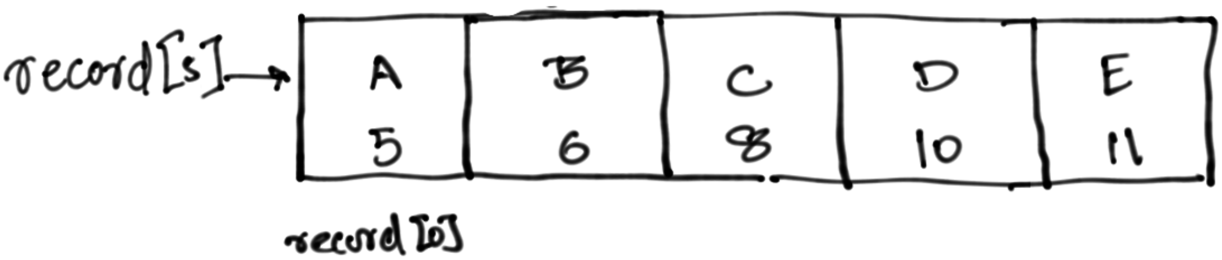
Data - structure Question

(1) Assume that you are the owner of a company that has 5 employees.

(2) Each employee is represented by {name, salary}

Which data-structure (that you know) will you use to represent these 5 employees?

Arrays



Arrays

record[5] →

A	B	C	D	E
5	6	8	10	11

record[i]

Suppose a new employee {F, 10} joins the company. How will you update your data-structure?

Arrays

record[5] →

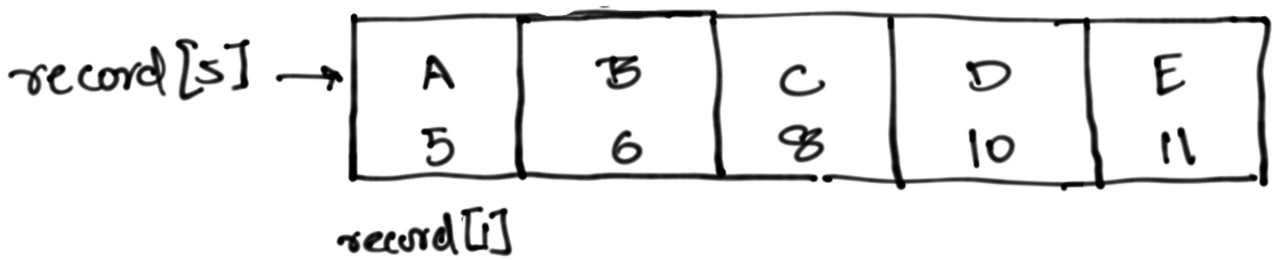
A	B	C	D	E
5	6	8	10	11

record[i]

Suppose a new employee {F, 10} joins the company. How will you update your data-structure?

A. Add {F, 10} @ record[6]

Arrays



Suppose a new employee {F, 10} joins the company. How will you update your data-structure?

A. Add {F, 10} @ record[6]

But record[6] does not exists. The array record is of size 5 only.

Now what will you do?

(1) Make a new array newrecord of size 6

(2) Copy the five records from record to newrecord.

(3) Add {F, 10} @ newrecord[6]

(1) Make a new array newrecord of size 6

(2) Copy the five records from record to newrecord.

(3) Add {F, 10} @ newrecord[6]

Q: What is the running time of this procedure?

(1) Make a new array newrecord of size 6

(2) Copy the five records from record to newrecord.

(3) Add {F, 10} @ newrecord[6]

Q: What is the running time of this procedure?

A: O(n) if there are n records in record array.

When an employee leaves

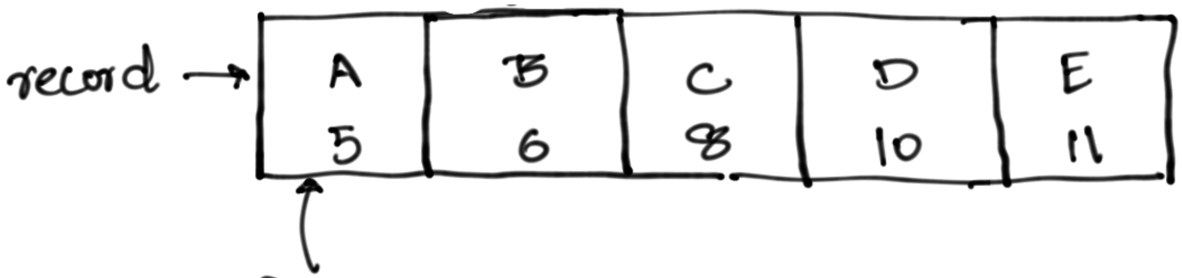
Suppose employee B leaves, so we have to remove the record of employee B.

Q: How would you do that?

When an employee leaves

Suppose employee B leaves, so we have to remove the record of employee B.

Q: How would you do that?

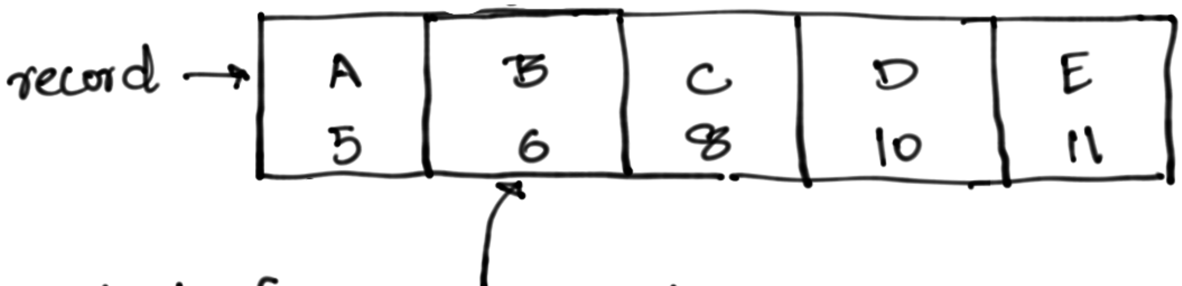


check if this is employee B.

When an employee leaves

Suppose employee B leaves, so we have to remove the record of employee B.

Q: How would you do that?

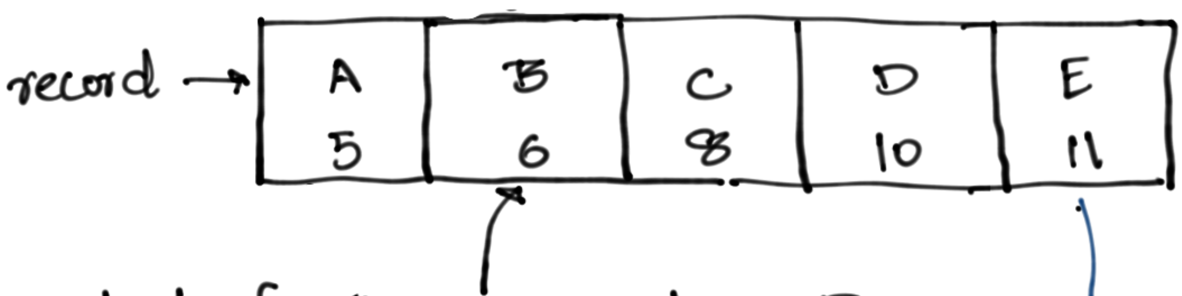


check if this is employee B.

When an employee leaves

Suppose employee B leaves, so we have to remove the record of employee B.

Q: How would you do that?



check if this is employee B.

When an employee leaves

Suppose employee B leaves, so we have to remove the record of employee B.

Q: How would you do that?

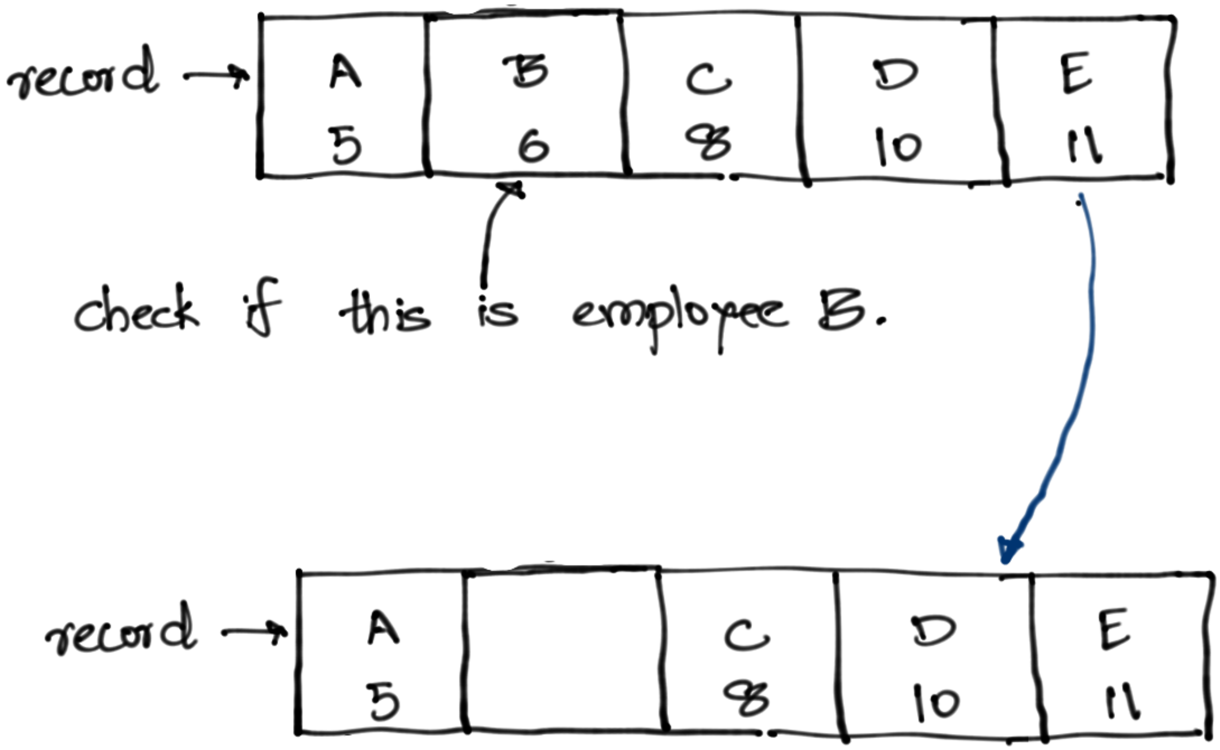
check if this is employee B.

Q: What is the running time of this method?

When an employee leaves

Suppose employee B leaves, so we have to remove the record of employee B.

Q: How would you do that?



Q: What is the running time of this method?

A: O(n) if there are n records.

Q: Is there any other problem with the deletion procedure?

Q: Is there any other problem with the deletion procedure?

A: (1) Wastage of Space: the number of employees may be much lesser than the record array.

Q: Is there any other problem with the deletion procedure?

A: (1) Wastage of Space: the number of employees may be much lesser than the record array.

(2) The time taken to search an employee will not be proportional to the # employees.

A 5				E 11

Q: Is there any other problem with the deletion procedure?

A: (1) Wastage of Space: the number of employees may be much lesser than the record array.

(2) The time taken to search an employee will not be proportional to the # employees.

A				E
5				11

Holy Grail for data-structure

The space taken by your data-structure should be proportional to the number of current employees in the company.

Performance of Arrays.

Insert	$O(n)$
Deletion	$O(n)$
Search	

Performance of Arrays.

Insert	O(n)
Deletion	O(n)
Search	O(n)

Performance of Arrays.

Insert	$O(n)$
Deletion	$O(n)$
Search	$O(n)$

So, array nearly always give worst case performance.

Q: Where are arrays good ?

Performance of Arrays.

Insert	$O(n)$
Deletion	$O(n)$
Search	$O(n)$

So, array nearly always give worst case performance.

Q: Where are arrays good?

A: Give me the 5th element of record array.

return record[5]

Running Time = ??

Performance of Arrays.

Insert	$O(n)$
Deletion	$O(n)$
Search	$O(n)$

So, array nearly always give worst case performance.

Q: Where are arrays good?

A: Give me the 5th element of record array.

return record[5]

Running Time = $O(1)$.

Performance of Arrays.

Insert	$O(n)$
Deletion	$O(n)$
Search	$O(n)$
Get the K^{th} element	$O(1)$.

Holy Grail for data-structure

The space taken by your data-structure should be proportional to the number of current employees in the company.

Holy Grail for data-structure

The space taken by your data-structure should be proportional to the number of current employees in the company.

There is a simple data-structure which solves this problem.

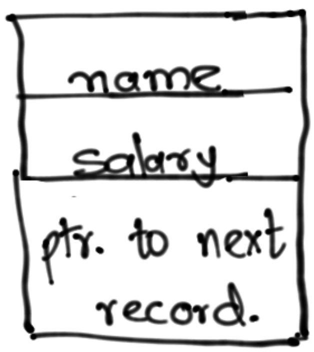
Mimics arrays.

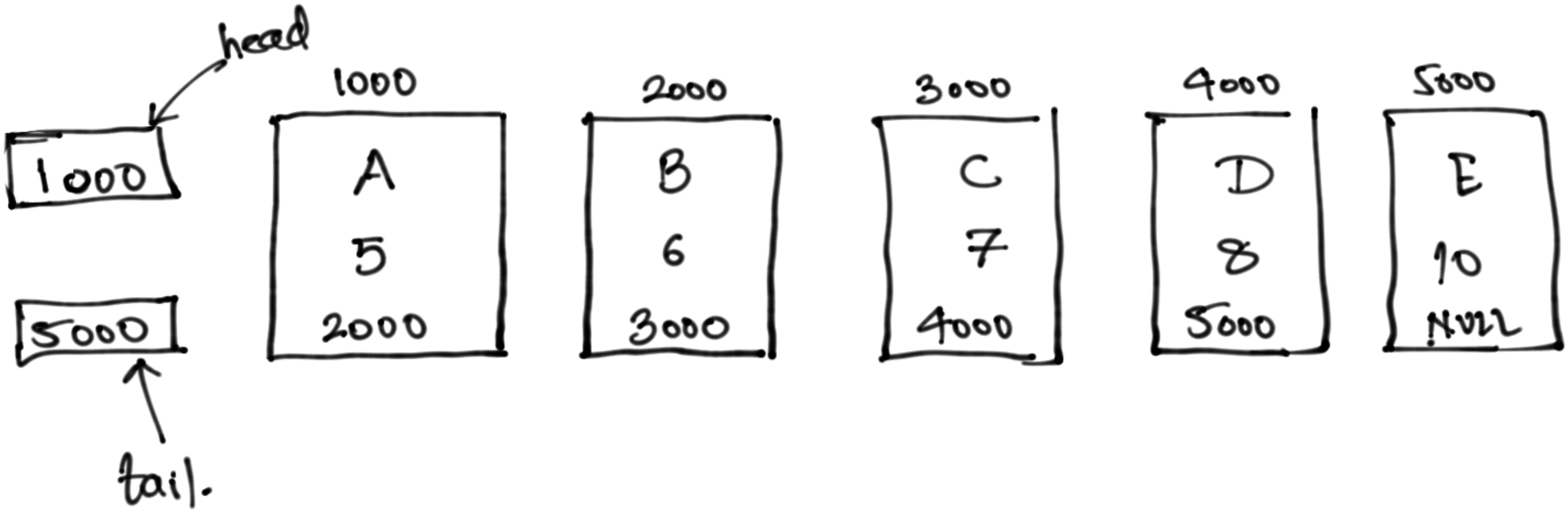
Holy Grail for data-structure

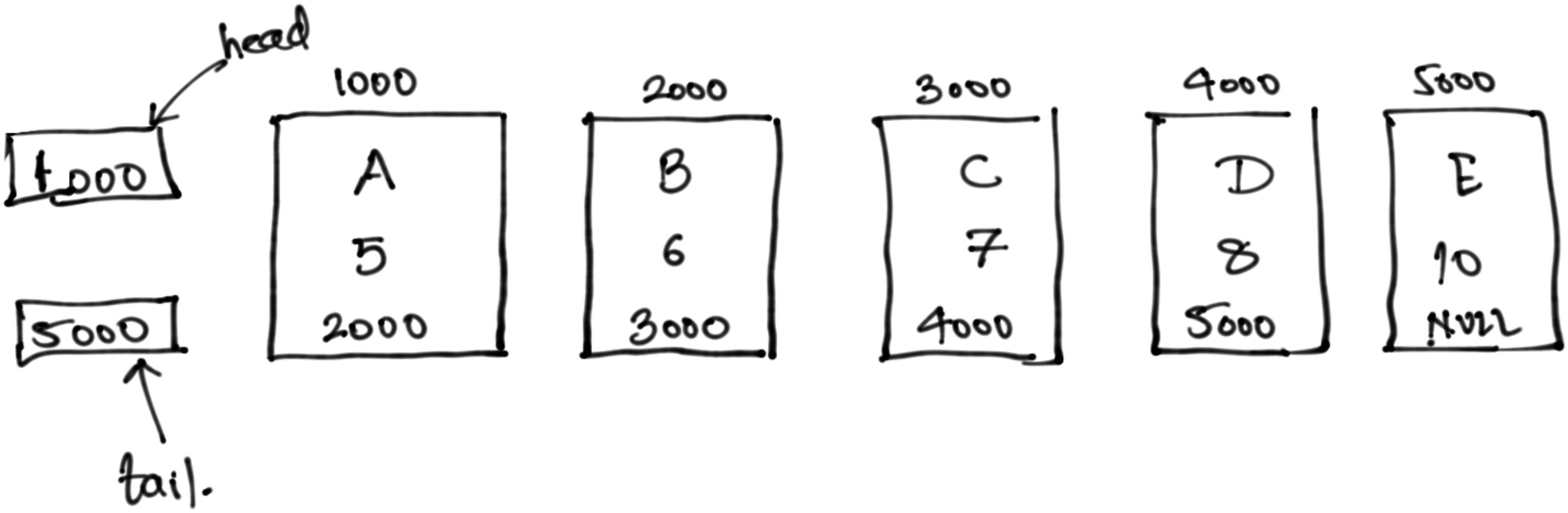
The space taken by your data-structure should be proportional to the number of current employees in the company.

