Binary tree

- In binary tree, there are At MOST 2^k nodes in level k.
  - ∴ a BT of depth d, has at most $1+2+4+\cdots+2^d = 2^{d+1}-1$ nodes

$2^{d+1}-1$

- Prove: If a binary tree has N vertices, than its depth is at most $N-1$

  root
  1
  2

  ∴ By adding one vertice we can increase the depth by at most 1

- Prove: If a BT has N vertices, than its depth is at least $\Omega(\log(N))$

  ∴ $N \le 2^{d+1}-1$ (d is depth) ∴ $\log_2(N+1) \le d+1$, $d \ge \log_2(N+1)-1$

⇒ Size of tree:

```
return getSize(root.left) + getSize(root.right) + 1
```

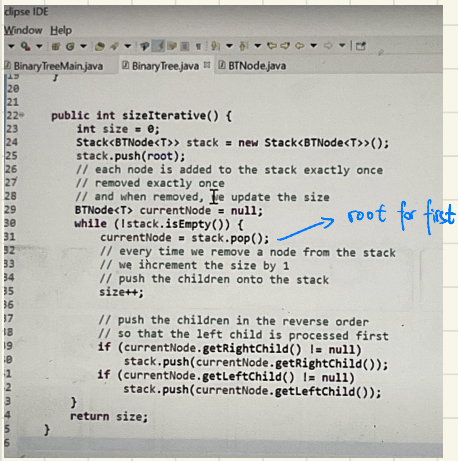
⇒ depth of tree:

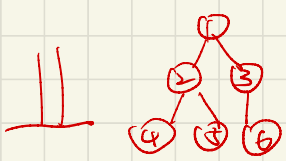
```
return max(getHeight(root.left), getHeight(root.right)) + 1
```

Excrise:

→ not used recursion to find size of tree

Using Stack, push and pop





1

2 3

2 6

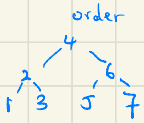
2

4, 5

F2: Current

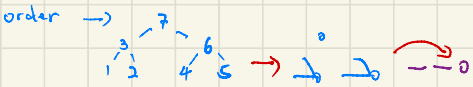
Traversing Binary Trees

- InOrder traversal : check tree first
  First visit left subtree → root → right subtree ⟹ merge them in Array



- PreOrder
  First visit root → Left subtree → right subtree

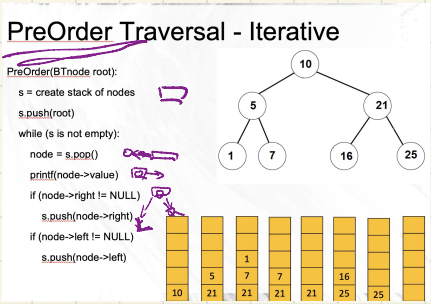
- PostOrder
  First Visit left subtree → right subtree → root

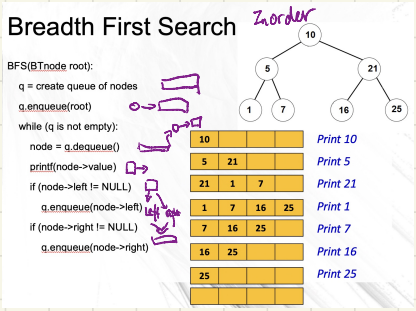


- Q: Change the algorithm to get the InOrder/postOrder/preOrder traversal
  A: Change Order print (root.data) (root.left) . (root.right)

- Traversal Iterative
  ↳ Use Stack (push . pop) first in last out to construct a tree

- Breath First Search
  ↳ Use Queue first in first out (same structure to Stack





InOrder

- Running time :
  O(size of the tree) ∵ Each node is processed in O(1) time.

Tips :
Queue in Java is LinkList

?

Evaluating arithmetic expression.

Eg. (operand 1) + (operand 2)

Solve: Find operand 1 → term 1, Find operand 2 → term 2, Apply term 1 and term 2

Q1: How we know where the first operand ends?

A1: Count parentheses

Q2: How we evaluate the first operand?

A2: recursion.

A1: () - Legal

(()) - Legal

(()()) - legal

()) - illegal 结束=0，不过为负数

Method: Add counter, Add and minus (Never negative) (end with zero)

Q3: What is running time for the expressions below?

- (1 + (1 + (1 + (... (1+1) ...))))

A3: Find first term take O(1) + recursion on n-1 terms

The total running time is $T(n) = O(1) + T(n-1)$

$\Rightarrow T(n) = O(n)$

Q4: What is running time for reverse

- ((((... (1+1) ...+1)+1)+1)

A4: Finding the term takes $O(n)$ + Recursions: $T(n-1) + O(1)$

So, $T(n) = O(n^2)$ in total (op1) + (op2)

Or

Change $O(n)$ change $C \cdot (n-1)$ / $C \cdot n$

$\Rightarrow$ So, $T(n) = C(n-1) + C(n-2) \ldots = O(n^2)$

$C(n)$ or not

It's fine

?

Prefix and Postfix notation

Polish notation

Prefix notation (No parentheses required)

Eg. $((16/(8-(2+2)))\cdot 3) \rightarrow \cdot/16-8+223$

* Operand Operand

Postfix notation (no parentheses required)

Eg. $((16/(8-(2+2)))\cdot 3 \longrightarrow 16822+-/3\cdot$

Operand Operand *

Evaluating Prefix notation

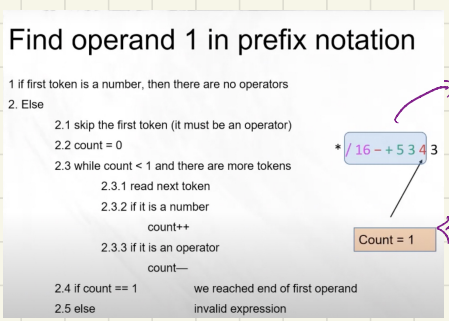
- Q.: How term ends

  A.: #number is n, operators are n-1, reach this → then ends

  like, in arithmetic expression, # operators = # number - 1

  And this holds for each term recursively

  If # operand = n, operator = n-1



Tips: may require quadratic time, to break the expression into operands every time.

- Q2: Can we evaluate an arithmetic expression in linear time?

  A2: Stack

Eg. Algorithm

1. Read right to left
2. Push number
3. Push operator
   a) remove the top two terms
   b) apply operator, and push result onto the stack

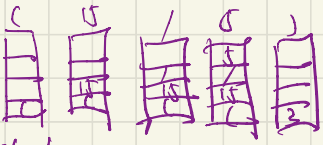
/84

Stack, number only, no operator

Evaluating Postfix notation

Algorithm

1. Read right to left
2. Push number
3. Push operator
   a) remove the top two terms
   b) apply operator, and push result onto the stack



Evaluating infix in linear time

Algorithm

1. If we see '(', push it to the stack
2. If we see an operator or a number, push it to the stack
3. If we see ')' → remove the last 4 items from the stack
   → evaluate, and push the result onto the stack

Represented by a binary tree?

Algorithm:

1. Evaluate left subtree using recursion
2. Evaluate right subtree using recursion
3. Apply the operator in the root.

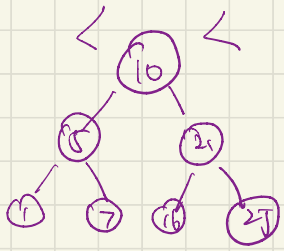
Q1: what is the stopping condition of the recursion
A1: leaf ← no child vertix

Binary Search Trees (BST)

Binary Search Trees

A *binary search tree (BST)* is a binary tree with the following property:

1. For every node, the value in the node is **greater or equal** than the values in its left child *and all its descendants*.
2. For every node, the value in the node is **smaller or equal** than the values in its right child *and all its descendants*.

