**Arrays**

Stored as a contiguous block in memory. This is why the length need to be defined creating a static array. Arrays are useful when you know the element and are good for memory size.

Each element within the array has the same amount of space in memory typically 4 bytes.

We can calculate the memory address “x” and the size of each element in the array “y”. To find the memory address of the “i” element the expression would be x + i\* y

**Quick Sort**

Quick sort is a divide and conquer algorithm, and is this implementation is recursive.

Uses pivot element to partition the array into two parts.

Quick sort is the fastest known algorithm for all sorting techniques.

As mention above the algorithm is divide and conquer algorithm by selecting an element in an array as the pivot and then the algorithm will partition the given array around the pivot element by placing the pivot in the sorted array.

To create the partition the pivot element can be the first element, last element, random element, or the middle are typical elements for the pivot. The pivot will help put all smaller element to the left and all the greater elements to the right. The partition is called recursively until the pivot is in the correct position in the array. The partition algorithm starts from the leftmost element that keeps track of the index and if there is a smaller element in the array then we swap that element.

An example is the following and we are given the array below. The pivot is in green

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **55** | **15** | **23** | **117** | **87** | **6** | **10** |

**i j pivot**

Pivot = 10, array[j] = 55; 55>10 so increase j

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **55** | **15** | **23** | **117** | **87** | **6** | **10** |

I j

Pivot = 10, array[j] = 15; 15>10 increase j; array[j] = 23; 23>10 increase j; array[j] = 117; 117>10 increase j; array[j] = 87; 87>10 increase j; array[j] = 6; 6<10 so swap i and j;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **55** | **15** | **23** | **117** | **87** | **6** | **10** |

I j

After the swap and then we increase i

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **6** | **15** | **23** | **117** | **87** | **55** | **10** |

i j

Now we want to looks for an element that is greater than the pivot to swap so we do not lose any values. We now compare the pivot to i instead of j. Reversing directions Pivot = 10, array[i] = 15; 15>10 so we swap;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **6** | **10** | **23** | **117** | **87** | **55** | **15** |

We continue this until the entire array is sorted by recursively calling the function.

Quick sort is an in-place algorithm, typically has a time complexity of (N Log(n)), and worst-case time complexity of O(N2), which occurs when the pivot is chosen poorly. It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot’s position (without considering their original positions). It has a low overhead, as it only requires a small amount of memory to function. Better on large data sets than smaller.

**List**

Since we know that the arrays size is immutable (fixed) we have to create what is referred to as list within java more specifically ArrayList, which is a resizable array that permits all elements, including null. Accessing elements with the list will be O(1) meaning great to iterate over, but removing and adding will take longer O(n). Keep in mind that you can specify the capacity, but if you set the capacity to a larger number then this will reduce the amount of incremental reallocation and the default capacity is set at 10.

**Singly Linked List**

In this list each node contains two buckets one bucket holds the data and the other holds the address of the next node. Traversing the list can only be done in one direction. The first item is referred to the head and the last always points to null. An example can be seen below.

|  |  |
| --- | --- |
| Data | next |

|  |  |
| --- | --- |
| Data | next |

Head 🡪 🡪 null

**Doubly Linked List**

**Stack**

A stack is an ordered list that is a linear data structure used for storing data. A good use for stacks is if we want to approach a problem and use an iterative method. Implemented by using Last in, first out (LIFO). You can think of LIFO as when you have a stack of plate you take from the top and add a new plate to the top of the stack. Adding an element to the top of the stack is done by the push method, removing is done by the pop method and if you want to check the element at the top you can call the peek method. All of these methods are O(1) when using a linked list, but when you use an array the time complexity becomes O(n) because you may need to resize the array. You can also access the bottom of the stack by using the First In, Last out (FILO). Stacks are dynamic because they do not have a fixed size and the size can be increased or decreased depending on the number of elements. Another note on stacks is that only limited access is possible since you are working with the top of the stack whereas an array you have random access and can access any element.

There are two ways you can implement a stack. One being using a one-dimensional array the second being using a singly linked list.

We can implement a stack using a one-dimensional array by the following.

1. Create a variable (int) called top to keep track of the “top” element.
2. Declare a function called stack that takes the input of capacity to create the array
3. Declare a function push() that takes in the parameter we want to put onto the stack
   1. Check if the stack if full, if full then we need to resize the array (typically double the length)
   2. Copy the old array into the new array.
   3. If the stack is not full then increase the top and place the new element in the top position.
4. Declare a function isEmpty() to check is the stack is empty by returning true or false because if the stack is empty there will be no element to pop.
5. Declare a function pop() this will remove the top element in the stack
   1. Check if the stack is empty
   2. Return the top element. Since the top of the stack there is nothing stored, the top item is actually top minus one.
6. Declare a function called peek() which returns the top element from the stack.
7. Declare a function called size() that returns the size of the array stack.