There is a simple data-stoucture which solves this problem.

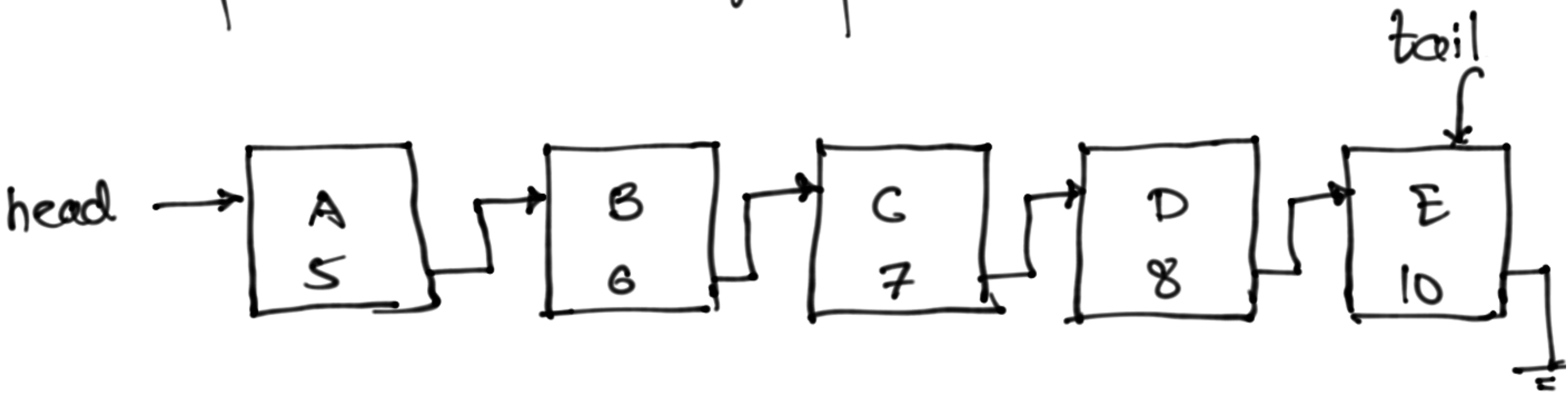
Mimics arrays.




head
1000
tail.
5000
1000
A
5
2000
2000
B
6
3000
3000
C
7
4000
4000
D
8
5000
5000
E
10
NULL



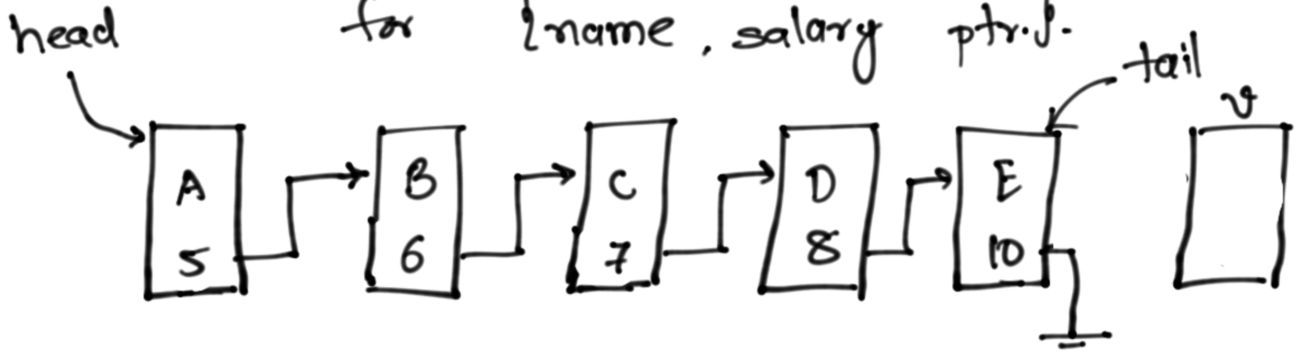
However, we will have a simpler pictorial view of pointers.



Insert $\{F, 10\}$.

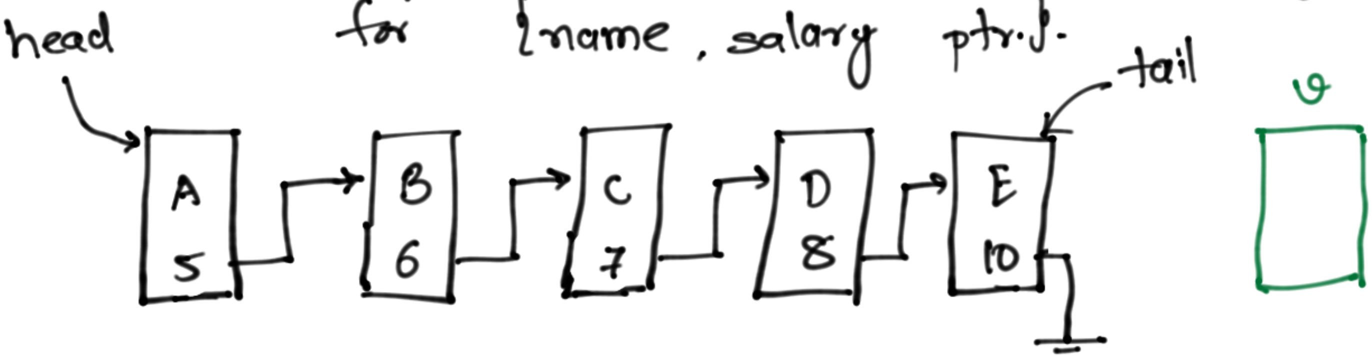
Insert {F, 10}.

① v ← Request the OS to allocate enough space for {name, salary ptr.}.

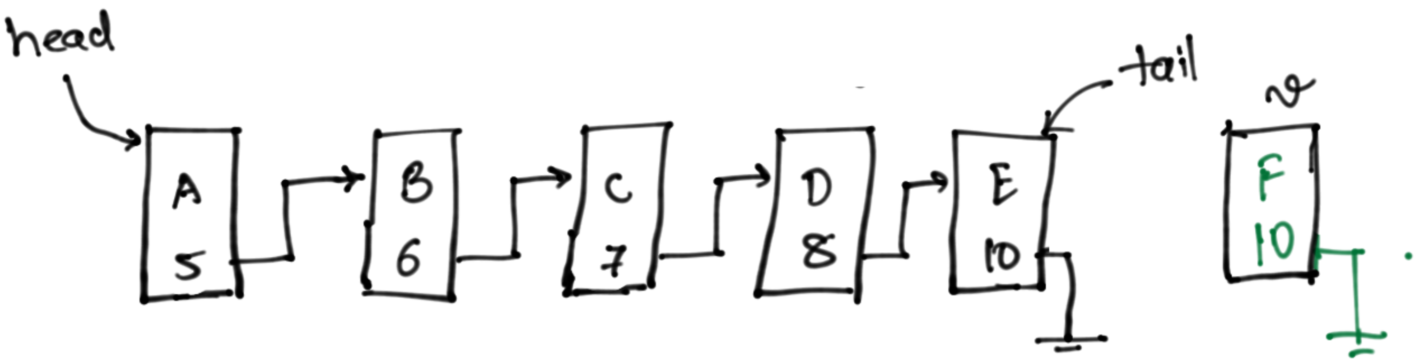


Insert {F, 10}.

① $v \leftarrow$ Request the OS to allocate enough space for {name, salary ptr.}.



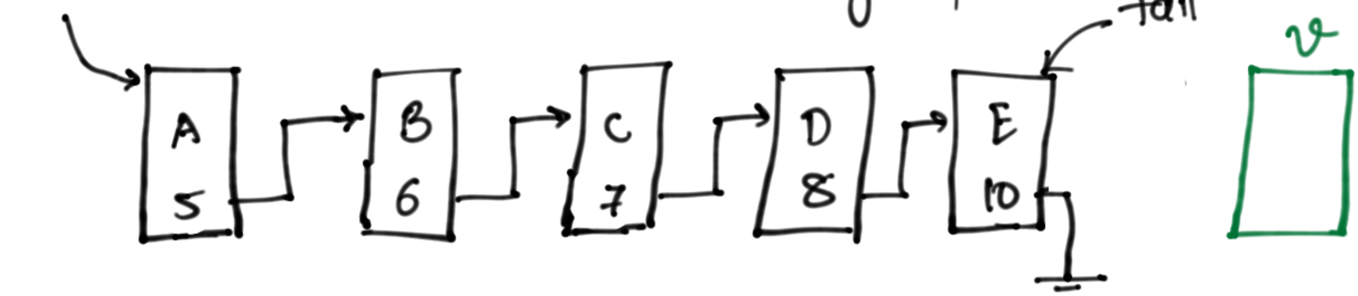
② v. name $\leftarrow$ F ;
v. salary $\leftarrow$ 10;
v. ptr $\leftarrow$ NULL



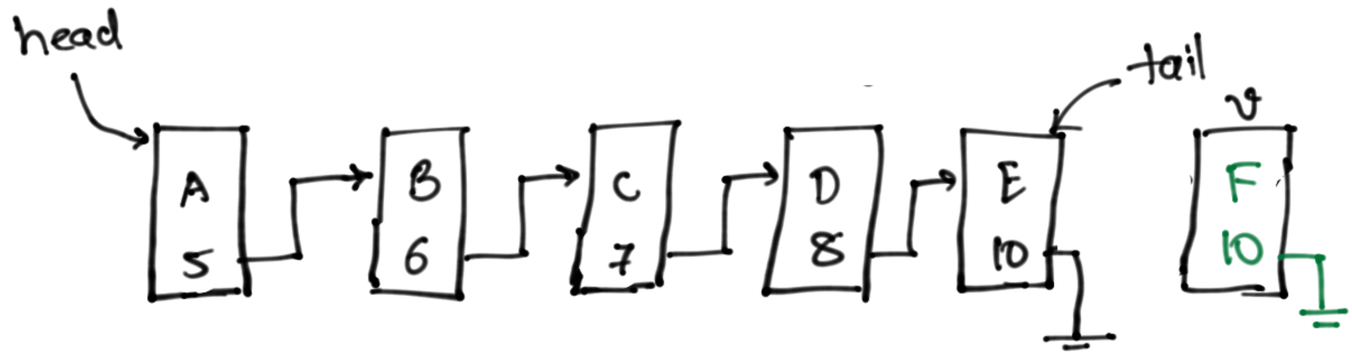
Insert {F, 10}.

① v ← Request the OS to allocate enough space for {name, salary ptr.}.

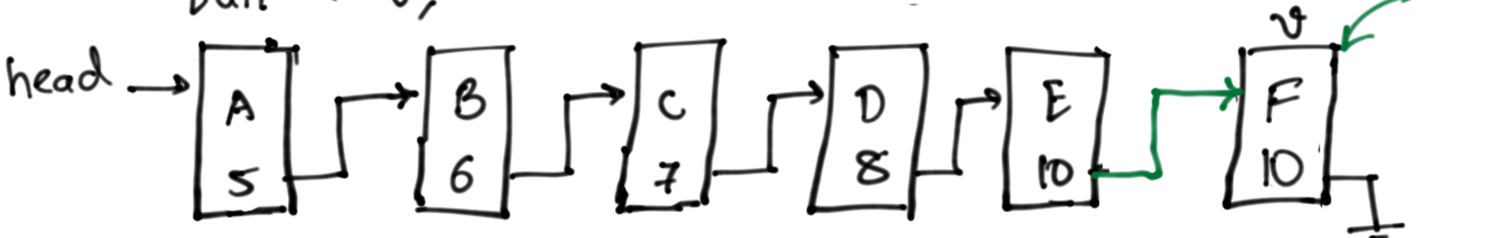
head



② v.name ← F ;
v.salary ← 10;
v.ptr ← NULL



③ tail.ptr ← v ;
tail ← v;



```
Insert (name, salary)
{
    r ← allocate me a new memory location;
    r.name ← name;
    r.salary ← salary;
    r.ptr ← NULL;

    if (head is NULL).
    {
```

```
Insert (name, salary)
{
    v ← allocate me a new memory location;
    v.name ← name;
    v.salary ← salary;
    v.ptr ← NULL;

    if (head is NULL).      //list is empty
    {
        head ← v;
        tail ← v;
    }
    else
    {
```

```
Insert (name, salary)
{
    v ← allocate me a new memory location;
    v.name ← name;
    v.salary ← salary;
    v.ptr ← NULL;

    if (head is NULL).     // list is empty
    {
        head ← v;
        tail ← v;
    }
    else
    {
        tail.ptr ← v;
        tail ← v;
    }
}
```

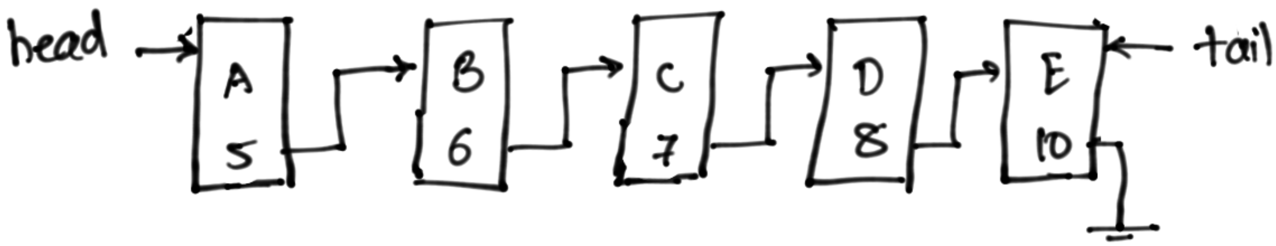
Q: What is the running time of this procedure?

```
Insert (name, salary)
{
    v ← allocate me a new memory location;
    v.name ← name;
    v.salary ← salary;
    v.ptr ← NULL;

    if (head is NULL).   // list is empty
    {
        head ← v;
        tail ← v;
    }
    else
    {
        tail.ptr ← v;
        tail ← v;
    }
}
```

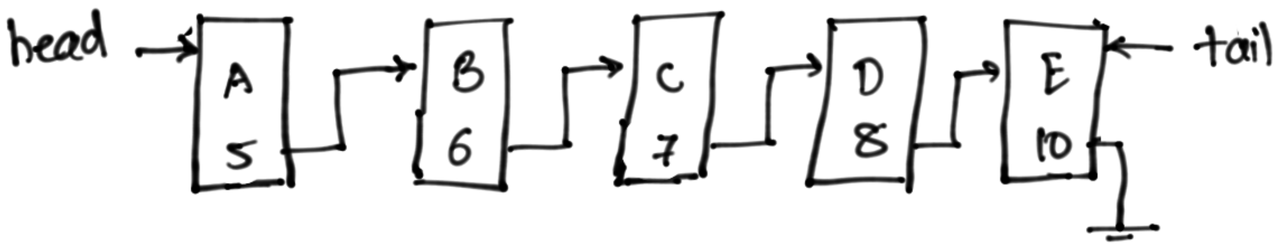
Q: What is the running time of this procedure?

A: $O(1)$.

delete Employee C

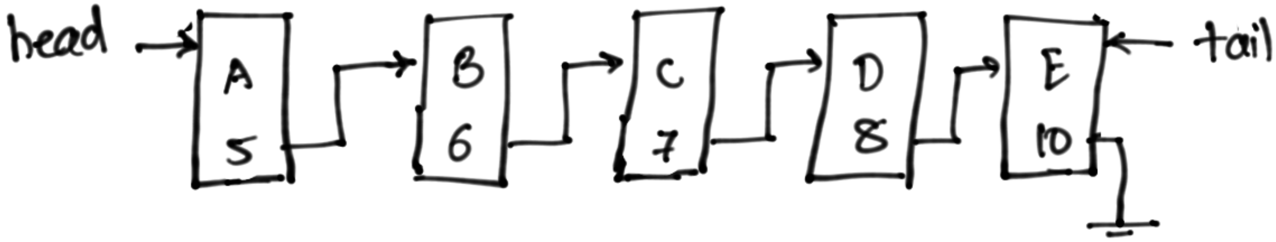


delete Employee C



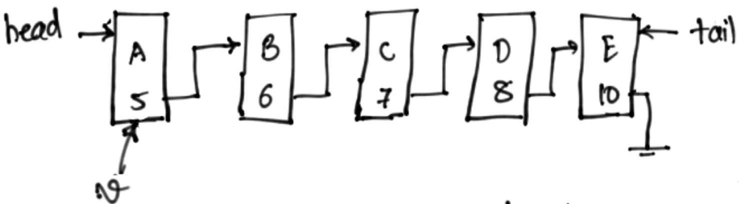
① v ← head (start with the head).