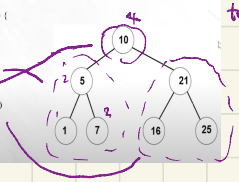


Binary Search Trees

Write a function that prints the values in a BST in the non-decreasing order.

maybe Increasing

```
void printSorted (BTNode root) {
    if ( root == null)
        return;
    printSorted(root.leftChild)
    print(root.data)
    printSorted(root.rightChild)
}
```

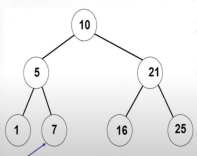


inorder traversal

Write a function that finds a given item in a BST (or returns NULL).

find(7)

```
BTNode find (BTNode root, T item) {
    if ( root == null )
        return null;
    if (root.data.compareTo(item) == 0)
        return root;
    if (root.data.compareTo(item) > 0)
        return find(root.left, item);
    if (root.data.compareTo(item) < 0)
        return find(root.right, item);
}
```



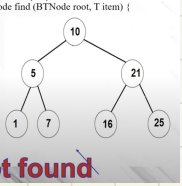
Found

Found : Compare

Write a function that finds a given item in a BST (or returns NULL).

find(8)

```
public <T extends <Comparable<T>> BTNode find (BTNode root, T item) {
    if ( root == null )
        return null;
    if (root.data.compareTo(item) == 0)
        return root;
    if (root.data.compareTo(item) > 0)
        return find(root.left, item);
    if (root.data.compareTo(item) < 0)
        return find(root.right, item);
}
```



Not found

Time:

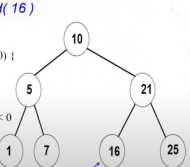
Running Time : (worst case)

O(depth of the tree)

Write an **iterative** function that finds a given item in a BST (or returns null).

find(16)

```
BTNode find (BinarySearchTree bst, T item) {
    BTNode current = bst.root;
    while ( current != null
        && current.data.compateTo(item) != 0) {
        if (current.data.compareTo(item) > 0)
            current = current.left.
        else  // current.data.compareTo(item) < 0
            current = current.right;
    }
    return current;
}
```



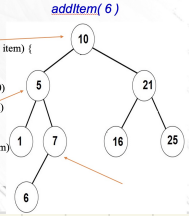
Found

iterative function:
自己創 Variable

Write a function that adds a given element to a binary search tree

addItem(6)

```
/** returns the added node */
BTNode addItem (BinarySearchTree bst, T item) {
    ...
    if (parent.data.compareTo(item)>0)
        parent.addLeftChild(item)
    else
        parent.addRightChild(item)
}
```



Add
L> CompareTo

> 0 -> 小數
< 0 -> 大數

L: 18 3: 12 2: 120

Q. : Describe all sequences of number 1 ... n that give a BST of full depth.
Write an algorithm that outputs all the sequences.

A. : First number must be either 1 or n (only 1 child, otherwise 2) (min or max)
what next

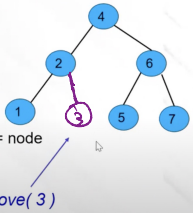
Removing an element from BST

Removing an element from Binary Search Trees

- Step 1: Get a pointer to the node we want to remove
- Removing a node with no children:
  - Just remove the vertex, and update its parent.
- Removing a node with one child:
  - Update the parent to skip over the removed node, and point to its (unique) child.

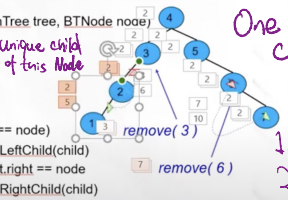
- Note that if the removed node is the root, then we should update the pointer to the root accordingly.

```
removeNode(BinarySearchTree tree, BTNode node)
    if (tree.root == node)
        tree.root = null;
    else
        if (node.parent.getLeftChild() == node)
            node.parent.setLeftChild(null)
        else  // node.parent.getRightChild() == node
            node.parent.setRightChild(null)
```




```
removeNode(BinarySearchTree tree, BTNode node)
    child = getChild(node)
    if (tree.root == node)
        root = child
    else
        if (node.parent.left == node)
            node.parent.setLeftChild(child)
        else  // node.parent.right == node
            node.parent.setRightChild(child)
```

Unique child of this Node



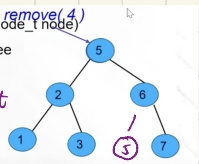
One child case

1. Splay node
2. delete

two children case

```
removeNode(BinarySearchTree tree, BTNode_t node)
    next = find successor of node in the tree
    // next has ≤1 child
    node.data = next.data
    removeNode(tree, next)
```

copy to root



Putting the predecessor in the root also works

good: no children (≤ 1 child)

Running time of the operations

What is the running time of findElement () ?

What is the running time of addElement() ?

What is the running time of removeElement() ?

Answer = O(depth of tree)

Conclusion: we want the tree to be balanced, i.e., have no *long strings*.

Best case: O(log(n)) for each operation.

Worst case: Θ(n) for each operation.

Average case: O(log(n)) – if elements are added in a random order

Eg. Fix a large n (1000000), random permutation $1 \cdots n$

Build a BST from this sequence

$\Rightarrow n = 1000000$. the depth $\approx \log_2 1000000$

Balancing the tree

Creating a balanced tree from a sorted array

Idea:

1) Set the median to be the root

2) Construct left and right subtrees recursively.

```
addArrayToTree ( array , first , last )
    if (first <= last)
        mid = (first+last)/2
    addToTree as the root( array[mid] )
    addArrayToTree as left subtree ( array , first , mid-1 )
    addArrayToTree as right subtree ( array , mid+1 , last )
```

找中间 element

Eg.

Example: Array = [1,2,3,4,5,6,7,8]

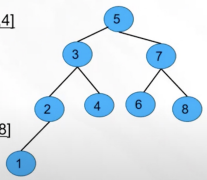
Root = 5

Create left subtree from [1,2,3,4]

- Root = 3
- Create subtree from [1,2]
- Create subtree from [4]

Create right subtree from [6,7,8]

- Root = 7
- Create subtree from [6]
- Create subtree from [8]



Running time

Idea:

1) Set the median to be the root

2) Construct left and right subtrees recursively.

Running time: O(size of the tree)

Because for each node we:

1. Add it to the tree
2. Make recursive calls to get its left and right children.

More formally: $T(n) = O(1) + 2T(n/2) = O(n)$

Q: Balancing tree How?

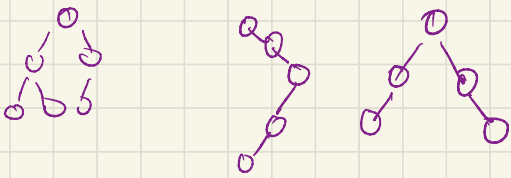
A: Have a self-balancing tree, that is, balanced after each modification (add/remove)

Like. AVL tree ...

BST operations:

- addElement ()
- findElement ()
- removeElement ()

It can be



AVL Tree (balanced BST)

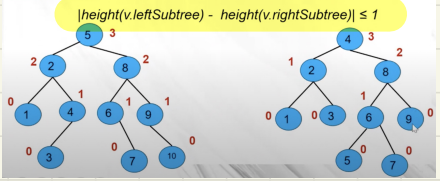
AVL Trees

AVL trees: named after the inventors
Georgy Adelson-Velsky and Evgenii Landis (paper from 1962).

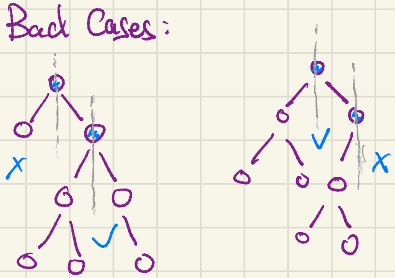
AVL tree is a Binary Search Tree, with the following property:

AVL property: *For vertex v in the tree*

$|height(v.leftSubtree) - height(v.rightSubtree)| \le 1$

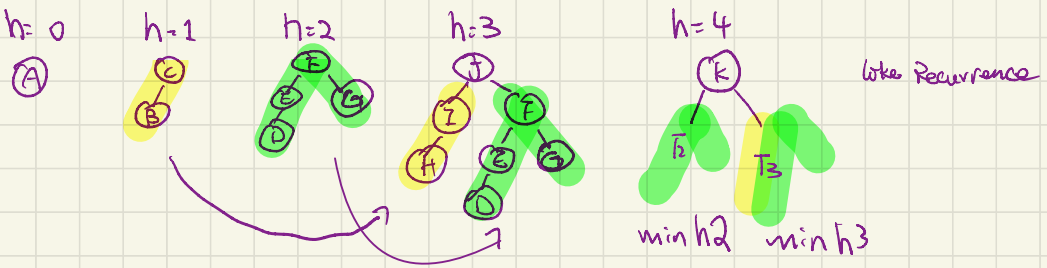


Bad Cases:



(Tips: Seeking Balance By Rotation) etc

Small AVL Trees:



like Recurrance

AVL Trees

AVL: $Size \ge 2^{h/2}$

Claim: If for vertex v of the tree

$|height(v.leftSubtree) - height(v.rightSubtree)| \le 1$

Then height $\le 2 \log_2(size)$. Equivalently size $\ge 2^{height/2}$

Proof: induction on the height of the tree.

- height=0 ➔ size=1 $\ge 2^{height/2}$
- height=1 ➔ size≥2 $\ge 2^{height/2}$.

Therefore, height = $1 < 2 = 2*\log_2(2) \le 2 \log_2(size)$

- Suppose height=h for h≥2. Then

size $\ge$ 1 + min-size(h-1) + min-size(h-2) $\ge 1 + 2^{(h-1)/2} + 2^{(h-2)/2} > 2^{h/2}$.

