Dynamic Set

* Mathematical sets are unchanging
* Sets used in computers can grow or shrink i.e. they are DYNMIC

Elements {

Key,

SatelliteData

}

Key data

The key is needed for the operation of the tree. Everything else isn't

Satellite data

Refers to any "payload" data which you want to store in your data structure and which is not part of the structure of the data structure. It can be anything you want. It can be a single value, a large collection of values, or a pointer to some other location that holds the value

Example

Node {

Node next; // key

int value; // satellite data

}

Queries operations

* Search(S, K)
* Minimum(S)
* Maximum(S)
* Successor(S, x)
* Predecessor(S, x)

In which, S is a Dynamic Set, K is the key and x is satellite data

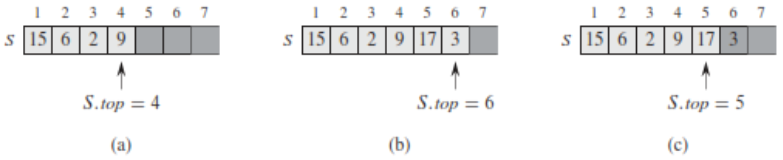
Modifying operations

Insert(S, x)

Delete(S, x)

Stack

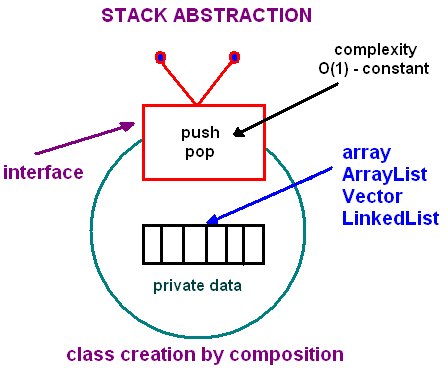
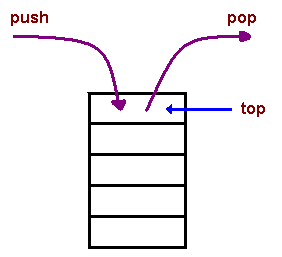
LIFO - Last In First Out



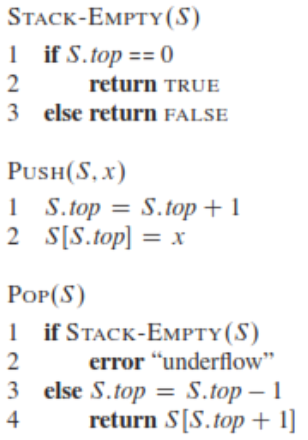
a) S has 4 elements and the top element is 9

b) After called PUSH(S, 17) and PUSH(S, 3)

c) After called POP(S)



Pseudo code



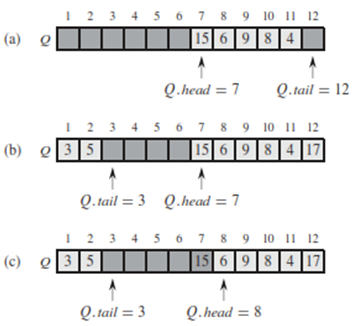
See more

lec8.MySimpleStack.java

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html

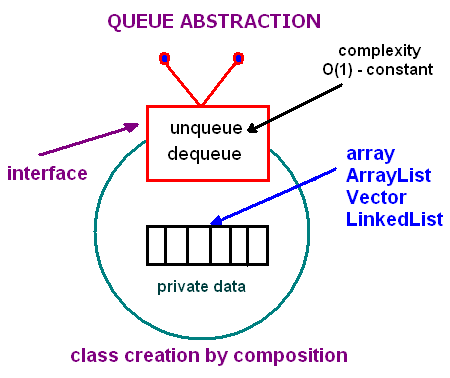
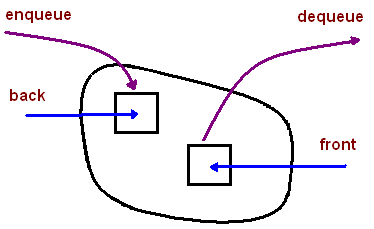
Queue

