Data Structure

Week 2

Kim, Mucheol

Definition of Data Structure

What is "Data Structure"?

- By Wikipedia
  - In computer science, a "data structure" is a **data organization, management**, and **storage format** that enables efficient access and modification.
  - A data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. 연산의 단순화, 반복적인 작업
  - Example
    - ✓ Array (list) → 동형 data type 저장 → 자주, 연속적으로 호출하는 data끼리 그룹화 ⇒ 호출의 효율성
    - ✓ Linked List 한정적인 메모리 에서의 단점 보완
    - ✓ Record
    - ✓ Union 서로 다른 data type 선택적 사용
    - ✓ Object

How can we implement data structures?

$c = a+b;$ → a, b의 메모리에서 찾아와서
연산한 후 c의 메모리 할당&저장

- By wikipedia
  - Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory
  - The array and record data structures are based on computing the addresses of data items with arithmetic operations
    - ✓ While the linked data structures are based on storing addresses of data items within the structure itself.
  - The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure.

Motivations

Data Structures & Algorithm

Insertion Sort

```
for (i = 1; i < n; i++)
{/* insert a[i] into a[0:i-1] */
   int t = a[i];
   int j;
   for (j = i - 1; j >= 0 && t < a[j]; j--)
      a[j + 1] = a[j];
   a[j + 1] = t;
}
```

∨ 어느정도 sorting 돼있는경우

∨ sorting된 리스트에 하나의 원소를 추가하는 경우

Complexity

- Space/Memory 공간복잡도 (점유하는 메모리의 크기) ⇒ 메모리 값을 하나씩 부르는 것은 비효율적.
  - Amount of memory program occupies
  - Usually measured in bytes, KB, MB, GB

  주어진 메모리 내에서 최대를 사용하는것이 효율적인것 (= 탐색 범위의 확장)
- Time 시간복잡도 (실행시간) ⇒ 데이터의 크기에 비례해서 계산 (단순히 시간을재면 가변성자료는 예측불가)
  - Execution time
  - Usually measured by the number of executions
    - ✓ Count a particular operation
    - ✓ Count number of steps
    - ✓ Asymptotic complexity (e.g., $O(n)$ and $O(n^2)$)

Comparison Count

```
for (i = 1; i < n; i++)
{/* insert a[i] into a[0:i-1] */
  int t = a[i];
  int j;
  for (j = i - 1; j >= 0 && t < a[j]; j--)
      a[j + 1] = a[j];
  a[j + 1] = t;
}
```

Comparison Count

- Pick an instance characteristic … n, n = a.length for insertion sort
- Determine count as a function of this instance characteristic.
- How many comparisons are made?

```
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
```

Comparison Count

```
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
```

- number of compares depends on a[]s and t as well as on i
  - Worst-case count = maximum count
  - Best-case count = minimum count
  - Average count

Worst-Case Comparison Count

```
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
```

- a = [1, 2, 3, 4] and t = 0 => 4 compares
- a = [1,2,3,…,i] and t = 0 => i compares 가변적

Worst-Case Comparison Count

```
for (i = 1; i < n; i++)
  for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
```

- total compares = 1 + 2 + 3 + … + (n-1) = (n-1)n/2

outer loop $i=1 \quad 2 \quad \cdots \quad n-1$

inner loop $1 \quad 2 \quad \cdots \quad n-1$

$\therefore 1+2+\cdots+n-1$

$= \frac{n(n-1)}{2}$

Step Count

- A step is an amount of computing that does not depend on the instance characteristic n
  - The operation-count method omits accounting for the time spent on all but the chosen operation ↳ 최종적으로 비교해야할 것
  - The step-count method count for all the time spent in all parts of the program

- 10 adds, 100 subtracts, 1000 multiplies can all be counted as a single step

- However, n adds cannot be counted as 1 step

Step Count

Statement	steps/execution(s/e)
`for (i = 1; i < n; i++)`	1
`{/* insert a[i] into a[0:i-1] */`	0
`int t = a[i];`	1
`int j;`	0
`for (j = i - 1; j >= 0 && t < a[j]; j--)`	1
`a[j + 1] = a[j];`	1
`a[j + 1] = t;`	1
`}`	0

Step Count

s/e isn't always 0 or 1

x = sum(a, n);

where n is the instance characteristic and sum adds a[0:n-1] has a s/e count of n

Step Count

	s/e	steps
`for (i = 1; i < n; i++)`	1	
`{/* insert a[i] into a[0:i-1] */`	0	
`int t = a[i];`	1	
`int j;`	0	
`for (j = i - 1; j >= 0 && t < a[j]; j--)`	1	i+ 1
`a[j + 1] = a[j];`	1	i
`a[j + 1] = t;`	1	
`}`	0	

Step Count

for (i = 1; i < n; i++)
{ 2i + 3}

step count for
    for (i = 1; i < n; i++)
is n

step count for body of for loop is
2(1+2+3+…+n-1) + 3(n-1)
= (n-1)n + 3(n-1)
= (n-1)(n+3)

Complexity of Insertion Sort

- **$O(n^2)$** ← n = 2n → 이어도 n^2 까지가 유효
- What does this mean?
  - Time or number of operations does not exceed $\mathbf{c.n^2}$ on any input of size **n** (**n** suitably large).
  - Actually, the worst-case time is $\Theta(\mathbf{n^2})$ and the best-case is $\Theta(\mathbf{n})$
  - So, the worst-case time is expected to quadruple each time **n** is doubled

Practical Complexities

10^9 instructions/second

n	n	nlogn	n^2	n^3
1000	1mic	10mic	1milli	1sec
10000	10mic	130mic	100milli	17min
10^9	1milli	20milli	17min	32years

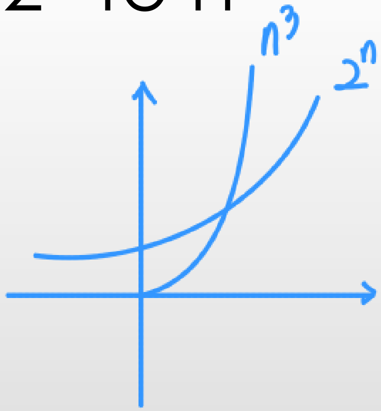
Efficient Algorithm

- Algorithmic improvement more useful than hardware improvement.