Claim: If for every vertex v of the tree it holds that

$|height(v.leftSubtree) - height(v.rightSubtree)| \le 1$

Then for a tree of height h we have $size_h \ge Fib(height+1) > 1.6^h$

Fib(0)=1, Fib(1)=1, Fib(2)=2, Fib(3)=3, Fib(4)=5 Fib(h) = Fib(h-1)+Fib(h-2)

Proof: induction on the height of the tree.

- height=0 ➔ $size_0 = 1 = Fib(1)$
- height=1 ➔ $size_1 \ge 2 = Fib(2)$.
- Suppose height=h for h≥2. Then

by induction hypothesis

$size_h \ge 1 + min\text{-}size(h-1) + min\text{-}size(h-2) \ge Fib(h) + Fib(h-1) = Fib(h+1)$.

Claim: If for every vertex v of the tree it holds that

$|height(v.leftSubtree) - height(v.rightSubtree)| \le 1$

Then for a tree of height h we have $size_h \ge Fib(height+1) > 1.6^h$.

Conclusion: if an AVL tree has N nodes, then height(tree) < $\log_{1.6}(N)$

$N > 1.6^h$ ↔ $\log_{1.6}(N) > h$

↔ $h < \log_{1.6}(N) = \log_{1.6}(2) * \log_2(N) = O(\log(N))$

$H < \log_{1.6}(N)$

AVL Trees

Implementation:

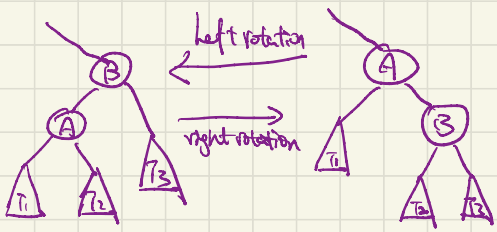
```
public class AVLNode<T> {
    private T data;
    private AVLNode<T> leftChild;
    private AVLNode<T> rightChild;
    private AVLNode<T> parent;
    private int height;
    // private byte balance; = leftChild.height - rightChild.height OPTIONAL
```

Better implementation:

```
public class AVLNode<T extends Comparable<T>> extends BTNode<T> {
    // inherits data, left, right, parent from BTNode
    private int height;
    // private byte balance; OPTIONAL
}
```

AVL trees – rotation

Eg.



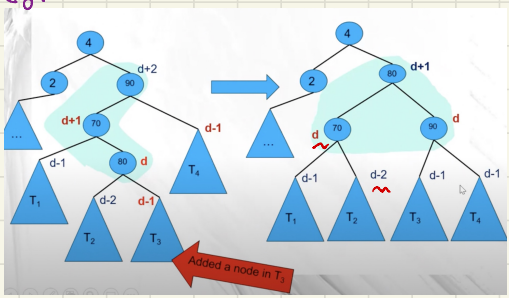
Same Value

⇒ AVL Trees – insertion (an idea)

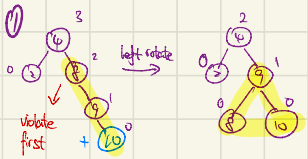
Add(element)

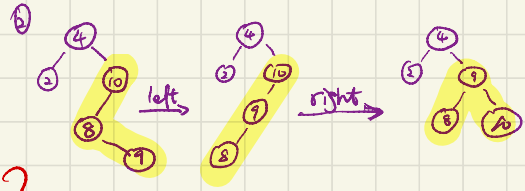
1) Insert a new node as usual, i.e., add as a leaf in BST
2) Update the heights of all ancestors of the added node
3) If a node becomes unbalanced (balance becomes +2 or -2)
   Apply "rotation" to fix the balance in this node.

Eg.



Eg.





depth?

Tips: If two grandchildren have the same height, rotate using the grandchild in the same direction as the child

AVL Tree - deletion (an idea)

remove (element)

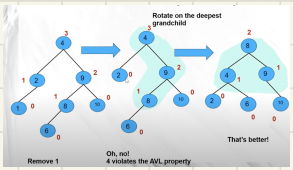
1) remove a node as in a BST (depending on the number of children)
2) Update the heights of all ancestors of the added node
3) If a node becomes unbalanced (balance becomes +2 or -2)

   Apply "rotation" to fix the balance in this node.

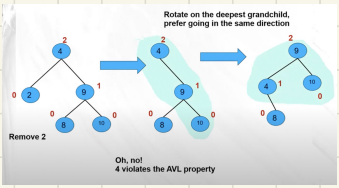
   Rotation is done on the grand child with greatest height

   If two grandchildren have the same height, we rotate using the grandchild in the same direction as the child

Eg



Eg.



AVL Tree compared to plain BST

AVL Trees Pros:

- The tree is always balanced – all operations run in O(log(n)) time
- Rebalancing costs a constant factor of each operation

AVL Trees Cons:

- Rebalancing is not free
- Maybe having O(n) time operations sometimes is no big deal, as long as it's O(log(n)) on average
- Inefficient when done in database systems on disk – writing on disk is costly, especially when writing is in different blocks.
- Difficult to program (you'd be surprised, but this is a real consideration in the industry.)

Conclusion: GOOD CODERS ARE APPRECIATED

Rebalancing is rotating 3 vertices this cost a Constant time (or log n)

Height related to Speed

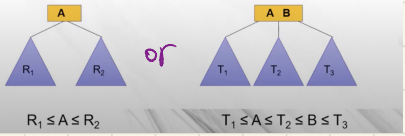
2^h = height $\log_2(n)$

2-3 Trees

2-3 trees

2-3 tree properties

- Every non-leaf vertex has 2 or 3 children
- Every vertex stores 1 or 2 values (as opposed to standard BSTs)
- The values in a node are more than everything on the right, and less than everything on the left
- All leaves has the same depth (not true for AVL trees)



2-3 trees - depth

- Every node has 1 or 2 values
- Every non-leaf vertex has 2 or 3 children
- All leaves has the same depth

Therefore, in a tree of height h with N elements

the number of vertices V is $N/2 \le V \le N$

- $V \ge 1 + 2 + 2^2 + \ldots + 2^h \ge 2^h$
- $V \le 1 + 3 + 3^2 + \ldots + 3^h \le 3^{h+1}$

In other words, the height of the tree is

$$\log_3(N/2)-1 \le \log_3(V)-1 \le h \le \log_2(V) \le \log_2(N)$$

In other words, we have $h = \Theta(\log(N))$

If every vertex has 2 elements → $\frac{N}{2}$

one element → N

element in node

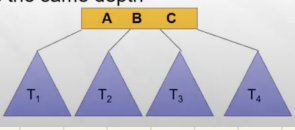
B-trees

B-trees

parameter

B-tree of order m (m个)

- Root has between 2 and m children (2-m个 children)
- Every non-leaf non-root has between m/2 and m children ($\frac{m}{2}$-m个 children)
- Every vertex stores a value between every two children (每2个children中存一个element)
- The values in a node are more than everything on the right, and less than everything on the left
- All leaves has the same depth

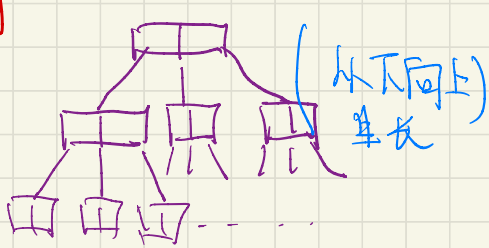


lecture only focus on m=3

B-trees – insertion

add it as a leaf

- Add the new value in a leaf
- If after the addition the leaf has 2 values ← 
  STOP
- If the leaf has 3 values (the leaf is "overfilled")
  We split it into two nodes:
  Take the median to be the parent (放上来的就是parent当)
  Values < than the median are put in the new left node
  Values > than the median are put in the new right node
  Add the median to the parent (mid)
  ** Fix the parent if needed (e.g. using recursion)



B-trees – deletion

- Find the value in the tree
- If the value is not in the leaf
  Find its predecessor (in a leaf)
  move the predecessor to the deleted value
- So we need to handle only deletions from leaves
- When deleting a node from a leaf, the leaf can become empty
  This sometimes called "underflow"
- In this case we need to rebalance the node

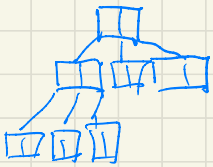
旁边node ∴需rotate value
Always delete ~~value~~, not inner node from leaves

On average the rebalancing time is O(1)

- When deleting a node from a leaf, the leaf can become empty
  This sometimes called "underflow"
- In this case we need to rebalance the node

Rebalancing:

1. If possible, borrow from the right sibling
2. Otherwise, if possible, borrow from the left sibling } 1 way
3. Else, all siblings have only one value
   Merge the parent with the right sibling, if possible
   Otherwise merge the parent with the left sibling } 1 way
   ** Fix the parent if needed (e.g. using recursion)



可删value
不删node

Master Theorem.

$T(n) = a*T(n/b) + f(n)$

$C = \log_b(a)$

$f(n) = O(n^d)$

Master Theorem

Master Theorem: Suppose we are given a recurrence relation

$$T(n) = a*T(n/b) + f(n)$$

for integers a, b, and some function f.