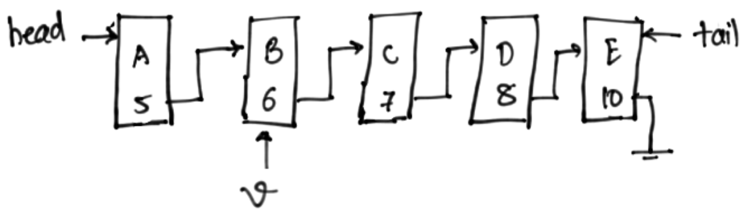
delete Employee C

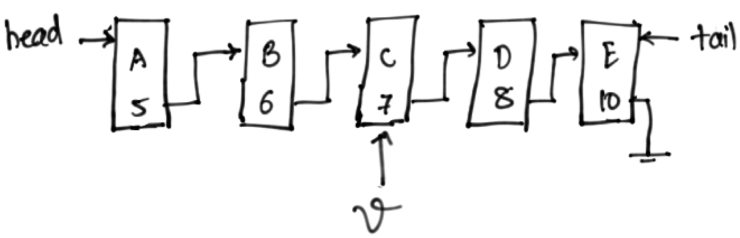


① $v \leftarrow$ head (start with the head).

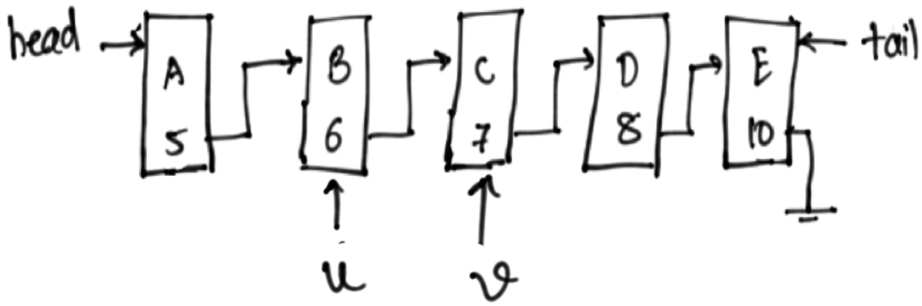
② Move through the list (using pointers) till you hit employee C



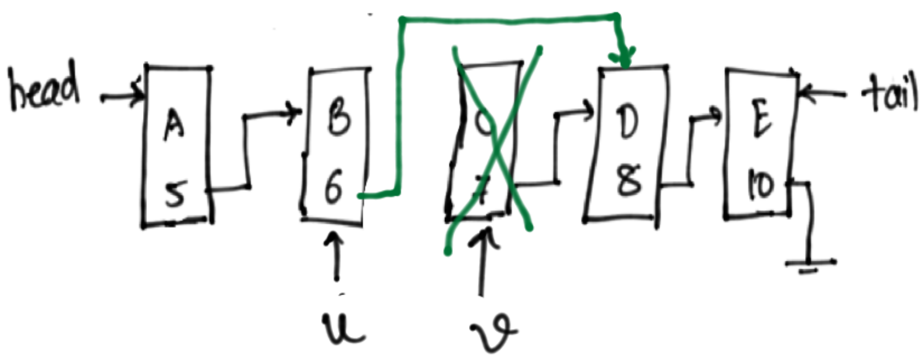




③ Maintain a ptr that follows v, say u.



④ u.ptr ← v.ptr
deallocate the memory associated with recor



```
Delete(name)
{
    if ( head is NULL)
        return;
```

```
Delete(name)
{
    if ( head is NULL)
        return;

    if (                        )
    {

    }
    else
    {   v ← head.ptr
        u ← head;
        while ( v is not NULL)
        {
            if (v.name = name)
            {   u.ptr ← v.ptr
                deallocate the memory allocated
                                to record v;
            }
            else
            {   u ← v;
            }   v ← v.ptr
        }
}
```

```
Delete(name)
{
    if ( head is NULL)
        return;

    if ( head.name = name)
    {   v ← head;
        head ← head.ptr;
        deallocate the memory allocated to
                                  record v
    }
    else
    {   v ← head.ptr
        u ← head;
        while ( v is not NULL)
        {
            if (v.name = name)
            {  u.ptr ← v.ptr
               deallocate the memory allocated
                           to record v;
            }
            else
            {     u ← v;
            }     v ← v.ptr
        }
    }
```

Q: What is the running time of Delete(·)

Q: What is the running time of Delete(·)

A $\Theta(n)$

Performance of Linked List

Insert	$O(1)$
Delete	$O(n)$
Search	?!

Performance of Linked List

Insert	$O(1)$
Delete	$O(n)$
Search	$O(n)$

Performance of Linked List

Insert	$O(1)$
Delete	$O(n)$
Search	$O(n)$

But the most important thing:

Holy Grail for data-structure

The space taken by your data-structure should be proportional to the number of current employees in the company.

Q: When is linked list worse than arrays?

Performance of Linked List

Insert	$O(1)$
Delete	$O(n)$
Search	$O(n)$

But the most important thing:

Holy Grail for data-structure

The space taken by your data-structure should be proportional to the number of current employees in the company.

Q: When is linked list worse than arrays?

Query: Get me k^{th} record.

Time taken by linked list = ?!

Performance of Linked List

Insert	$O(1)$
Delete	$O(n)$
Search	$O(n)$

But the most important thing:

Holy Grail for data-structure

The space taken by your data-structure should be proportional to the number of current employees in the company.

Q: When is linked list worse than arrays?

Query: Get me k^{th} record.

Time taken by linked list $= O(k)$

Performance of Arrays.

Insert	$O(n)$
Deletion	$O(n)$
Search	$O(n)$
Get the k^{th} element	$O(1)$.

Performance of Linked List

Insert	$O(1)$
Deletion	$O(n)$
Search	$O(n)$
Get the k^{th} element	$O(k)$

Balanced Parenthesis

```
int main()
{
    for(i=0; i<n; i++)
    {
        printf("Hello World");
    }
}
```

Balanced Parenthesis

```
int main()
{
    for(i=0; i<n; i++)
    {
        printf("Hello World");
    }
}
```

The compiler does a lot of things out of which one thing is check if the paranthesis are balanced.

Q: What is a balanced parenthesis?

Balanced Parenthesis

```
int main()
{
    for(i=0; i<n; i++)
    {
        printf("Hello World");
    }
}
```

The compiler does a lot of things out of which one thing is check if the paranthesis are balanced.

Q: What is a balanced parenthesis?

(1) Each opening symbol has a corresponding closed symbol.

(2) Parenthesis are properly nested.

Examples

(1) $\{\}()$

(2) $((\{)\})$

(3) $((\}\{))$

Examples

(1) {}()

(2) (({)})

(3) ((}{))

Q: Given a string of '{', '}', '(', ')', find if it is balanced.

What will be your algorithm?

(	)	{	(	)	{	}	}

Examples

(1) {}()

(2) (({)})

(3) ((}{))

Q: Given a string of '{', '}', '(', ')', find if it is balanced.

What will be your algorithm?

(	)	{	(	)	{	}	}

find the first closing element

Examples

(1) {}()

(2) (({)})

(3) ((}{))

Q: Given a string of '{', '}', '(', ')', find if it is balanced.

What will be your algorithm?

(	)	{	(	)	{	}	}

find the first closing element

Q: Where will the corresponding opening element lie?

Examples

(1) { } ( )

(2) (({) })

(3) ((} {))

Q: Given a string of '{', '}', '(', ')', find if it is balanced.

What will be your algorithm?

(	)	{	(	)	{	}	}

find the first closing element

immediate left of closing element

Examples

(1) { } ( )

(2) (({) })

(3) ((} {))

Q: Given a string of '{', '}', '(', ')', find if it is balanced.

What will be your algorithm?

~~(~~	~~)~~	{	(	)	{	}	}

find the first closing element

immediate left of closing element

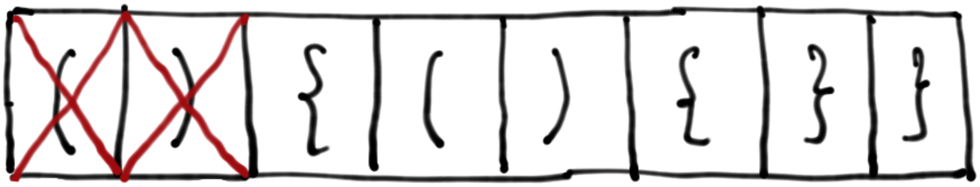
Delete these two elements

~~(~~	~~)~~	{	(	)	{	}	}

Repeat till all the elements are deleted

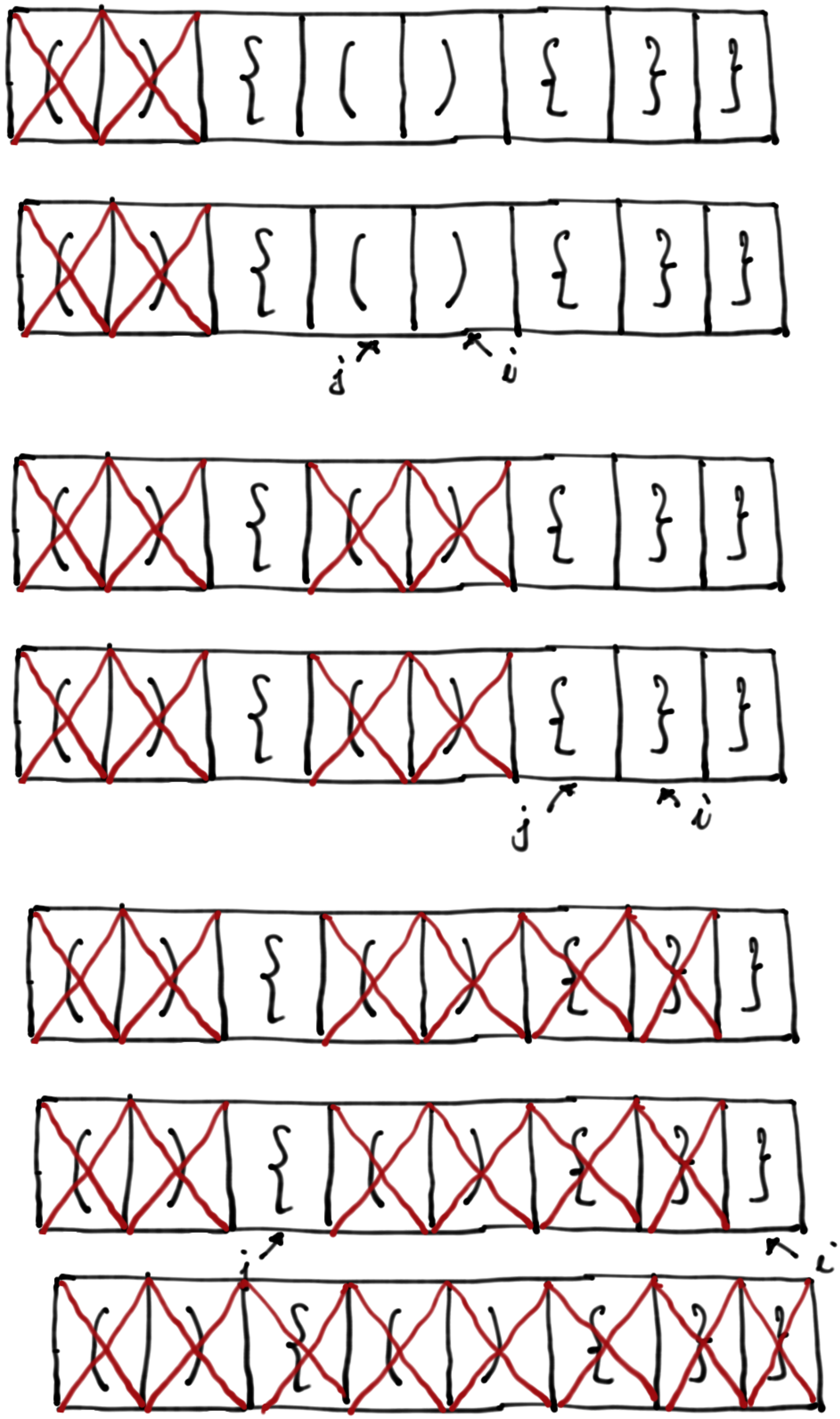
(1) Find the first closing element, say at location i.

(2)



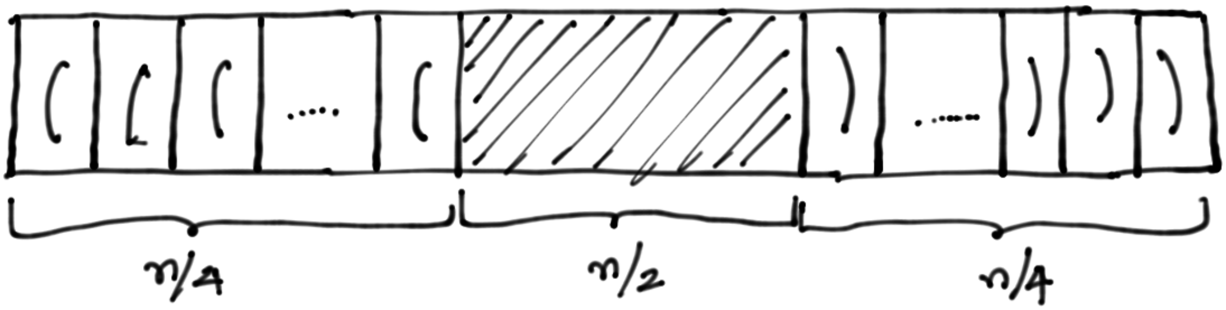
Repeat till all elements are deleted

1. Find the first closing element, say at location i.
2. Let j be the first non-deleted location to the left of i.
3. If the closing element does not matches with opening element, then report that string is not balanced.

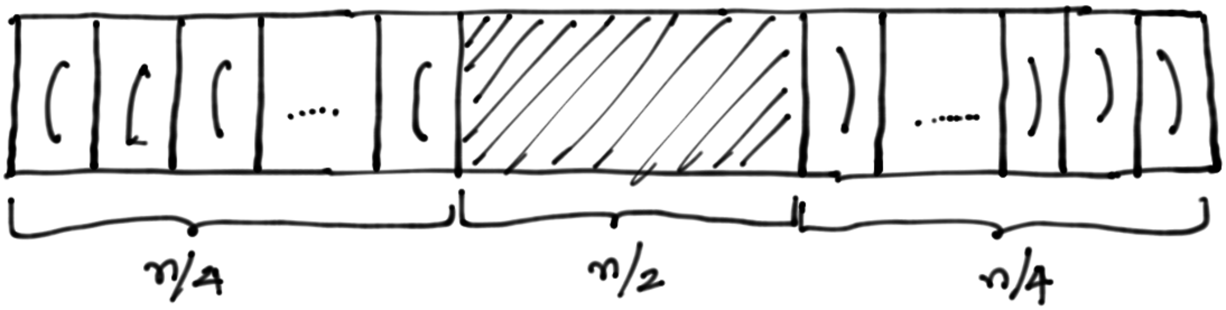

( ) { ( ) { } }
j
i
j
i
j
i

Q: What is the worst case running time of this algorithm?

Q: What is the worst case running time of this algorithm?

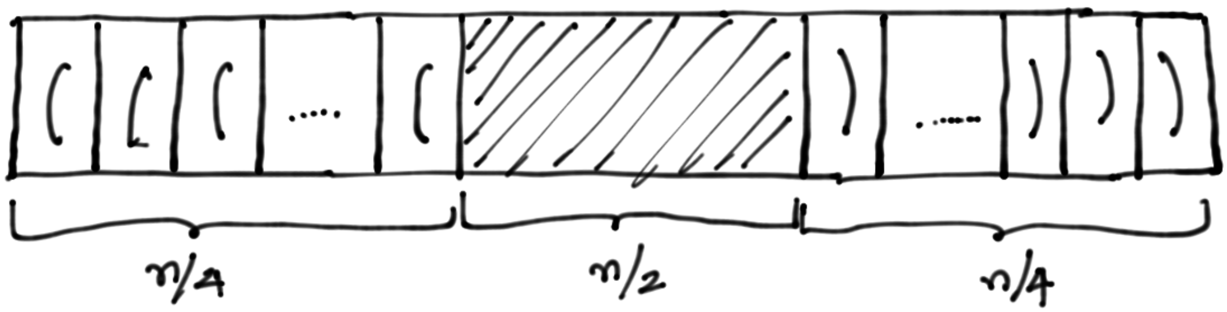


Q: What is the worst case running time of this algorithm?



For each closing parenthesis on the right, we travel $\geq \frac{n}{2}$ deleted elements.

Q: What is the worst case running time of this algorithm?



For each closing parenthesis on the right, we travel $\geq \frac{n}{2}$ deleted elements.