We can implement a stack using a linked list by the following.

1. Declare a variable linked list that will hold our elements and will be our stack.
2. Declare the linked list function that will create our linked list (doubly for memory).
3. Declare a function push() that takes in the parameter we want to put onto the stack.
4. Declare a function pop() that will pop the last element of the Linked List.
5. Declare a function peek() that will show the last element of the Linked List
6. Declare a function isEmpty() to check is the stack is empty by returning true or false because if the stack is empty there will be no element to pop.

**Queue**

A queue is an abstract linear data structure that uses the first in, first out (FIFO) approach. This means the element is inserted at the end of the queue (Enqueue) and removed in the front of the queue (Dequeue). Accessing elements can only be done at the front and end of the queue. An example of this can be a line of people waiting to order food. The person at the front of queue was first to end and the first to leave whereas someone at the end of the line is last to arrive and the last to leave. Applications are job scheduling and multiprogramming.

We can implement a queue using an array by the following.

1. Create two int variables called front and back to keep track of the front and back queue positions. Create an array for the elements that going to held.
2. Declare a function to create an array queue that takes in the capacity of the array
3. Declare a function add() this is the enqueue operation to add an element to the front of the queue
   1. Check if the queue is full (when the back index equals the queue length), if full then we need to resize the array (typically double the length).
   2. Copy the old array into the new array.
   3. If the queue is not full place the new element in the back queue position and then increment the back index because we just added an element.
4. Declare a function remove() this is the dequeue operation to remove an element from the rear.
   1. Check if the queue is empty by seeing if the back minus front is equal to zero. Since we cannot remove an element if the queue is empty
   2. Set an element to the front index value since this is what will be removed.
   3. Since the front index is being removed it needs to be assigned a null value, the front index needs to be increased so we can remove the element at position one.
   4. Check if the queue is empty again, if so set front and back to zero.
   5. Return the element that was set in “b”.
5. Declare a function called peek() which returns the front most element in queue. First check if the queue is empty. If there is a value return the value at the front index.

**Hashing**

Hashing is the process of mapping keys, values into the hash table by using a hash function. Mapping is a way of linking two elements together as one the elements is a key and the other a value. A key is the identifier of the element(data/entity). A value is the actual data/entity that is being stored. An example of this would be a tax number, each citizen gets a tax number associated with the citizens details such as name, address, age, and so on. The benefit of the mapping is that if there are duplicate name, age, or address the key (tax id) creates a unique id. The key is used as the memory location for the key value. A type of problem that arises with a hash function is when a hash function has the same index for two values. This problem is called collision. Hashing benefits are insert and lookup of an element are done by the index and will be done in O(1) time and worst O(n) if too many elements were hashed into the same index.

Hashing in Data Structure is an important method designed to solve the problem of efficiently finding and storing data in an array.

1. Create an hashtable array with the capacity preset.
2. Declare a hash function to set the key value. Using the division method is where if you have an array of 10 and the key inserted is 15 that is outside the range index of 10 we need a way to get the value into the array. We can do this by dividing so the method would be HashFunction = key % size = 15 % 10 = 5 so the key (15) would be inserted into the 5th index of the array.
3. Declare a insert function to insert the key and value. After the key has been created for the element, the element can now be stored into the array.
   1. First get the key associated with the element.
   2. Next to handle collision the hash key needs to check if the key is in the array already if not then store the value as the hash key.
   3. Add a get() function to retrieve the key value from the hash table.

A way to address collision is a technique referred to as open address or closed hashing. The idea of open addressing is if try to put an employee into the table and there is already an employee at the location in the table that the element is being put then this element needs to be put at a different location in the table (array). One way to solve this is linear probing the collision is resolved by searching the hash table until an empty slot in the table is found. The linear function of x is f(x) = x. For inserting the searching would be hash(x)+0, hash(x)+1,… hash(x) +k until an empty slot is found to be inserted.

1. Declare a function occupied() to check if a spot within the array is filled.
2. Within our put() method check to see if a spot is occupied. If a spot is occupied then wrap around the array to find the next open index.
   1. Declare a variable to stop searching at the current hash key
   2. If the hash index has reached the end of the hash table then set the hash key to 0 so that the index can wrap around the array.
   3. Else increase the hash key to the next index
   4. While there is a occupied spot in the array and the hashkey does not equal the stop. Create probe position by the formula hash(key) = key % table size.

