→ contiguous area of memory

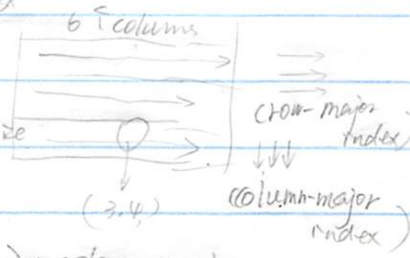
Array : Advantage ① constant-time access (read / write)

can figure out address of particular element of an array

↳ array-addr + elem_size × (i − first_index)

multi-dimensional Arrays : (row , colom)

ex: (3,4) array-addr + ((3−1)×6 + (4−1)) × element_size



↓

array_addr + elem_size × ((row_i − first_row) × colom_number + (coloum_i − first_colum))

Times for common operations :

	Add	Remove
Beginning	O(n)	O(n)
End	O(1)	O(1)
Middle	O(n)	O(n)

② Advantage : easier add/remove at end.

ex

5	8	3	12			

room for 7

add/remove here (end)

remove first

~~5~~	8	3	12			

8	3	12				

O(n)

insert first

5	8	3	12			

Summary : 1. contiguous area of memory consisting of equal-size elements indexed by contiguous integer

2. Constant-time access to any element (read / write)

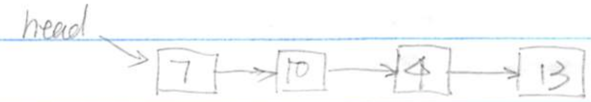
3. Constant time to add / remove at the end

4. Linear time to add / remove at an arbitrary location.

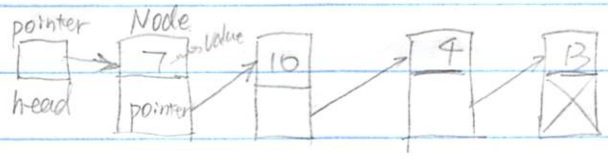
cb a d

Linked List

1. Singly - Linked List.



Actually



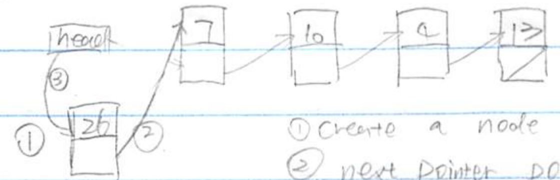
Node contains : key, next pointer

List API :

O(1)	PushFront (key)	: add to front
	Key TopFront ()	: return front item
O(1)	PopFrom ()	: remove front item.
O(1)/O(n) (with tail / no tail)	PushBack (Key) (Append)	: add to back
	Key TopBack ()	: return back item
O(n)	PopBack ()	: remove back item
	Boolean Find (key)	: is key in List?
	Erase (key)	: remove key from list
	Boolean isEmpty ()	: is the list empty?
	AddBefore (Node, key)	: adds key before the node.
	AddAfter (Node, Key)	: adds key after the node

Times for Some Operations

- PushFront : O(1).



① Create a node (key + next pointer)
② next pointer points to first node
③ update head pointer to the added new first node

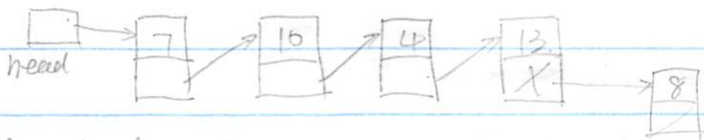
PushFront (key)
node = new node;
node. key = key;
node. next = old head node
head → node
if tail = nil : (before insertion, list empty, the head / tail nil)
tail = head;

- PopFront : O(1) ① update head pointer ② remove first node.

PopFront()
if head = nil;
ERROR : empty list tail ← nil
head ← head. next

(no tail pointer)

- PushBack : O(n)



Start from head, go down until the end, add a node here

else: old tail

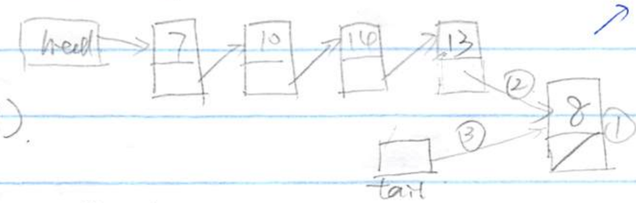
tail.next ← node.

tail ← node.