$\Rightarrow$ Total number of steps in the algorithm $\geq \frac{n}{4} \times \frac{n}{2} = \frac{n^2}{8}$

$\Rightarrow$ Running time $= O(n^2)$.

```
// Input: S is an array of strings

while ( S contains a closing element).
{
    Let i be the location of first closing
        element.
    Let j < i be the first non-empty location
        to the left of i
    if ( S[j] & S[i] are of same type)
        remove S[i] & S[j]
    else
    {   print ("Not Balanced");
        return;
    }
}
```

```
// Input: S is an array of strings

while ( S contains a closing element).
{
    Let i be the location of first closing
       element.
    Let j < i be the first non-empty location
       to the left of i
    if ( S[j] & S[i] are of same type)
        remove S[i] & S[j]
    else
    {   print ("Not Balanced");
        return;
    }
}
if ( S contains non-deleted element).
{   print (" Not balanced"):
    return;
}

print ("Balanced"):
return;
```

```
// Input: S is an array of strings

O(n) → while ( S contains a closing element).
{
    Let i be the location of first closing
        element.
O(n) → Let j < i be the first non-empty location
           to the left of i
    if ( S[j] & S[i] are of same type)
         remove S[i] & S[j]
    else
    {    print ("Not Balanced");
         return;
    }
}
if ( S contains non-deleted element).
{     print (" Not balanced"):
      return;
}

print ("Balanced"):
return;
```

Q: Can you do better? May be $O(n)$.

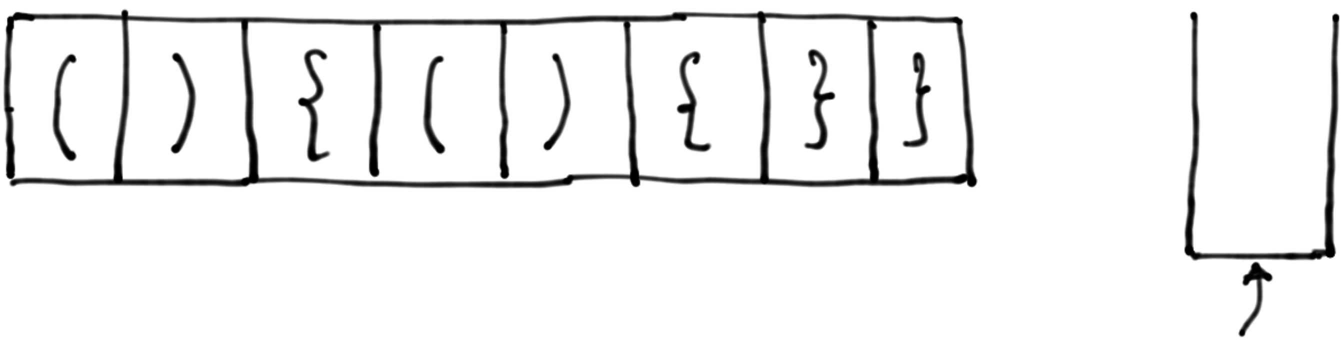
Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

A: Use another data-structure.

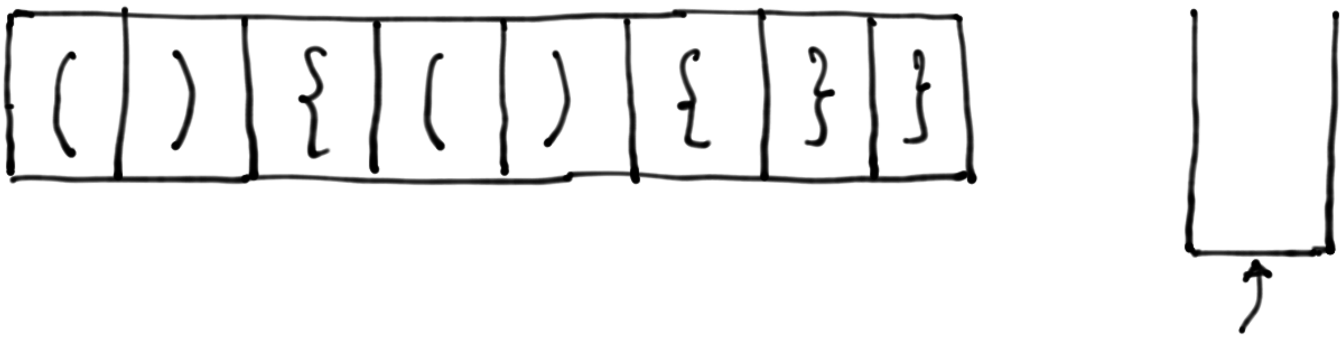


Opening elements as seen in the array

Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

A: Use another data-structure.

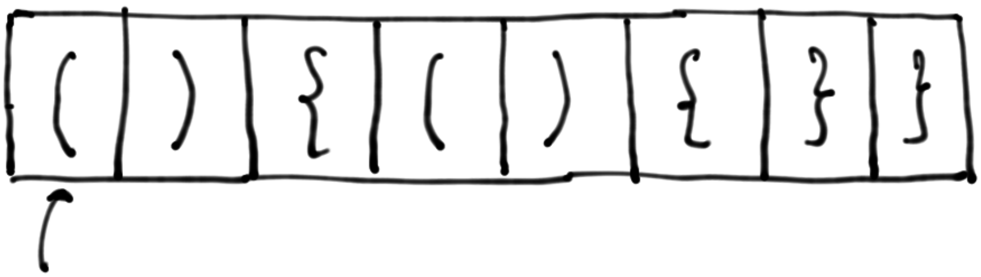


"Non deleted" Opening elements as seen in the array

Q: Can you do better? May be O(n).

Q: Given a closing element i, can we find the corresponding location i in O(1) time?

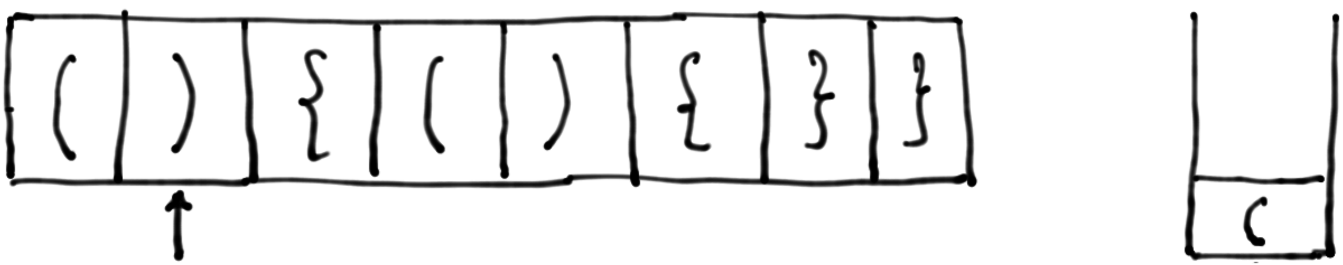
A: Use another data-structure.



Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

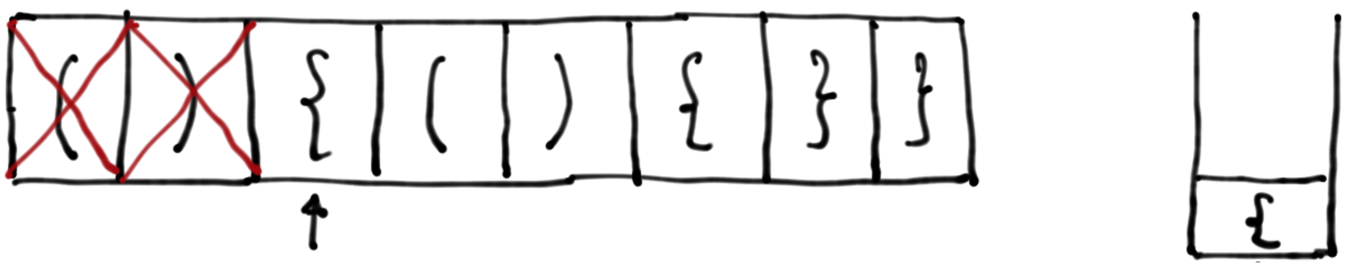
A: Use another data-structure.



Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

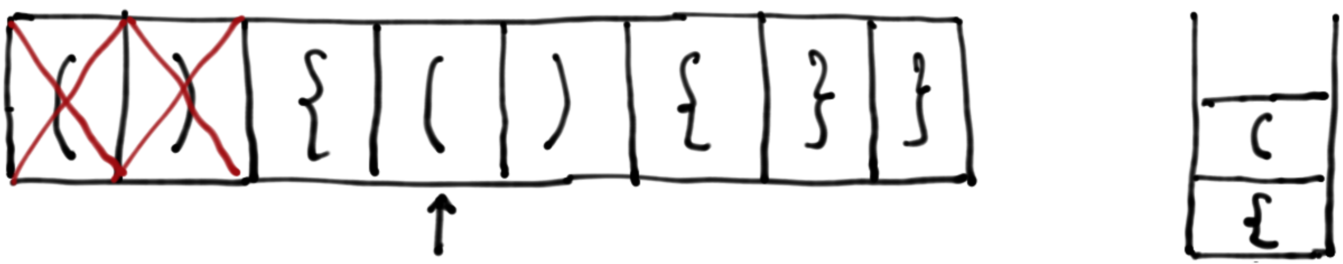
A: Use another data-structure.



Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

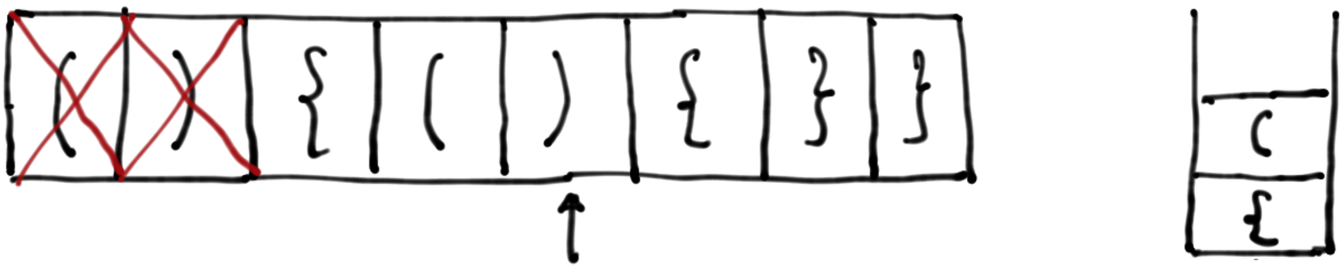
A: Use another data-structure.



Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

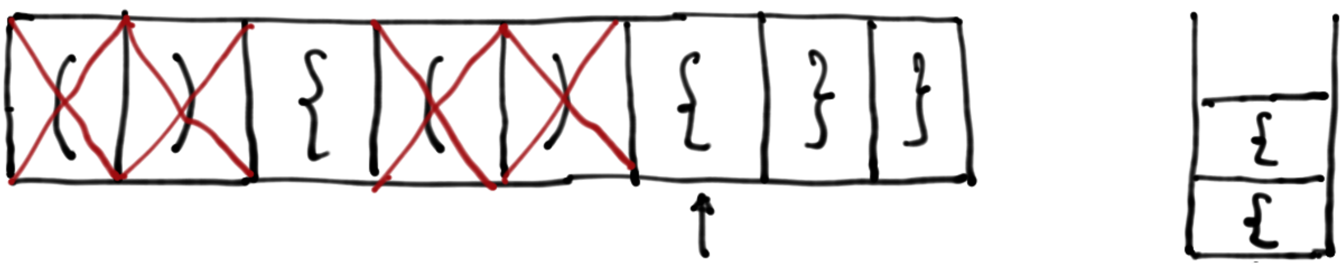
A: Use another data-structure.



Q: Can you do better? May be O(n).

Q: Given a closing element i, can we find the corresponding location i in O(1) time?

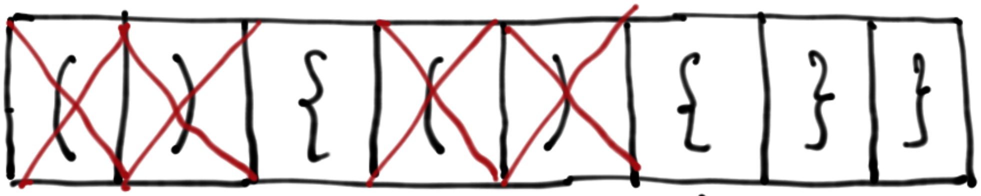
A: Use another data-structure.

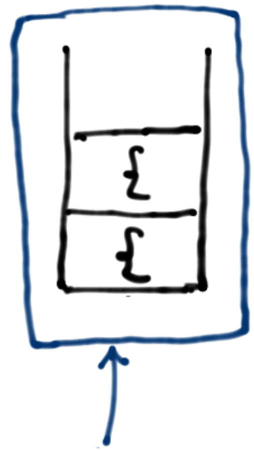


Q: Can you do better? May be $O(n)$.

Q: Given a closing element i, can we find the corresponding location i in $O(1)$ time?

A: Use another data-structure.





Stack

Following important observation about stack.

(1) The last element inserted onto stack is the first element removed.
(Last In First Out) LIFO.

(2) Two operations:
Push(element e)
Pop()

Q: How would you implement Stack?

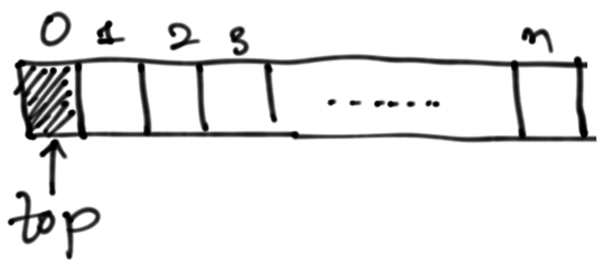
Q: How would you implement Stack?

(1) Using Arrays
(2) Using Lists.

Using Arrays : A[1...n]

```
Create_Empty()
{
    top ← 0;
}
```

represents the topmost element in the stack.

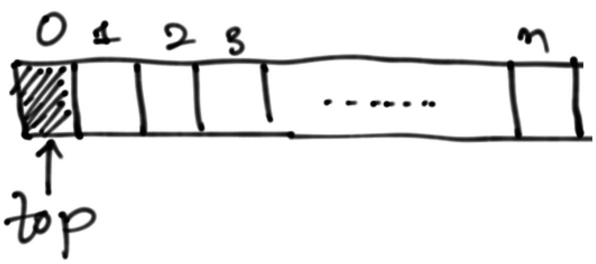


Push(a)

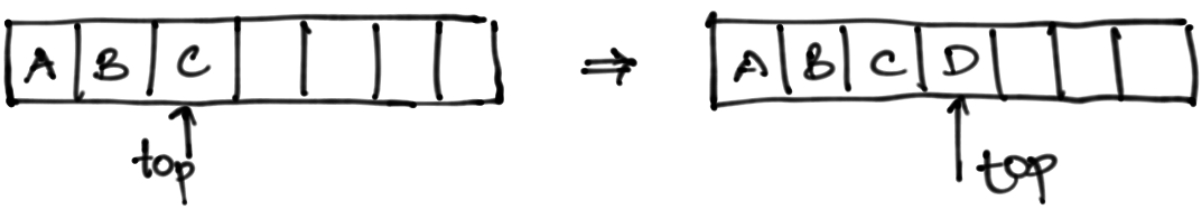
Using Arrays : A[1...n]

```
Create_Empty()
{
    top ← 0;
}
```

represents the topmost element in the stack.




```
Push(a)
{   if( top < n)
    {
        top ← top+1;
        A[top] ← a
    }
    else  print ("Stack Full")
}
```



Pop()

```
Pop()
{   if ( top = 0)
        print "Stack Empty"
    else
    {   a ← A[top] ;
        top ← top-1 ;
        return a ;
    }
}
```

What is the running time of push & pop procedure?

```
Pop()
{   if ( top = 0)
       print "Stack Empty"
    else
    {    a ← A[top] ;
         top ← top-1 ;
         return a ;
    }
}
```

What is the running time of push & pop procedure?

A. O(1).

```
Pop()
{   if ( top = 0)
        print  "Stack Empty"
    else
    {    a ← A[top] ;
         top ← top-1 ;
         return a ;
    }
}
```

What is the running time of push & pop procedure?

A: $O(1)$.

Q: What is the problem with this implementation?

```
Pop()
{   if ( top = 0)
        print "Stack Empty"
    else
    {   a ← A[top] ;
        top ← top-1 ;
        return a ;
    }
}
```

What is the running time of push & pop procedure?

A: O(1).

Q: What is the problem with this implementation?

A: Array size is fixed.