FIFO – First In First Out

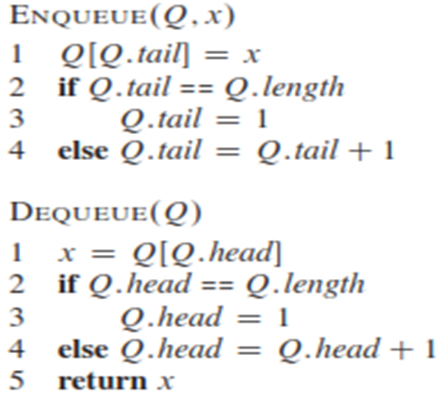


Q [1…12]

1. Queue has 5 elements from 7…11
2. Call enqueuer(17), enqueuer(3) and enqueuer(5)
3. Calls dequeue(Q), returns the value 15 at the head of the queue. New head has value 6



Pseudo code



See more

lec8.ArrayQueue.java

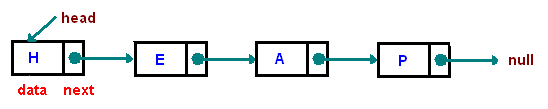
<https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html>

Linked List

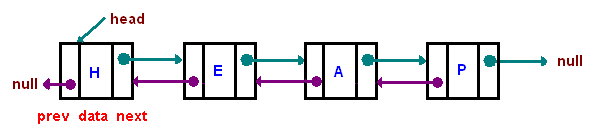
One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements

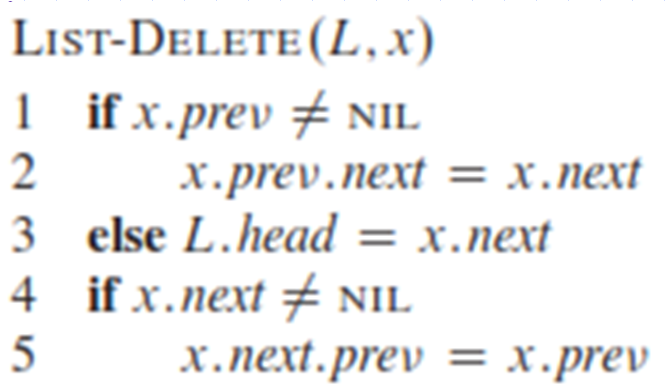
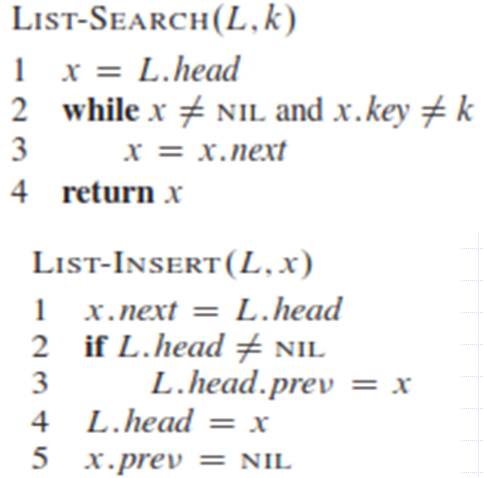
Singly Linked List



Doubly Linked List

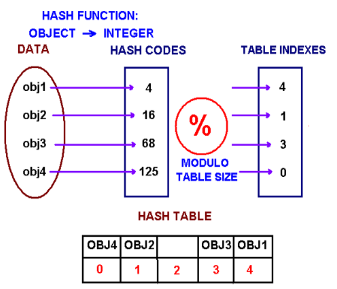


Pseudo code



Hashing

The problem at hands is to speed up searching if we know in advance the index at which that value is located in the array. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a hash function. A hash function is a function which when given a key, generates an address in the table.