- Let $c = \log_b(a)$.

Then

1. If $f(n) = O(n^d)$ for some $d<c$, then $T(n) = \Theta(n^c)$

Example: If a=3, b=2, then c=1.585.

If $f(n) = O(n)$, then $d=1 < 1.58 = c$. then $T(n) = \Theta(n^{1.58})$

2. If $f(n) = \Theta(n^c)$, then $T(n) = \Theta(n^c \log(n))$

Example: a=8, b=2. This means we make 8 recursive calls on T(n/2)

Then $c=\log_b(a)=\log_8(2)=3$

Suppose $f(n) = O(n^3)$. So d=3, and c=3=d.

In this case $T(n)= \Theta(n^3 \log(n))$

3. If $f(n) = \Omega(n^d)$ for some $d>c$, then $T(n) = \Theta(f)$

Example: a=4, b=2. This means we make 4 recursive calls on T(n/2)

Then $c=\log_b(a)=\log_2(4)=2$

Suppose $f(n) = O(n^3)$. So d=3, and d > c.

In this case $T(n)= \Theta(n^3)$

1. 找 c → d 来自 f(n), n 的 power
2. c 与 d 对比

- $d<c$
- $d=c$
- $d>c$

不满足 Master Theorem:

1. $T(n) = T(n/2) + T(n/4) + O(n)$
2. $T(n) = 2*T(n/2) + O(n*\log n)$ 不是 poly
3. $T(n) = T(n-1) + O(n)$ → 不是 n/b
4. $T(n) = 4*T(n/2) + O(n^3) + 2*T(n/6) + O(n)$ → 两个 T(递归)

Eg.

1. $T(n) = 2T(n/2) + O(1)$
   ∵ $C = \log_2(2) = 1$ and $f(n) = \Theta(n^0)$
   ∴ Then $T(n) = \Theta(n)$
2. $T(n) = 2T(n/2) + O(n)$
   ∵ $C = \log_2(2) = 1$ and $f(n) = \Theta(n^1)$
   ∴ Then $T(n) = \Theta(n \log(n))$
3. $T(n) = 3T(n/2) + 3n^2$
   ∵ $C = \log_3 2 = 1.58$ and $f(n) = \Theta(n^2)$
   ∴ Then $T(n) = \Theta(n^2)$
4. $T(n) = 4T(n/2) + 4n^{1.5}$
   ∵ $C = \log_2(4) = 2$ and $f(n) = \Theta(n^{1.5})$
   ∴ Then $T(n) = \Theta(n^{\log_2(4)}) = \Theta(n^2)$

Some Cases Master not applied:

a, b must Constant

n/b needed, Only one T might solve.

Faster Integer Multiplication -

Eg.

Input: two positive integers A, B

Output: their product, A*B

Example of an explicit computation:

26*43 = (20+6) * (40+3)

= (2*4) * 100 + (2*3 + 6*4) *10 + (6*3) *1

= 8*100 + 30*10 + 18*1 = 800 + 300 + 18 = 1118

Crazy method:

26*43 = ?

Units = 3*6 = 18

Hundreds = 2*4 = 8

Tens = (2+6)*(4+3) – Unit – Hundreds = 8*7 –18 – 8 = 30

Answer = 8*100 + 30*10 + 18*1 = 1118

Goal: Given two positive integers A and B in decimal with n digits each, compute A*B.

Idea: divide and conquer - use recursion.

Let's write A = $a_n a_{n-1} a_{n-2} \ldots a_1 a_0$ and B = $b_n b_{n-1} b_{n-2} \ldots b_1 b_0$

represent them as

$$A = A_1 * 10^{n/2} + A_0 \text{ and } B = B_1 * 10^{n/2} + B_0$$

Then, we have

$$A*B = A_1*B_1 * 10^n + (A_0*B_1 + A_1*B_0) * 10^{n/2} + A_0*B_0$$

We can solve each product recursively and then compute the sum?

We make 4 recursive calls. Therefore, the runtime will be

$$T(n) = 4T(n/2)+O(n) = O(n^2).$$

Integer multiplication in time $O(n^{1.58})$

Goal: Given two positive integers A and B in decimal with n digits each, compute A*B.

Algorithm: divide and conquer - use a sophisticated recursion.

1. Let's write as before

$$A = A_1*10^{n/2} + A_0 \text{ and } B = B_1*10^{n/2} + B_0$$

2. Define:
   - $U = A_0*B_0$
   - $H = A_1*B_1$
   - $T = (A_0+A_1)*(B_0+B_1) - U - H$.
3. Return $H*10^n + T*10^{n/2} + U$

Key observation:

$T = (A_0+A1)*(B_0+B_1) - U - H = A_0*B_1 + A_1*B_0$

We make only 3 recursive calls (!!!). Therefore, the runtime is

$$T(n) = 3T(n/2)+O(n) = O(n^{\log_2(3)}) = O(n^{1.58}).$$

Unbalanced Merge Sort / Merge Sort

Consider the following variant of Merge Sort

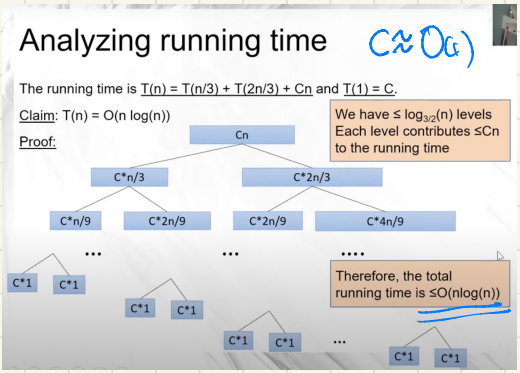
- Given an array A[0…n-1] of length n do
  - Let k = n/3
  - Sort A[0…k]
  - Sort A[k+1…n]
  - Merge the two halves // can be done in time O(n)
- The running time of the algorithm can be written as
- T(n) = T(n/3) + T(2n/3) + O(n), and T(1) = O(1)

two recursive call but unbalanced

running time: $T(n) = O(n \log(n))$

Recall the Merge Sort algorithm

- Given an array A[0…n-1] of length n do
  - Let mid = n/2
  - Sort A[0…mid]
  - Sort A[mid+1…n]
  - Merge the two halves // can be done in time O(n)
- The running time of the algorithm can be written as
- T(n) = 2*T(n/2) + O(n), and T(1) = O(1)
- Then T(n) = O(n log(n))

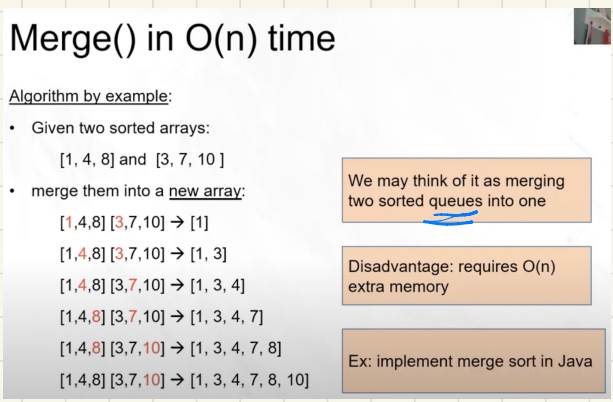


$C \approx O(1)$

$\log_{1.5}$ $\log_2$

every level → $C \cdot n$

contribute



Queue

Q1: Implement Merge Sort without recursion?

A1: Yes. Iterative

Big-O notation

Big-O notation

- We use Big-O notation to express the amount of resources used by algorithms:
  - time complexity (running time)
  - space complexity (used memory).
- Denote by f(N) the running time of a program for inputs of size N. Examples:
  - $f(N) = 1.5N^2 + 4N + 3$
  - $f(N) = 2N^4 + 10N^3 + 4N^2 + 900 \log(N) + 3000$
  - $f(N) = 2N - 100$ (for N>10)
- The big-O notation will be the *fastest increasing term* for large inputs.
- Example: if $f(N) = 2N^4 + 10N^3 + 4N^2 + 900 \log(N) + 3$ we will say that the time complexity is $O(N^4)$.

Q1: Why are we ignoring low order terms?
A1: negligible as N grows

- Definition: Let f(N) and g(N) be two functions on positive integers.

*We say that f = O(g) if there exists a large constant C (e.g. C = 1000) such that f(N) < C*g(N) for all sufficiently large N.*

- Equivalently:

f = O(g) if there C>0 (e.g. C = 1000) such that f(N)/g(N) < C for all N large enough.

f = O(g) if $\limsup_{N\to\infty} f(N)/g(N) < \infty$

It is possible that $\limsup_{N\to\infty} f(N)/g(N) = 0$

O(n)

Ex

- Definition: Let f(N) and g(N) be two functions on positive integers.
- We say that f = O(g) if there exists a large enough constant C (e.g. C = 1000) such that f(N) < C*g(N) for all sufficiently large N.
- Example1: $f(N) = 5N^2 + 4N + 3$. Want to show $f = O(N^2)$

  Let C=12.