List Implementation

value
next

Create. Empty ()
{

List Implementation

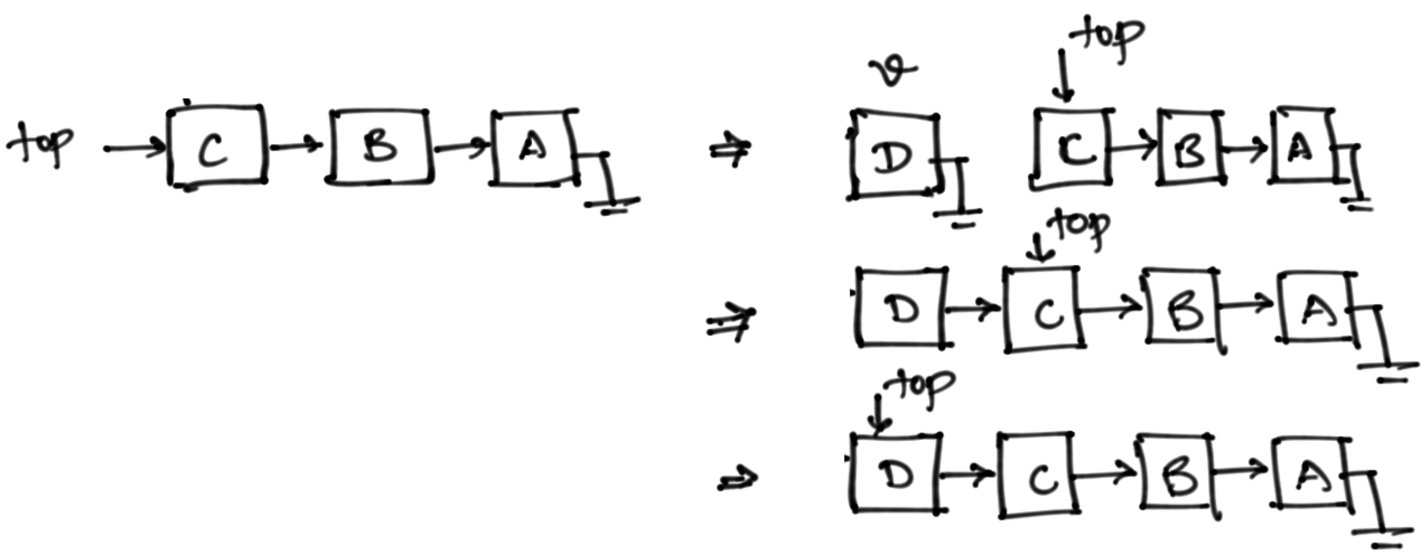
```
Create. Empty()
{    top ← null;
}

Push(a)
{
```

List Implementation

```
Create. Empty ()
{    top ← null ;
}

Push(a)
{ v ← ask for memory for a new node ;
    v.value ← a ;
    v.next ← null ;
    if ( top is not null)
    {    v.next ← top ;
    }
    top ← v ;
}
```

List Implementation

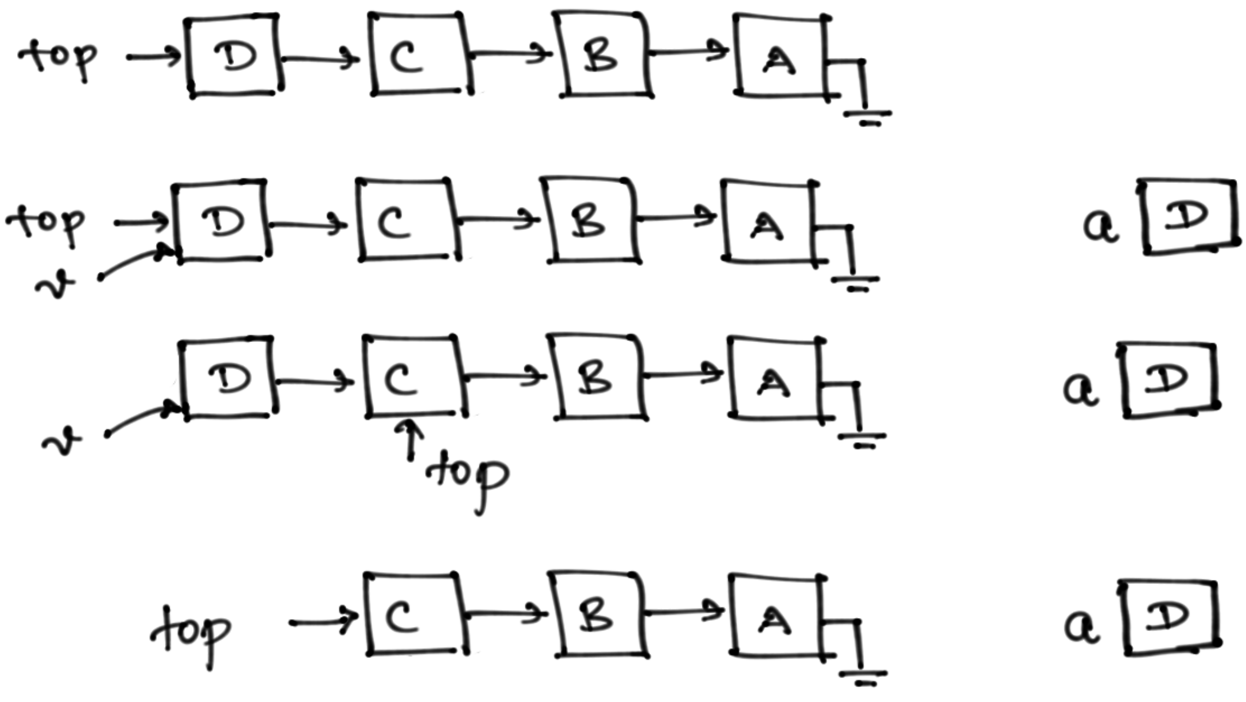
```
Create. Empty ()
{    top ← null;
}

Push(a)
{ v ← ask for memory for a new node;
    v.value ← a;
    v.next ← null;
    if ( top is not null)
    {   v.next ← top;
    }
    top ← v;
}
```



```
Pop()
{   if( top is null)
    {   print "Stack Empty";
    }
    else
    {
```

```
Pop()
{   if( top is null)
    {   print "Stack Empty";
    }
    else
    {   v ← top;
        a ← v.value;
        top ← top.next;
        deallocate the memory associated with
        node v;
        return a;
    }
}
```

```
Pop()
{   if ( top is null)
    {   print "Stack Empty";
    }
    else
    {   v ← top;
        a ← v.value;
        top ← top.next;
        deallocate the memory associated with
        node v;
        return a;
    }
}
```



Running time of push & pop = ??

Running time of push & pop = $O(1)$.

Q: Now you have all the ingredients in hand. Can you design an $O(n)$ time algorithm for balanced parenthesis problem?

```
// Input: S is an array of strings

while ( S contains a closing element).
{
    Let i be the location of first closing
        element.
    Let j < i be the first non-empty location
        to the left of i
    if ( S[j] & S[i] are of same type)
        remove S[i] & S[j]
    else
    {   print ("Not Balanced");
        return;
    }

if ( S contains non-deleted element).
{   print (" Not balanced");
    return;
}

print ("Balanced");
return;
```

```
for ( i ← 1 to n)
{    if ( S[i] is an opening element)
```

```
for ( i ← 1 to n)
{    if ( S[i] is an opening element)
         Push ( S[i] ) ;
     else
```

```
for ( i ← 1 to n)
{    if ( S[i] is an opening element)
         Push ( S[i]) ;

     else
     {     a ← Pop() ;
           if ( a = '(' & S[i] = ')' or
                a = '{' & S[i] = '}')
```

```
for ( i ← 1 to n)
{    if ( S[i] is an opening element)
        Push ( S[i]) ;
     else
     {    a ← Pop();
          if ( a = '(' & S[i] = ')' or
               a = '{' & S[i] = '}')
                continue;
          else
          {    print "Not balanced";
               return;
          }
     }
}
```

```
for ( i ← 1 to n)
{    if ( S[i] is an opening element)
         Push ( S[i]) ;

     else
     {       a ← Pop();
             if ( a = '(' & S[i] = ')' or
                  a = '{' & S[i] = '}')
                   continue;

             else
             {     print "Not balanced";
                   return;
             }
     }
}
if ( Stack is not empty)
     print " Not balanced"
else
     print "Balanced" ;
```

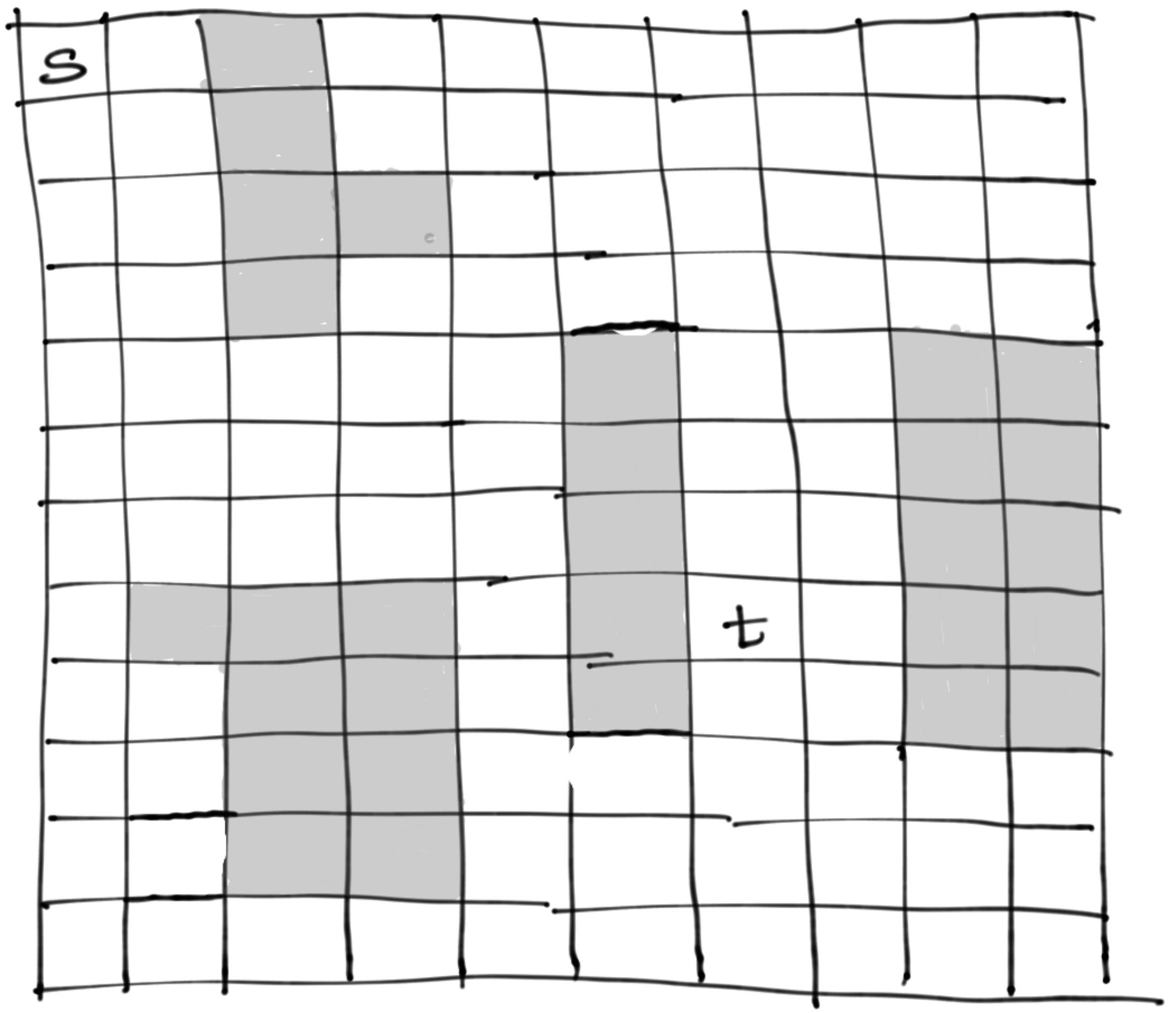
Running Time :

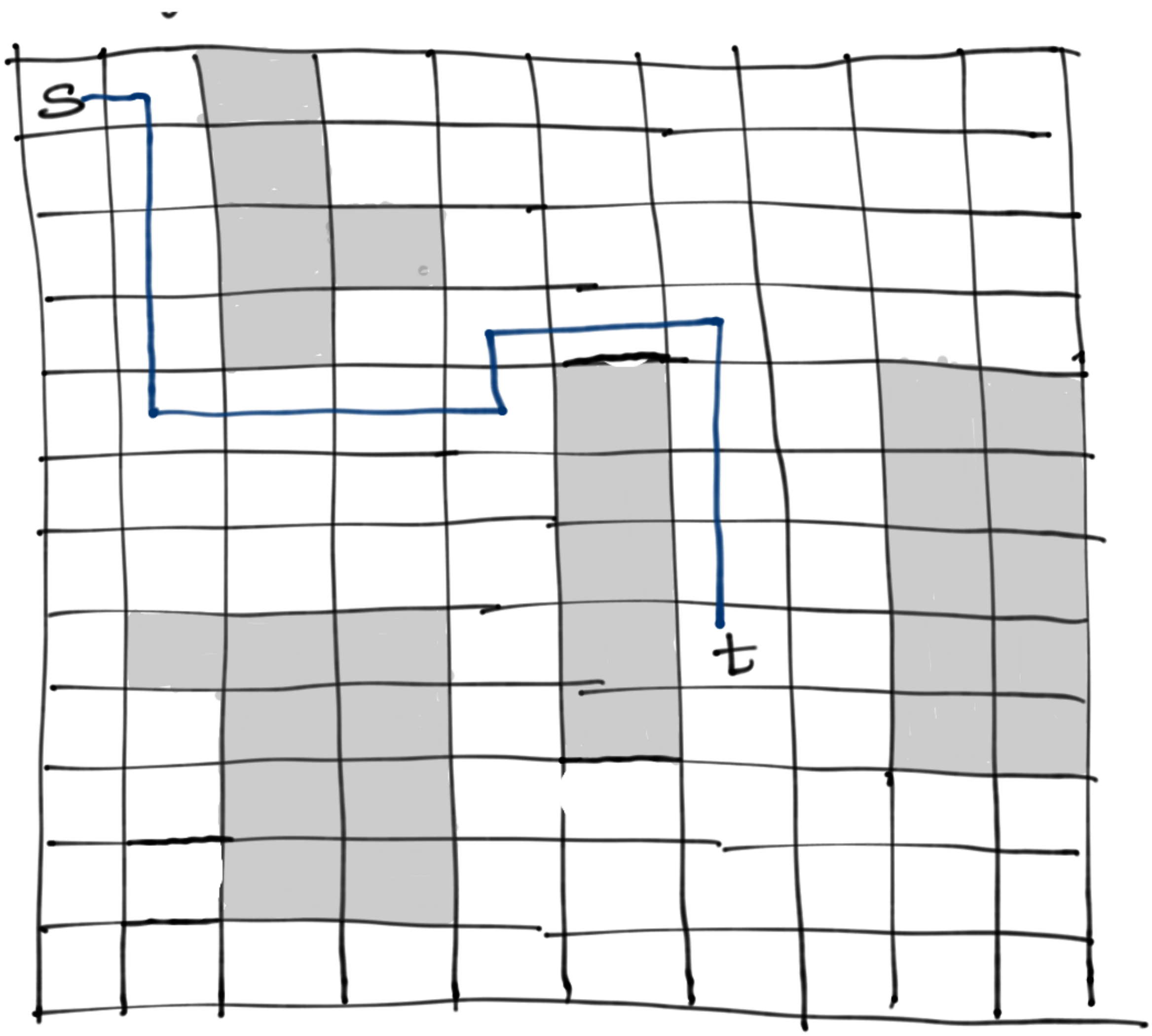
```
for ( i ← 1  to n)
{     if ( S[i]  is  an  opening  element)
          Push ( S[i] ) ;

      else
      {       a ← Pop();
              if ( a = '(' & S[i] = ')'  or
                   a = '{' & S[i] = '}')
                     continue;

              else
              {      print "Not balanced";
                      return;
              }
      }
}
if ( Stack is not empty)
     print "Not balanced"
else
     print "Balanced" ;
```

Running Time : O(n).

- Assume that you are given a $n \times n$ grid
- Some of the cells in the grid are obstacles.
- Start location $s(0,0)$.
- Ending location $t(i,j)$

Find the shortest path from s to t avoiding obstacles?





Q: Give such a grid, find the shortest path between s & t?

