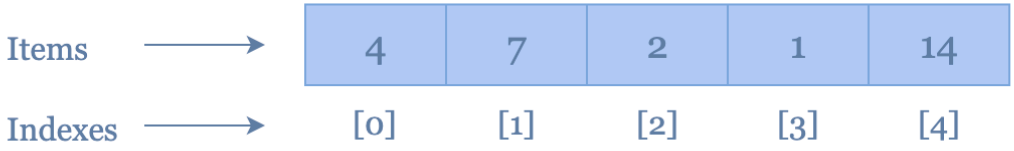
fredag 10 september 2021

Data Structures

Summary

Array

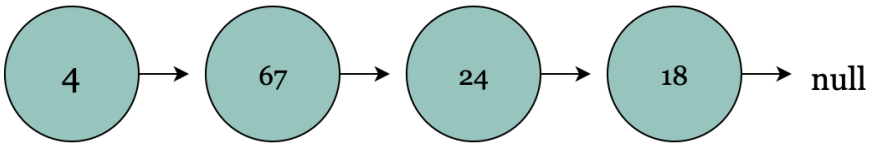


- Collection of items stored sequentially in memory that has to be of the same variable type. One of the most popular and simple data structures, used to implement a lot of other data structures.
- Each item is indexed, and the indexing starts at 0, each item known as an element. In Java, you cannot change the size of an array. For dynamic sizes, use Lists or Vectors (that has the same properties as arrays).
- Insertion and deletion is expensive, since we have to copy the information and create a new array each time.
- In sorted array we can perform binary search O(log(n))
- Declaration with initializing in Java (Declaration to the left, initializing to the right):
    - int[] name = new int[14];

Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Space Complexity
O(1)	O(n)	O(n)	O(n)	O(n)

Worst case and average complexities

Singly-Linked-List

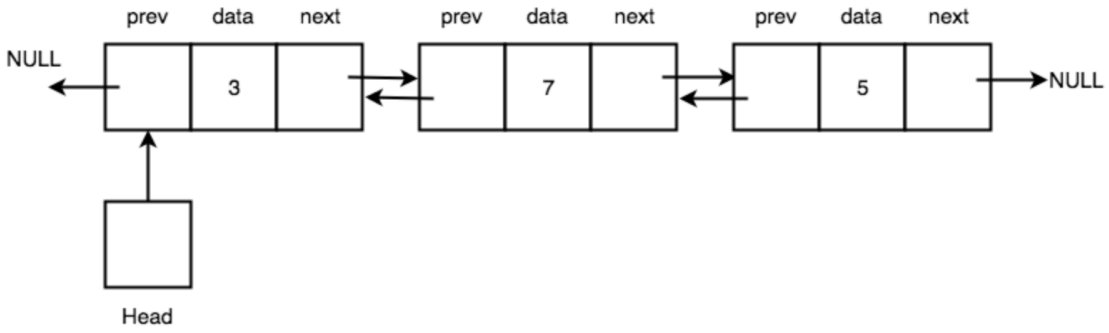


- Each node holds a value, and a pointer to the next node. Arrays uses indexes, in a linked list, we have to traverse the list to find the k:th element. The last node points to a null element, and thus we know where to end.
- We call the first node the **head** and the last node the **tail**.
- Nodes cannot be accessed directly
    - Although dynamic size, and fast to insert and delete elements
    - Deletion is O(1) when given a pointer to the element. Deletion is O(n) when given the key to the element in the worst case, since we have to traverse n elements to fins it.
    - Insertion is O(1) since we simply insert the element at the beginning of the list. Insertion at a specific position is O(n), since we have to traverse the array to find this position.
- Used when implementing stacks, queues and hash tables
- Decleration and initializing in Java:
    - LinkedList<String> name = new LinkedList<String>();
    - Operations:
        - name.add(”A”)
        - name.addLast(”C”)
        - name.addFirst(”D”)
        - name.add(2, ”E”)
        - name.remove(”B”)
        - name.removeFirst()
        - name.removeLast()

Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Space Complexity
O(n)	O(n)	O(1) (O(n))	O(1) (O(n))	O(n)

