

# Data structures

09 June 2021 07:07

## 1. Array

W

## 2. Linked list allocations are scattered in heap. Use only memory needed

When u need allocate else release.

However LL uses more memory than array for eg

Dynamic memory alloc uses LL to keep track of alloc memory. OS scheduler process probably uses Circ LL.

// NOTES:

// 1. This is a sequential data structure and is used to store data. ✓  
// 2. Linked list is special data structure in which data elements are linked to one another.  
// 3. The pictorial look of a linked list is:

// OPERATIONS:  
// Insertion, Deletion, Traversing.

// PROS  
// 1. Linked list is dynamic data structure, it can grow and shrink at run time.  
// 2. It is used to efficiently utilize memory  
// 3. Insertion and Deletion is very easy at any position.

// CONS  
// 1. More memory is required if list is too big.  
// 2. Accessing elements is time consuming.

// TYPES OF LINKED LIST: ✓  
// 1. Singly Linked List  
// 2. Doubly Linked List  
// 3. Circular Linked List  
// 4. Doubly Circular Linked List

## 3. XOR LL

XOR of prev and next stored. Hence we need just one next. Behave like Doubly LL.

// NOTES:  
// 0. This is singly linked list but act like doubly linked list.  
// 1. We can go forward and reverse in XOR Linked List.  
// 1. It is memory efficient doubly linked list.  
// 2. Instead of storing address of next node, it stores XOR of previous and next address  
// 3.

// General Linked List  
// A -> B -> C -> D -> NULL

// XOR Linked List  
// Data      A      B      C      D  
// Next    0 xor B   A xor C    B xor D    C xor 0

// PROS:  
// 1. It takes less memory than doubly linked list but act like doubly linked list.

// CONS (Wiki):  
// 1. Debugging becomes hard. (because we don't store actual address)  
// 2. Code complexity.  
// 3. Garbage Collection is difficult (because no proper address is stored)

## 4. Circ LL

// NOTES:  
// 1. Circular linked list is modified version of singly linked list.  
// 2. In circular linked list last node point to first node.



// INSERTIONS: ✓  
// 1. Insertion in an empty list ✓  
// 2. Insertion at the beginning of the list ✓  
// 3. Insertion at the end of the list ✓  
// 4. Insertion in between the nodes ✓

// ADVANTAGES:  
// 1. CPU scheduling ✓  
// 2. Queue can be implemented using one pointer (last). Because next to last is always first!  
// so we don't have to implement first pointer at all.  
// 3. Applications where you don't want to reset once you reach to last node. ✓

I

5. F

6. F

I

7. F

I

8. F

I

## 9. Stacks

I

### use cases

- backtracking
  - finding the correct path through a maze
- compile-time memory management
  - programs use them to store local data and procedure info
  - nested and recursive functions
- depth-first search
- undo (*pop*) / redo (*push*)

- Items are **pushed** onto the top of a stack
- Items are **popped** off the top of the stack
- A last in first out data structure (**LIFO**)
- A stack is an abstract data type (ADT)
- Call stack stores return addresses, parameters and register contents when sub routines are called
- Used by compilers to evaluate mathematical expressions which require intermediate steps

## 10. Queues

I

### use cases

- operating systems
  - CPU and disk scheduling
- Spotify
- breadth-first search

## 11. F

I

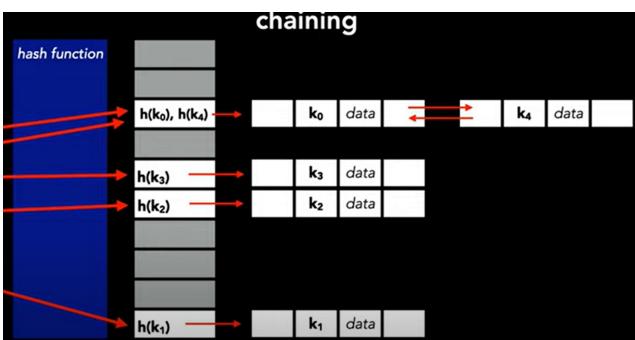
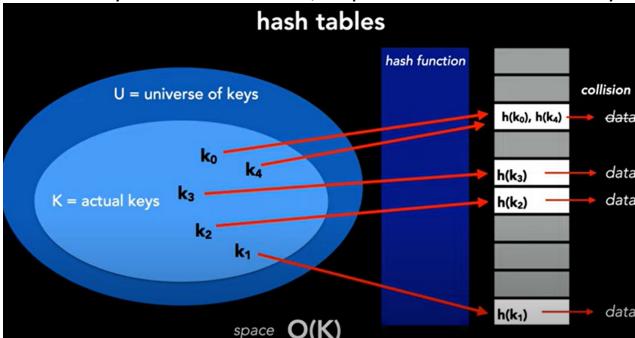
## 12. Hash tables