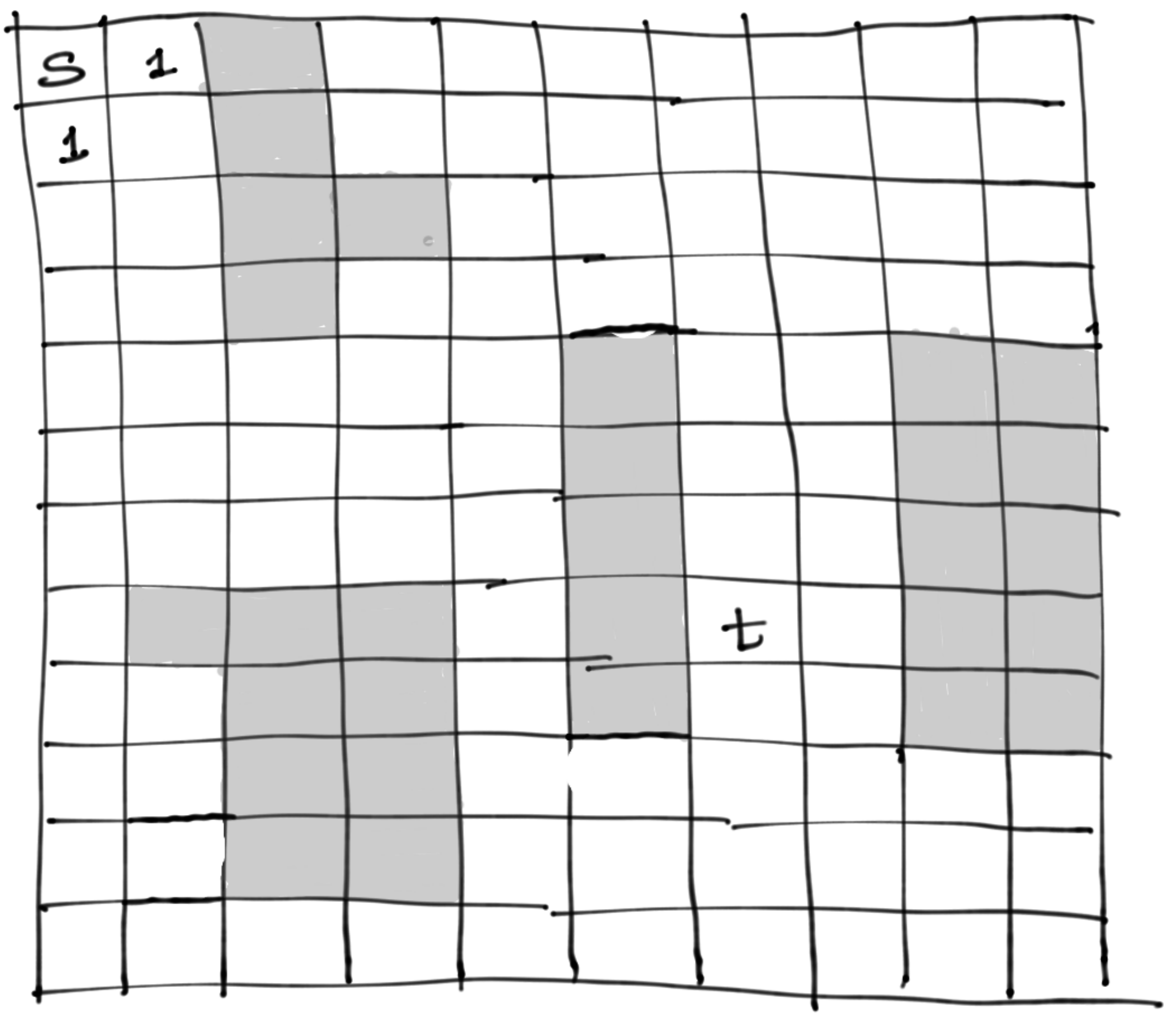
Q: What will you do?

Q : Can you find all the cells that are at distance 0 from s?

Q: Can you find all the cells that are at distance 0 from s?

A: $L_0 = \{s\}$

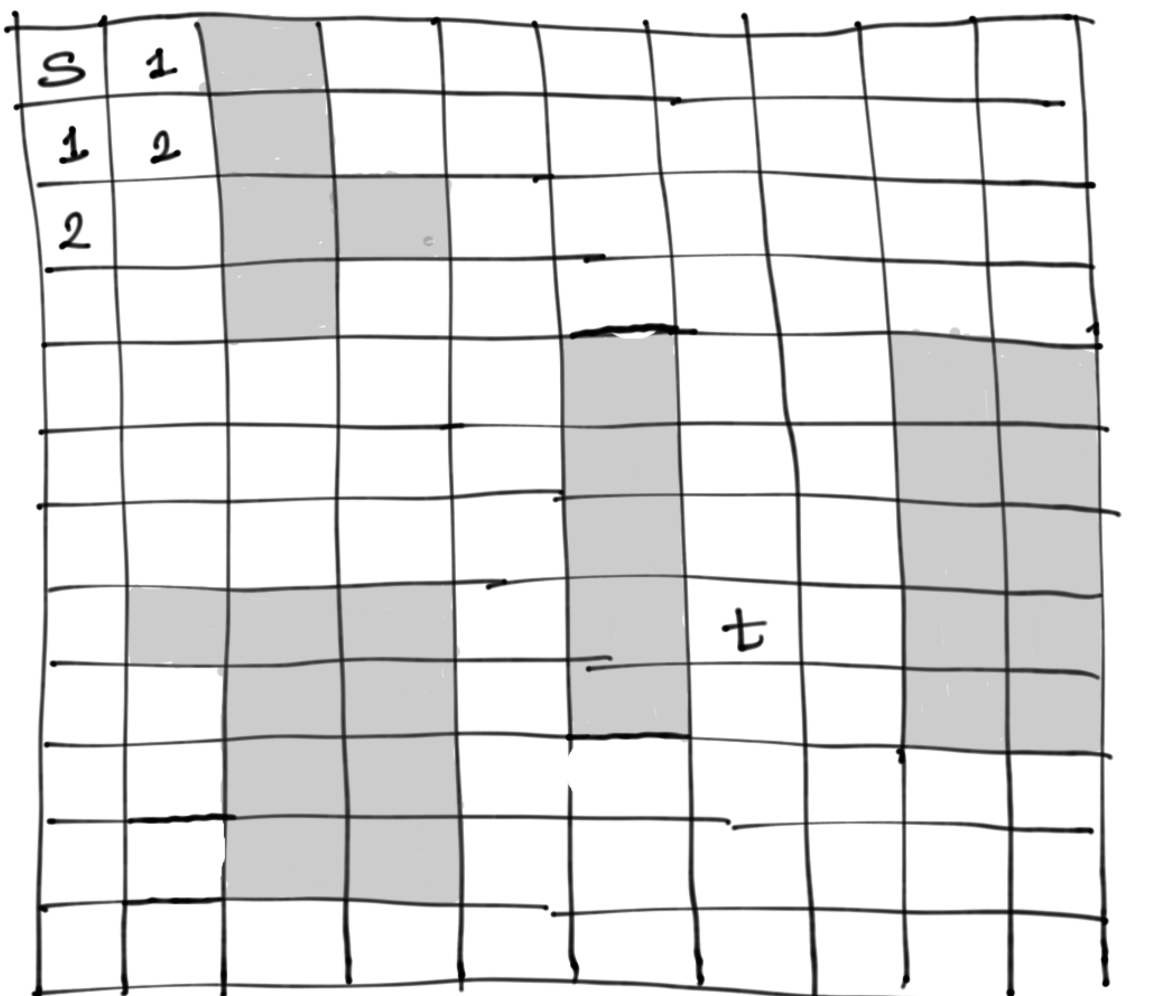
Q: Given L_0, can you find all the cells at a distance 1 from s?



Q: Can you find all the cells that are at distance 0 from s?

A: $L_0 = \{s\}$

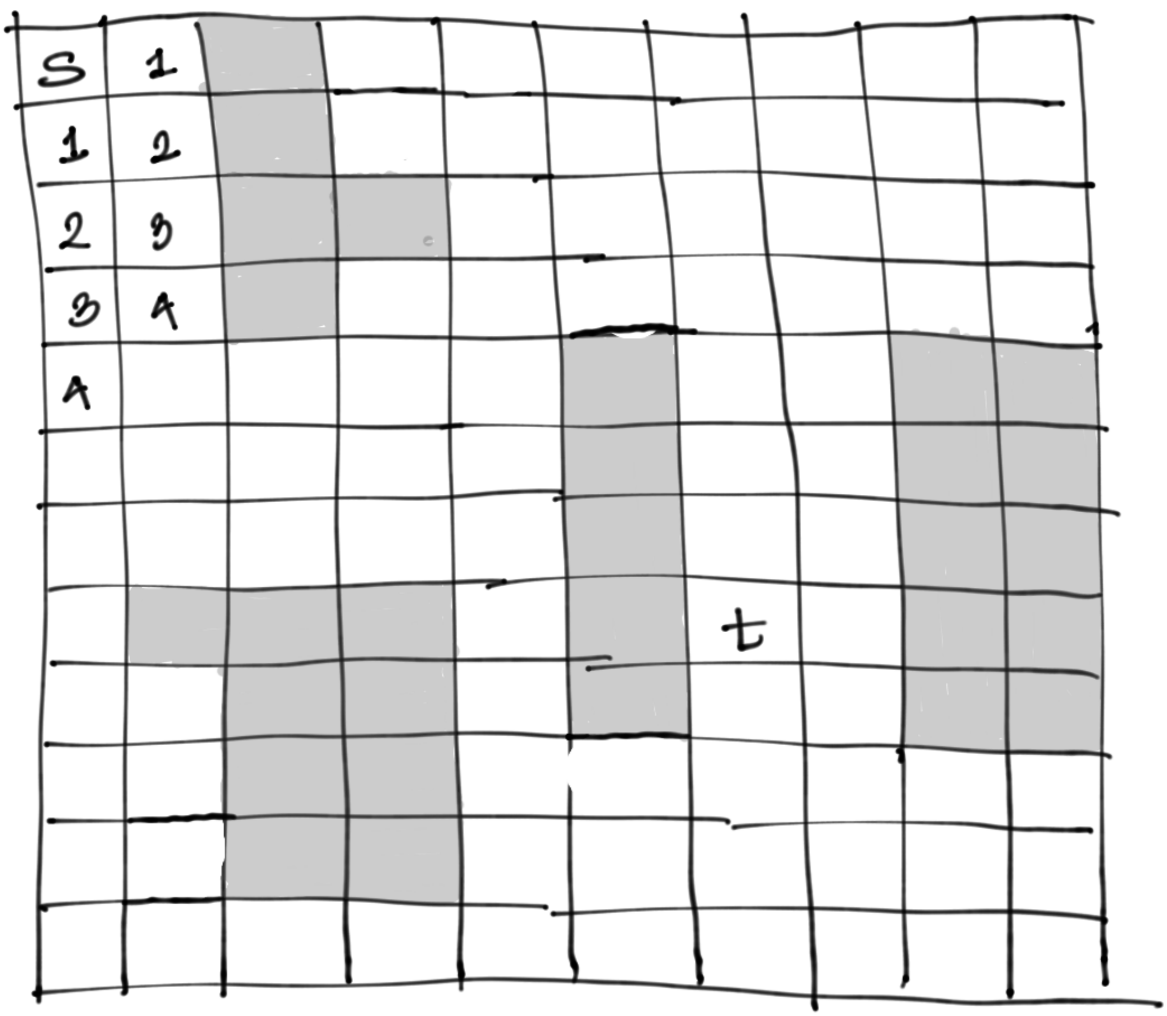
Q: Given L_0, can you find all the cells at a distance 1 from s?



Q: Can you find all the cells that are at distance 0 from s?

A: $L_0 = \{s\}$

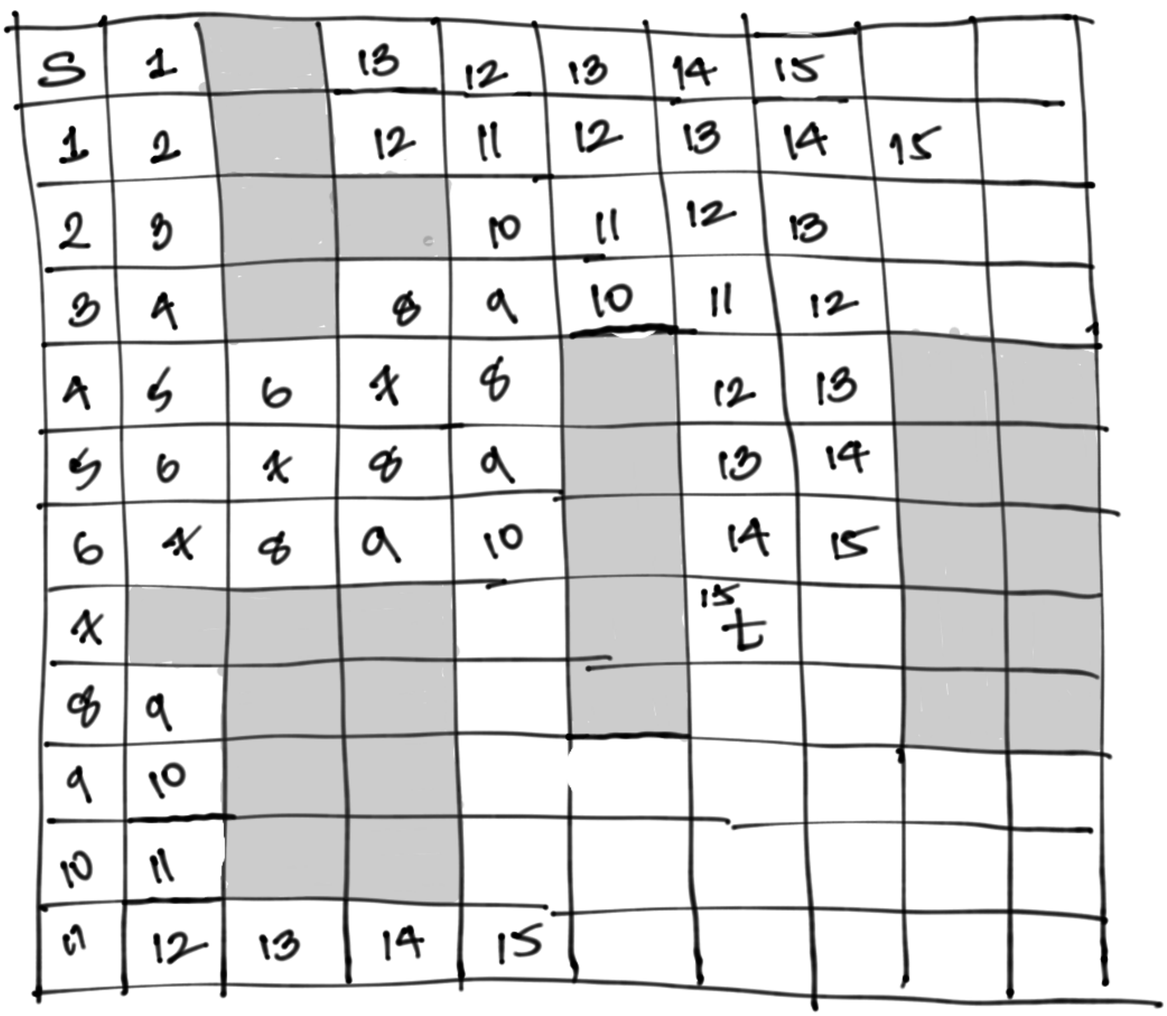
Q: Given L_0, can you find all the cells at a distance 1 from s?



Q : Can you find all the cells that are at distance 0 from s?

A : $L_0 = \{s\}$

Q: Given L_0, can you find all the cells at a distance 1 from s?



Q: Any idea about the running time of this algorithm

Q: Any idea about the running time of this algorithm

A: At least n^2 as the grid size is n^2.

```
main()
{    for each cell c in the grid
          distance [c] ← ∞ ;
     L0 ← {s}
     i ← 0;
     while ( Li does not contain t or
               Li is not empty)
     {    Find-Next-Layer(Li)
     }    i ← i+1
}
```

```
main()
{   for each cell c in the grid
        distance [c] ← ∞ ;
    L0 ← {s}
    i ← 0 ;
    while ( Li does not contain t or
              Li is not empty)
    {   Find-Next-Layer (Li)
    }   i ← i+1
}

Find-Next-Layer ( Li)
{   Li+1 ← ϕ ;
    for each cell c in Li
    {
```

```
main()
{   for each cell c in the grid
        distance [c] ← ∞ ;
    Lo ← {s}
    i ← 0 ;
    while ( Li does not contain t or
            Li is not empty)
    {   Find-Next-Layer(Li)
    }   i ← i+1
}

Find-Next-Layer(Li)
{   Li+1 ← φ ;
    for each cell c in Li
    {   for each neighbor b of c s.t
               b is not an obstacle
          {
```

```
main()
{   for each cell c in the grid
        distance [c] ← ∞ ;
    L0 ← {s}
    i ← 0;
    while ( Li does not contain t or
            Li is not empty)
    {   Find-Next-Layer(Li)
    }   i ← i+1
}

Find-Next-Layer(Li)
{   Li+1 ← φ ;
    for each cell c in Li
    {   for each neighbor b of c s.t
            b is not an obstacle
        {  if ( distance[b] = ∞)
           {   distance[b] ← i+1;
               Li+1 ← Li+1 U {b};
           }
        }
}   }
```

Q: Running Time

```
main()
{   for each cell c in the grid              O(n^2)
        distance [c] <- ∞ ;
    L0 <- {s}                                 O(1)
    i <- 0 ;
    while ( Li does not contain t or
              Li is not empty)
    {     Find-Next-Layer (Li)
    }     i <- i+1
}


Find-Next-Layer (Li)
{       Li+1 <- φ ;                                      O(1)
|Li|    for each cell c in Li
        {   for each neighbor b of c s.t
                  b is not an obstacle
            {  if ( distance [b] = ∞)                     O(1)
                {   distance [b] <- i+1 ;
                    Li+1 <- Li+1 ∪ {b} ;
                }
            }
        }
}
```

Running Time $= O\left(n^2 + \sum_{i=1}^{\infty} |L_i|\right)$

Q: How much is $\sum_{i=1}^{\infty} |L_i|$

Q: How much is $\sum_{i=1}^{\infty} |L_i|$

<u>Observation</u> : Each cell is a part of one layer L_i.

Q: How much is $\sum_{i=1}^{\infty} |L_i|$

Observation: Each cell is a part of one layer L_i.

$$\Downarrow$$

$$\sum_{i=1}^{\infty} |L_i| = O(n^2).$$

$\Rightarrow$ Running Time $= O(n^2)$.

We will focus on an implementation problem.

Q: What data-structure will you use to implement it?