Worst case and average complexities

Doubly-linked-list

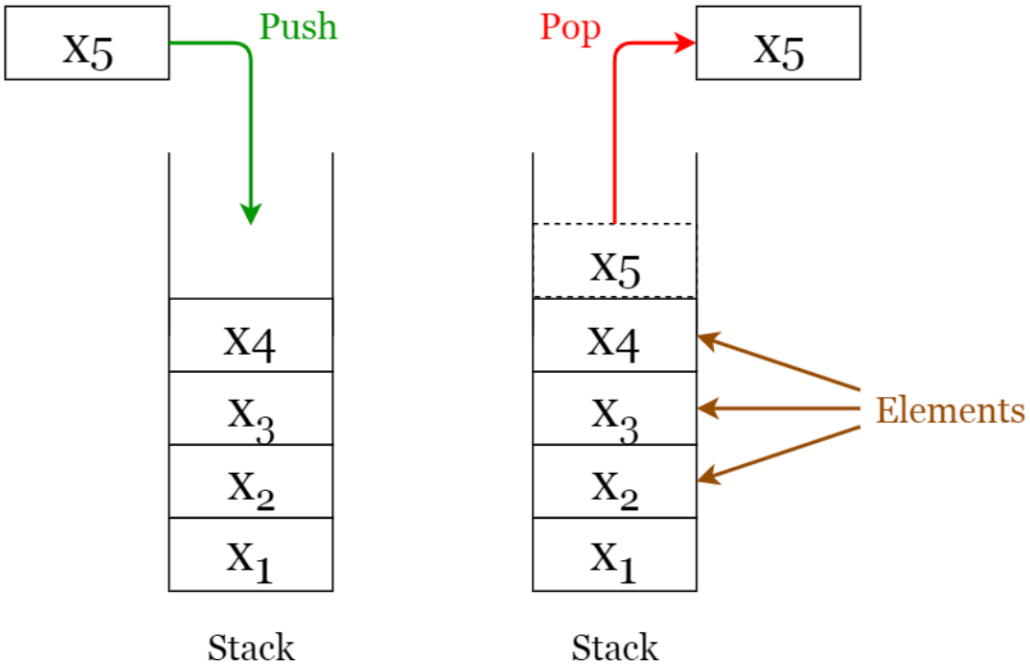


- Decleration and initializing in Java:
    - LinkedList<String> name = new LinkedList<String>();
    - LinkedList is implemented as doubly linked list in java, you have to use a ListIterator to use the prev pointer
    - Implementations:
        - Used in allocation of dynamic memory for example. Useful when you want to find an element, and want to manipulate the elements that are next and previous to that one. For eg when merging the free blocks in a freelist when performing malloc.

Access (finding element with knowing the position)	**Searching (finding position given the element)**	**Insertion**	**Deletion**	**Space Complexity**
O(n)	O(n)	O(1) (O(n))	O(1) (O(n))	O(n)

Worst case and avarage complexities

Stack



- push() - insertion at top of the stack
- pop() - delete top element and return it
- empty() - Is the stack empty
- peek() - access top element without deleting it
- search() - returns the 1-based position where the object is on stack
- LIFO - last in first out
    - Imagine the plate stack analogy, you take the plate at the top of the stack first.
- Useful applications:
    - Backtracking to a previous state
        - Recursion algorithms for eg
- Can be implemented using array, vector or linkedlist depending on need
    - Array

        - Memory is saved as pointers are not involved
        - Not dynamic, doesn't grow and shrink depending on runtime
    - Vector (Dynamic array - **Java**)
        - Memory is saved as pointers are not involved
        - Like an array, elements can be accessed using index.
        - However the size of the Vector can grow or shrink during runtime.
    - LinkedList
        - Dynamic, can grow and shrink depending on the needs we have during runtime
        - Requires extra memory due to pointers
- In Java:
    - Import java.util.stack
    - Stack <Integer> stack = new Stack<Integer>();