- PopBack : O(n). 从头开始

If has a tail pointer

PushBack → O(1).



PushBack(key)

node ← new node

node.key ← key

node.next = nil (empty list)

if tail = nil:

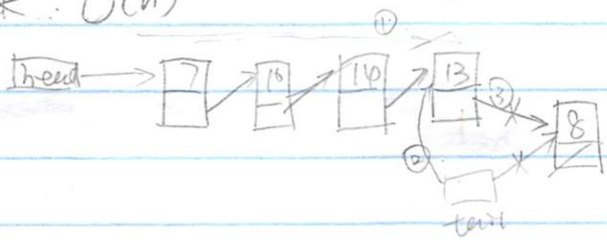
head ← tail ← node.

① Create a new node (key)

② update the current end node's next pointer to the ①

③ update the tail pointer

PopBack : O(n)

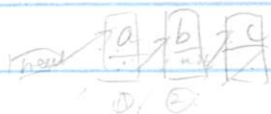


Do not have a pointer from 8 to 13. (end to end before)

So ① walk down from start to (13) Node (the node contains a next pointer to current-end node)

② update the tail pointer to the node found in ①

③. update ①'s next pointer



PopBack:

if head = null : ERROR : empty list

if head = tail : (one element node)

head ← tail ← nil

else:

p ← head

while p.next.next ≠ nil:

p ← p.next

p.next ← nil ; tail ← p

AddAfter (node, key)

node2 <- new node

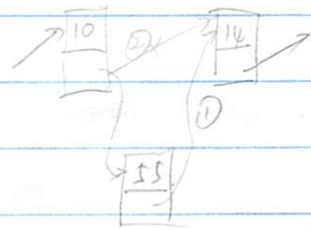
node2.key <— key

node2.next = node.next

node.next = node2.

if tail = node (~~在tail~~ 表示node在tail)

tail ← node2

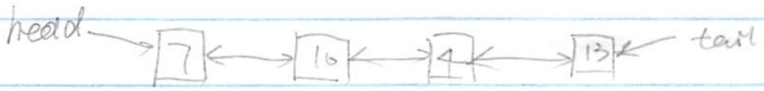


AddBefore (node, key) 和PopBack类似，无pointer to privons.

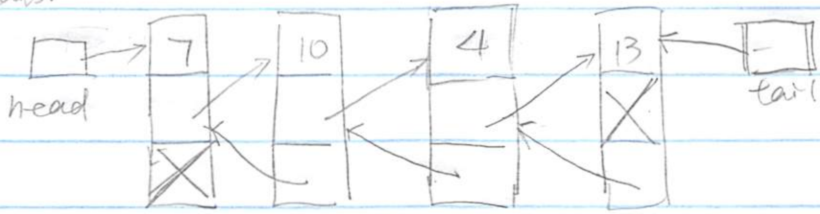
Summary: Double-Linked List.

Single-Linked List	no tail	with tail
PushFront (key)	O(1)	
TopFront ( )	O(1)	
PopFront ( )	O(1)	
PushBack (key)	O(n)	O(1)
TopBack ( )	O(n)	O(1)
PopBack ( )	O(n) → O(1)	
Find (key)	O(n)	
Erase (key)	O(n)	
Empty ( )	O(1)	
AddBefore(Node, key)	O(n) → O(1)	
AddAfter (Node, key)	O(1).	

Doubly - Linked List

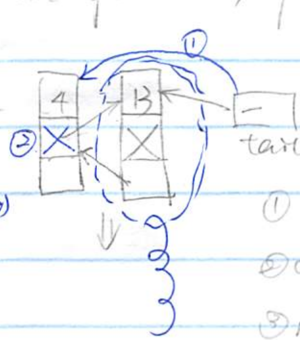


in details.