We will focus on an implementation problem.

Q: What data-structure will you use to implement it ?

A: Linked- List.

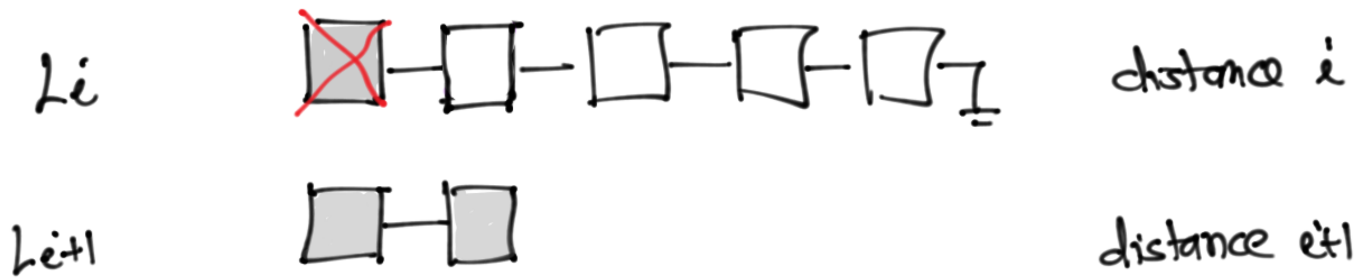
So, we would be using many linked list.

Q: Can we use only one linked list instead ?

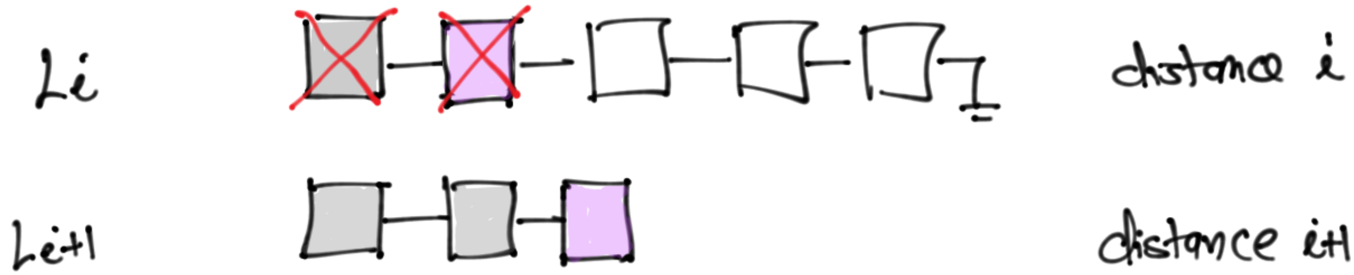
Current algo

L_i  distance i

Current algo

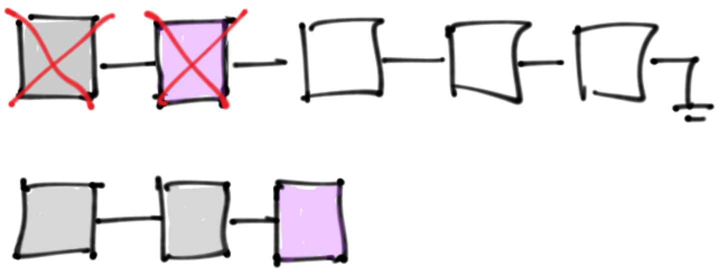


Current algo



Current algo

L_i distance i



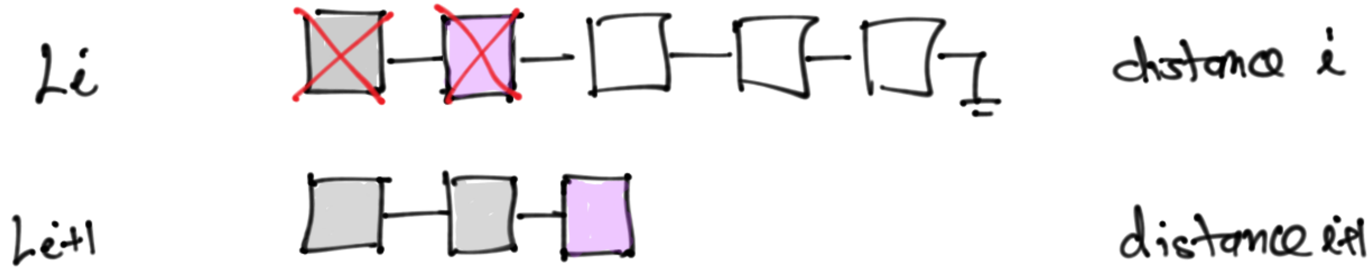
L_{i+1} distance $i+1$

<u>Observation</u>

(1) We are visiting cells in the non-decreasing order of their distance from s.

(2) We can add L_{i+1} at the end of L_i

Current algo

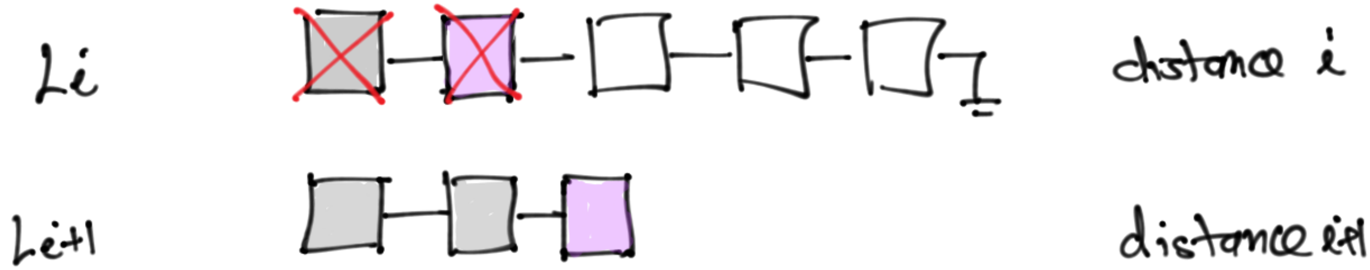


Observation

(1) We are visiting cells in the non-decreasing order of their distance from s.

(2) We can add L_{i+1} at the end of L_i

Current algo



<u>Observation</u>

(1) We are visiting cells in the non-decreasing order of their distance from s.

(2) We can add L_{i+1} at the end of L_i

<u>Observation</u>: Analoguous to stack
Push – Not at top but at bottom
Pop – Element at top

F I F O

First In First Out

Datastructure : Queue

<u>Two operations</u>

(1) Enqueue(a) : Add an element a at the end of the queue.

(2) Dequeu() : Remove an element from the start of the queue.

```
Q ← Create-Empty()
for each cell c in the grid
    distance[c] ← ∞ ;

distance[s] ← 0;
Q.enqueue(s) ;      //setting up to

while (Q is not empty)
{
    c ← Deque();
    for each neighbor b of c that is not
                  an obstacle
    {
        if   distance[b] = ∞
        {
            distance[b] ← distance[c] +1;
            Enqueue(b);
        }
    }
}
```

Q: How would you implement Queues:

Q: How would you implement Queues:

1. Arrays
2. Linked List

Queues Using Linked Lists

```
Create-Empty ()
{
    front ← null;
    rear ← null;
}
```

Enqueue (a)

Queues Using Linked Lists

```
Create-Empty ()
{
    front ← null;
    rear ← null;
}


Enqueue (a)
{   v ← allocate memory for a new node;
    v.value ← a;
    v.next ← null;

    if (                    )
    {



    }
    else
    {

    }
}
```

Queues Using Linked Lists

```
Create-Empty ()
{
    front ← null;
    rear ← null;
}


Enqueue (a)
{   v ← allocate memory for a new node;
    v.value ← a;
    v.next ← null;

    if (                    )
    {



    }
    else
    {   rear.next ← v;
        rear ← v;
    }
}
```

Queues Using Linked Lists

```
Create-Empty()
{
    front ← null;
    rear ← null;
}


Enqueue(a)
{   v ← allocate memory for a new node;
    v.value ← a;
    v.next ← null;

    if (rear = null)
    {
        front ← v;
        rear ← v;
    }
    else
    {   rear.next ← v;
        rear ← v;
    }
}
```

```
Dequeue()
{
    // Queue is empty
```

```
Dequeue()
{
    // Queue is empty
    if ( front = null )  //or rear = null)
        print "Queue Empty";
    else
    {
```

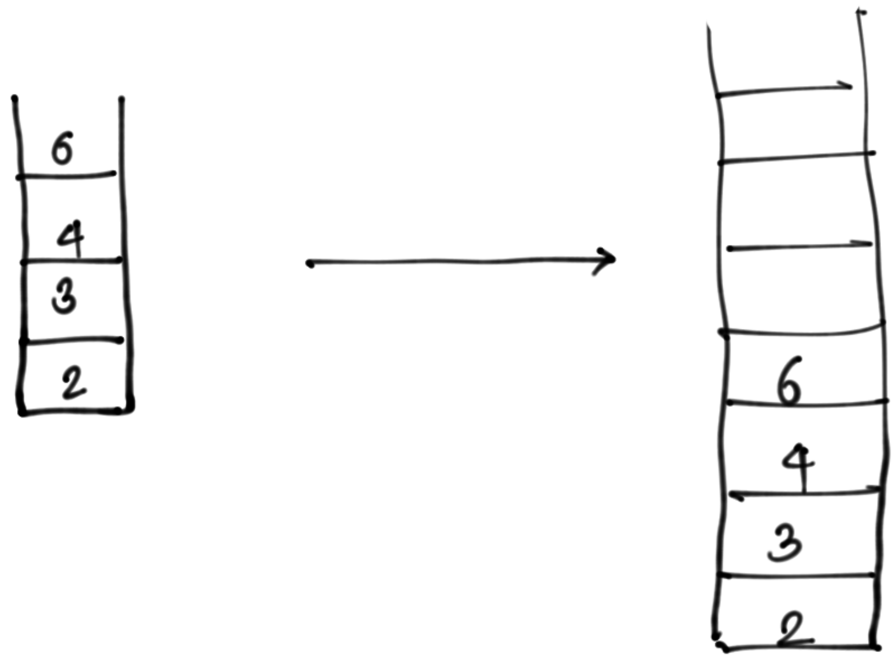
```
Dequeue()
{
    // Queue is empty
    if ( front = null )  //or rear = null)
        print "Queue Empty";
    else
    {   v ← front ;
        a ← v. value ;
        if (            )
        {



        }
        else
        {
        }




    }
}
```

```
Dequeue()
{
    // Queue is empty
    if ( front = null )  //or rear = null)
        print "Queue Empty";
    else
    {   v ← front;
        a ← v.value;
        if (                )
        {



        }
        else
        {   front ← front.next
        }
        deallocate the memory associated
         to node v;
        return a;
    }
}
```

```
Dequeue()
{
    // Queue is empty
    if ( front = null )  //or rear = null)
        print "Queue Empty";
    else
    {   v ← front;
        a ← v.value;
        if ( front = rear )
        {   // Queue is empty now
            front ← null;
            rear ← null;
        }
        else
        {   front ← front.next
        }
        deallocate the memory associated
         to node v;
        return a;
    }
}
```

PROBLEM: ASSUME THAT YOU HAVE IMPLEMENTED STACK USING AN ARRAY. WHILE PUSHING AN ELEMENT, IF THE ARRAY IS FULL, THEN WE DO THE FOLLOWING.

PROBLEM: ASSUME THAT YOU HAVE IMPLEMENTED STACK USING AN ARRAY. WHILE PUSHING AN ELEMENT, IF THE ARRAY IS FULL, THEN WE DO THE FOLLOWING.



ASSUMING THAT WE START WITH AN ARRAY OF SIZE 1, SHOW THAT A SEQUENCE OF n PUSH TAKES $O(n)$ TIME.

$$\sum_{i=1}^{n} \text{TIME TAKEN TO PUSH } i^{th} \text{ ELEMENT} = O(n).$$

PROBLEM 2: GIVEN A LINKED LIST & TWO NUMBERS a & b, REVERSE THE SUBLIST BETWEEN a^{th} & b^{th} NODE IN THE LINKED LIST.

$a = 3, \quad b = 6$

$$\boxed{2} \to \boxed{4} \to \boxed{1} \to \boxed{3} \to \boxed{7} \to \boxed{9} \to \boxed{10} \to \perp$$

$$\Downarrow$$

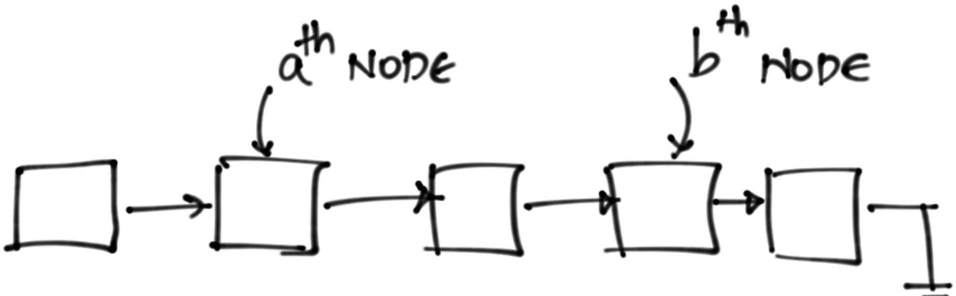
$$\boxed{2} \to \boxed{4} \to \boxed{9} \to \boxed{7} \to \boxed{3} \to \boxed{1} \to \boxed{10} \to \perp$$

TIME TAKEN: $O(b)$.

p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVp ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

```
tempp ← p;
NEXTp ← p.next
WHILE (NEXTp IS NOT NEXTq)
{
    NEXTNEXTp ← NEXTp.next;
    NEXTp.next ← tempp;
    tempp ← NEXTp;
    NEXTp ← NEXTNEXTp;
}
PREVp.next ← q;
p.next ← NEXTq
```

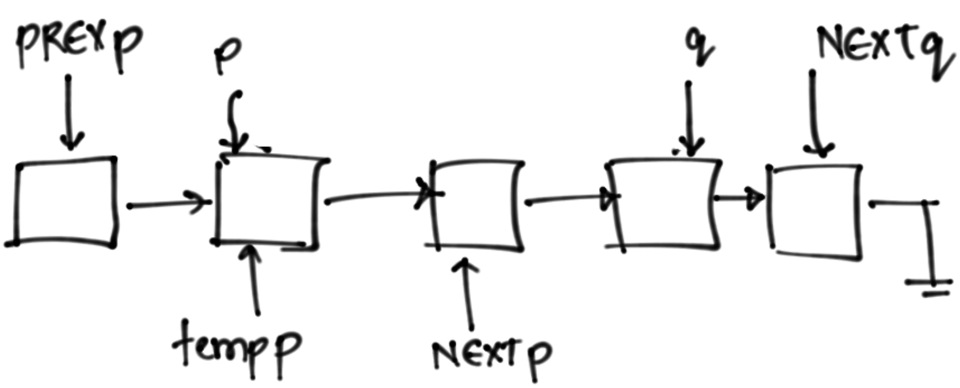


→ p ← THE a^{th} NODE IN THE LINKED LIST
→ q ← THE b^{th} NODE IN THE LINKED LIST

→ PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
→ NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

→ tempp ← p;
→ NEXTp ← p.next

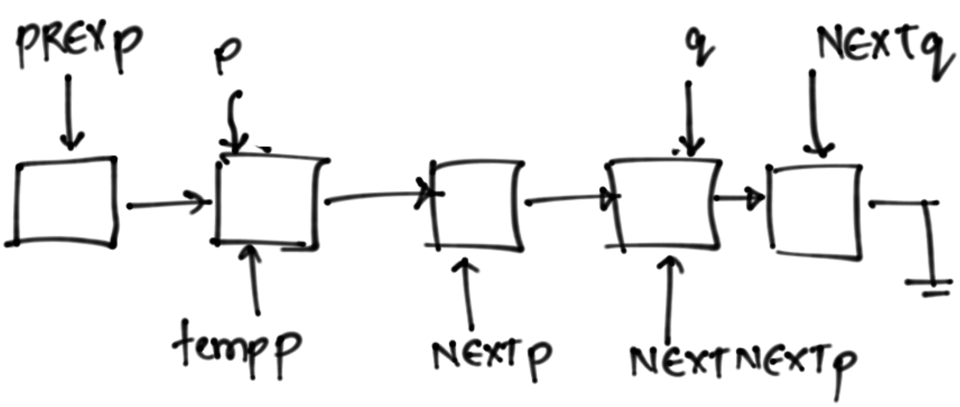
```
WHILE(NEXTp IS NOT NEXTq)
{
    NEXTNEXTp ← NEXTp.next;
    NEXTp.next ← tempp;
    tempp ← NEXTp;
    NEXTp ← NEXTNEXTp;
}
  PREVp.next ← q;
  p.next ← NEXTq
```



p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

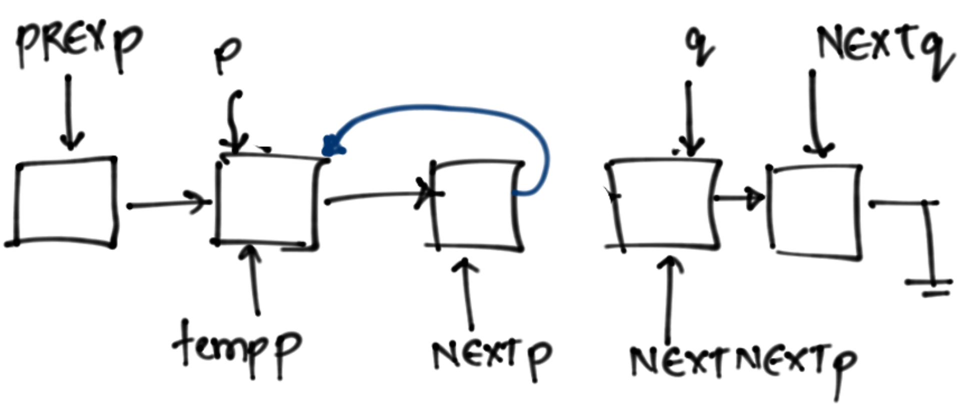
```
tempp ← p;
NEXTp ← p.next
WHILE (NEXTp IS NOT NEXTq)
{
→  NEXTNEXTp ← NEXTp.next;
   NEXTp.next ← tempp;
   tempp ← NEXTp;
   NEXTp ← NEXTNEXTp;
}
PREVp.next ← q;
p.next ← NEXTq
```



p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVp ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