$O(n^2) \rightarrow O(n\log n) \rightarrow O(n)$

$\log n \approx c$

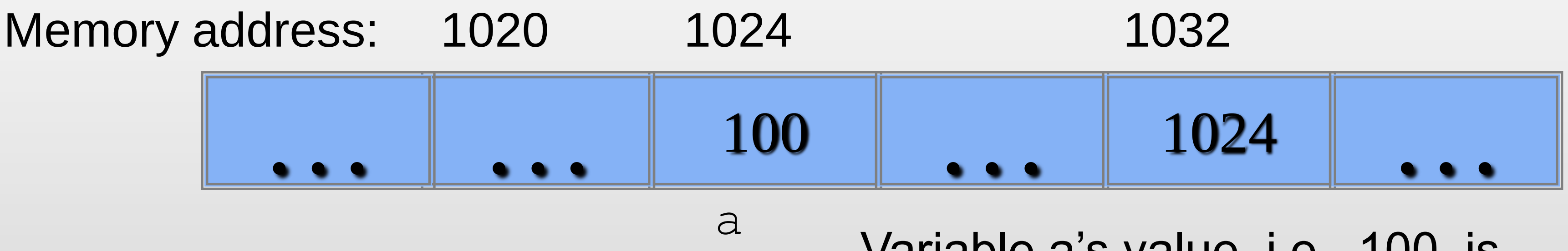
E.g. 2^n to n^3



Review of C Pointers

Pointers and Arrays

- Physical memory consideration
  - Different sizes and access speeds in actual physical memory
  - Optimization could ensure your code makes the best use of physical memory available.
  - OS moves around data in physical memory during execution.

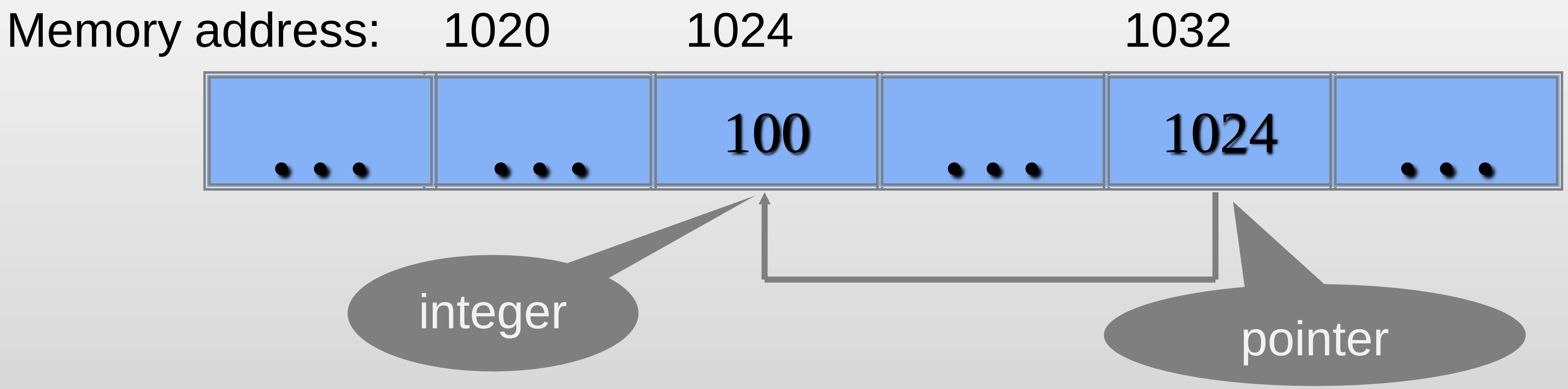



```
int a = 100;
```

Variable a’s value, i.e., 100, is stored at memory location 1024

Pointers and Arrays

- A Pointer is a variable that contains the address of a variable.
  - To express a computation
  - Lead to more compact and efficient code than can be obtained in other ways
  - Extremely useful, especially for data structures



Pointers and Arrays

- Declaration of Pointer variables

  where *type* is the type of data pointed to (e.g. int, char, double)

```
type* pointer_name;
 //or
type *pointer_name;
```

- Examples:

```
int *n;
CProgramming *c;
int **p;    // pointer to pointer
```

Pointers and address

- The "*address of* " operator (&) gives the memory address of the variable
  - **&variable_name**

Memory address: 1020 1024

…	…	100	…	…	…

- Pointers and address

p = &c;

  - Assigns the address of c to the variable p, and p is said to “point to” c.
  - The “&” operator only applies to objects in memory.

Pointers and address

- Example

```
int V = 101;
int *P = &V;          /* P points to int V */
int **Q = &P;         /* Q points to int pointer P */

printf(“%d %d %d\n”,V,*P,**Q); /* prints 101 3 times */
```

Casting pointers

- When assigning a memory address of a variable of one type to a pointer that points to another type it is best to use the cast operator to indicate the cast is intentional.