**dictionary** = generic way to map keys to values

**hash table** = implementation of a dictionary using a hash function

Because keys aren't unbounded, it's possible that two data may hash to same key. So we have LL to maintain list of data that hash to same key



## hash function

### goals:

- maximize randomness
- produce the least amount of collisions

### examples:

- division
- multiplication
- universal hashing
- dynamic perfect hashing
- static perfect hashing

## Objectives of Hash Function

- Minimize collisions
- Uniform distribution of hash values
- Easy to calculate
- Resolve any collisions

Linear probing/ Open addressing.. Linear search o find next avail slot (or) to find item

$$\text{Load Factor} = \frac{\text{Total number of items stored}}{\text{Size of the array}}$$

[Hash Tables and Hash Functions](#)

## Collision Resolution

- Open addressing
  - Linear probing
  - Plus 3 rehash
  - Quadratic probing (*failed attempts*)<sup>2</sup>
  - Double hashing
- Closed addressing

I

### 13. Heap use nearly complete binary tree-i.e. all levels are filled except last

What is a heap?

A heap has to be a **balanced** binary tree, i.e. either the lowest level or the level above the lowest level must all have **leaves**.

In other words, this tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

In simple terms, it's a nearly complete binary tree.

#### **Max-Heap:**

for every node  $i$  other than the root:  $A[\text{parent}(i)] \geq A[i]$   
 in other words, the value of a node is at most the value of its parent  
 which indicates, the largest element in a max-heap is at the root

#### **Min-Heap:**

the opposite of Max-Heap  
 for every node  $i$  other than the root:  $A[\text{parent}(i)] \leq A[i]$   
 the value of a node is at least the value of its parent  
 the smallest element in a min-heap is at the root

**height**

$O(\log n)$

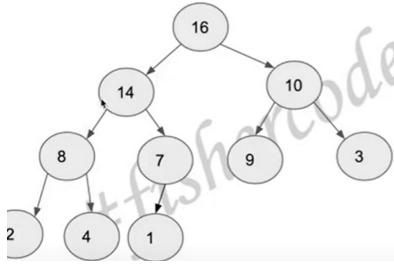
This could be simplified to:

$$2^h \leq n < 2^{(h+1)}$$

which could be transformed to:

$$h \leq \lg n < h + 1$$

so, in  $\Theta$  notation, it could be written as  $h = \Theta(\lg n)$



What's the definition of height?

The number of edges on the longest simple downward path from the **node** to a **leaf**.

So, the node with value 7 has a height of one.

And the height of a heap is the height of its root.

So, this heap on the left has a height of three.

Height - 0th to xth ...  $\log_2(n)$

Min number of nodes for height =  $2^h$

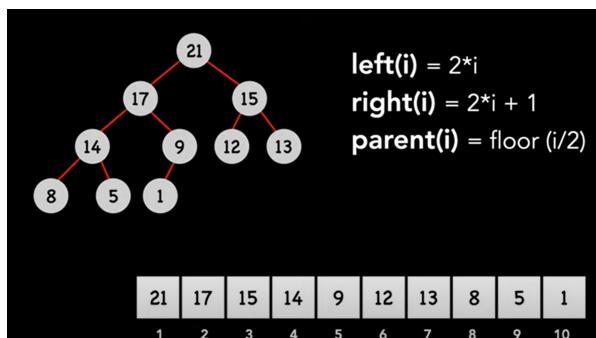
Max number of nodes for height =  $2^{h+1} - 1$

$$2^h \leq n \leq 2^{h+1} - 1$$

$$h \leq \log_2(n) \leq h+1$$

$$\text{Height} = \Theta(n)$$

All operations are in terms of height i.e.  $\Theta(n)$



Max-heapify assumes already bottom nodes are heaps. So if imbalance is found then it has to go down by  $O(\log n)$  only to fix it  
**MAX-HEAPIFY**

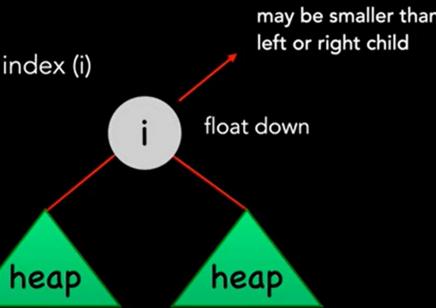
This is a procedure call to maintain the max-heap property.

MAX-HEAPIFY assumes the subtrees rooted at LEFT( $i$ ) and RIGHT( $i$ ) are max heaps, only that  $A[i]$  might be smaller than its children, thus violating the max-heap property.

So, MAX-HEAPIFY lets the value at  $A[i]$  float down in the max heap so that the subtree still obeys the max-heap property.