Node contains: key, next pointer, prev pointer

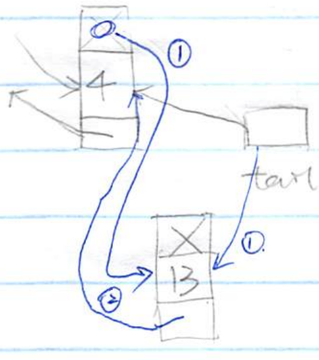
- Pop Back: O(1)



  if head = nil : Error
  if head = tail : (one element)
    head ← tail ← nil
  else : tail ← tail.prev
    tail.next ← nil

① update tail pointer
② update it's next pointer to nil
~~③ remove the original back~~

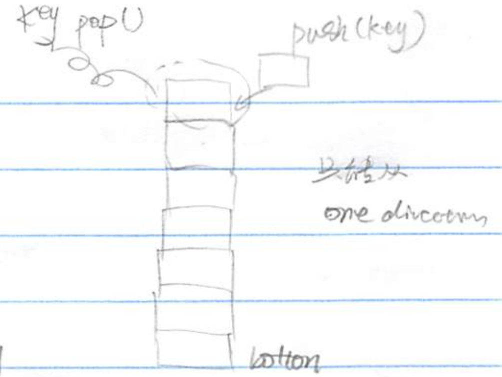
- Push Back (key)



node ← new node
node.key ← key ;
node.next ← nil ;
if tail = nil : (empty list)
  head ← tail ← node
  node.prev ← nil
else :
  tail.next ← node
  node.prev ← tail
  tail ← node

Summary:

- Constant time to insert at or remove from the front
- With tail and doubly-linked, constant time to insert at or remove from the back.
- O(n) time to find arbitrary element.
- List element needs not be contiguous.
- With doubly-linked list, constant time to insert between nodes or remove a node.



Stack:

abstract data type with follwing operations:

push(key): adds key to collection

Key Top(): return most recently-added key

Key Pop(): removes and returns most recently-added key

Boolean Empty().

Balanced Brackets: Input: A string str consisting with "(", ")" "[" "]"

Output: returns whether or not balanced (all matching)

IsBalanced(str). [([(( ))])]

Stack stack

for char in str:

{ if char in ['(', '[']:

stack.push(char)

else

if stack.Empty(): returns false ⇒ all close],) not balanced

top ← stack.Pop()

if ((top = '[' and char != ']') or

(top = '(' and char != ')')):

return false;

} return stack.Empty (if empty, return T)

- Stack Implemention with Array → { space wasting / liminartion (maxium size)

allocate an array with the max size of the stack ex: 6.

a	b	d	e	f	g

push(a) O(1). Top() → b.

push(b)

push(c) Pop() → c (remove)

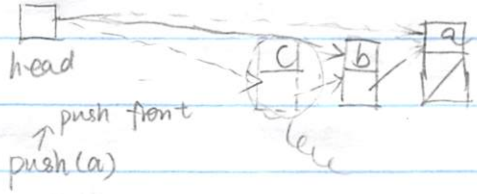
push(d) keep pop()

push(e) Empty() → T

push(f) push(x) → Error. Empty()? No

push(g)

Stack Implementation with Linked List.



push front

push(a)

push(b)

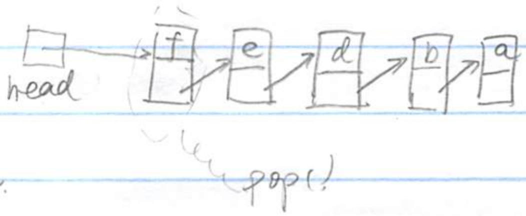
Top() → b (top is head element) top front)

push(c)

pop() → c

push(d)

⋮ keep adding no limitation of size.



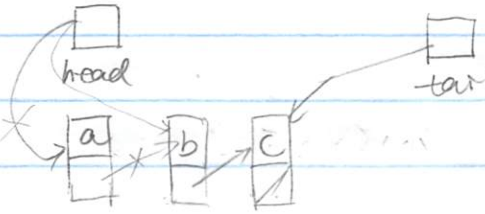
Summary:
- Stack can be implemented with either array or a linked list
- each stack operation is O(1): Push Top() pop() Empty
- Stack are ocassionally known as LIFO (last in first out)

Queue: abstract data type

first come first serve.
(FCFS)
(FIFO)

- Enqueue(key): adds key to collection
- Key Dequeue(): removes and returns the least recently added key
- Boolean Empty()?