In the above image, create an array of size M *(= 5 in this case)*. Choose a hash function h that is a mapping from objects into integers 0, 1...M-1. Put these objects into an array at indexes computed via the hash function index = h(object). Such array is called a hash table.

Division method

h(k) = k mod m

Example:

Insert the following sequence of numbers 23, 46, 12, 21, 75, 5, 3 into a hash table of size 9 using h(x) = x%9 as a hash function, where % mean “mod”. Use Chaining with Linked List to avoid collision.

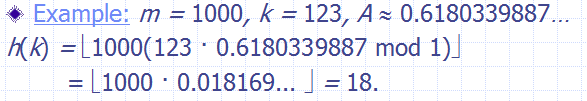
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|  | 46 |  | 12 |  | 23 |  |  |  |
|  |  |  | 21 |  | 5 |  |  |  |
|  |  |  | 75 |  |  |  |  |  |
|  |  |  | 3 |  |  |  |  |  |

Multiplication method



* Disadvantage: Slower than the division method.
* Advantage: Value of m is not critical. Typically chosen as a power of 2, i.e., m = 2p, which makes implementation easy.

Example:



A hash function that returns a unique hash number is called a universal hash function which properties as below:

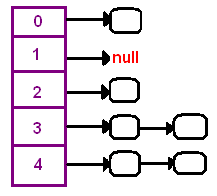
* It always returns a number for an object.
* Two equal objects will always have the same number
* Two unequal objects not always have different numbers

For Division or Multi. Method, the worst case is that all keys map to ONE Slot! To avoid so, Universal Hashing selects hash functions randomly. The key idea is to select Hash Function at random at run time from set of selected functions, provides good average case performanceCollisions

When we put objects into a hash table, it is possible that different objects (by the equals() method) might have the same hash code. This is called a collision. Here is the example of collision. Two different strings ""Aa" and "BB" have the same key:

"Aa" = 'A' \* 31 + 'a' = 2112

"BB" = 'B' \* 31 + 'B' = 2112



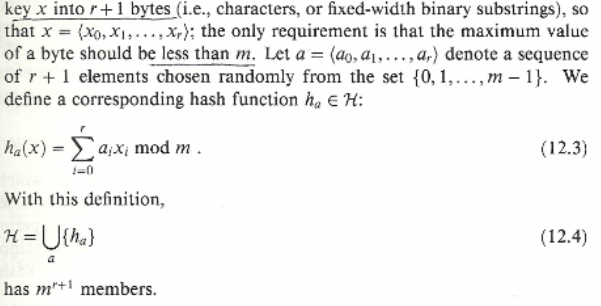
If the number of hash functions h, for which h(x) = h(y) is |H| / m , then h is a Universal Hash Function. Chance of collision between x and y is 1/m.

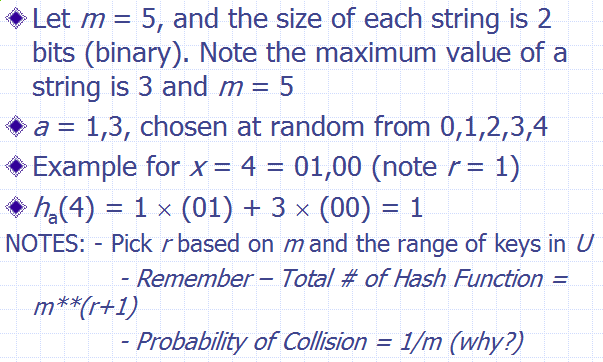
There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a separate chaining collision resolution.

The big attraction of using a hash table is a constant-time performance for the basic operations add, remove, contains, size.

Another technique of collision resolution is a linear probing. If we cannot insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on.