## max-heapify

- maintains max-heap property
  - value of  $i \leq$  value of parent
  - value of  $i \geq$  value of children
- inputs: array ( $a$ ), heap size, index ( $i$ )



```

def max_heapify(a, heap_size, i):
    l = 2*i
    r = 2*i + 1

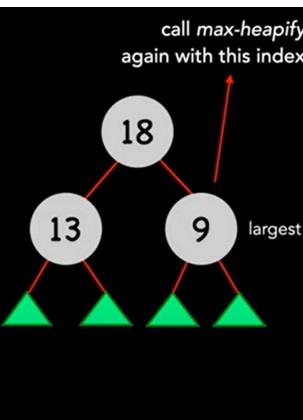
    largest = i

    if l < heap_size and a[l] > a[i]:
        largest = l

    if r < heap_size and a[r] > a[largest]:
        largest = r

    if largest != i:
        # swap elements
        a[i], a[largest] = a[largest], a[i]
        max_heapify(a, heap_size, largest)

```



## running time

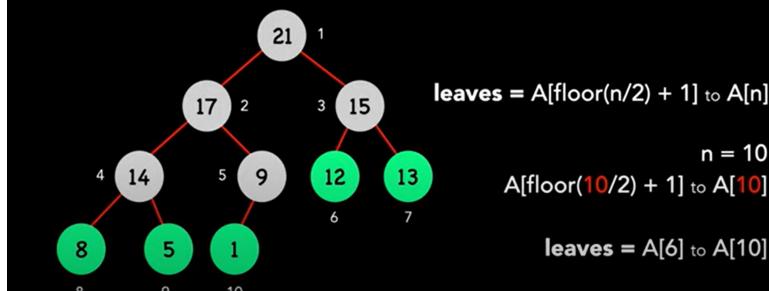
$O(\log n)$

height of binary tree

Leaves are already heaps. So start from non-leaf node and call max-heapify on it since nodes below it are already heaps.

## build-max-heap

– wrapper function that calls max-heapify



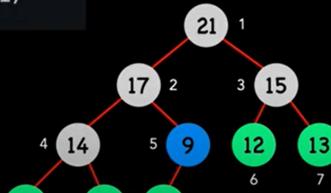
```

def build_max_heap(a):
    heap_size = len(a)

    for i in range(heap_size//2, 0, -1):
        max_heapify(a, heap_size, i)

def build_max_heap(a):
    for i in range(5, 0, -1):
        max_heapify(a, 5, i)

```



## running time

$O(n)$

I

## 14. Priority Queue

## Priority Queue:

### Max PQ -> backed by Max Heap

Scheduled jobs on a shared computer, etc.

### Min PQ -> backed by Min Heap

Event-driven simulator, etc.

I

15. F

I

16. F

17. F

I

18. F

I

19. F

I

20. F

I

21. Tree

```
// TOPIC: Tree Data Structure Introduction

// NOTES:
// 0. Tree is a hierarchical data structure which stores the information in the form of hierarchy.
// 1. Tree is a non-linear data structure compared to arrays, linked lists, stack and queue.
// 2. Tree represents the nodes connected by edges.
// 3. Tree is one of the most powerful and advanced data structures.

// TREE TERMINOLOGIES:
// Root, Parent Node, Child Node, Siblings, Path, Height of Node, Height of Tree, Depth of Node, Degree of Node, Edge.

// TYPES OF TREE WE WILL COVER IN THIS SERIES:
// Binary Tree, Binary Search Tree, AVL Tree, Red-Black Tree, Splay Tree, N-ary Tree, Tri Structure,
// Suffix Tree, Huffman Tree, Heap Structure, B Tree, B+ Tree, R Tree, Counted-B Tree, K-D Tree,
// Decision Tree, Markel Tree, Fenwick Tree, Range Tree

// ADVANTAGES OF TREE DATA STRUCTURE:
// 0. Tree reflects structural relationships in the data.
// 1. It is used to represent hierarchies.
// 2. It provides an efficient insertion and searching operations.
// 3. Trees are flexible. It allows to move subtrees around with minimum effort.
```

<https://towardsdatascience.com/5-types-of-binary-tree-with-cool-illustrations-9b335c430254>

Edge - connecting line

Degree - number of children

Height - number of nodes below

Depth = height?

