpoint once in S.*

*Remove e from E.*

1. *Repeat Step 2 until E is empty.*

*Return S.*

**Prim’s Minimum Spanning Tree**

This algorithm utilizes the fact that all vertices of a graph G connects to another vertex via an edge in a minimum spanning tree S. When an edge not in S connects to a vertex v in S, the edge must be the smallest edge out of all edges that v connects with in G. Even more precise, that same edge must be the smallest out of all edges adjacent to any vertex in S. The algorithm is below.

*Let S be the minimum spanning tree, initially empty.*

*IF empty graph:*

*Exit algorithm.*

1. *Pick a starting vertex and add it to S.*
2. *Out of all edges not in S connected to any vertex in S, pick the smallest one and add it to S.*
3. *Repeat Step 2 until all vertices in G are included in S.*

*Return S.*

**Breadth-First Search**

*Let G be an actual object-oriented graph structure or an array with elements being vertices, and each vertex must have a way of knowing its immediately adjacent vertices.*

*Let VISITED be a container of all vertices previously visited already.*

*Let S1 be any vertex chosen to be the starting vertex.*

*Let v.ADJ be a container of all immediately adjacent vertices for some arbitrary vertex .*

*Let Q be a queue that has the same data type as G.*

*Push S1 onto Q.*

*Append S1 to VISITED.*

*WHILE(){*

*S = Q.pop.*

*Print(S). //Optional*

*FOR()*

*IF(){*

*Push S.ADJ[i] onto Q.*

*Append S.ADJ[i] to VISITED.*

*}*

*}*

\*Note: if the ID for the vertices are evenly spaced numbers that can be mapped to the integer sequence , then an array of pairs and a counter will fit the situation very well. Suppose the array below with a helper counter is used.

counter = 1 //counter initialized to 0 before any operations

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| S1 | null | null | null |  | null | null | null |
| false | false | false | true |  | false | false | false |

The array of pairs have 2 mutually independently rows that each serve a different function. The top row holds copies of vertices in the order they are visited by the algorithm. Each time a vertex is visited, a copy replaces null at index in the top row. Regardless of the order of traversal, the bottom row corresponds to the mapped 0-based integer ID and denotes whether the vertex with that ID has been visited or not. In the above example, the starting vertex S1 is the first to be visited and has ID = 3. That is why the vertex object is first in top row, and boolean value at index 3 of bottom row is set to true.

**Depth First Search**

*Let G be an actual object-oriented graph structure or an array with elements being vertices, and each vertex must have a way of knowing its immediately adjacent vertices.*

*Let VISITED be a container of all vertices previously visited already. Specifically for this case, VISITED is a static variable.*

*Let S1 be any vertex chosen to be the starting vertex.*

*Let v.ADJ be a container of all immediately adjacent vertices for some arbitrary vertex .*

*Let a function be defined as:*

*FUNCTION DFSGraph(){*

*Append v to VISITED.*

*FOR(){*

*IF()*

*DFSGraph().*

*}*

*Simply call DFSGraph().*

**Topological Sort**

The graph to be sorted must be a directed acyclic graph (DAG), which have directed edges but does not form any directed cycles. This type of sort orders a sequence of vertices such that for every directed edge , comes prior to in the sorted sequence. One way to test whether a graph is a DAG is by finding one or more vertices with zero (0) incoming edges, meaning . If each vertex has both incoming and outgoing edges, then . There are 2 main methods of topological sort.

**Depth First Search (DFS) with Stack**

*Let G be an actual object-oriented graph structure or an array with elements being vertices, and each vertex must have a way of knowing its outgoing edges and corresponding neighbor vertices.*

*Let VISITED\_STACK be an empty stack and a static variable.*

Let ROOT be a list of all vertices , meaning zero (0) vertices point to .

FOR()

DFSGraph() but replace append to VISITED with push onto VISITED\_STACK.

**Kahn’s Algorithm**

Assume that the in-degree, that is, , has been computed prior.