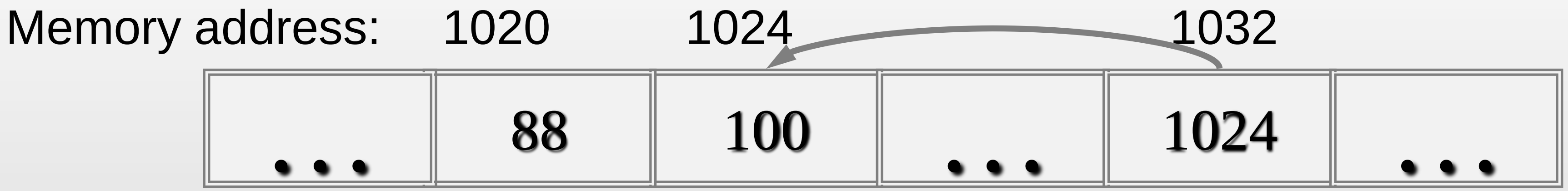
```
int V = 101;
float *P = (float *) &V; /* Casts int address to float * */
```

Removes warning, but is still a somewhat unsafe thing to do

Casting pointers

- Can explicitly cast any pointer type to any other pointer type
  - ppi = (double *)pn; // *pn originally of type (int*)

- Dereferenced pointer has new type, regardless of real type of data



Pointers and Function Arguments

- There is no direct way for the called function to alter a variable in the calling function.

```
void swap(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

  - The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed.
  - Since the operator “&” produces the address of a variable, “&a” is a pointer to a.
- Pointer arguments enable a function to access and change objects in the function that called it.

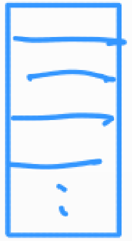
```
void swap(int *px, int *py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

메모리의 구조를 제어하는 것처럼 사용

Arrays and pointers

- Primitive arrays implemented in C using pointer to block of continuous memory

배열



&arr ; 시작주소

- Consider array of 8 ints:
  - int arr[8];
- Accessing arr using array entry operator:
  - int a = arr[8];
- arr is like a pointer to element 0 of the array:
  - int *pa = arr; ⇔ int *pa = & arr[0];
- There is a modifiable/reassignable variable like a pointer.

Arrays and pointers

- int A[5] - A is the address where the array starts (first element), it is equivalent to &(A[0])
  - A is in some sense a pointer to an integer variable
- To determine the address of A[x] use formula:
  - (address of A + x * bytes to represent int)
  - (address of array + element num * bytes for element size)
- The + operator when applied to a pointer value uses the formula above:
  - A + x is equivalent to &(A[x])
  - *(A + x) is equivalent to A[x]
- Address value increments by i times size of data type
  - Suppose arr[0] has address 100. Then arr[3] has address 112.
  - Why?

Character Pointers and Functions

- Example
  - char amessage[] = “now is the time”;
  - char *pmessage = “now is the time”;
  - What is difference between them?
- amessage is an array, just big enough to hold the sequence of characters and ‘\0’.
- pmessage is a pointer, initialized to pointer to a string constant; the pointer may subsequently be modified to point.

Character Pointers and Functions

- 1st Comparisons
  - A :

```
void strcpy(char *s, char *t)
{
  int i;
  i = 0;
  while ({s[i] = t[i]) != '\0')
     i++;
}
```

  - B :

```
void strcpy(char *s, char *t)
{
   while ((*s = *t) != '\0')
   {
      s++;
      t++;
   }
}
```

- What is difference between them?

Character Pointers and Functions

- 2nd Comparison
  - strcmp 1 :

```
int strcmp(char *s, char *t)
{
  int i;
  for (i = 0 ; s[i] == t[i] ; i++)
    if (s[i] == '\0')
      return 0;
    return s[i] - t[i];
}
```

  - strcmp 2:

```
int strcmp(char *s, char *t)
{
  for ( ; *s == *t ; s++, t++)
    if (*s == '\0' )
      return 0;
  return *s - *t;
}
```

Reference Variable

- A reference is an additional name to an existing memory location
- A reference variable serves as an alternative name for an object.
- A reference variable always refers to the same object.
  - Assigning a reference variable with a new value actually changes the value of the referred object.
- Reference variables are commonly used for parameter passing to a function

Limits of Static Allocation 정적할당의 한계

- What if we don't know how much space we will need ahead of time?

char 선언할 때 가변적으로 사용하려면? 가장 큰 배열 선언 → 비효율적

- Example:
  - ask user how many numbers to read in
  - read set of numbers in to array (of appropriate size)
  - calculate the average (look at all numbers)
  - calculate the variance (based on the average)

- Problem: how big do we make the array??
  - using static allocation, have to make the array as big as the user might specify (might not be big enough)

Dynamic Memory Allocation

- Allow the program to allocate some variables (notably arrays), during the program, based on variables in program (dynamically)

- Previous example: ask the user how many numbers to read, then allocate array of appropriate size

- Idea: user has routines to request some amount of memory, the user then uses this memory, and returns it when they are done
  - memory allocated in the Data Heap

Memory Management Function

- calloc - routine used to allocate arrays of memory
- 동적 malloc - routine used to allocate a single block of memory
- realloc - routine used to extend the amount of space allocated previously
- free - routine used to tell program a piece of memory no longer needed
    - note: memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Array Allocation with calloc

prototype: void * calloc(size_t num, size_t esize)

- size_t is a special type used to indicate sizes, generally an unsigned int
- num is the number of elements to be allocated in the array
- esize is the size of the elements to be allocated generally use sizeof and type to get correct value
- an amount of memory of size num*esize allocated on heap
- calloc returns the address of the first byte of this memory generally we cast the result to the appropriate type
- if not enough memory is available, calloc returns NULL

Releasing Memory(free)

어디선가는 memory release 해야함.

while 문에서 메모리 할당하면?

⇒ 개에바...

prototype: void free(void *ptr)

- memory at location pointed to by ptr is released (so we could use it again in the future)
- program keeps track of each piece of memory allocated by where that memory starts
- if we free a piece of memory allocated with calloc, the entire array is freed (released)
- results are problematic if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

Releasing Memory(free)

- When function problem called, space for array of size N allocated, when function ends, variable nums goes away, but the space nums points at (the array of size N) does not (allocated on the heap) - furthermore, we have no way to figure out where it is)
- Problem called memory leakage

Memory allocation with malloc

prototype: void * malloc(size_t esize)

- similar to calloc, except we use it to allocate a single block of the given size esize
  - as with calloc, memory is allocated from heap
- NULL returned if not enough memory available
- memory must be released using free once the user is done
- It can perform the same function as calloc if we simply multiply the two arguments of calloc together
  - malloc(N * sizeof(float)) is equivalent to
  - calloc(N, sizeof(float))

