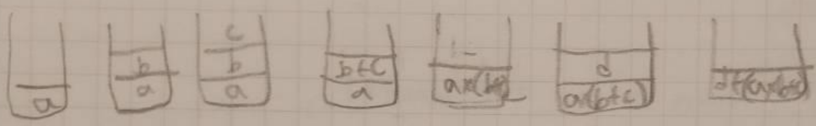
10928156 電機三甲 蔡宜呈

Application of Algebaric Expression

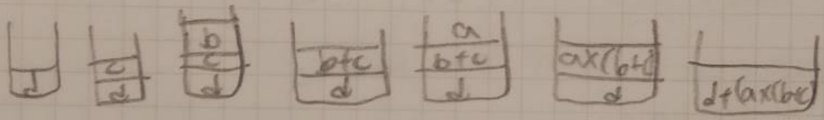
ex. postfix : pop the top 2 operand from the stack
while operator is entered
pushes the result back to the stack

ex.

abc + ×d+ : postfix



Prefix +×a+bcd



Infix to postfix

first save all the operands and operators to each two stacks
and compare the two operator's precedence, execute
the operator who has the higher precedence first
or if u have braces than no need to compare the precedence

$((a\times(b+c))+d)$

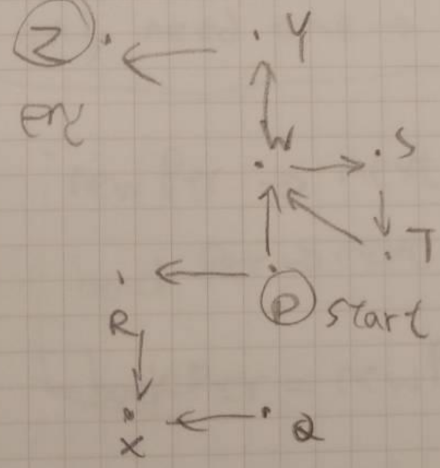
+
+
*

Search-a path

find the next city and marked the origin as visited

if ~~w~~ the next city is visited then than go back to the city where it starts and do it again

and if there is no way to go ~~and also we~~ can't and it didn't reach to the destiny ~~then~~ go back to the first location



push P → next unvisited

R

X

pop X ← No unvisited

R

push W

S

T

pop T

S

push Y

Z

Queue

FIFO

Application

Reading astring

Recognize Palindrome

simulation

→ anything about lining

is Empty

enqueue

dequeue

getFront

~~de~~

palindromes

→ save the string to queue and stack

while(!queue.isEmpty) queue.pop == stack.pop

stack.push as same as Queue.enqueue

~ .pop (top) ~ ~ .dequeue(front)

if (top != front)

cout~~ ~~n it isn't palindrome

Implementations of the ADT Queue

A linear linked list with two external references

one front →
one back

A circular ——— one

only need one → back

newPtr → next = NULL;
backPtr → next = newPtr
backPtr = newPtr;

enqueue →

```
QueueNode *newPtr = new QueueNode
newPtr -> item = newItem
newPtr -> next = NULL
if (isEmpty())
    frontPtr = newPtr.
else backPtr->next = newPtr
backPtr = newPtr
```

newPtr -> next = NULL
backPtr -> next = newPtr
backPtr = newPtr; →

dequeue

tempPtr = frontPtr

frontPtr = frontPtr -> next

tempPtr -> next = NULL

delete tempPtr

tempPtr = frontPtr

frontPtr = NULL

backPtr = NULL;

tempPtr -> next = NULL;

delete tempPtr

simulation

→ modeling the behavior

event - driven simulation

26 5 25 0

22 4

arrival event & departure event

caculate statics
- total waiting time
- average waiting time

Make Decision
- should we add teller

Big (O) Notation

Time efficiency
space efficiency

three factors that will effect (without algorithm)

specific implementation

computer

data

execution time related to the number of operations

```
for (a=1; a<=n; a++)
   for (b=1; b<=a; b++)
      for (c=1; c<=k; c++)
```

loop take $(n^2+n)k$ time

if Algorithm A requires time proportional to $n^2 \rightarrow O(n^2)$

B $\quad n \rightarrow O(n)$

if run time sometimes be n or n^2 or n^3
although n^2 or n^3 may be, also be the runtime of the algorithm
but we will select the best answer as the time proportion
that is $O(n)$

$O(1)$ $\quad O(\log_2 n)$ $\quad O(n)$ $\quad O(n\log_2 n)$ $\quad O(n^2)$ $\quad O(n^3)$ $\quad O(2^n)$