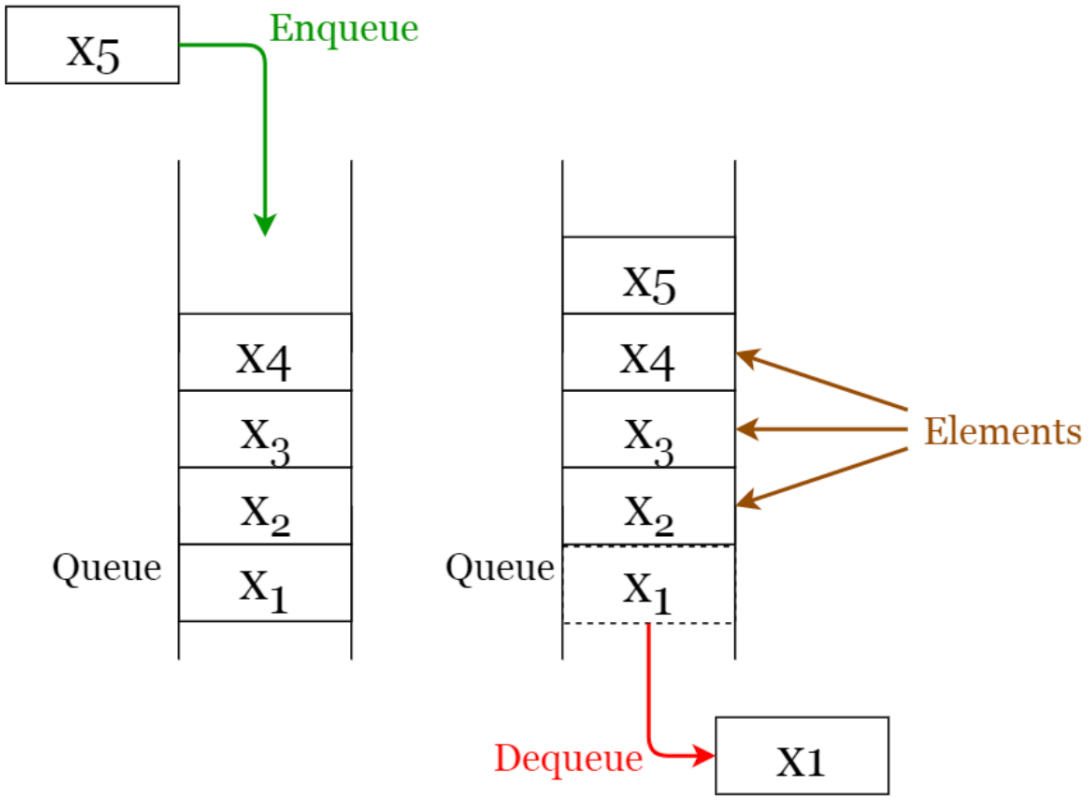
Access (finding element with knowing the position)	**Searching (finding position given the element)**	**Insertion**	**Deletion**	**Space Complexity**
O(n)	O(n)	O(1)	O(1)	O(n)

Worst case and avarage complexities (Implemented with LinkedList)

Access (finding element with knowing the position)	**Searching (finding position given the element)**	**Insertion**	**Deletion**	**Space Complexity**
O(1)	O(n)	O(1)	O(1)	O(n)

Worst case and avarage complexities (Implemented with Vector or Array)

Queue



- boolean add(E e) - Inserts element into queue, returns true or throws IllegalStateException
- remove() - removes the head of the queue and returns this object - throws an exception id empty
- poll() - same as remove, returns null if empty
- peek() - returns but do not remove head of queue or returns null if empty
- isEmpty()
- Application:
  - Used to manage threads in multithreading
  - Used to implement queuing systems (priority queues)
- In Java:
  - Since Queue is an interface, objects cannot be created of the type queue. We always needs a class that extends this interface to create an object. We say that