Queue Implementation with Linked List



Enqueue(a) (add in end)

Enqueue (b)

Empty()? No → head is not null

Dequeue() → a (from head) poping from front

Enqueue: List. PushBack(a) < O(1) / O(n) no tail

Dequeue: List. TopFront() → O(1)
+ List. PopFront() → O(1)

Empty: List. isEmpty() → O(1)

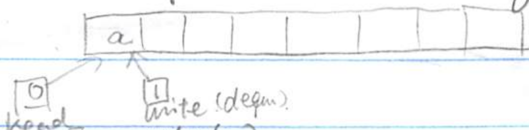
if: Enqueue: List. Pushfront(a) → O(1)

Dequeue. List TopBack() < O(n) no tail / O(1) with tail

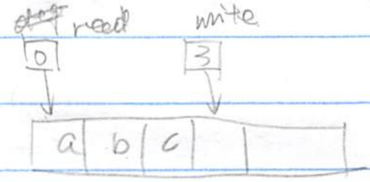
+ List. PopBack() → O(n) - single. / O(1) - double

Queue Implementation with Array

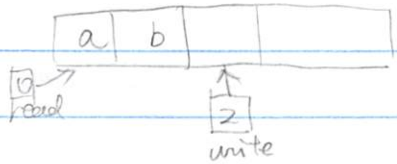
write index: where next enqueue happens
read index: where next dequeue happens



enqueue ① Enqueue(a)



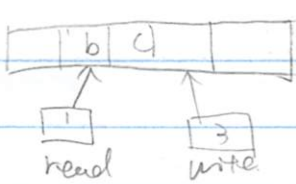
② Enqueue(b)



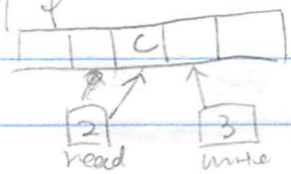
③ Enqueue (c)

Empty() ? → write is 0 and read is write : ~~empty~~
~~write is not 0. not empty~~

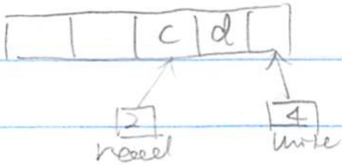
④ Dequeue(a) read(0)



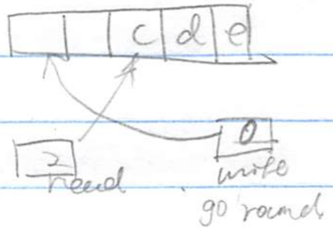
⑤ Dequeue (b)



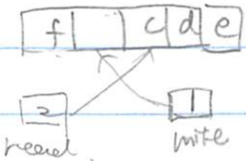
⑥ Enqueue (d)



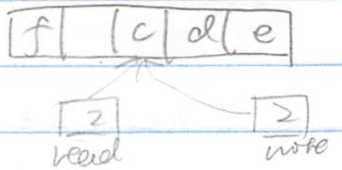
⑦ Enqueue (e)



⑧ Enqueue (f)

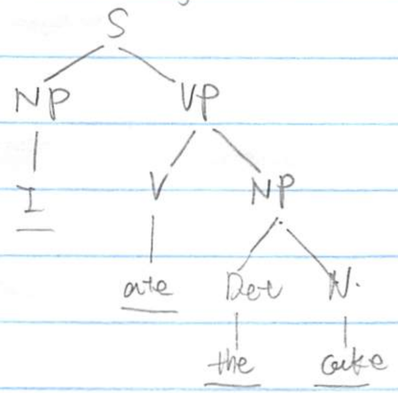


⑨ Enqueue (g) → Error

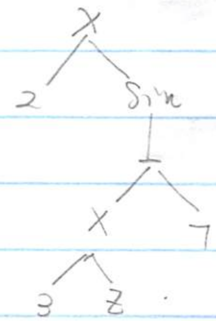


Summary: • Queue can be implemented with either
every element needs pointer ← linked-list (with tail pointer) or an array → with boundary limitation
• Each queue operation is O(1): Enqueue Dequeue Empty

