1. Linked List (single):
2. findNode(value):

In the linked list, this function will take value and search it through the list via iterating from the beginning till the end. When it finds a corresponding match it will return its index or the required info of the node

1. addNode(node):

Given a node, we need to insert it into the linked list. For simplicity, assume we are dealing with a sorted linked list, consisting of integer values, first we need to iterate through the list, and stop where the corresponding place is found (the first node that has a bigger value than our node’s value). When we find the first node that has a bigger value than our node’s value (call it ‘x’), we make our node’s ‘next’ pointer assigned to ‘x’. And the ‘next’ pointer of the node we encounter before x (call it ‘y’), is assigned to our node. In case of double-linked list, we should also deal with the previous pointers accordingly. This scenario is valid when the node to be added is in the middle.

1. deleteNode(value):

It can be reviewed as the reverse version of add. We again have 3 different scenarios:

Head: if the node to be deleted is the current head, delete that node and its pointers, and assign the next node as new head

Middle: make the previous node of to be deleted node’s ‘next’ value assigned to the node that comes right after node to be deleted, and delete the node to be deleted with its pointers

Tail: just delete it and make the previous node’s next pointer assigned to NULL

1. FIFO
2. POP:

Make the second item the new ‘head’, delete the first item with its pointers, copy its value in a temporary variable, return the temporary variable

1. PUSH(item):

Make the last item’s ‘next’ pointer assigned to our new item, our new item is the new TAIL.

1. Hash
2. Insert(value, table\_size, table\_address):

First we will hash the value with the hashing function. After getting the hash value, we are going to reach the hash tables respective index and store our value in that index of the table. If we have a collision meaning that the location is occupied with another element – checked via hash of the occupying element, if hash fails to point to the same address, we understand that it is not a valid element thus not occupying (for simplicity of understanding the collisions). Then we resolve the collision with f(i) such that new location = hash(value) + f(iteration) and try again.

1. Delete(value):

Again we need to hash the value first, after that we reach the respective index of the hash table and delete the value, making that index of the table NULL, for our example we change the value in the address to a value which is not matching to the current address.

1. Find(value):

Hash the value and find the index in the hash table, then retrieve the wanted info from the element

1. Hash(value):

Hashing the value and getting a hash value as a result, mod(table\_size)

1. Binary Tree
2. Insert(value):

Using left and right pointers, reach to the respective node of the tree and create a new leaf with the given value

1. Find(value):

Nearly same as insert, first reach to the respective node of the tree, if the value of the node is matching with the given value, we have found the node, now can retrieve other info of the node

1. Delete(value):

Again find the corresponding node of the tree, if the value is matching, make the pointer of the parent, NULL. If the node to be deleted is leaf, there is no problem, if the node to be deleted has a single child and a parent, attach parent’s respective pointer to the child of the to be deleted node. If node to be deleted has 2 children, some rotation might be required.

Note: For a BinaryTree rather than having a tree structure we can implement a Heap. As a fixed size tree, heap can be easier to implement.

a.a. Insert(value): We append the value at the next available slot in the array, then percolate the element. We divide the address by 2, check if the parent is bigger or smaller depending on if heap is minheap or maxheap. Then we repeat again until we find the appropriate location. We return the address.

b.b. find(value): we start with the root, if value is equal to root return the address. If not, progress to children, multiply the address by 2. Repeat until finding or till the end of the list. If not find, return root address -1. O log(N)

c.c. deleteMin(): Take the root element, than take the last element from the array, put it on the root and percolate accordingly. Decrease size by 1.

d.d. percolate(value): a. If a minHeap : percolateUp(value), divide by 2 and compare value with parent. Can compare divide by 2 + 1 too. If value is smaller than parent, swap. Continue till finding a smaller parent. percolateDown does the same by multiplying parent by 2 and checking if bigger or not. If current value is bigger, swap. Continue until heap order property is established.

b. If a maxHeap, rather than expanding the tree with parent.value < child.value policy, parent values are bigger than child values. While percolating, we try to establish heap order property accordingly.