$\longleftrightarrow$

good $\qquad\qquad\qquad\qquad\qquad\qquad$ bad

$O(5n^3+3n)$

ignore the lower order term

and ignore the higher order term's constant

$O(f(n)) + O(g(n))$

$= O(f(n)+g(n))$

Sequential search

Strategy -> Stop when the desired item is found

worst case

best case

Average case

Efficiency of sorting Algorithm

internal sort

external sort

stable sort

stabble	unstabble
bubble	quick
insertion	heap
merge	selection
radix	

2 5_a 5_b 6 8_a 8_b 19 27

in stabble it always be the same sequence

but in unstable

5_a & 5_b may switch

or

8_a & 8_b

bubble sort

find the small one ~~[illegible]~~ if the next one is bigger than swap

selection sort

swap the smallest one to the first

and swap the second smallest one to the second ... etc

Insertion Sort

put the next value to sorted list ~~[illegible]~~ and insert as it's value region

may swap the front letter to the very back end → will usually mess the sort

$$n + \sum_{pass=1}^{n-1} (n - pass + 1) + \sum_{pass=1}^{n-1} (n - pass)$$

$$= n + 2\left[n^{*}(n-1) - n^{*}(n-1)/2\right] + (n-1) = n^2 + n - 1 \rightarrow O(n^2) \leftarrow \text{loop}$$

$$4^{*} n(n-1)/2 = 2n^2 - 2n \rightarrow O(n^2) \leftarrow \text{comparison}$$

Data exchange $O(n)$

compare $O(n^2)$ → still $O(n^2)$

although no earlier termination even ~~[illegible]~~ it have been sorted

because it will only move the data $O(n)$, so when we met a list with single large data than it's appropriate

bestcase $O(n)$

worst case $O(n^2)$

when the data is not complete

Shell Sort → stronger version insertion sort

Insert but don't sort percisely

Merge sort

divide and conquer and compose together

if there is n number

seperate so n-1 space swap = 2n

→ worst $n-1+2n=3n-1$ → $O(n)$

levels : $\log_2 n$

average : $(n * \log_2 n)$

→ the second array will large as origin array

quick sort

also divide and conquer

Choose a pivot

Partitionc the array about the pivot

→ to ~~divid~~ split items ~~pivot~~ to two parts

→ left of the split

right of the split

do it again

Radix Sort

sort as the based of Radix

compare the number each by each from high to small

but also need to seperate to group

Binary Tree

Binary search Tree

position oriented ADT

insert to n^{th} position

Delete ~~from~~

ex.

list stack queue binary tree

Value oriented ADT

Insert according to its value

Delete only knowing

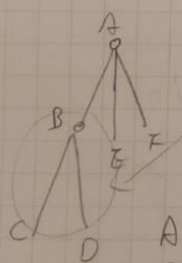
ex.

sorted list, binary search tree

Trees : are compose of nodes and edges

hiearchical → parint-child (two nodes

Ancestor - descendant (among nodes)



subtree of a tree

→ any node and its desendant

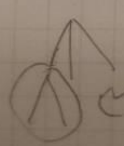
A single node r, The root

sets

Parent of node B = A

child = C, D

Root : the only node in the tree with no parent



Subtree of node B contain child of node B and it's decendant's

Leaf
- A node with no child

Siblings
- node with common parent

Ancestor from node to root

Descendant
from node to leaf

Binary Tree

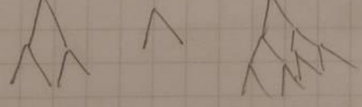
→ ~~can~~ b one can have one siblings or non

Height=3 H=5 H=n

the max of from root to leaf is Height

Full Binary tree

every node levels $<h$ has two childrem



complete Binary Trees

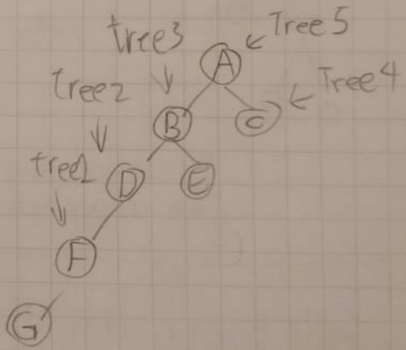
→ is full to level h-1
fill from left to right