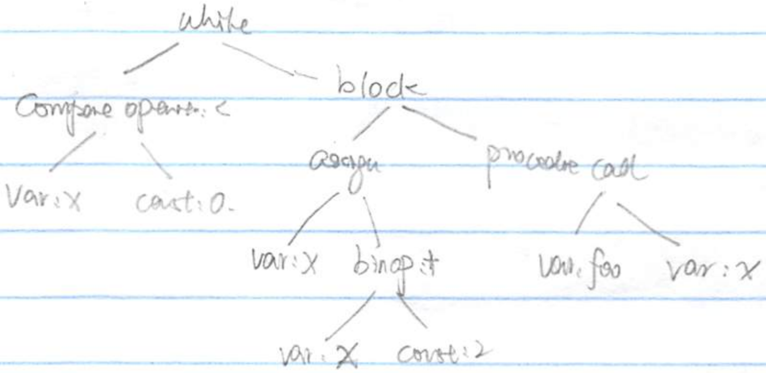
Syntax Tree for a Sentence / Arithmetic / Code ....



$2\sin(3z-7)$



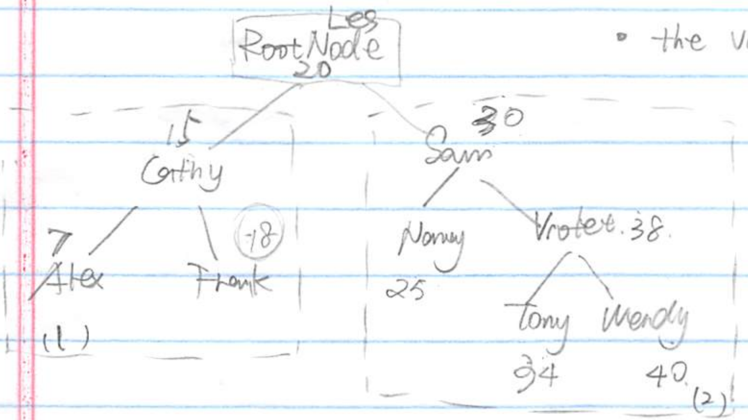

```
while x<0:
    x=x+2;
    foo(x);
```

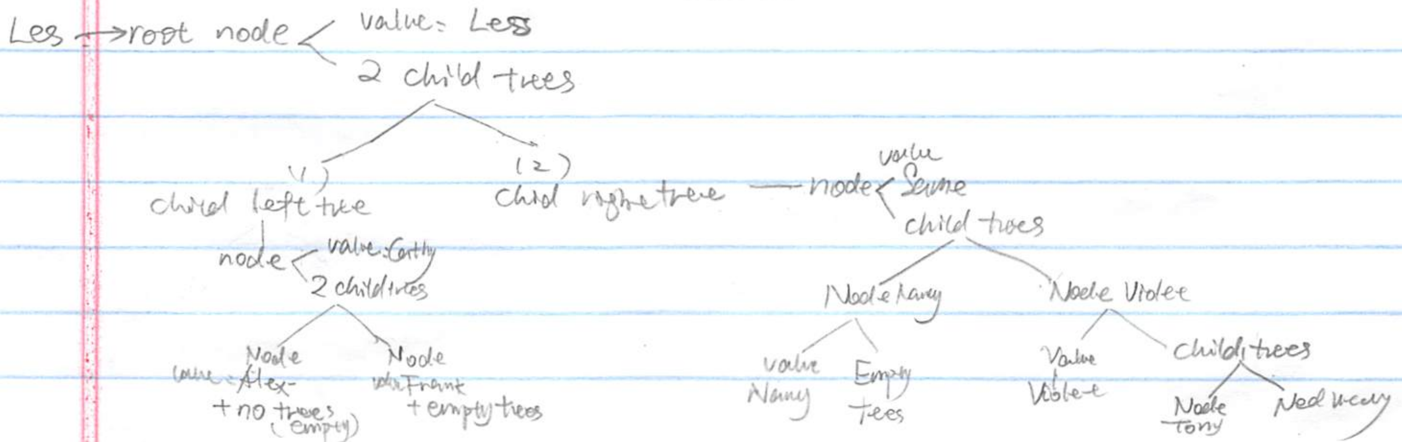


Binary Search Tree. (左边的 < 这 Node)
(右边的 > 这 Node)

• each node has 2 children

• the value of root node ≥ left child
≤ right child.





Definition

Tree (recursive):
- empty
- a node with:
  - a key
  - a list of child trees

root: top node in the tree.