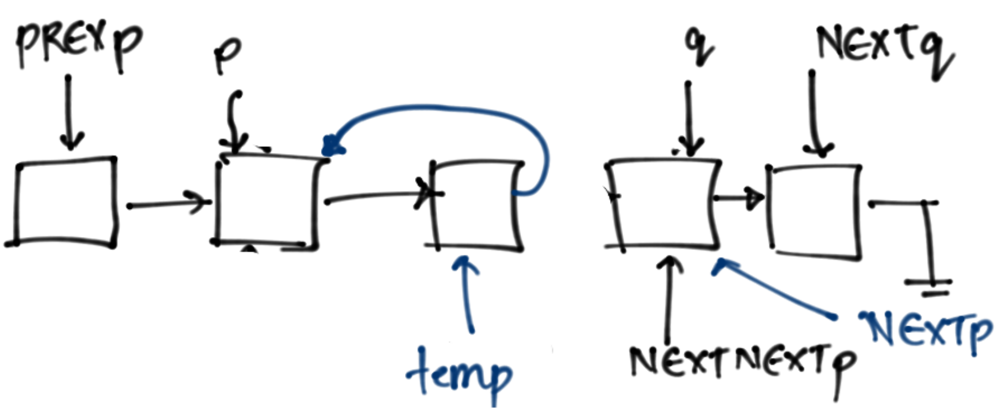
```
tempp ← p;
 NEXTp ← p.next
WHILE (NEXTp IS NOT NEXTq)
{
   NEXTNEXTp ← NEXTp.next;
→  NEXTp.next ← tempp;
   tempp ← NEXTp;
   NEXTp ← NEXTNEXTp;
}
 PREVp.next ← q;
 p.next ← NEXTq
```



p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

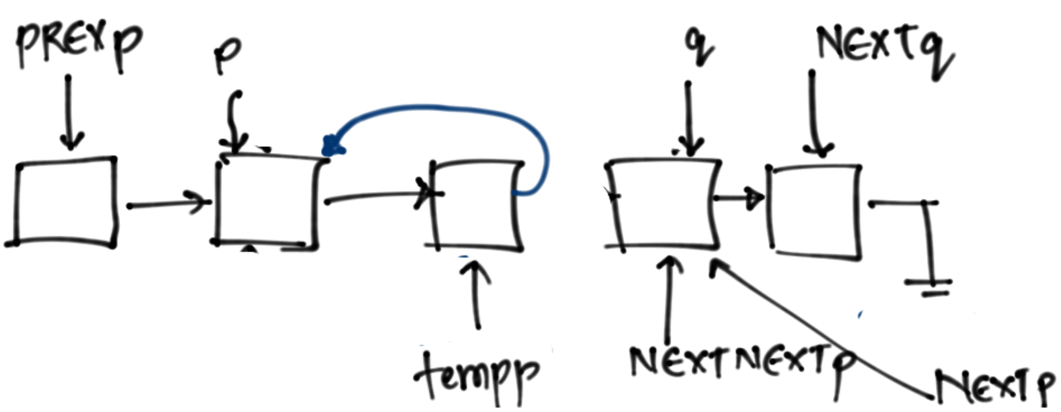
```
tempp ← p;
NEXTp ← p.next
WHILE (NEXTp IS NOT NEXTq)
{
    NEXTNEXTp ← NEXTp.next;
    NEXTp.next ← tempp;
    tempp ← NEXTp;
    NEXTp ← NEXTNEXTp;
}
PREVp.next ← q;
p.next ← NEXTq
```



p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVp ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

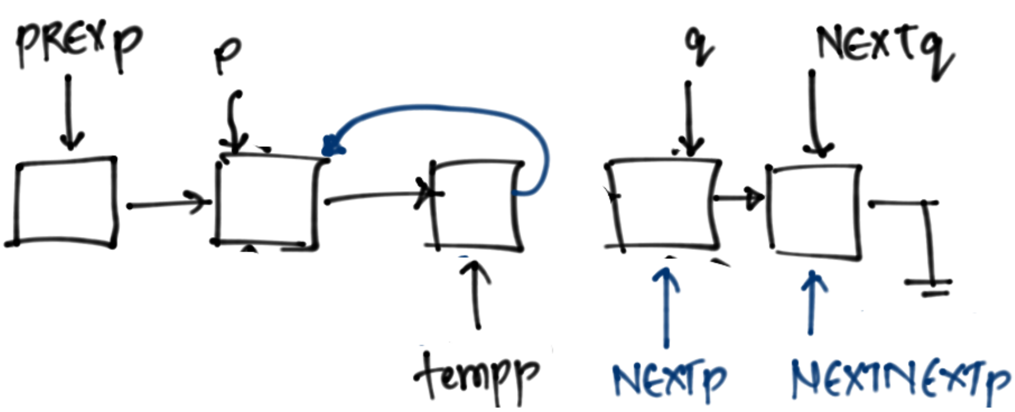
```
  tempp ← p;
  NEXTp ← p.next
→ WHILE(NEXTp IS NOT NEXTq)
  {
      NEXTNEXTp ← NEXTp.next;
      NEXTp.next ← tempp;
      tempp ← NEXTp;
      NEXTp ← NEXTNEXTp;
  }
  PREVp.next ← q;
  p.next ← NEXTq
```



p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVp ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

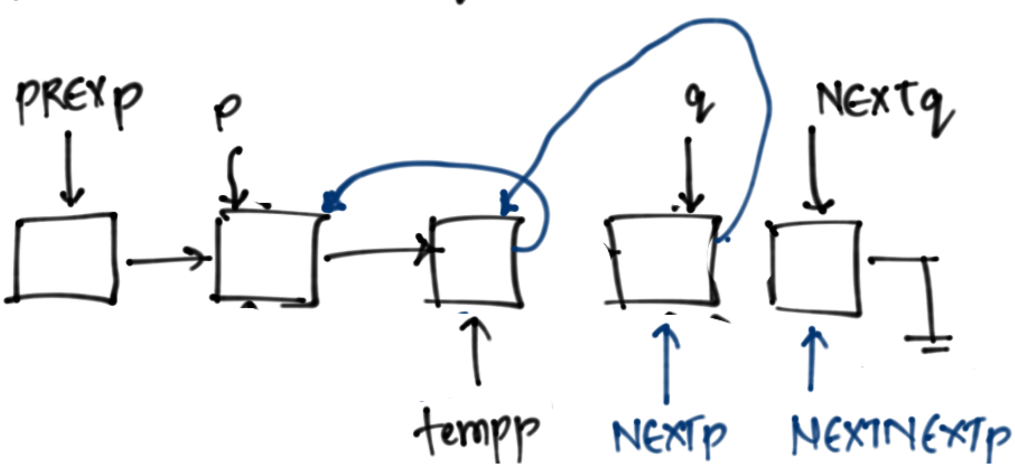
```
tempp ← p;
NEXTp ← p.next
WHILE (NEXTp IS NOT NEXTq)
{
    NEXTNEXTp ← NEXTp.next;
    NEXTp.next ← tempp;
    tempp ← NEXTp;
    NEXTp ← NEXTNEXTp;
}
PREVp.next ← q;
p.next ← NEXTq
```



p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

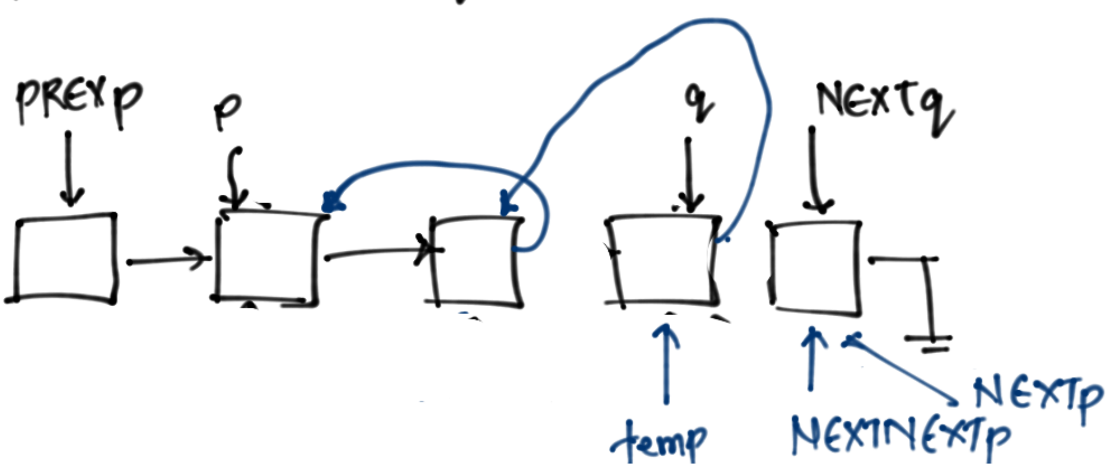
```
tempp ← p;
NEXTp ← p.next
WHILE (NEXTp IS NOT NEXTq)
{
    NEXTNEXTp ← NEXTp.next;
→   NEXTp.next ← tempp;
    tempp ← NEXTp;
    NEXTp ← NEXTNEXTp;
}
PREVp.next ← q;
p.next ← NEXTq
```



p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

```
tempp ← p;
NEXTp ← p.next
WHILE (NEXTp IS NOT NEXTq)
{
    NEXTNEXTp ← NEXTp.next;
    NEXTp.next ← tempp;
    tempp ← NEXTp;
    NEXTp ← NEXTNEXTp;
}
PREVp.next ← q;
p.next ← NEXTq
```

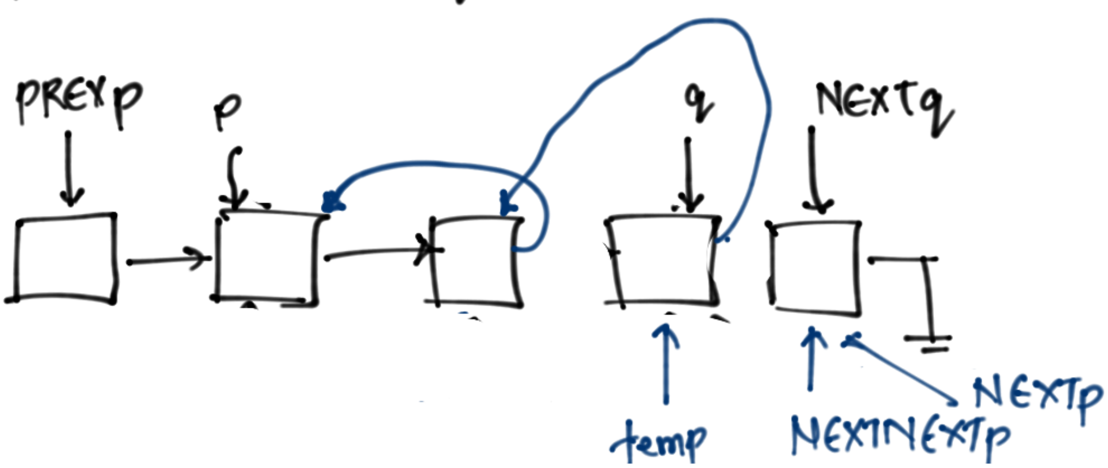


p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

```
  tempp ← p;
  NEXTp ← p.next
→ WHILE (NEXTp IS NOT NEXTq)
  {
      NEXTNEXTp ← NEXTp.next;
      NEXTp.next ← tempp;
      tempp ← NEXTp;
      NEXTp ← NEXTNEXTp;
  }
  PREVp.next ← q;
  p.next ← NEXTq
```

COME OUT OF WHILE LOOP

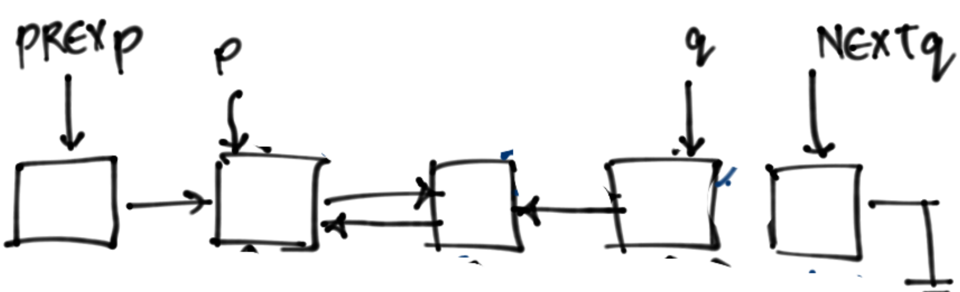


p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

```
tempp ← p;
NEXTp ← p.next
→ WHILE (NEXTp IS NOT NEXTq)
{
    NEXTNEXTp ← NEXTp.next;
    NEXTp.next ← tempp;
    tempp ← NEXTp;
    NEXTp ← NEXTNEXTp;
}
PREVp.next ← q;
p.next ← NEXTq
```

COME OUT OF WHILE LOOP

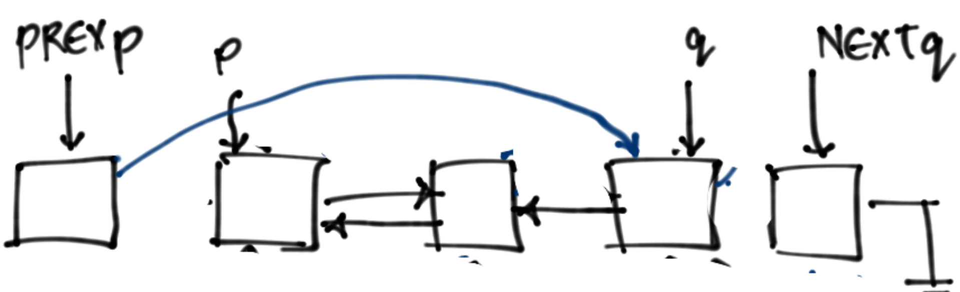


p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

```
   tempp ← p;
   NEXTp ← p.next
→ WHILE (NEXTp IS NOT NEXTq)
   {
      NEXTNEXTp ← NEXTp.next;
      NEXTp.next ← tempp;
      tempp ← NEXTp;
      NEXTp ← NEXTNEXTp;
   }
→  PREVp.next ← q;
   p.next ← NEXTq
```

COME OUT OF WHILE LOOP

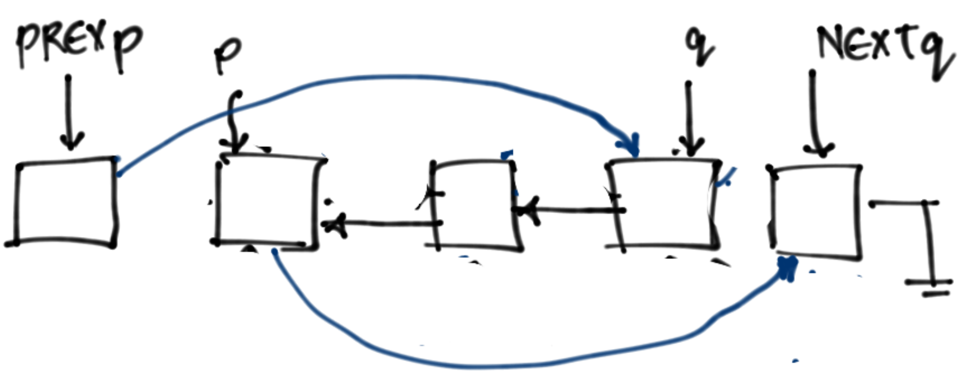


p ← THE a^{th} NODE IN THE LINKED LIST
q ← THE b^{th} NODE IN THE LINKED LIST

PREVP ← THE $(a-1)^{th}$ NODE IN THE LINKED LIST
NEXTq ← THE $(b+1)^{th}$ NODE IN THE LINKED LIST.

```
    tempp ← p;
    NEXTp ← p.next
→ WHILE(NEXTp IS NOT NEXTq)
  {
      NEXTNEXTp ← NEXTp.next;
      NEXTp.next ← tempp;
      tempp ← NEXTp;
      NEXTp ← NEXTNEXTp;
  }
    PREVp.next ← q;
→   p.next ← NEXTq
```

COME OUT OF WHILE LOOP



END