Realloc

prototype: void * realloc(void * ptr, size_t esize)

- ptr is a pointer to a piece of memory previously dynamically allocated
- esize is new size to allocate (no effect if esize is smaller than the size of the memory block ptr points to already)
- program allocates memory of size esize,
    - then it copies the contents of the memory at ptr to the first part of the new piece of memory,
- finally, the old piece of memory is freed up

Data Structure

Week 3

Kim, Mucheol

Sparse Matrices

Matrix & Sparse Matrix

- Sparse Matrix
  - A sparse matrix or sparse array is a matrix in which most of the elements are zero
  - Common Criteria
    - ✓ The number of non-zero elements is roughly equal to the number of rows or columns
- Why Sparse Matrix?
  - Because its exploitation can lead to enormous computational savings and because many large matrix problems that occur in practice are sparse.
  - Ex)
    The example has 13 zero values of the 18 elements in the matrix, giving this matrix a sparsity score of 0.722 or about 72%.

$$\left\{ \begin{matrix} 1,0,0,1,0,0 \\ 0,0,2,0,0,1 \\ 0,0,0,2,0,0 \end{matrix} \right\}$$

Sparse_Matrix Structure

- Objects
  - A set of triples, <row, column, value>, where row and column are integers and form a unique combination, and value comes from the set item
- Functions
  - Sparse_Matrix Create(max_row, max_col) :
  - Sparse_Matrix Transpose(a)
    - ✓ return the matrix produced by interchanging the row and column value of every triple.
  - Sparse_Matrix Add(a,b)
    - ✓ if the dimensions of a and b are the same return the matrix produced by adding corresponding items, namely those with identical row and column values.
  - Sparse_Matrix Multiply(a,b)
    - ✓ If number of columns in a equals number of rows in b return the matrix d produced by multiplying a by b
      - ➢ $d[i][j] = \sum(a[i][k]*b[k][j])$ where $d(i,j)$ is the (i,j)th element.

Sparse Matrices

- Matrix
  ➔ table of values

$$\begin{array}{cl} \begin{matrix}0&0&3&0&4\\0&0&5&7&0\\0&0&0&0&0\\0&2&6&0&0\end{matrix} & \begin{matrix}\\ \text{Row 2}\\ \\ \\ \end{matrix} \\ \text{Column 4} & \end{array}$$

4 x 5 matrix

4 rows

5 columns

20 elements

6 nonzero elements

Sparse Matrices

- Sparse matrix $O(n^2) \rightarrow O(kn)$
  ➔ (#nonzero elements)/(#elements) is small.
- Examples:
  - Diagonal
    - ✓ Only elements along diagonal may be nonzero
    - ✓ n x n matrix → ratio is $n/n^2 = 1/n$
  - Tridiagonal
    - ✓ Only elements on 3 central diagonals may be nonzero
    - ✓ Ratio is $(3n-2)/n^2 = 3/n - 2/n^2$
  - Lower triangular (?)
    - ✓ Only elements on or below diagonal may be nonzero
    - ✓ Ratio is $n(n+1)(2n^2) \sim 0.5$
- These are structured sparse matrices.
  - Nonzero elements are in a well-defined portion of the matrix.

Sparse Matrices

- An n x n matrix may be stored as an n x n array.
  ➔ $O(n^2)$ space.
  - The example structured sparse matrices may be mapped into a 1D array so that a mapping function can be used to locate an element quickly;
  - The space required by the 1D array is less than that required by an n x n array.

Unstructured Sparse Matrices

- Airline flight matrix.
  - airports are numbered 1 through n
  - flight(i,j) = list of nonstop flights from airport i to airport j
  - n = 1000 (say)
  - n x n array of list pointers => 4 million bytes
  - total number of nonempty flight lists = 20,000 (say)
  - need at most 20,000 list pointers => at most 80,000 bytes

Unstructured Sparse Matrices

- Web page matrix.
  - web pages are numbered 1 through n
  - web(i,j) = number of links from page i to page j

- Web analysis.
  - authority page … page that has many links to it
  - hub page … links to many authority pages

~3/14

Web Page Matrix

- n = 2 billion (and growing by 1 million a day)
- n x n array of ints => 16 * 10^{18} bytes (16 * 10^{9} GB) = 16 exa bytes
- each page links to 10 (say) other pages on average
- on average there are 10 nonzero entries per row
- space needed for nonzero elements is approximately 20 billion x 4 bytes = 80 billion bytes (80 GB)

Representation Of Unstructured Sparse Matrices

- Single linear list in row-major order.
  - scan the nonzero elements of the sparse matrix in row-major order (i.e., scan the rows left to right beginning with row 1 and picking up the nonzero elements)
  - each nonzero element is represented by a triple (row, column, value)
  - the list of triples is stored in a 1D array

Single Linear List Example

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

list =

$$
\begin{array}{l}
\text{row} \\
\text{column} \\
\text{value}
\end{array}
\begin{bmatrix}
1 & 1 & 2 & 2 & 4 & 4 \\
3 & 5 & 3 & 4 & 2 & 3 \\
3 & 4 & 5 & 7 & 2 & 6
\end{bmatrix}
$$

One Linear List Per Row

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

row1 = [(3, 3), (5,4)]
row2 = [(3,5), (4,7)]
row3 = []
row4 = [(2,2), (3,6)]

Single Linear List

- Array representation
  - 1D Array of triples of type term
    - ✓ int row, col, value
- Size of 1D array generally not predictable at time of initialization.
  - Start with some default capacity/size (say 10)
  - Increase capacity as needed
  - Use REALLOC

Approximate Memory Requirements

- 500 x 500 matrix with 1994 nonzero elements, 4 bytes per element

2D array 500 x 500 x 4 = 1million bytes
1D array of triples 3 x 1994 x 4
= 23,928 bytes