Decompose

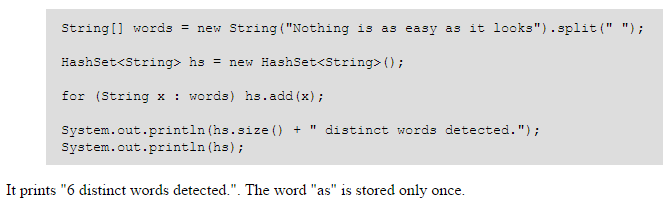




HashSet

HashSet is a regular set - all objects in a set are distinct.

HashSet stores and retrieves elements by their content, which is internally converted into an integer by applying a hash function. Elements from a HashSet are retrieved using an Iterator. The order in which elements are returned depends on their hash codes.

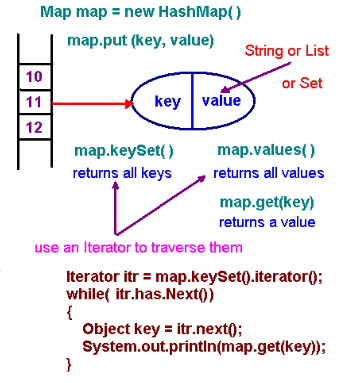


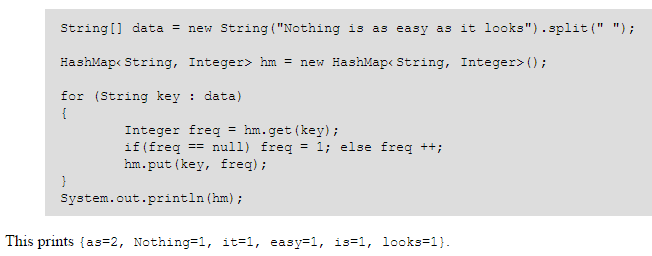
HashSet methods:

* set.add(key) -- adds the key to the set.
* set.contains(key) -- returns true if the set has that key.
* set.iterator() -- returns an iterator over the elements

HashMap

HashMap is a collection class that is designed to store elements as key-value pairs. Maps provide a way of looking up one thing based on the value of another.





HashMap methods

* map.get(key) -- returns the value associated with that key. If the map does not associate any value with that key then it returns null. Referring to "map.get(key)" is similar to referring to "A[key]" for an array A.
* map.put(key,value) -- adds the key-value pair to the map. This is similar to "A[key] = value" for an array A.
* map.containsKey(key) -- returns true if the map has that key.
* map.containsValue(value) -- returns true if the map has that value.
* map.keySet() -- returns a set of all keys
* map.values() -- returns a collection of all value

NOTE: HashSet and HashMap will be printed in no particular order. If the order of insertion is important in your application, you should use LinkeHashSet and/or LinkedHashMap classes. If you want to print dtata in sorted order, you should use TreeSet and or TreeMap classes

Binary Search Tree

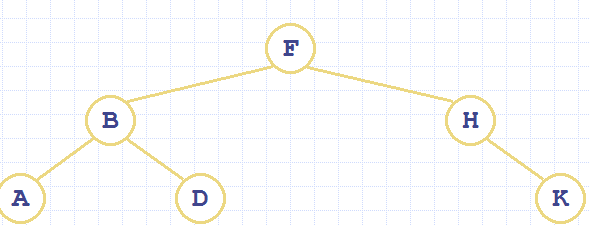
A binary tree is a tree such that each node has *at most* 2 children

BST property

key[leftSubtree(x)] <= key[x] <= key[rightSubtree(x)]

* The key of a node is always greater than the keys of the nodes in its left subtree.
* The key of a node is always smaller than the keys of the nodes in its right subtree.