Balanced Binary Trees

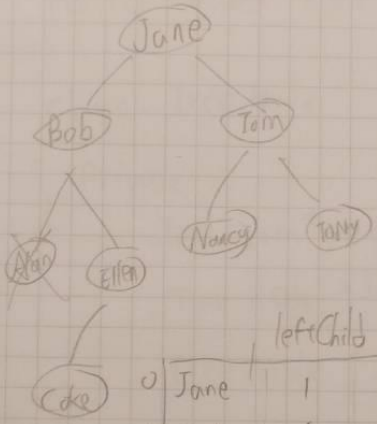
$\Delta height_1 \leq 1$

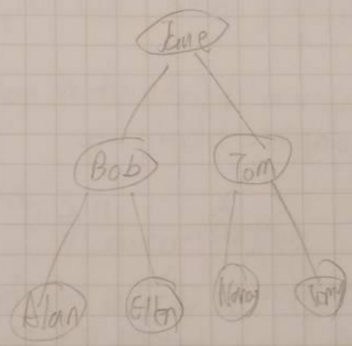
$\Delta height_2 \leq 1$

The ADT Binary tree



tree1. setRootData (F)
tree1. attachLeft (G)
tree2. setRootData(D)
tree2.attach Left Subtree(tree1)
tree3.setRoot Data(B)
tree3.attach Left Subtree (tree2)
tree3. attach Right(E)
tree4 .setRootData(C)
tree5. createBinaryTree('A', tree3, tree4)





		leftChild	Right
0	Jane	1	2
1	Bob	-1	4
2	Tom	5	6
3	Coke	-1	-1
4	Ellen	3	-1
5	Nancy	-1	-1
6	Tony	-1	-1
7	?	-1	8
	?	-1	9

root

0

free

76

Free list

Jane	1	2
Bob	3	4
Tom	5	6
Coke	-1	-1
Ellen	-1	-1
Nancy	-1	-1
TONY	-1	-1
?	-1	8
?	-1	9

root

0

free

7

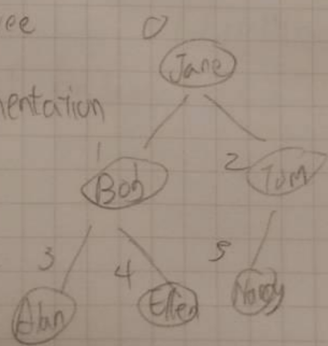
a complete binary tree

with array-based implementation

leftChild = 2* parent+1

rightChild = 2*parent+2

→ parent = (child-1/2)



0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy

Height and No. of Nodes

minimum height of N Nodes

- complete binary tree

$n \le 2^{h}-1$

$\Rightarrow h = \lceil \log_2(n+1) \rceil$

maximum height of Nodes

$(2^{h-1}-1)+1 \le n$

$h = \log_2(n)+1$

---

properties

N_2: 3 nodes with 2 children

N_0: 4 Leaves

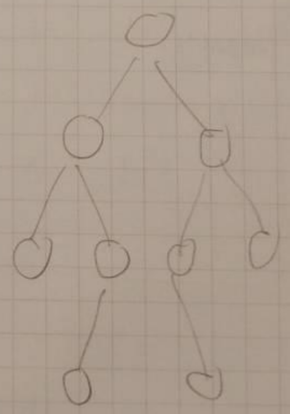
- $N_0 = N_2 + 1$

B: 8 Branches

$B = |E| = 2*N_2 + 1*N_1$

$N_0 + N_2 + N_1 = |V| = |E| + 1$

$= 2*N_2 + 1*N_1 + 1$



Pointer Based ADT Binary Tree

class Tree Node

{ Private

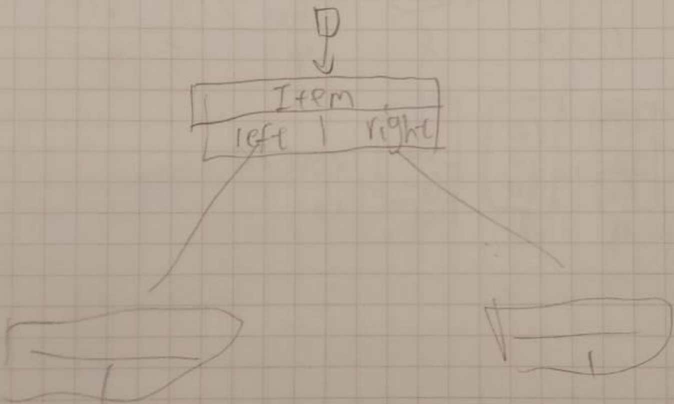
TreeItemType item // data portion

Tree Node *leftChildPtr // pointer to left child

Tree Node *rightChildPtr // pointer to right child

}

TreNode *root // Pointer to the root



recursive traversal algorithm

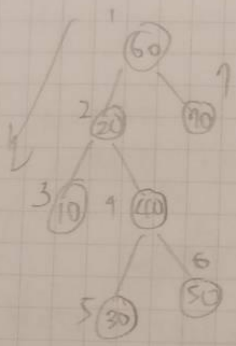
traverse (in binTree : BinaryTree)

if (bin Tree is not empty

preorder → traverse(left subtree of binTree's root)

inorder → (right)