Matrix Transpose

$$\begin{array}{ccccc} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{array} \longrightarrow \begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 3 & 5 & 0 & 6 \\ 0 & 7 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{array}$$

sparse의 transpose

Matrix Transpose

$$\begin{matrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{matrix} \longrightarrow \begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 3 & 5 & 0 & 6 \\ 0 & 7 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{matrix}$$

$$\begin{matrix} \text{row} & 1 & 1 & 2 & 2 & 4 & 4 \\ \text{column} & 3 & 5 & 3 & 4 & 2 & 3 \\ \text{value} & 3 & 4 & 5 & 7 & 2 & 6 \end{matrix} \longrightarrow \begin{matrix} 2 & 3 & 3 & 3 & 4 & 5 \\ 4 & 1 & 2 & 4 & 2 & 1 \\ 2 & 3 & 5 & 6 & 7 & 4 \end{matrix}$$

Matrix Transpose fast transpose

① 현재 열의 nonzero수 세기

$$\begin{matrix} 0&0&3&0&4 \\ 0&0&5&7&0 \\ 0&0&0&0&0 \\ 0&2&6&0&0 \end{matrix} \longrightarrow \begin{matrix} 0&0&0&0 \\ 0&0&0&2 \\ 3&5&0&6 \\ 0&7&0&0 \\ 4&0&0&0 \end{matrix}$$

row	1	1	2	2	4	4
column	3	5	3	4	2	3
value	3	4	5	7	2	6

Step 1: #nonzero in each row of transpose.

= #nonzero in each column of original matrix

= [0, 1, 3, 1, 1]

②

Step2: Start of each row of transpose

= sum of size of preceding rows of transpose

= [0, 0, 1, 4, 5]

Step 3: Move elements, left to right, from original list to transpose list.

See in program 2.8

Matrix Transpose

Step 1: #nonzero in each row of transpose.

= #nonzero in each column of original matrix

= [0, 1, 3, 1, 1]

Step2: Start of each row of transpose

= sum of size of preceding rows of transpose

= [0, 0, 1, 4, 5]

Step 3: Move elements, left to right, from original list to transpose list.

Complexity

m x n original matrix

t nonzero elements

Step 1: O(n+t)

Step 2: O(n)

Step 3: O(t)

2n+2t

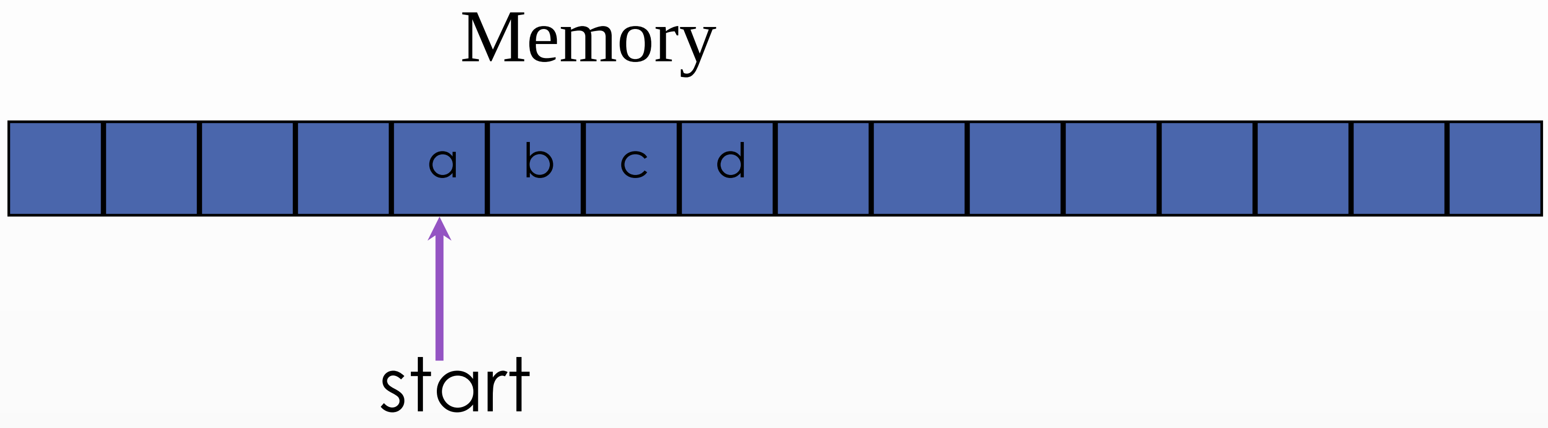
Overall O(n+t)

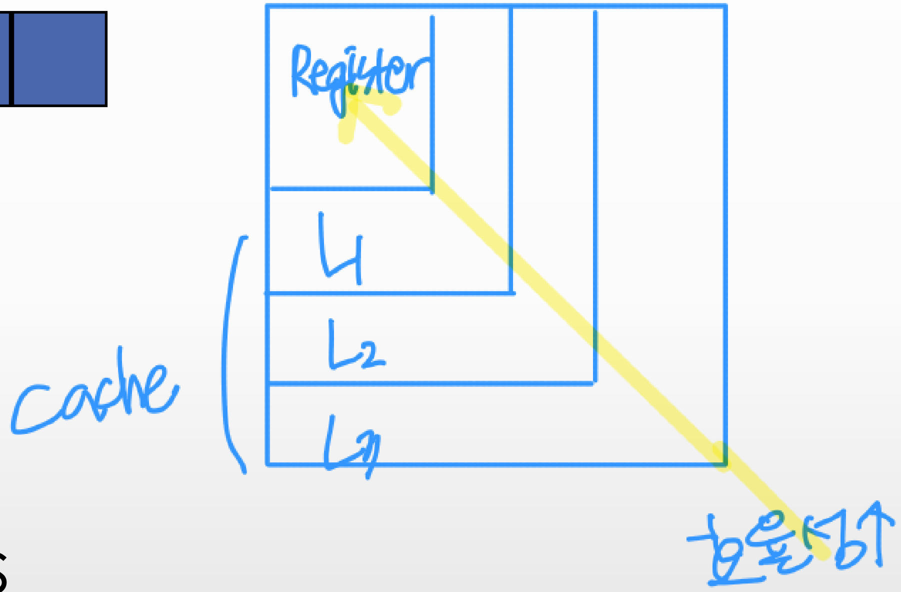
Matrix Multiplication

- Easy. Do it yourself. (see in 2.4.3 section)