Height Of Binary Tree

```
struct Node {
    int data;
    Node* left;
    Node* right;
};

int getMaxHeight(Node* root) {
    if(root == NULL) {
        return 0;
    }

    int leftHeight = getMaxHeight(root->left);
    int rightHeight = getMaxHeight(root->right);

    return std::max(leftHeight, rightHeight) + 1;
}
```

Size = number of nodes

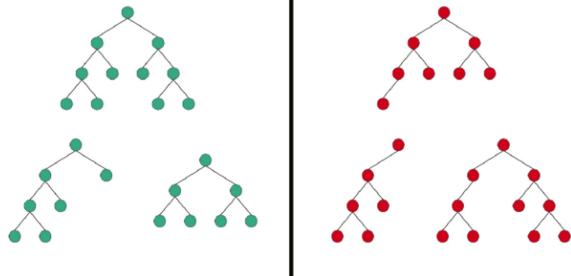
Size of Binary Tree

```

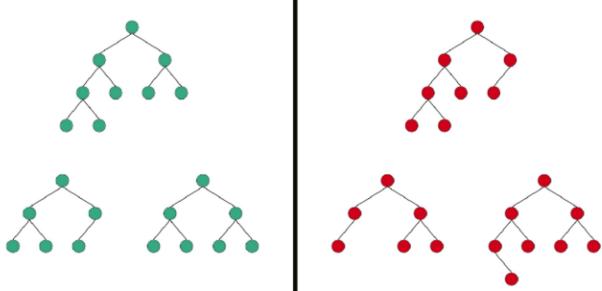
int getSize(Node* root) {
    if(root == NULL) {
        return 0;
    }
    int leftSize = getSize(root->left);
    int rightSize = getSize(root->right);
    return leftSize + rightSize + 1;
}

```

Full Binary tree - every node except leaf will have 0 or two children



Almost complete - every level except last and last-1 levels have no children  
Complete - every level is filled except last level.. & filled from left

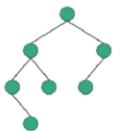


Left/Right skewed - every node has left or right children only

Perfect Binary tree



Balanced - height of left and right part of tree differ by 1



# balanced search trees

guaranteed height of  $O(\log n)$  for  $n$  items

D

22. F

I

23. BST

**Binary Search Tree** ✓

**NOTES:**

0. In BST, a node at max can have two children (Left, Right)
1. Binary Search Tree is a binary tree data structure which has the following properties.
  - a. The left subtree of a node 'A' should contain all the nodes with lesser value than 'A'.
  - b. The right subtree of a node 'A' should contain all the nodes with higher value than 'A'.
  - c. The left and right subtree also must be a binary search tree.
2. It is like having sorted data in tree.
3. In-order to have sorted data in tree, tree should have above three properties.

**BENEFITS:**

0. Instead of using sorted array if we use BST then complexity of Insert, Delete becomes  $\log(n)$ .
1. As it maintains sorted elements you have all advantages of data being sorted.

**COMPLEXITY:**

1. Search, Insert, Delete Complexity:  $\log_2(n)$
2. Space Complexity:  $O(n)$

I

24. AVL Tree

[https://www.youtube.com/watch?v=20cnalwJY9I&list=PLk6CEY9XxSIBG2Gv6-d1WE3Uxqx94o5B2&index=19&ab\\_channel=CppNuts](https://www.youtube.com/watch?v=20cnalwJY9I&list=PLk6CEY9XxSIBG2Gv6-d1WE3Uxqx94o5B2&index=19&ab_channel=CppNuts)  
Balance tree based on difference of L-R at every insertion

Prerequisite is BST; /

NOTES:

1. AVL tree is height balancing BST OR self balancing BST.
2. Why BST was not enough? and we needed balancing BST (AVL Tree)
  - a. Normal BST is having issue when data is sorted or almost sorted (skewed Binary tree).
  - b. Using AVL tree we keep balancing when it becomes non balanced while inserting data.
3. AVL tree take the height of left and right sub tree and find the difference (L-R) and it should be either of {-1,0,1}.
4. This difference is called Balance Factor.
5. If the balance factor of left and right sub tree is not either of {-1,0,1} then tree is balanced using rotation techniques.
  - A. left left ==> Right Rotation
  - B. left right ==> Left and Right Rotation
  - C. right left ==> Right and Left Rotation
  - D. right right ==> Left Rotation

I

25. Red black tree

## Why Red Black Tree? ✓

1. They are self balancing Binary Search Tree. ✓
2. RB trees are good at frequent insertion and deletion than lookup (search).
3. It has less rotations than AVL tree.
4. It has less memory footprint than AVL tree. ✓
5. Red Black Trees are used in most of the language libraries like map, multimap, multiset in C++ whereas AVL trees are used in databases where faster retrievals are required.

I

a b L

## red-black tree

1. A node is either red or black.
2. The root and leaves (NIL) are black.
3. If a node is red, then its children are black.
4. All paths from a node to its NIL descendants contain the same number of black nodes.

[Red-black trees in 4 minutes — Intro](#)



26. F

I

27. F

I

28. F

I

29. F

I

30. F

I

31. F

I

32. F

I

33. F

I

34. D

I

35. F

I

36. F  
I
37. F  
I
38. F
39. F  
I
40. F  
I
41. F  
I
42. F  
I
43. F  
I