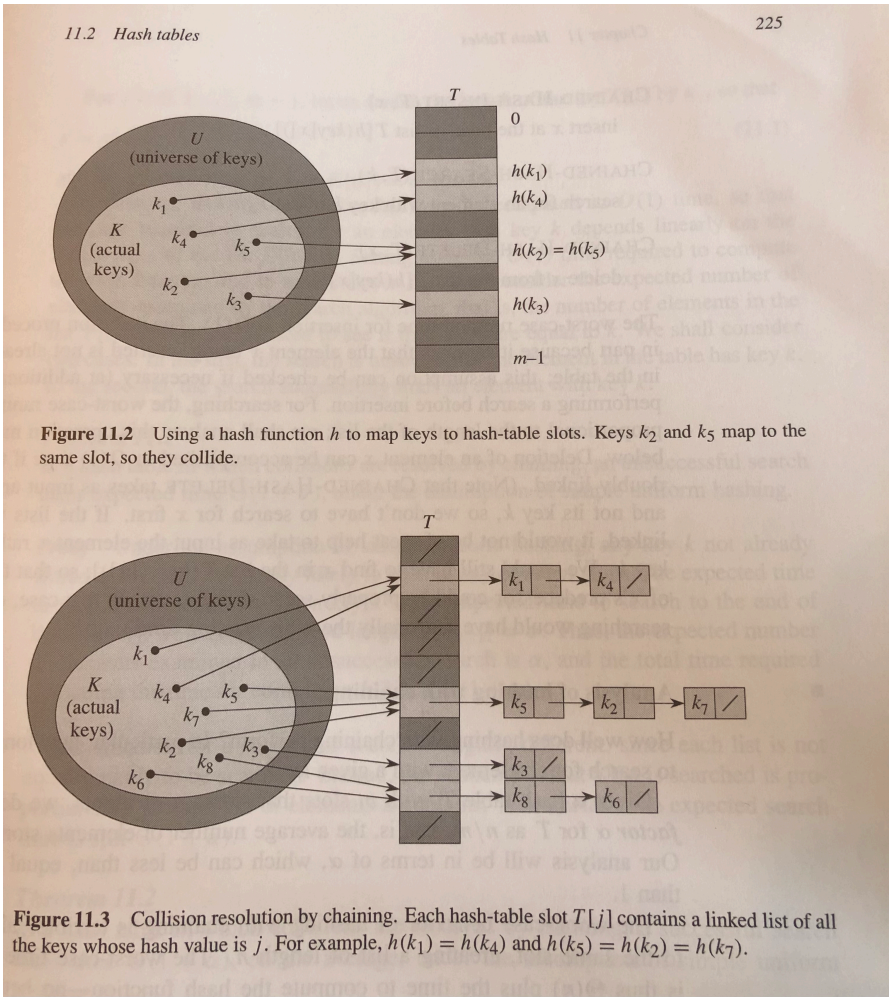
we need a concrete class for the declaration, most common ones are PriorityQueue and LinkedList:

- Eg:
  - Queue<Integer> queue = new LinkedList<Integer>();

Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Space Complexity
O(n)	O(n)	O(1)	O(1)	O(n)

Worst case and average complexities (Implemented with LinkedList)

HashTable



- put(key, value) - adding elements
- put(same_key, new_value) - changing element in hash table
- remove(key) - remove element with key
- Stores key-value pairs in hash table
- We use a hash function to map the key k to slot h(k), where h(k) is called the hash value
- In java, every element in the array, which is the foundation how the hash table, contains a bucket. In case of collision, that is two different keys maps to the same bucket, that bucket points to a linked list.

    - This method to handle collisions is called chaining. Another method is called linear probing or open addressing.
- Initialized as:
    - Hashtable<K, V> ht = new Hashtable<K, V>();
- Worst case is quite bad for hash tables when using chaining, that is if the hash function is bad and we get all of the elements in one bucket
- Average case is very good though and this is why it is used frequently

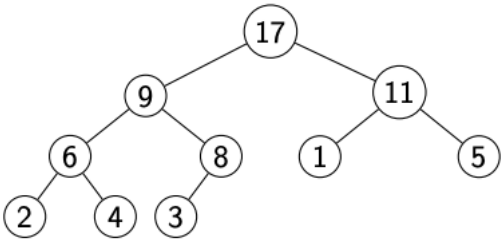
Access (finding element with knowing the position)	**Searching (finding position given the element)**	**Insertion**	**Deletion**	**Space Complexity**
N/A	O(n)	O(n)	O(n)	O(n)