Let ROOT denote a queue this time, holding only vertices with no incoming edges. The order the vertices are enqueued does not matter.

*Let v.ADJ be a container of all immediately adjacent vertices for some arbitrary vertex*

WHILE(){

Q = ROOT.dequeue.

FOR(){

Decrement by 1.

IF()

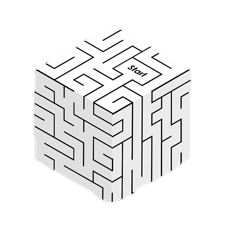
ROOT.enqueue(v).

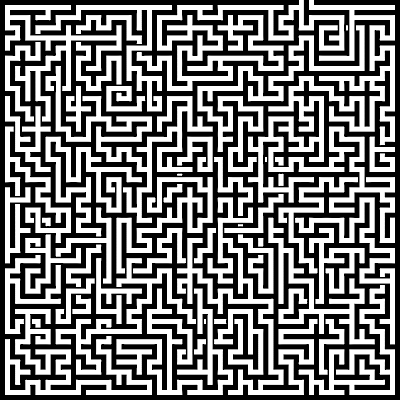
}

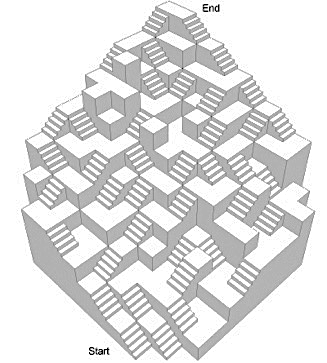
}

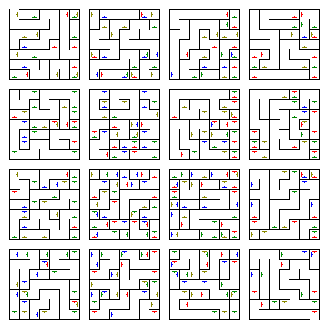
**Maze and Algorithms**

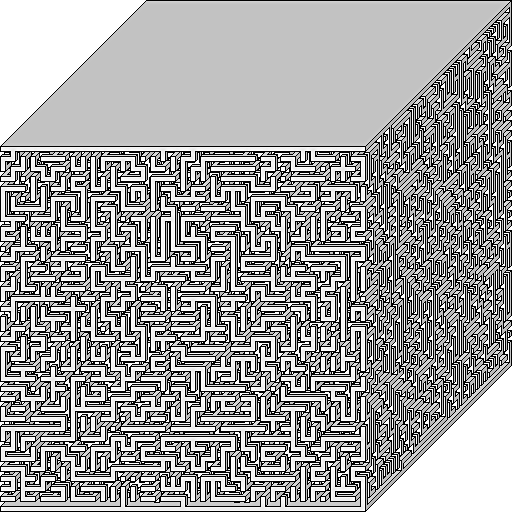
Types of Mazes (Dimensionality)











**Algorithms for 2D Mazes**

**Pledge Algorithm**

**Trémaux’s Algorithm**

*Let S and E stand for start and end of a maze M. Each discrete passage in M must be able to hold in memory an integer MARKING, initialized to 0.*

*Let P be the player, a dot representing the current location inside of M.*

*Let CURRENT denote the current continuous path that P is inside.*

*WHILE(){*

*Never enter a path with MARKING == 2.*

*++CURRENT.MARKING.*

*IF()*

*IF(some passage p at junction has p.)*

*Go into p.*

*ELSE IF(){*

*++CURRENT.MARKING.*

*Go back along CURRENT.*

*} ELSE IF()*

*Go into passage with minimum value for MARKING.*

*ELSE*

*RETURN -1. //M has no E (exit)*

*ELSE IF(){*

*CURRENT.MARKING == 2.*

*Go back along CURRENT.*

*}*

*}*

Dead-End Filling

Maze-Routing Algorithm

**Games and Puzzles**

**Boggle Algorithm**

**String Algorithms**

Levenshtein Distance

Damerau-Levenshtein Distance

Jaro Distance

**BK Tree**

**Tango Tree**