Example

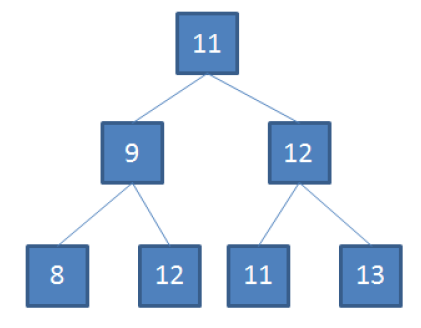


Traversals

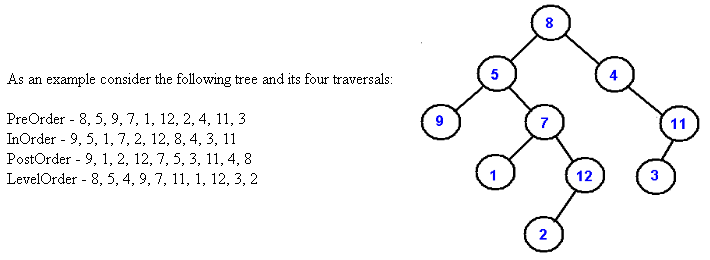
* Pre-order tree walk: print root, then left, then right
* In-order tree walk: print left, then parent and then right
* Post-order tree walk: print left, then right, then root
* Level-order tree walk: print the root, and then its children, then its grandchildren

Example

Use Pre-Order, In-Order and Post-Order Tree-Walk to print all keys in the following Tree:



* Pre-order tree walk: 11,9,8,12,12,11,13
* In-order tree walk: 8,9,12,11,11,12,13
* Post-order tree walk: 8,12,9,11,13,12,11
* Level-order tree walk: 11, 9, 12, 8, 12, 11, 13



Algorithm Design

1. Devide and Conquer

Involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then “merging” the solutions to the subproblem(s) to produce a solution to the original problem

The methods

* Divide the problem into subproblems (divide input array into left and right halves)
* Conquer the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
* Combine the solutions to the subproblems into a solution to the problem (return value found or indicate not found)

It doesn’t work if

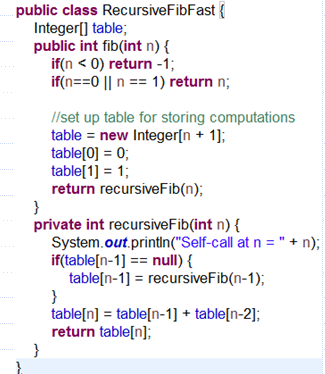
* For Divide and Conquer to be effective, it must be possible to break up the original problem into non-overlapping subproblems (Ex: In MergeSort, the steps of recursive sorting of the left half of the list do not affect, and are not affected by, the steps of the sorting of the right half of the list)
* If something similar to Divide and Conquer is attempted when problems are overlapping, it may result in many redundant computations (Ex: Recursive Fibonacci)
* Binary Search (and operations on a BST)
* MergeSort
* QuickSort
* QuickSelect

1. Dynamic Programming

Sometimes problems can be broken down into overlapping subproblems, which can be solved, and whose solutions can be combined in some way to obtain a solution to the main problem. Solutions to subproblems are stored and combined stage by stage to produce a solution to the main problem

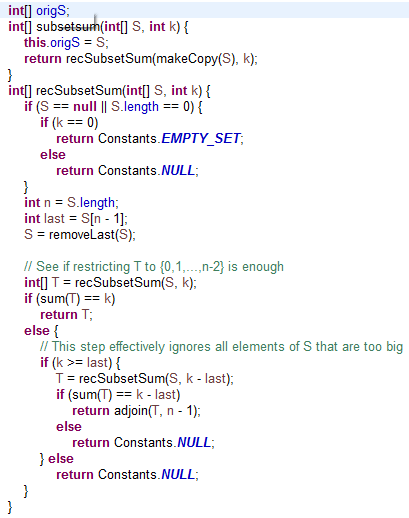
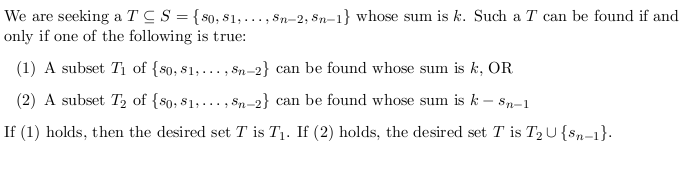
When such a problem exhibits the following characteristics, it can in many cases be tackled using dynamic programming:

* Overlapping subproblems – the subproblems “overlap” – the recursion tends to solve the same subproblems over and over (example: recursive fibonacci)
* Optimal substructure – an optimal solution is composed of a combination of optimal solutions to subproblems
* Revised Recursive Fibonacci
* To generate the nth Fibonacci number, the subproblems are computation of the kth Fibonacci numbers for k < n.
* To prevent redundant computation, solutions to subproblems can be stored in a table and accessed whenever needed during execution of the algorithm



* SubsetSum

The Subset Sum optimization problem says: We have set S = {s0, s1 … sn-1} of n positive integers and a non-negative integer k. Find a subset T of S so that the sum of the s(r) in T is k



Explanation

The recursive algorithm tries to find a solution T for ({s0, s1, …, sn-2, sn-1}, k) by checking if a solution exists for either of the subproblems

* ({s0, s1, …, sn-2}, k)
* ({s0, s1, …, sn-2}, k - sn-1)

To find these, it seeks solutions to smaller subproblems

As n gets larger, the recursive solution will repeatedly recalculate solutions for the smaller subproblems (recall how this happened with recursive Fibonacci)

We can speed up the recursive approach by storing solutions to subproblems in a table (memoization). See code Demo

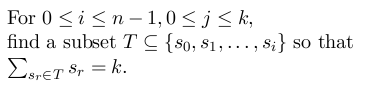
We can organize the stored computations in the recursive algorithm in a table (see RecDynamSubsetSum-Demo.pdf)

Typically, the amount of work done to fill in the table is polynomial bounded, and the rest of the running time is insignificant.

A “bottom-up” approach is typically used to fill in the table from the 0th row to the last row. The correct output is then read from the bottom right corner of the table. See DynamicSubsetSum-Demo.pdf

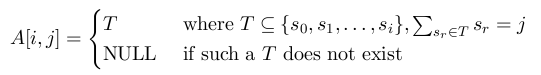
Bottom up approach

There are only (k+1) \* n problems to solve, namely:

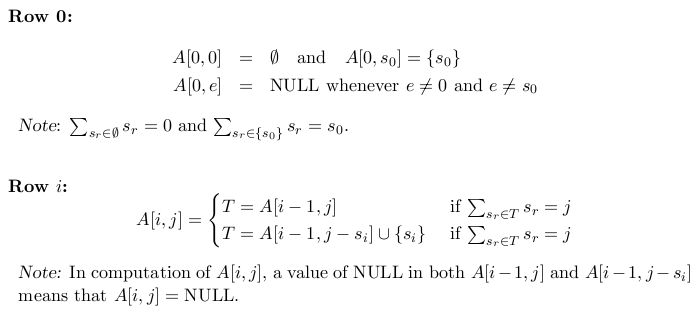


Build a solution for bigger values of i and j using stored solutions for smaller values of i and j.

Obtain a 2-dimensional array (a matrix) A so that



* If S contains values > k, we ignore them since they don’t contribute to the solution (computations for which j is too big are skipped – see the implementation in code)
* Fill row i = 0 first, then fill later rows based on values of earlier rows.



Pseudo-polynomial time algorithm

The dynamic programming solution to SubsetSum runs in O(kn). However, k may be much bigger than n, and even if k is Θ(n), the true running time is based on the number of bits in k, not on the value of k. So even this algorithm runs in exponential time in terms of input size.

Note: For a pseudo-polynomial time algorithm, its running time is polynomial in the numeric value of the input, but is exponential in the length of the input – the number of bits required to represent it

* Knapsack
* Shortest Path (in a graph - later)

1. The Greedy Method

* Fractional Knapsack
* Shortest Path (in a graph - later)
* Minimum Spanning Tree (in a graph - later)