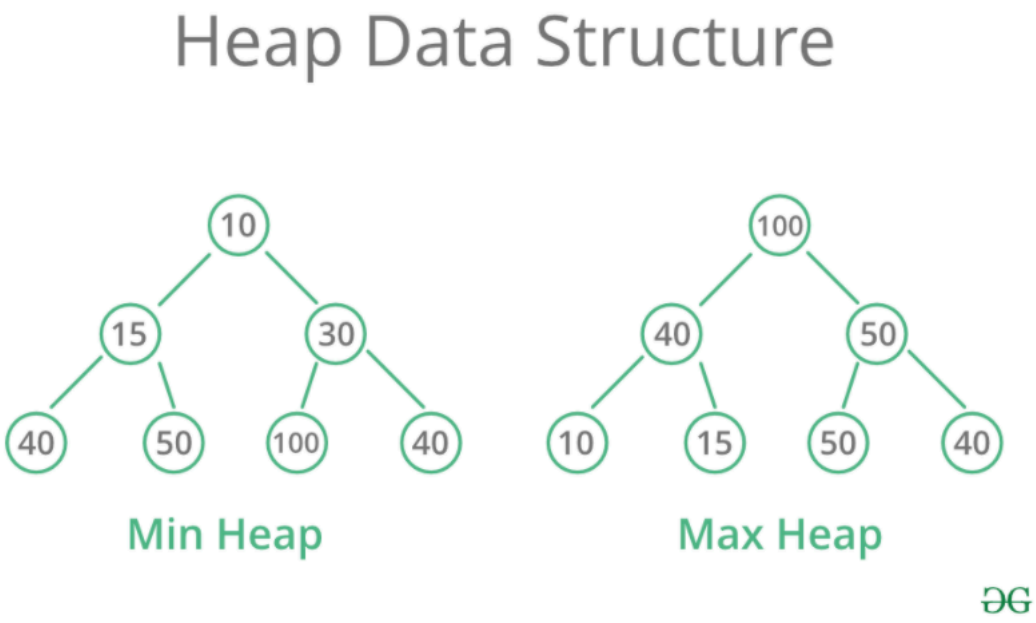
Worst case complexities using chaining

Access (finding element with knowing the position)	**Searching (finding position given the element)**	**Insertion**	**Deletion**	**Space Complexity**
N/A	O(1)	O(1)	O(1)	O(1)

Average complexities using chaining

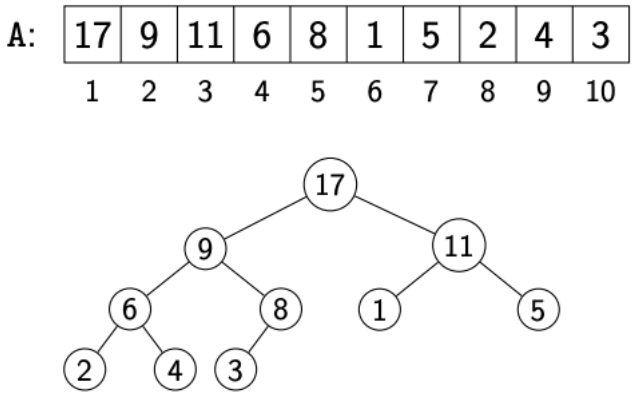
Heap





- Heap is a special kind of tree structure, also referred to as a complete binary tree
- All levels of the tree should be filled apart from the last level, which should be filled from left to right
- Max Heap:
    - Greatest element in the root. All children should be smaller, and this should be true for all subtrees in the tree, ie every node's children should be smaller then the parents (or as small).
- Min Heap:
    - Smallest element in the root. All children should be larger, and this should be true for all subtrees in the tree, ie every node's children should be larger then the parent (or as large).

- Effective storage of heap as array:
    - Store in array A so that:
        - Root is in A[1]
        - Children to A[I] is in A[I+1] and A[I+2]



- Remove operation (remove root element A[1]):
    1. Save root element
    2. Remove last element in the heap and place it in root position
    3. If root element is smaller than it's largest child, swap their placement.
    4. Do 3. recursively until no longer true.
    5. Heap structure is now in order
    6. O(log(n)) in worst case (depth of a balanced tree is log(n))
- Java Min Heap, use PriorityQueue:
    - PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();
- Java Max Heap, use PriorityQueue and Comparator:
    - PriorityQueue<Integer> maxHeap = new PriorityQueueeu<>(Comparator.reverseOrde());