Terminology: tree with arrows (通常不用 arrow, 上面是下面的 parent)

level ——— Fred

level 2 ——— Kate Sally Jim

level 3 ——— Sam Hugh

Parent

Child: Hugh is not a child of Fred

Ancestor

Decendent: Hugh is a decendent of Fred

Sibling: Sharing same parents

leaf: node with no children: Sam Hugh Sally Jim

Level: (1+ num edges between root and node)

Interior Node: (non-leaf): ~~all~~ nodes are not leafs (do have children)
Fred Kate

Height: maximum depth of subtree node and farthest leaf

leaf height is 1

Fred Height: 3.

Kate ---: 2.

Recursively:

```
Height(tree)
    if tree = nil:
        return 0;
    return 1 + Max(Height(tree.left),
                   Height(tree.right));
```

ex: leaf:
max < no child left Height → 0 / no --- right Height →
↓ 0
1+0=1 → Height of leaf

Forest: Collection of trees.

Node contains:
- key
- Children (lists of children)
- (optional) parent

Binary tree: nodes Contains:
- key
- left
- right
- (optional) parent

Size of tree (number of nodes)

```
if tree = nil
    return 0
return 1 + Size(tree.left)
         + Size(tree.right);
```

Walking a Tree (traverse)

Often we want to visit the nodes of a tree in a particular order

For ex: print the nodes of the tree: (if binary search tree, get all elements in sorted order)

Depth-first: Completely traverse one sub-tree before exploring a sibling sub-tree (先children 再sibling)

Breadth-first: We traverse all nodes at one level before processing to next level. (先siblings 再children)

Depth-first: (Stack)

InOrderTraversal(tree) → binary tree

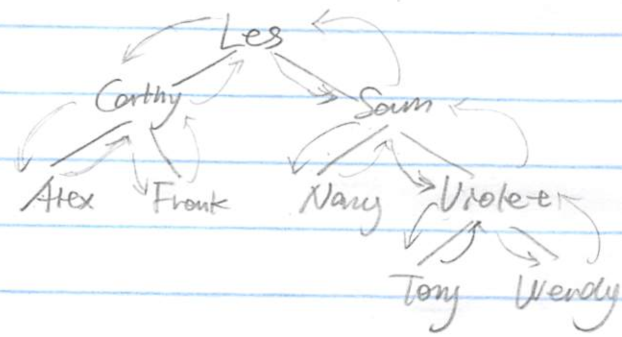
if tree = nil:
    return

left child → node → right child

InOrderTraversal(tree.left)

Print(tree.~~right~~ key)

InOrderTraversal(tree.right).



Output: Alex Cathy Frank Les. Nancy Sam Tony Violet Wendy

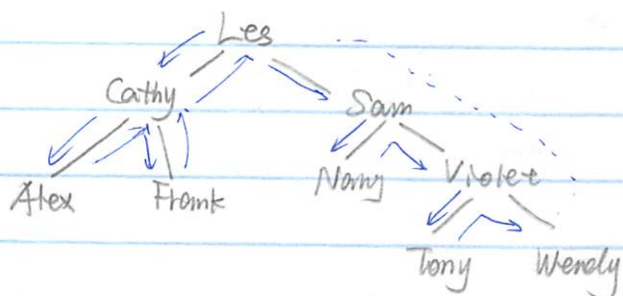
PreOrderTraversal(tree). → general tree

if tree = nil
    return

Print(tree.key)

PreOrderTraversal(tree.left)

PreOrderTraversal(tree.right)



Output: Les, Cathy, Alex. Frank Sam Nancy Violet. Tony Wendy

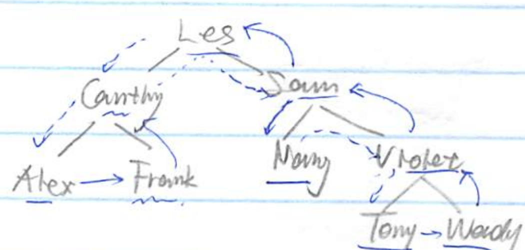
PostOrderTraversal(tree)

if tree = nil
    return

PostOrderTraversal(tree.left)

PostOrderTraversal(tree.right)

Print(tree.key)



Output: Alex, Frank, Canthy, Nancy. Tony. Wendy. Violet. Sam. Les.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]