Arrays

1D Array Representation In C

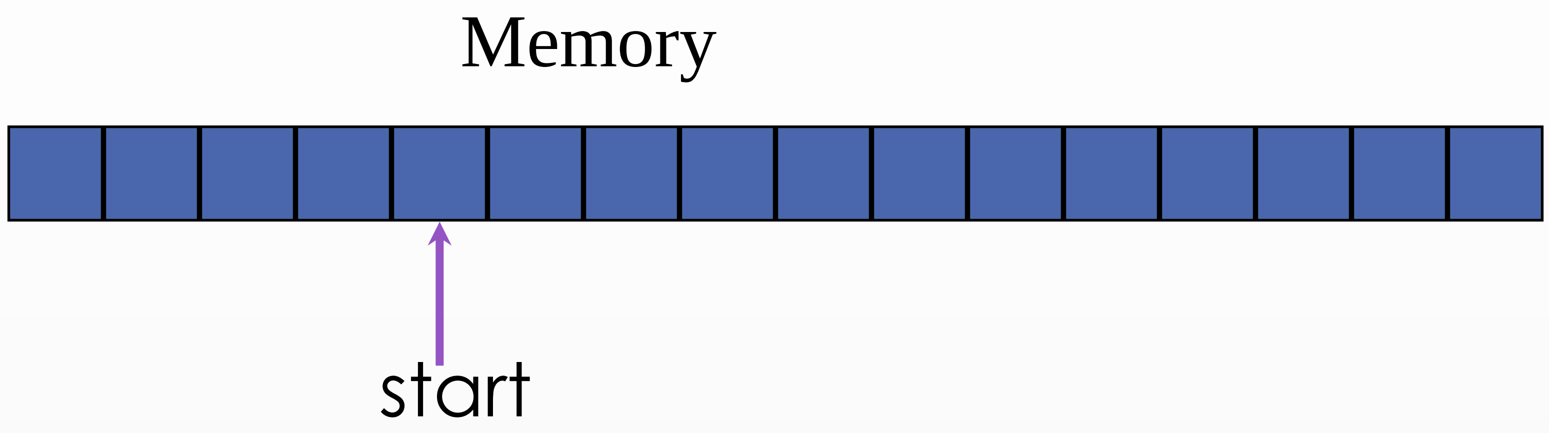




- 1-dimensional array x = [a, b, c, d]
- map into contiguous memory locations

location(x[i]) = start + i

Space Overhead



- space overhead = 4 bytes for start

(excludes space needed for the elements of x)

2D Arrays

정적메모리할당

- The elements of a 2-dimensional array a declared as:

```
int arr[3][4];
```

or

```
int *arr = (int*)malloc(sizeof(int) * 4);
```

- may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

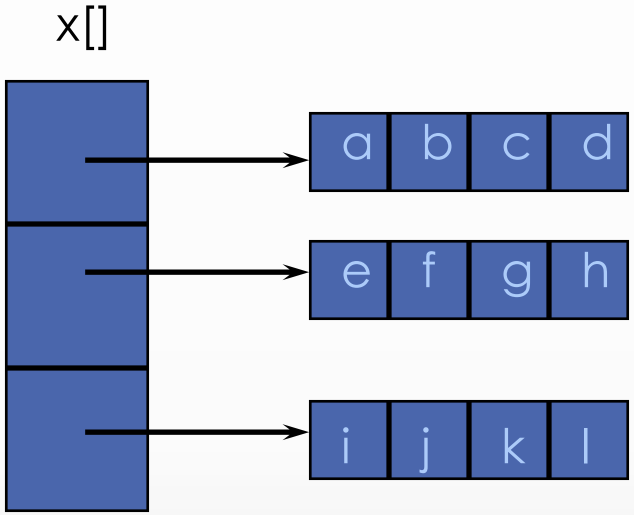
Rows Of A 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]	row 0
a[1][0]	a[1][1]	a[1][2]	a[1][3]	row 1
a[2][0]	a[2][1]	a[2][2]	a[2][3]	row 2

Columns Of A 2D Array

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
column 0	column 1	column 2	column 3

2D Array Representation In C



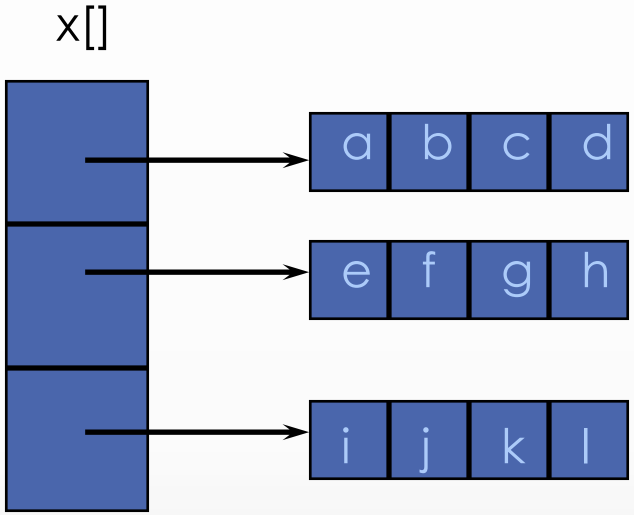
space overhead = space required by the array x[]