  Then $f(N) = 5N^2 + 4N + 3$

  $< 5N^2+4N^2+3N^2$ [for N>2]

  $= 12N^2 = CN^2$

  Therefore $f = O(N^2)$

- Definition: Let f(N) and g(N) be two functions on positive integers.

We say that f = Θ(g) if f=O(g) and g = O(f).

- Equivalently:

f = Θ(g) if there C,d>0 (e.g. C = 1000, d=0.001) such that d < f(N)/g(N) < C for all N large enough.

f = Θ(g) if $\limsup_{N\to\infty} f(N)/g(N) < \infty$ *and* $\liminf_{N\to\infty} f(N)/g(N) > 0$.

Θ(n) =

Definition: Let f(N) and g(N) be two functions on positive integers.

We say that f = Θ(g) if f=O(g) and g = O(f).

Example: $f(N) = N^3 + N + 3$. Then f is not $\Theta(2^N)$

Let $g(N) = 2^N$

Then $\lim_{N\to\infty} f(N)/g(N) = 0$

So f is not $\Theta(2^N)$.

- Definition: Let f(N) and g(N) be two functions on positive integers.

We say that f = Ω(g) if g = O(f).

- Equivalently:

f = Ω(g) if there d>0 (e.g. d=0.001) such that f(N)/g(N) > d for all N large enough.

f = Ω(g) if $\liminf_{N\to\infty} f(N)/g(N) > 0$.

Ω <

Eg.

- Addition: Design an algorithm that gets two numbers each with n digits, and computes their sum.
- Q: What is the running time of the algorithm?
- A: O(n) for the standard long addition.

If a decimal number has n digits, how large is this number?

- Multiplication: Design an algorithm that gets two numbers each with n digits, and computes their product.
- Q: What is the running time of the algorithm?
- A: $O(n^2)$ for the standard long multiplication.
- Fact: can do much faster than n^2.
- Relatively easy to get $O(n^{1.58})$
- Can be done in O(n log(n))

Priority Queue

? LL can't Binary Search

- Priority Queue: an order collection of items O(nlogn) best possible
  - Some operator: (5个)
    - add item (item): 给加一个 item 到 Queue
    - getTop(): 返回一个 highest priority 的 item, 但 not removing it
    - removeTop(): remove 一个 highest priority 的 item, 返回之
    - getSize/isEmpty

但是我们不想这么做, instead, 用 Heap

Heap

Min-heap

Heap is an implementation of Priority Queue.

Sometimes priority queues are called "heaps", because heaps are their most common implementation.

We will typically use min heap. That is, the minimal element is removed first from the heap

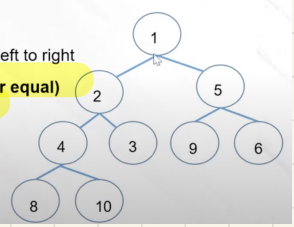
Public methods:

- addElement(element)
- getMin()
- removeMin()
- getSize()

How to use

Properties:

- Heaps are complete binary trees
- The vertices in the last row are added left to right
- **The value in each node is smaller (or equal) than the values of its children**
- No requirements about other relatives



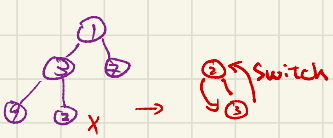
getMin():

1. Return the value at the root. 返回 root 值

Running O(1)

- Add Item(item)
  - Add item to the next available position
  - Satisfy min-heap condition

Running O(log(n))



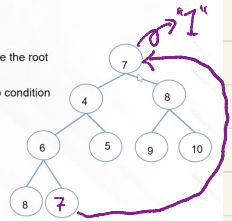
- Remove Min

Equal Value in tree? Okay

operation {

Remove min:

1. min = root.value
2. Move the last element in the tree be the root
3. Propagate the root down to satisfy the min-heap condition
4. Return min (from step 1)



1. 把 3 换上来 switch
2. 重新排序 rearrange order

- Heap Using Array Representation

Set Heap as Array

array[0]

array[2i+1]

array[2i+2]

array[(j+1)/2-1]

Since the for heaps the tree is always full, it is convenient to represent it using an array:

More specifically, we want to store

1. The size of the tree
2. The BFS traversal of the heap

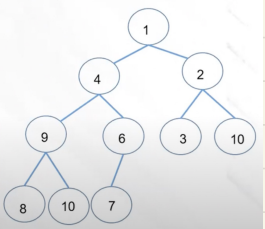
In this example: [1,4,2,9,6,3,10,8,10,7]

How can we find the root in the array?

Where is the left child of array[i]?

Where is the right child of array[i]?

Where is the parent of array[j]?



- Public operations: 3
  1. addElement(element) - in O(log(n)) time
  2. getMin() - in O(1)
  3. removeElement - in O(log(n))

Use in other: Can use heap to design an O(nlog(n)) sorting algorithm

↳ Given an array of elements

Add all to a heap ← linear time O(n)

Remove one by one, insert them into array in the increasing order.

O(nlogn) ←

HeapSort

Graphs:

Graph: a collection of nodes and a collection of edges, where each edge connects a pair of nodes. denoted $G = (V, E)$

Graph representation.

→ 2 standard representation:

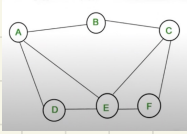
1. Matrix representation
2. Adjacency list representation.

Matrix representation

Given a graph G=(V,E) with |V|=N vertices

A symmetric matrix of size NxN with

- M[i,j]=1 if (i,j)∈E
- M[i,j]=0 if (i,j)∉E



	A	B	C	D	E	F
A	0	1	0	1	1	0
B	1	0	1	0	1	1
C	0	1	0	0	1	1
D	1	0	0	0	1	0
E	1	1	1	1	0	1
F	0	1	1	0	1	0

对称矩阵

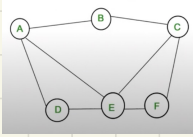
→ check if they are connected

Adjacency list representation

Given a graph G=(V,E) with |V|=N vertices

We have N lists (linked list or array), one for each vertex

The list of a vertex v∈V hold all neighbours of v.



A:	B	D	E	
B:	A	C		
C:	B	E	F	
D:	A	E		
E:	A	C	D	F
F:	C	E		

邻接表

Implicit graph representation

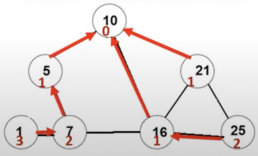
It is also possible to work without having the graph

- For example, because the graph is too large to hold in the memory
- We can only store the nodes that we have seen so far.
- Similar to exploring a new territory, and creating a map on the fly.

Breadth First Search. → exploring a graph

Goal: an algorithm that gets a graph G = (V,E), and a starting vertex, and computes the distance from start to all other vertices in G.

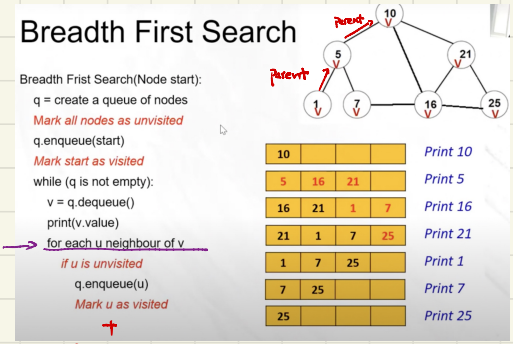
BONUS: if for each node v∈V we have the distance from start to v, then we can also compute a shortest path from start to v.



$O(|E|)$ ⇒ 快

Recall BFS on trees

Breadth First Search



Set parent(u) = v in the tree ≈ visited code

Q1: Running Time of BFS algorithm

A1: We touch every edge exactly twice.

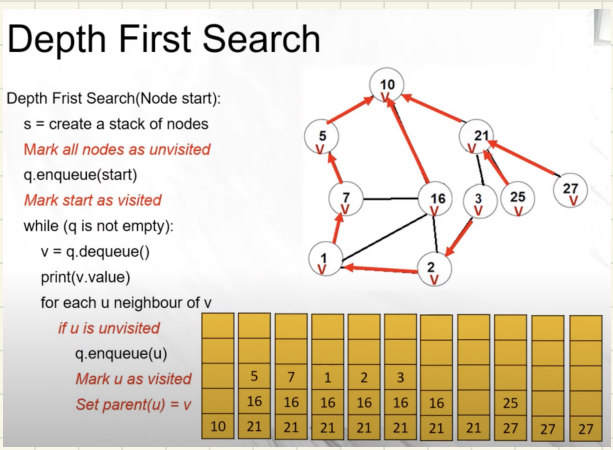
This is assuming G is connected.