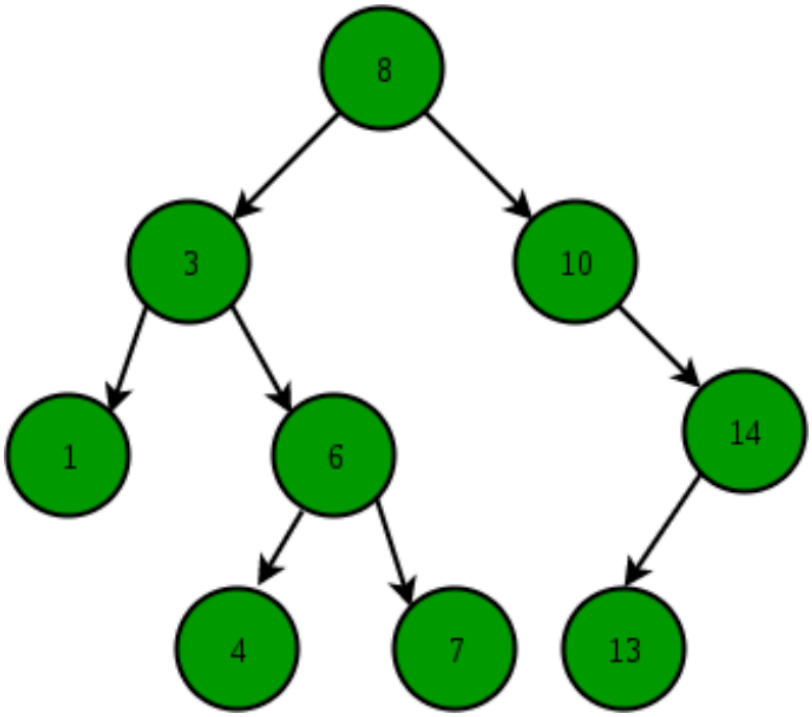
Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Find min	Delete min	Space complexity
O(1)	O(n)	O(log(n))	O(1)	O(1)	O(log(n))	O(n)

Worst case complexity (Min Heap)

Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Find min	Delete min	Space complexity
O(1)	O(n)	O(1)	O(log(n))	O(1)	O(log(n))	O(n)

Avarage case complexity (Min Heap)

Binary Search Tree



- Every BST contains a key and the fields left, right (pointers to the children), p (pointer to parent).
- A leaf in the tree is a tree that points to NIL as it's children
- A root in the tree is a node that has a NIL parent
- Property of binary search tree:
    - X is a node in the tree.
    - If y is a node in the left subtree of x, then y.key <= x.key.
    - If z is a node in the right subtree, then z.key >= x.key
- For a complete BST, the height is log(n)
    - The worst case runtime of basic operations is then O(log(n))
- If the search tree is a linear chain om elements, runtime is expected to be O(n)

- We can use red-black-trees to guarantee a height of O(log(n))

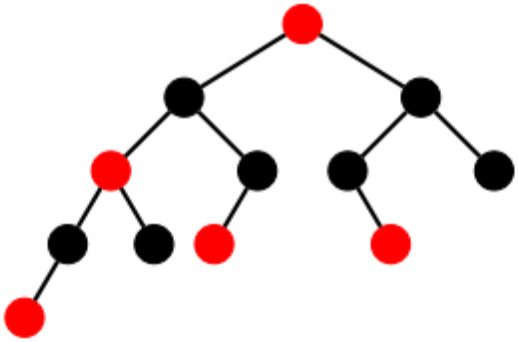
Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Space Complexity
O(n)	O(n)	O(n)	O(n)	O(n)

Worst case complexities

Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Space Complexity
O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Average case complexities