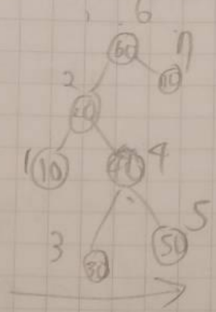
postorder →



preorder

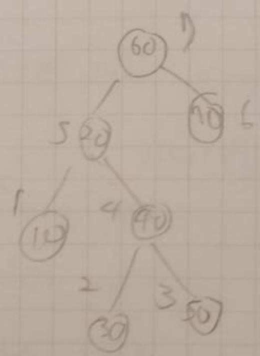
↑

遇見則輸出

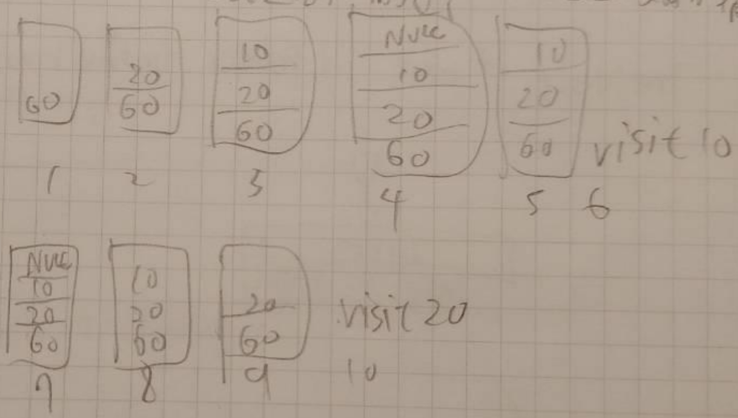


inorder

回去遇到才輸出



回去且沒有才輸出

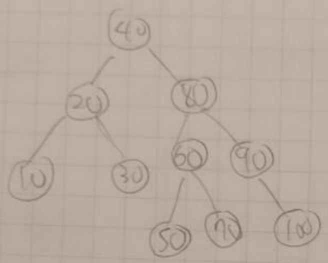


Binary search tree

left < righ >

40 20 10 80 90 100 30 60 50 70

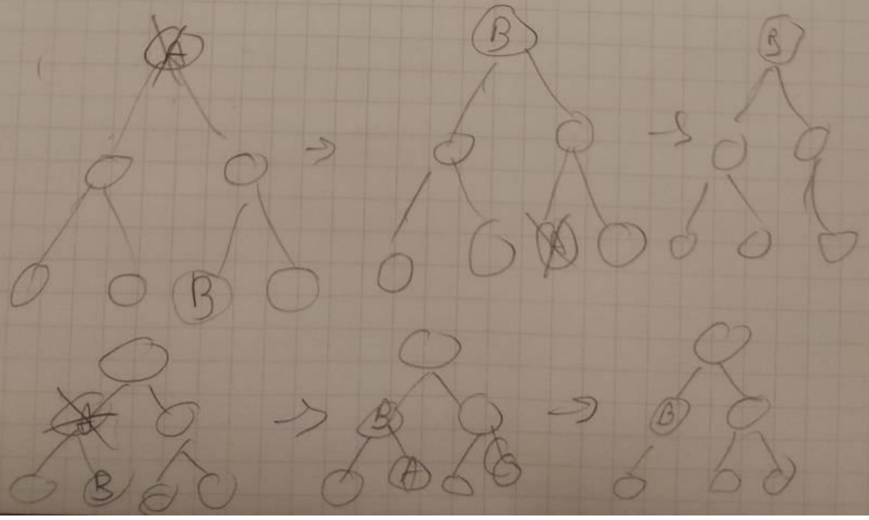
↓

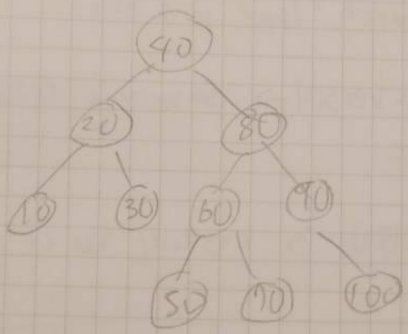


h=4

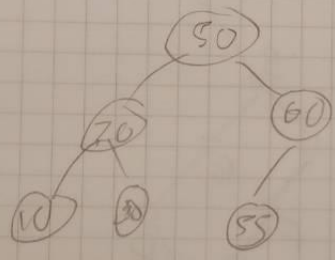
Delete ⇒ when we delete a node we still need to connet the last, when the node will connect more than 2 node than we have special moves

ex.





+55 -90 -70 -100 -40 -80



Efficiency of Binary Search Tree

Operation	Average	worst ← a line
Retrieval	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$

→ Tree sort

Build a binary search tree by n Insertions

Average $O(n \cdot \log n)$

worst $O(n \cdot n)$

Saving a Binary Search Tree in a File

preorder traversal → save and restore to original tree

inorder traversal → restore to a balanced tree

Left child → the leftmost child

Right child : right next siblings