Therefore, the total running is O(|E|) in adjacency list model

If the graph isn't connected, then the running time is linear in the number of edges in the connected component of start

Depth First Search (using stack instead of Queue)

Depth First Search



Running time

Q: what is the running time of the Depth First Search algorithm?

A: Same as for BFS, we touch every edge exactly twice (once from each side).

This is assuming that G is connected.

Therefore, the total running time is O(|E|) in the adjacency list model.

Same algorithm above

If the graph is not connected, then the running time is linear in the number of edges in the connected component of start.

*In the matrix representation the running time is $O(|V|^2)$

O(|E|)

BFS vs DFS

BFS	DFS
Finds the shortest distance from the starting vertex to all v∈V.	Usually does not give the shortest distance.
Efficient if the target is close to the starting vertex.	Reaches far away vertices quicker.
More structured	More useful for solving puzzles than BFS.
	Useful for other applications (topological search)

- Both create a spanning tree of the connected component.
- Both algorithms are uninformed. That is, they don't have a guess about distance to the target.
- Next time: A* - algorithm that uses estimated distance to the target.

Queue

Stack

对当前target比较近，efficient

A* algorithm

A*

A* (Node start, Node target):

For each node define:

- g(v) - distance travelled from start to v 已经走的
- h(v) – estimated distance from v to target - heuristic estimated走的
- f(v) = g(v) + h(v) 全程

Let OpenQueue be a Priority Queue ordering vertices by f(v), min goes out first BFS

- Will be used to hold nodes that are currently processed

Let ClosedSet be a Set BT or HashSet

- Will be used to hold nodes that have already been processed
- Probably implemented using HashSet.
- We'll discuss Sets and HashSets in a bit.
- For now, ClosedSet is just a Set, supporting the operations: insert, remove, find → node

2个 Data structure

OpenQueue：正在processed

ClosedSet：以完成processed

A* (Node start, Node target)

Could be more one target

1. OpenQueue.add(start)
2. while (OpenQueue is not empty)
   - 2.1 v = OpenQueue.getMin() --// min w.r.t f(v) takes O(1) time
   - 2.2 for each u∈v.neighbours() do
     - 2.2.1 if u is a target
       - Set g(u) = g(v) + 1, set u.parent = v
       - Return u
     - 2.2.2 Otherwise: 不is target
       1. If u is already in OpenQueue *with a higher f(u)*, then update f(u) — 比较的 shorter
       2. Else if u is in the ClosedSet *with a higher f(u)*, *then update f(u)* and move it to OpenQueue — shorter? retry
       3. Otherwise add u to OpenQueue
     - 2.2.3 In all three cases in 2.2.2 set u.parent = v
   - 2.3 Move v to ClosedSet

If *h(v)=0* for all v, and we use max-PriorityQueue, we get the **DFS** algorithm

If *h(v)=0* for all v with min-PriorityQueue, we get the **BFS** algorithm

If *h(v)=dist(v,target)* for all v, we go straight to the target

找点 父亲节点，且neighbours

A* with admissible heuristics

Theorem: if we use min-PriorityQueue and h(v) ≤ dist(v, target) for all v, then A* will *eventually* find the target.

Note that *eventually* depends on:

1. Computing of h(v) – trade-off between efficiency and precision
2. Exact implementation of OpenQueue, ClosedSet
3. How fast .equals() works.
4. Size of the search space / total number of vertices
5. Memory used / size of each node
6. Some luck (e.g., use randomness to compute h, or prune some branches in the computation)

Hashing and Hash Tables

- Hashing

operations: Insert / Remove / Contains

best / ideal time: O(1), worst: O(n)

- hash table: an array of some fixed size.
  A hash function: mapping objects into the array

Eg. hash("A", "a") = 9

- Hash functions hashCode()

Hash function

A hash function converts any key (any class) into an int.

The value of the hash function will be an index in the hash table.

Want different keys to be *usually* mapped to different values.

Example: Let's say the hash table is of size 100.

- hash(name, number) = (name.hashCode() + number) mod 100

and name.hashCode() = name[0]*31^{n-1}+name[1]*31^{n-2} +…+name[n-1]

Then

- hash("CMPT225") = (hash("CMPT") + 225) mod 100
- hash("CMPT125") = (hash("CMPT") + 125) mod 100

Both hashes are the same.

HashMap (key, Value)

HashSet (Value)

- Eg. back Example:
  - hash (name, date of birth, SFU ID) = (first two digits of date birth + first two digits of SFU ID) % 100

Then:

hash("Jack", 1997, 301876) = 69 + 30 = 99

hash("Bob", 1967, 300012) = 19 + 80 = 99

Another example: Let's say the hash table is of size 127.

- Hash(id, name, age, height)

  = (id*31 + 19*age*height^2 + name.hashCode()) % 127

and name.hashCode() = name[0]*31^{n-1}+name[1]*31^{n-2} +…+name[n-1]

Looks more difficult to find a collision.

or.

Hashing String.

hashCode("a") = 'a' = 97 (ascii) hashCode("b") = 'b' = 98

hashCode("ab") = 'a' x31 + 'b' = 3105 hashCode("abc") = a x31^2 + b x31 + c = 96354

- Hash tables

Hash tables

A possible solution for this data structure would be the following:

- Create an array of some fixed size, say 1019 (some large prime)
- Add(element):
  - array[element.hashCode()] = element;
- Remove(element)
  - array[element.hashCode()] = **null**;
- Contains(element)
  - **return** (array[element.hashCode()] != **null**);

Q: What if two elements are mapped to the same index?

How should we handle collisions?

↓

A: Two Standard solutions:

1. Seperate Chaining / Closed addressing
2. Open Addressing / Probing

Seperate Chaining — "closed addressing"

Separate Chaining

Idea: each entry in the hash table contains a linked list of all elements mapped here.

When creating the hash table, we create an empty LinkedList for each hash value.

contains?(element)

1. index = element.hashCode() key
2. list = table[index]
3. return list.contains(element)

//equiv.

return table[element.hashCode()].contains(element)

add(element)

1. index = element.hashCode()
2. list = table[index]
3. If (!list.contains(element))

   3.1 list.add(element) – let's say add to front

remove(element)

1. index = element.hashCode()
2. list = table[index]
3. If (list.contains(element))

   3.1 list.remove(element)

Rehashing

Load factor = number of elements / size of the array

To guarantee good performance, the load factor should be less than 1 (often it is expected to be below 0.75).

When array becomes too full (i.e. the load factor exceeds some predefined value), all the values stored in the table need to be rehashed to a larger table

Look at the implementation of HashMap.class

It is a pretty complicated class, but you will learn quite a bit from it.

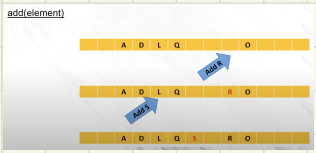
Load factor = # elements / size Array

Open Addressing – linear probing 线性探测

Linear probing: when insertion causes collision, check if the next entry is empty, and put the element there. If not go to the right one more…

add(element)

1. index = element.hashCode()
2. while (table[index] != null)
   2.1 index = index+1 (mod table.length)
3. table[index] = element



delete 标记

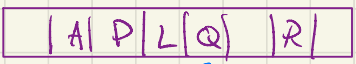
not good

add

contains(element)

1. index = element.hashCode()
2. while (table[index] != null && !(table[index].equals(element)))
   2.1 index = index+1 (mod table.length)
3. return (table[index] != null)

存在 && 相等



If contains(Q)? - No

Yes

remove(element)

1. How should we handle removals?
2. Let's say we went to remove from table[i], but there is another element with same hash, that we added to table[i+1]. What should we do?
3. Or that element was added to table[i+3] because table[i,i+1,i+2] were taken.

When removing mark that location as "deleted"

remove(element)

1. index = element.hashCode()
2. table[index] = deleted

标记成

Here deleted is a special object used by our Hash, e.g.

```
private static final Object deleted = new Object();
```

This will require us to revise our insert() and contains() methods.

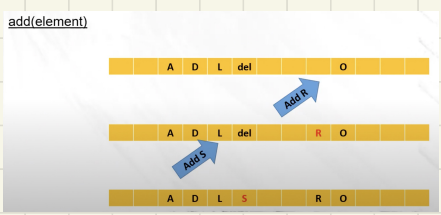
We have a special object *deleted*

```
private static final Object deleted = new Object();
```

add(element)

1. index = element.hashCode()
2. while (table[index] != null && table[index] != deleted)
   2.1 index = index+1 (mod table.length)
3. table[index] = element

存在 && 无标记

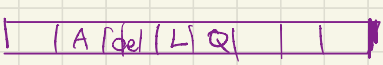


contains(element)

1. index = element.hashCode()
2. while (table[index] != null && !(table[index].equals(element)))
   2.1 index = index+1 (mod table.length)
3. return (table[index] != null)

存在 && 相等(包括不相等)

Same



We have a special object *deleted*

```
private static final Object deleted = new Object();
```

remove(element)

1. index = element.hashCode()
2. while (table[index] != null && !element.equals(table[index]))
   2.1 index = index+1 (mod table.length)
3. if (table[index] != null)
   3.1 table[index] = deleted

A	D	L	del	S		

delete marked as "del"

Clustering — saving time → Resize table to reduce the load factor

Linear probing - clustering

Cluster: A consecutive block of keys

W				F	N	X				A	D	L	del	S		R	O	del	Z	K

Problem: New keys proportionately likely to hash into big clusters.