Red-Black Tree

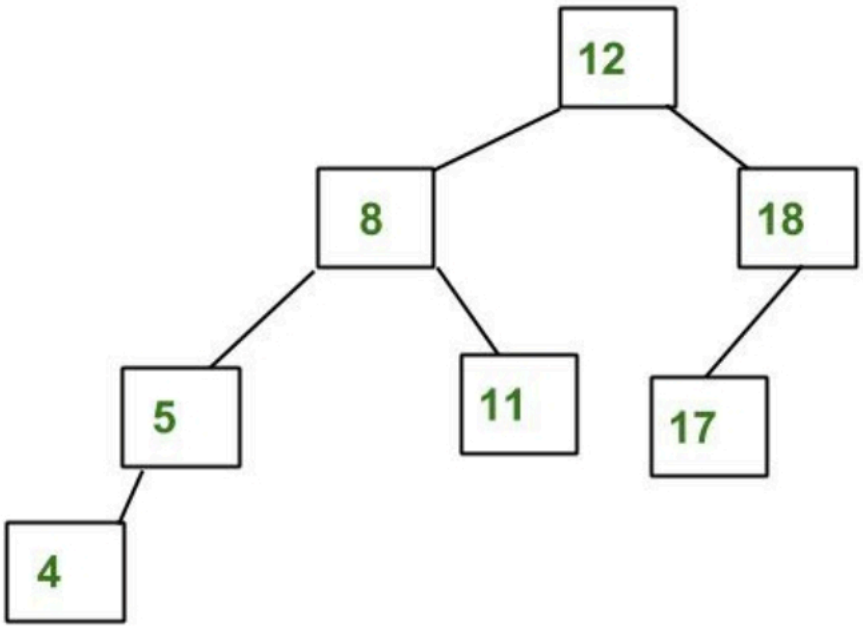


- Problem with BSTs are that they take a long time to traverse if they are not balanced properly
- Red-black trees solve this by balancing the tree after each insertion, so that a height of log(n) is guaranteed
- A red-black tree is a binary search tree with one extra field: it's color
    - Color can be either red or black
- A binary search tree is a red-black tree if it has the following properties:
    1. Every node is either red or black
    2. The root is black
    3. There are no two adjacent red nodes
    4. For each node, all paths from the node to descendent leaves contain the same number of black nodes
- After each insert and rotation, the tree has to be modified to keep the properties of a BST (through left and right rotations), and a red-black tree
- In Java:
    - TreeSet ts = new TreeSet();

Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Space Complexity
O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Worst and average case complexities

AVL Tree



- Another example of a self balancing BST
- The difference between heights of the left and right subtrees cannot be more than one for all nodes
- Checks that this is maintained after every modification of the tree

Access (finding element with knowing the position)	Searching (finding position given the element)	Insertion	Deletion	Space Complexity
O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Worst and average case complexities

Splay Tree

- Another self balancing BST
- Keeps it self roughly balance through splaying (avarage height is O(log(n)))
- But unlike AVL and red-black tree, splay trees are not guaranteed to be height balanced
- A splay tree rotates the latest accessed element to the root, so that it can be accessed in O(1) the next time
    - A side effect of these rotations is a tendency to balance the tree
- The tree isn't guaranteed to perform an operation in O(log(n)), although it is guaranteed to perform m number of operations in O(mlog(n)) If m >= n
- Application:
    - Used for cache algorithms

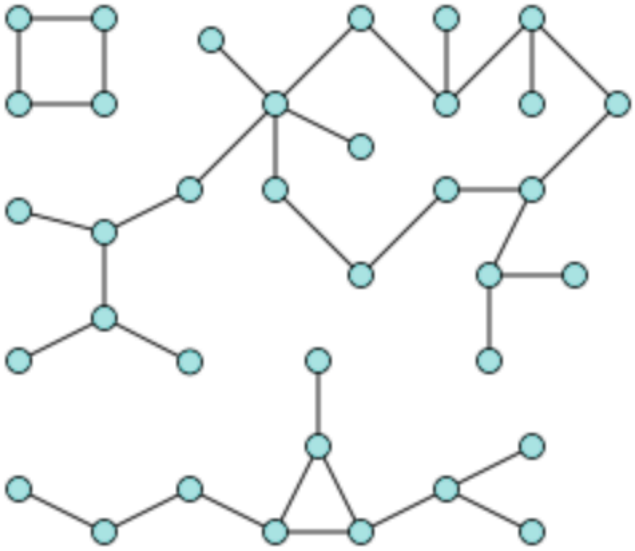
Access (finding element with knowing the position)	**Searching (finding position given the element)**	**Insertion**	**Deletion**	**Space Complexity**
O(n)	O(n)	O(n)	O(n)	O(n)

Worst case complexities

Access (finding element with knowing the position)	**Searching (finding position given the element)**	**Insertion**	**Deletion**	**Space Complexity**
O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

Average case complexities

Graphs



- A graph is represented as G = (V, E)
    - Where V = vertices and E = Edges
- Two main ways to represent graphs
    - Collection of adjacency lists
        - The most common representation
    - Adjacency matrix
- The vertex set of G is denoted V(G)
- The edge set of G is denoted E(G)
- A self loop is an edge whose endpoints is a single vertex
- Multiple edges are two or more edges that join the same two vertices
- Graph called simple if it has no self-loops or multiple edges, and a multigraph it does have multiple edges
- A cycle is a path v1, v2, …, vk for which k > 2, the first k-1 vertices are all different, and v1 = vk
- A path is a series of vertices v1, v2, …, vk such that there are edges between vi and vi+1.