The next implementation is to find a given index within our table. This is very similar to our insertion method. For searching the function would be hash(x)+0, hash(x)+1,… hash(x) +k until an element is found if the element is not found then the element is not present within the table.

1. TODO
2. …

If an element needs to be removed from the table then the element needs to be searched for within the table. This is very similar to adding an element the function would still be hash(x) + 0, hash(x) +1, …. Hash(x) +k to find if the element matches the inputted key then remove that matched values.

1. Use the find key function to find if the key is in the hash table.
2. If the element is found then set the value of the hash key to null.

Another way to deal with collision is Chaining also known as open hashing. This is done by using linked list. If two elements are hash to the same location then these elements are represented by using a single-linked list. Once again the function hash(key) = key % table size will be used.

1. Create a LinkedList array for the hashtable
2. Declare a function hashkey() to assign the key by using the function hash(key) = key % table size.
3. Declare a function to add an element that has the key and element attribute.
   1. Declare a variable hashedKey that uses the function hashKey to find the value of the key.
   2. Add the element using the hashedKey to the hashtable with the elements attributes.
4. Declare a function get() If the key needs to be retrieved. Keep in mind using chaining each index will used a linked list.
   1. Declare a variable hashedKey that uses the function hashKey to find the value of the key.
   2. Using java’s List Iterator assign the hashed key values to the hash table.
   3. While there is iteration in the linked list. Check if the element equals the key. If there is not then there is no element that has the key.
5. Delcare a function remove() if the key needs to removed from the table. Keep in mind using chaining each index will used a linked list.
   1. Declare a variable hashedKey that uses the function hashKey to find the value of the key.
   2. Using java’s List Iterator assign the hashed key values to the hash table.
   3. While there is iteration in the linked list. Check if the element equals the key if so remove the element. If there is not then there is no element that has the key.

Chaining while each to understand, implement, and add and remove elements there is wasted space since linked list are being created.

Bucket Sorting is a use for hashing. Distributes the items into buckets based oh hashed values and then sorts the buckets and the merge by concatenation the buckets together. This means that the values in the first bucket need to be greater then values in the second bucket and so on. Let say we sorted the numbers into buckets and then did our sort in bucket one we get 1, 3, 5 and bucket two we get 2, 4, 6. When we merge these together we get 1,3,5,2,4,6, which is not in order. This means we might want to use an insertion sort to sort the buckets since the stability is based on the sort algorithm.

Uses hashing

**Trees**

A tree data structure is a way to organize data hierarchically like a tree. The tree is made up of nodes, where the top node is called the root. The root cannot have parents and the branches, but can have children and those children could have children and are connected by edges. In summary the tree is a collection of objects or entities referred to as node and are linked together by edges. Below is a diagram of the recap.

Parent Node Root

----------------------------------- A Node ------------------------------------- Level 1

-------------- B Node -------------- C Node -------------- D Node -------------- Level 2

-------- E Node ------ F Node ------ G Node ------ H Node ------ I Node ------- Level 3

Subtree

--------------------- J Node ------------------ K Node -------------------------- Level 4

Child Node Siblings

| **Terminology** | **Description** | **Diagram** |
| --- | --- | --- |
| Root | Root node is a node from which the entire tree originates. It does not have a parent | Node A |
| Parent Node | An immediate predecessor of any node is its parent node. | B is parent of E & F |
| Child Node | All immediate successors of a node are its children. The relationship between the parent and child is considered as the parent-child relationship. | F & E are children of B |
| Leaf | Node which does not have any child is a leaf. Usually the boundary nodes of a tree or last nodes of the tree are the leaf or collectively called leaves of the tree. | E, F, J, K, H, I are the leaf nodes. |
| Edge | Edge is the connection represented by a line between one node to another.In a tree with n nodes, there will be ‘n-1’ edges in a tree. | Connecting line between A&BOR A&C OR B&F OR any other nodes connecting each other. |
| Siblings | Siblings in real life means people with the same parents, similarly in the case of trees, nodes with common parents are considered to be siblings. | H&I are siblings |
| Path | Path is a number of successive edges from source node to destination node. | A ,C, G, K is path from node Ato K |
| Height of Node | Height of a node represents the number of edges on the longest path between that node and a leaf. | A, C, G, K form a height. Height of A is no. of edges between A and K,, which is 3. Similarly the height of G is 1as it has just one edge until the next leaf node. |
| Levels of node | Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on | Level of H, I & J is 3. Level of D, E, F & G is 2 |
| Degree of Node | Degree of a node implies the number of child nodes a node has. | Degree of D is 2 and of C is 3 |
| Visiting | When you’ve iterated or traversed to a specific node programmatically, accessing value or checking value of the current node is called visiting. |  |
| Internal Node | A node that has at least one child is known as an internal node. | All the nodes except E, F, J, K, H, I are internal. |
| Traversing | Traversing is a process of visiting each node in a specific order in a tree data structure. | There are three types of traversals: inorder, preorder, postorder traversal. |
| Ancestor node | An ancestor or ancestors to a node are all the predecessor nodes from root until that node. I.e. any parent or grandparent and so on of a specific node are its ancestors. | A, C & G are ancestor to K and J nodes |
| Descendant | Immediate successor of a node is its descendent. | K is descendent of G |
| Sub tree | Descendants of a node represent subtree. Tree being a recursive data structure can contain many subtrees inside of it. | Nodes B, E, F represent one subtree. |