= 3 * 4 bytes

= 12 bytes

= number of rows x 4 bytes

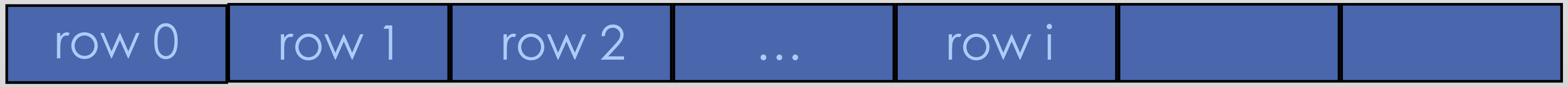
2D Array Representation In C



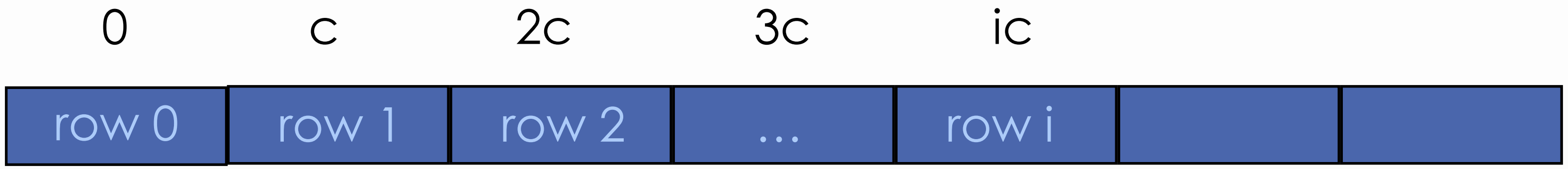
- This representation is called the array-of-arrays representation.
- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- 1 memory block of size number of rows and number of rows blocks of size number of columns

Row-Major Mapping

- Example 3 x 4 array:
  a b c d
  e f g h
  i j k l
- Convert into 1D array y by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get y[] = {a, b, c, d, e, f, g, h, i, j, k, l}

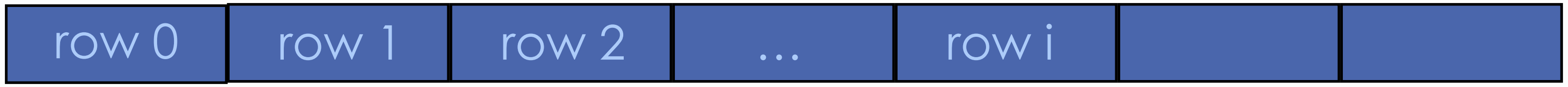


Locating Element x[i][j]



- assume **x** has **r** rows and **c** columns
- each row has **c** elements
- **i** rows to the left of row **i**
- so **ic** elements to the left of **x[i][0]**
- so **x[i][j]** is mapped to position **ic + j** of the 1D array

Space Overhead



4 bytes for start of 1D array +

4 bytes for c (number of columns)

= 8 bytes

Disadvantage

- Need contiguous memory of size **rc**.

Column-Major Mapping

a b c d
e f g h
i j k l

- Convert into 1D array y by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- We get y = {a, e, i, b, f, j, c, g, k, d, h, l}

Arrays in Programming Languages

- Programming languages or their standard libraries that support multi-dimensional arrays typically have a native row-major or column-major storage order for these arrays.
  - Row-major order is used in C/C++/Objective-C (for C-style arrays), PL/I, Pascal
  - Column-major order is used in Fortran, MATLAB, R 통계학적인언어 → column-major
- A typical alternative for dense array storage is to use Iliffe vectors, which typically store pointers to elements in the same row contiguously (like row-major order), but not the rows themselves.
  - Java,[13] C#/.Net, Scala, and Swift.
- less dense is to use lists of lists, e.g., in Python

✓ 왜 row-major 쓰는가? 메모리구조 → 연산의 효율성위해

Matrix

- Table of values. Has rows and columns, but numbering begins at 1 rather than 0.

a b c d	row 1
e f g h	row 2
i j k l	row 3

- Use notation x(i,j) rather than x[i][j].
- May use a 2D array to represent a matrix.

Shortcomings Of Using A 2D Array

- Indexes are off by 1.
- C arrays do not support matrix operations such as **add**, **transpose**, **multiply**, and so on.
  - Suppose that x and y are 2D arrays.
    Can't do x + y, x –y, x * y, etc. in C.

Diagonal Matrix 대각행렬

- An n x n matrix in which all nonzero terms are on the diagonal.

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{matrix}$$

- **x(i,j)** is on diagonal iff **i = j**
- number of diagonal elements in an **n x n** matrix is **n**
- non diagonal elements are zero
- store diagonal only vs $\mathbf{n^2}$ whole

sparse matrix

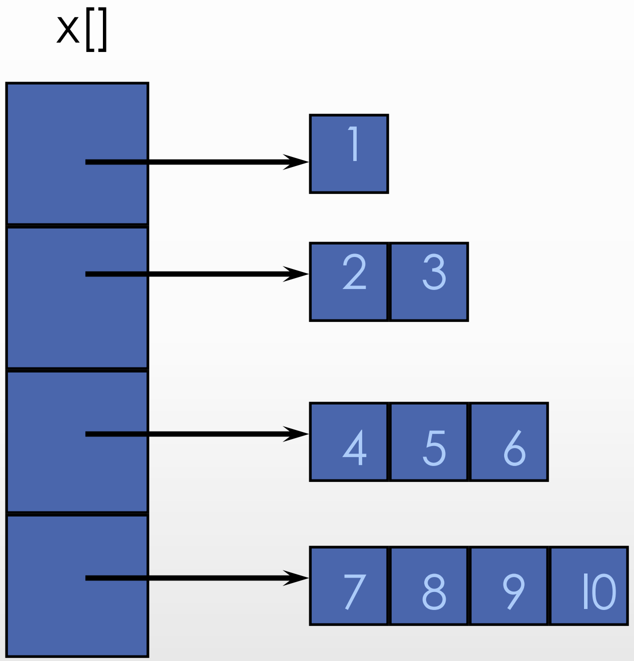
Lower Triangular Matrix

- An n x n matrix in which all nonzero terms are either on or below the diagonal.

$$
\begin{matrix}
1 & 0 & 0 & 0 \\
2 & 3 & 0 & 0 \\
4 & 5 & 6 & 0 \\
7 & 8 & 9 & 10
\end{matrix}
$$

- **x(i,j)** is part of lower triangle iff **i >= j**.
- number of elements in lower triangle is **1 + 2 + … + n = n(n+1)/2**. 원소 $\frac{n(n+1)}{2}$ 개
- store only the lower triangle

Array Of Arrays Representation



Use an irregular 2-D array … length of rows is not required to be the same.