This causes long insertions, deletions, contains?

Solution: resize the table to reduce the load factor.

For example:

- if load factor > ½, double the length
- if load factor < ¼ , halve the length

Remember: when resizing we rehash all keys, which takes long time.

Variants:

1. Quadratic probing:

contains(element)

1. index = element.hashCode()
2. i=0
3. while (table[index + i^2] != null && table[index+ i^2].equals(element)))
   2.1 i++
4. return (table[index + i^2] != null)

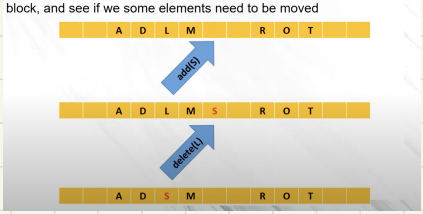
2. Double hashing: two hash functions

contains(element)

1. index = element.hashCode()
2. i=0
3. while (table[index + i*$hash_2$(element)] != null &&
   table[index+i*$hash_2$(element)].equals(element)))
   2.1 i++
4. return (table[index + i*$hash_2$(element)] != null)

3.

Back to Linear Hashing: when removing elements, we can set it to null instead of marking it at deleted. But this requires looking forward at the block, and see if we some elements need to be moved



3-SUM

3-SUM

Input: an array A[0…n-1] of ints

Goal: find i,j,k such that A[i] + A[j] = A[k].

The indices don't have to be all different

Solution in $O(n^2)$ time (on average): or $O(n^2 \log(n))$

1. Store all elements of A in a hash table of size O(n)
2. For i=0…n-1

   For j=0…n-2

   If A[i]+A[j] is in the hash table,

   return i,j, and the index of the value in the array

4. ideas for 3-Sum by hash

- no modulo p

Idea 0: compute M = max(A[i]), create an array of length M, and store each A[i] is HashTable[A[i]]. – In other words: create a boolean table with table[i] = true if and only if I belongs to A.

Problem: the values in A might be huge compared to n. if A too big?

So most of the table will be empty…

Idea 1: Choose a prime p∈[10n, 20n], defined a hash table of size p

For each i=0…n-1

add A[i] to HashTable[A[i] mod p]

Problem: If an "adversary" knows p, the input could be such that all A[i]'s are equal modulo p, and hence go to the same cell.

If we want slow speed

Idea 2: Choose a random prime p∈[10n, 20n], and declare a hash table of size p.

Define hash(x) = x mod p

For each i=0…n-1

add A[i] to HashTable[hash(A[i])]

Looks better now…

Idea 3: Choose a random prime p∈[10n, 20n]

Declare a hash table of size p.

Choose a,b ∈ {0,1…p-1}

Define hash(x) = ax+b mod p

For each i=0…n-1

[Same General formula different parameter]

not Randomness

hash function is fixed

next run would different hash function

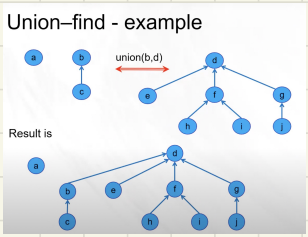
Union-Find

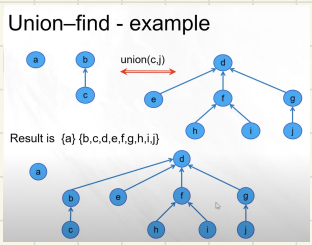
We want a data structure with following operations:

- makeset(x) - Creates an set containing only x
- union(i,j) - merge the Set containing i with the Set containing j
- find(x) - returns the name of the Set containing x

In particular, if i, j are the same set, then find(i) = find(j)

Eg.





Union–find

Q: How should we implement union(i,j)?

Idea:

- We can simply connect the root of one set to the root of the other.
- Connect the try not to increase the depth, by connecting the shallower tree to the root of the deeper one

** We will need to keep track of the height of every tree.

*** No need to remember the height of every node, only the height of the root of each tree.

→ makeSet(x):

```
x.parent = null
x.rank = 0
```

→ find(x):

```
Current = x
while(Current.parent != null)
    Current = current.parent
return current
```

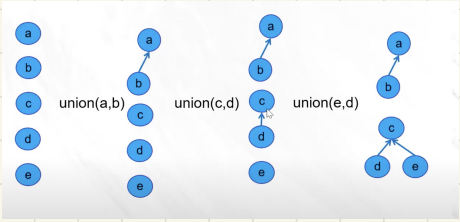
→ Union(u,v):

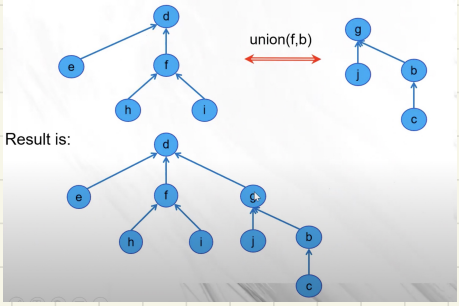
1. $root_u$ = find(u) // root of u
2. $root_v$ = find(v) // root of v
3. if $root_u == root_v$
   return "SAME SET"

height ⇔ rank

4. if $root_u$.height > $root_v$.height
   $root_v$.parent = $root_u$ // $root_u$ will be the new root
5. if $root_u$.height < $root_v$.height
   $root_u$.parent = $root_v$ // $root_v$ will be the new root
6. if $root_u$.rank == $root_v$.rank
   $root_u$.parent = $root_v$ // $root_v$ will be the root
   $root_v$.height++ [// update the rank.]

Eg.





Union Find

```
1
2  public class UnionFind {
3
4      /**
5       *
6       * @param data
7       * @return a Node (a ticket) that can be used to run find and union
8       */
9      public static Node makeSet(Object data) {
10         return new Node(data);
11     }
12
13     /**
14      *
15      * @param node
16      * @return returns the root of the tree containing node
17      */
18     public static Node find(Node node) {
19         if (node.getParent() == null)
20             return node;
21         else
22             return find(node.getParent());
23     }
24
25
26     public static void union(Node u, Node v) {
27         Node ru = find(u);
28         Node rv = find(v);
29         if (ru.getRank() > rv.getRank())
30             rv.setParent(ru);   // ru will be the common root
31         else if (ru.getRank() < rv.getRank())
32             ru.setParent(rv);   // rv will be the common root
33         else {  // ru.getRank() == rv.getRank()
34             ru.setParent(rv);
35             rv.setRank(rv.getRank()+1);
36         }
37     }
38 }
39
```

```
public static void main(String[] args) {
    Node n1 = UnionFind.makeSet(new LinkageError());
    Node n2 = UnionFind.makeSet(new Vector<Integer>());
    Node n3 = UnionFind.makeSet(3.6);
    Node n4 = UnionFind.makeSet("HELLO");
    Node n5 = UnionFind.makeSet(5);

    testSameSet(n1, n2);

    System.out.println("union(n1, n3)"); // {1,3} {2} {4} {5}
    UnionFind.union(n1, n3);

    System.out.println("union(n4, n5)");  // {1,3} {2} {4,5}
    UnionFind.union(n4, n5);

    System.out.println("union(n2, n5)"); //  // {1,3} {2,4,5}
    UnionFind.union(n2, n5);
```

每个Node是以Object类建立
∴每次union需以n1, n2..(有效tickets)的形式所使用，而不是其中的值

Union–find

Running time:

- find(x): If the tree of x has height h, then find(x) runs in O(h) time
- union(u,v): If the tree of u has height h_u and the tree of v has height h_v, then union(u,v) runs in $O(h_u + h_v)$ time
- Conclusion: we should try to keep the trees as shallow as possible

Key Property: If a vertex has rank k has at least 2^k vertices in the subtree rooted at that vertex.

Conclusion: If there are n vertices, then maximal rank=depth is at most $\log_2(n)$. Therefore, find and union run in O(log(n)) time.

Key Property: If a vertex has rank k has at least 2^k vertices in the subtree rooted at that vertex (including itself).

Proof by induction:

- In makeset() we set u.rank = 0, and u has 1 node under it (just itself).

For induction step:

- If in the union(u,v) we had v.rank > u.rank, then the ranks don't change, but the number of vertices under v increases.
- Same for v.rank < u.rank
- If in union(u,v) we had u.rank == v.rank, then both subtrees had at least 2^k vertices by induction. Merging them gives at least 2^{k+1} vertices.