- A graph is connected if for every pair of vertices u and v, there is a path from u to v
- If there is a path connecting u and v, the distance between these vertices is defined as the minimal number of edges on a path from u to v
- A connected component is a subgraph of maximum size, in which every pair of vertices are connected by a path.

Trees

- A tree is a connected simple acyclic graph.

Directed graphs

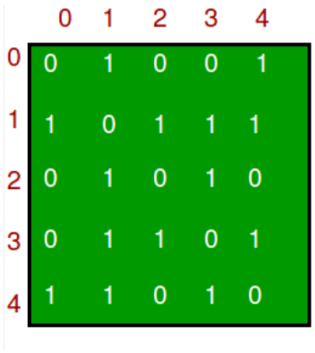
- A directed graph G = (V, E) consists of a vertex set V and an edge set of ordered pairs E of elements in the vertex set.

DAG

- Directed acyclic graph

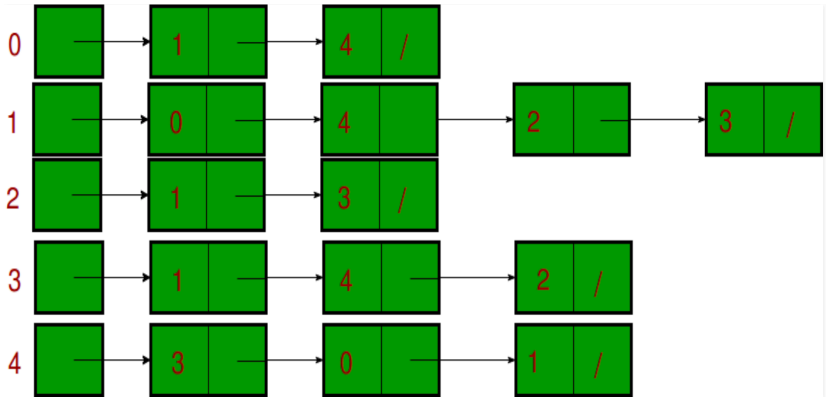
Data structures

Adjacency matrix



- An adjancy matrix is a |V| x |V| matrix of integers, representing a graph
- The vertices are number from 1 to |V|
- The number at position (I, j) indicates the number of edges from i to j
- 2D array adj[][]
- Pros
    - Easier to implement
    - Removing an edge takes O(1)
    - Operations like checking if edge between u to v is O(1)
- Cons
    - Consumes more space O(V^2)
    - Adding a vertex is O(V^2) in time

Adjacency list



- An array of lists
- Size of array is |V|
- Pros
    - Saves space O(|V + E|)
- Cons
    - Operations like checking whether there is an edge between u and v can be done in O(V), not that efficient

Algoritmer

Algoritm	Time Complexity
DFS (Adjacency List)	O(|V|+|E|)
BFS (Adjacency List)	O(|V|+|E|)
DFS (Adjacency Matrix)	O(|V|^2)
DFS (Adjacency Matrix)	O(|V|^2)
Djikstra's	O((|V|+|E|)*log(|V|))
Topological sort	O(|V|+|E|)

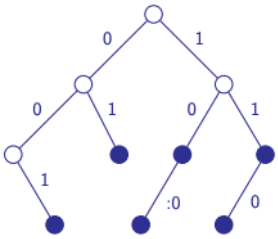
Average case complexities

Skip list

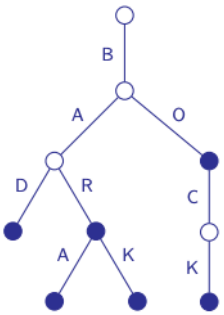
- Probabilistic data structure
- Easy to implement
- As fast as balanced trees to search but easier to modify
- Search time is as fast as for balanced tree
- Insert and delete time is doubled for balanced trees in comparison with skip list

Trie

Exempel på binär trie:
{001, 01, 10, 100, 11, 110}



Exempel på bokstavstrie (automat):
{BAD, BAR, BARA, BARK, BO, BOCK}



Svårt att lagra effektivt!

- Trie is an efficient information retrieval data structure
- Trie supports search, insert and delete operations in O(L) time where L is the length of the word.
- Very efficient for prefix search (or auto-complete)
- The main issue is that they need a lot of memory for storing, since we have one node per letter instead of one node per word. This creates a need for a lot of pointers, which increases the need of memory substantially.