Construction a binary tree in Java. This is similar to a linked list where if we have the head then we can get to any other object. The program to do this needs to construct a root node.

1. Create a class that will hold the node of the tree.
   1. Create a variable for the data of the node
   2. Create variables for the left and right nodes
   3. Declare getters and setters for the all of the variables
   4. Create a constructor that accepts the variable for the data of the node.
2. Create a class for the tree
   1. Create a variable for the root node.

To insert a new element into the tree.

1. Declare a function that takes in a value
2. Compare the value to root node to see if the value should be on the left or right.
3. If the value is less than the root. See if the left child has any other nodes if not insert the child. If the child does have other nodes compare the other nodes by recursively calling left child.
4. If the value is greater than the root, check if the right child has any other nodes if not then insert the child if the root does have a right child then compare the value to the other nodes by recursively calling.

Tree traversal is a way to check the values of the nodes within the tree. There are a couple ways to do this. One is to visit all nodes on the level, another is pre-order where the program visits the root of the subtree first, another is post-order the subtree is visited after the root, the last is in-order this is where the left child is visited, then the root, and last the right child.

To return an element in the tree.

1. Declare a function that takes in a value
2. Compare the value to root node to see if the value should be on the left or right.
3. If the value is equal to the node return the value.
4. If the value is less than the node See if the left child is not null and then check the value. Recursively calling this method for all additional node.
5. If the value is greater than the node See if the left child is not null and then check the value. Recursively calling this method for all additional node.
6. If the value does not equal any of the nodes then the value does not exist in the tree.

To return the min value of a tree

1. Traverse the left side of the tree

To return the max value of a tree

1. Traverse the right side of the tree

To delete a node from a tree

**Heaps**

To understand heaps, it is helpful to understand binary trees. A binary tree is composed of nodes where the top node is referred to the root and is not a child of another node. A node can have a max of two nodes referred to as children or left and right. The max amount of children a node can have is two and the least is zero also a node can have one child. If a node has zero children the nodes is referred to as a leaf node.

One way to represent a tree is using an array. If we have six elements in an array.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F |

Index: 0 1 2 3 4 5

If the index of a given element is i,

1. 2i + 1: This will give us the left child of element i
2. 2i +2: This will give us the right child of element i
3. (i-1)/2: The element at the lower bound of this index gives the parent of the element at i.

If the index is 1, so I = 1 in our example.

1. 2(1) +1 = 3. The left child would be D.
2. 2(1)+2 = 3. The right child would be E.
3. (1-1)/2 = 0. The parent would be A.

A complete binary tree is a tree where all the elements are arranged without missing a sequence. This means that all levels are completely filled except the last level and elements are filled left to right.

Now that a binary tree has been clarified in order for a heap the heap need to be a complete binary tree some refer to heap as binary heaps. A min-heap is where the root node is always the smallest of all the elements and each node is smaller than their children. A max-heap is the opposite of a min-heap the root node is the largest and each parent node is larger than their children. In order for a binary tree maintains the heap data structure there is a process called heapify, where elements within the tree structure are rearranged to either be a min or max heap.

Construction of a heap

1. TODO

**Priority Queue**

This is a type of queue that arranges elements based on their priority value. Elements that have a higher priority are retrieved before elements with lower priority values. A priority queue has a few characteristics, one of them is that each element has a priority value, the next is the higher priority elements are dequeued before the smaller elements, and if two elements have the same value then the first in first out rule is applied. A priority queue can either be in ascending order or descending order. For example if elements are arranging using the smallest value such as 1,2,3 this would be in ascending order where if the element values were arranged in order such as 3,2,1 this would be descending order.

Construction of a priority queue.

1. TODO