An Application : Minimum Spanning Tree

All vertices
n-1 edges

undirected graph with weighted edges

Minimum Spanning Tree

The Minimum Spanning Tree Problem (MST)

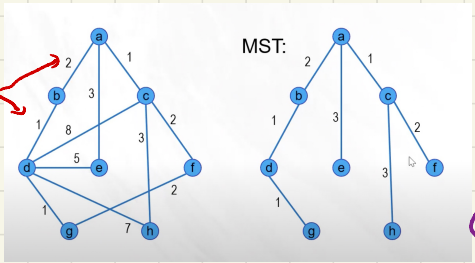
Input: an undirected graph G = (V,E) and costs on the edges $\{c_e : e \in E\}$

Output: a spanning tree of minimum cost

Definition: A spanning tree of a connected graph G=(V,E) is a subset $T \subseteq E$ of the edges such that

1. T has |V|-1 edges
2. the graph (V,T) is connected.

The costs of a spanning tree is defined as $\Sigma_{e \in T} c_e$.



Cost: 13

Connected cycle

Input: an undirected graph G = (V,E) and costs on the edges $\{c_e : e \in E\}$

Output: a spanning tree of minimum cost

Kruskal's Algorithm:

1. Set T = empty set
2. While |T| < |V|-1 do
   a) Pick an edge $e \in E \setminus T$ with minimum weight such that $T \cup \{e\}$ does not contain a cycle.
   b) Add e to T
3. Return T

Q: how can you check if e=(u,v) closes a cycle in T?

A: run BFS/DFS on T from u, and check if v is reachable

run BFS/DFS, if it's reached then, add one more edge will make it a cycle

kruskal : 从最小的 weight 开始

Running Time $O(nm)$

Running time: Suppose the graph has n vertices and m edges (m>n-1).

Naively: We add n-1 edges, each time looking for the lightest edge that does not close a cycle.

Kruskal's algorithm runs in time at most $(n-1) * m * O(n) = O(n^2 m)$

An improvement: We can first sort the edges by their costs

Go over the edges in the increasing order, and for each edge see if it closes a cycle.

Then the running time will be $O(m \log(m)) + m * O(n) = O(nm)$

$O(m \log(m))$

Improvement using union-find:

First sort the edges by their costs.

Go over the edges in the increasing order, and for each edge see if it closes a cycle.

Check if an edge closes a cycle using union-find. How?

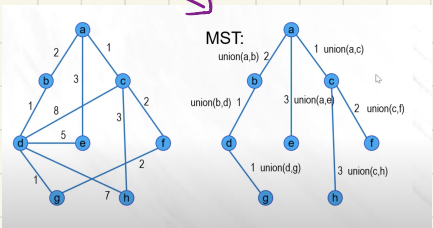
Then the running time will be $O(m \log(m)) + O(m \log(n)) = O(m \log(m))$

Input: an undirected graph G = (V,E) and costs on the edges $\{c_e : e \in E\}$

Output: a spanning tree of minimum cost

Kruskal's Algorithm:

1. Set T = empty set
2. Sort the edges according to c_e's in the non-decreasing order.
3. For each vertex v ∈ V
   makeset(v)
4. For each edge e=(u,v) in the sorted set
   If find(u) != find(v) // if u, v are in different connected components
   union(u,v)
   add e to T
5. Return T



So far: We saw a data structure for union find where each operation takes O(height) time.

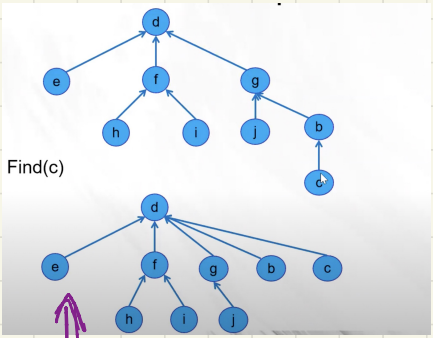
If we added n items, then height ≤ $\log_2(n)$

Therefore, we get O(log(n)) for each operation.

Q: Can we do faster than O(log(n))?

For example, what if in the MST problem the edges of the graph are already sorted? Or if the weights are between 1 and |E|, so we can sort them in O(|E|=m) time

A: Yes, we can get to logloglog(n)… and even faster... amortized time



find(x):

```
current = x
while (current.parent != null)
    current = current.parent

root = current

for each node on the path fro
    node.parent = root
```

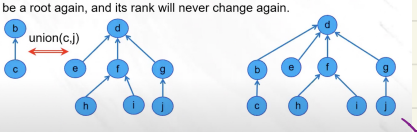
Next time find(x) will Have running time O(1)

Next time find on any ancestor of x Will have running time O(1)

We pay just a bit more each time, But this will give us an improvement compared to O(log n).

Not a key observation: makeSet(x) makes x a root(of itself)

Key observation: once a node is not a root (due to union), it will never be a root again, and its rank will never change again.



Therefore, the ranks of all nodes are unchanged by path compression, even though these are no longer the heights.

Theorem: If we make the following calls:

- makeSet() - n times
- find() - m times
- union() – at most n-1 times

The total running time of all operations is O(n log*(n)) + O(m log*(n)).

Here log*(n) = the number of time you apply $\log_2()$ to get ≤1.

log*()

log*(n) = is the number of time you apply $\log_2()$ to get ≤1.

Examples:

- $\log_2(\log_2(\log_2(16))) = \log_2(\log_2(4)) = \log_2(2) = 1$
  Therefore, log*(16) = 3
- $\log_2(\log_2(\log_2(\log_2(2^{16}=65,536)))) = \log_2(\log_2(\log_2(16))) = 1$
  Therefore, log*(65,000) = 4
- $\log^*(2^{65,536}) = 5$
- For all practical purposes log*(n) ≤ 5 for all numbers n that represent anything semi-meaningful in the universe.

Finding Median

Input: An array of A of length n

Output: the median element in the array in the sorted order.

Example:

- On input A = [1,4,3,7,2,9]
- The output should be 4

Application: we can it in QuickSort
- Helps us find a pivot that partitions the array into two halves

A naïve solution: sort A, and output A[n/2] → O(nlogn)

Q: Can we do fast than O(n log(n)) ?

A: Yes, we can do it in O(n) time

Finding k-th smallest element

Algorithm: Given an array of A of length n, and integer k

1. Divide A into ⌊n/5⌋ arrays each of length 5 (or less)
2. Sort each 5-tuple
3. Call the medians of the 5-tuple their representatives.
4. Recursively find the median of these n/5 representatives
5. Let M be the median of the n/5 representatives.
6. Move all elements greater than M to the right, and all element smaller than M to the left.
7. Let i* be the index of M in the array.
8. If i* == k, return A[k]
9. If i* > k, recursively search in A[0…i*-1]
10. If i* < k, recursively search in A[i*+1…n-1]

find pivot M

i>k ; i<k

M

- Partition into 5 tuples

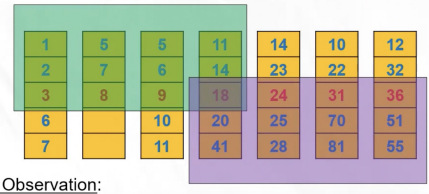
1	5	5	11	14	10	12
2	7	6	14	23	22	32
3	8	9	18	24	31	36
6		10	20	25	70	51
7		11	41	28	81	55

- Find the median of the representatives (using recursion): M = 18
- Let S = all elements ≤M
- Let G = all elements ≥M
- If |S| = k-1, return M
- If |S| ≥ k, apply recursion on (S, k)
- If |S| ≤ k-2, apply recursion on (G, k-|S|-1)

First Recursion for M

Second Recursion for seperate two parts Small / greater

Finding k-th smallest element



Key Observation:

- At least 0.3n numbers are ≤ M
  because there are n/10 representatives ≤ M, and for each 3 numbers are ≤ M
- At least 0.3n elements are ≥ M
  because there are n/10 representatives ≥ M, and for each 3 numbers are ≤ M
- Therefore, the last recursive step is applied on a subarray of size ≤ 0.7n.

Algorithm: Given an array of A of length n, and integer k

1. Divide A into [n/5] arrays each of length 5 (or less)
2. Sort each 5-tuple — O(n) time
3. Call the medians of the 5-tuple their representatives.
4. ***Recursively*** find the median of these n/5 representatives — T(n/5)
5. Let M be the median of the n/5 representatives.
6. Move all elements greater than M to the right, and all element smaller than M to the left. — O(n)
7. Let i* be the index of M in the array.
8. If i* == k, return A[k]
9. Else ***apply recursion*** either on elements <M or on elements >M — At most T(0.7n)

- Running time: T(n) = T(0.2n) + T(0.7n) + O(n)