Map Lower Triangular Array Into A 1D Array

- Use row-major order, but omit terms that are not part of the lower triangle.

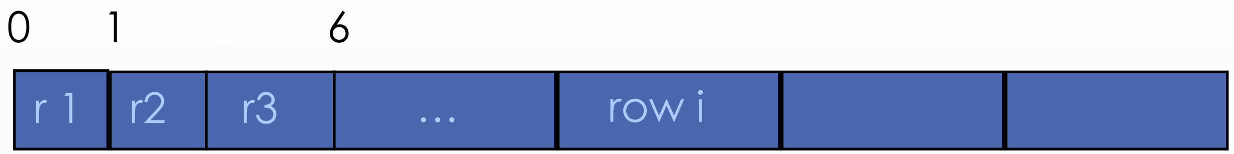
- For the matrix

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{matrix}$$

we get

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Index Of Element [i] [j]



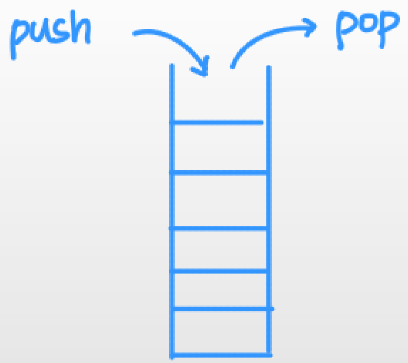
- Order is: row 1, row 2, row 3, …
- Row **i** is preceded by rows **1**, **2**, **…**, **i-1**
- Size of row **i** is **i**.
- Number of elements that precede row **i** is
  1 + 2 + 3 + … + i-1 = i(i-1)/2
- So element **(i,j)** is at position **i(i-1)/2 + j -1** of the 1D array.

Stacks

Stacks

선형자료의의미
→ 1차원으로 들어온자료를 처리

- Definitions - an abstract data type that serves as a collection of elements, with two principal operations:
  - new 원소 in — push, which adds an element to the collection, and
  - last 원소 out — pop, which removes the most recently added element that was not yet removed.



- Characteristics
  - Linear list.
  - LIFO(Last In First Out)
  - One end is called top.
  - Other end is called bottom. index 역할
  - Additions to and removals from the top end only.

System Stack

```
main()
{
  int a=p();
  return 0;
}

int p()
{
  return q();
}

int q()
{
  return(1+2);
}
```

- It is used by a program at run-time to process function calls.
  - Whenever a function is invoked, the program creates a structure, referred to as an activation record or a stack frame, and places it on top of the system stack. stack을 왜 쓰는가? 외부요소 영향X → 무결성 보장
- Procedure
  - The activation record for the invoked function contains only a pointer to the previous stack frame and a return address.
  - The previous stack frame pointer points to the stack frame of the invoking function, while the return address contains the location of the statement to be executed after function terminates.
  - If this function invokes another function, the local variables and the parameters of the invoking function are added to its stack frame.
  - If this function terminates, stack frame is removed and the processing of the invoking function continues.

Stacks as an Abstract data type

- Object: Stack
  - An order list in which insertions and deletions are made at only one end.

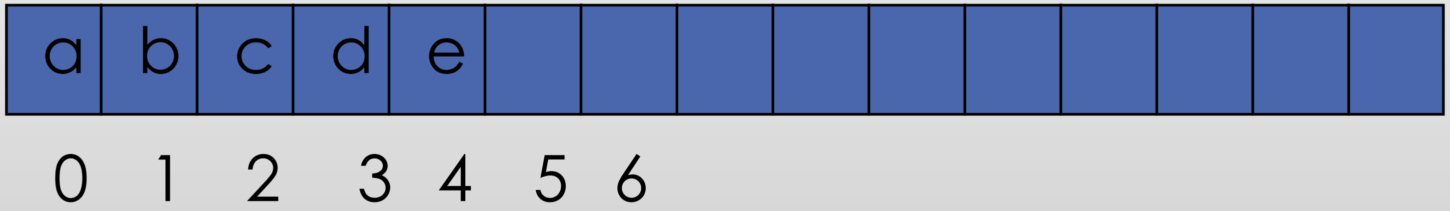
- Standard operations:
  - IsEmpty - return true iff stack is empty
  - IsFull - return true iff stack has no remaining capacity
  - Top - return top element of stack
  - Push - add an element to the top of the stack
  - Pop - delete the top element of the stack

Stacks

- Use a 1D array to represent a stack.
- Stack elements are stored in stack[0] through stack[top].

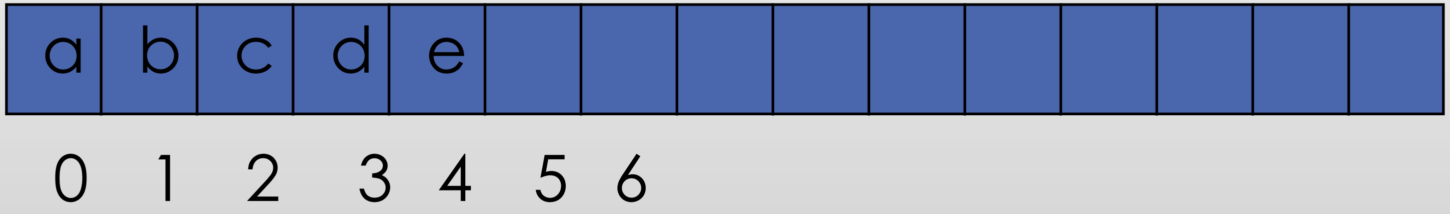
Stacks

- stack top is at element e
- IsEmpty() => check whether top >= 0
  - O(1) time
- IsFull() => check whether top == capacity – 1
  - O(1) time
- Top() => If not empty return stack[top]
  - O(1) time

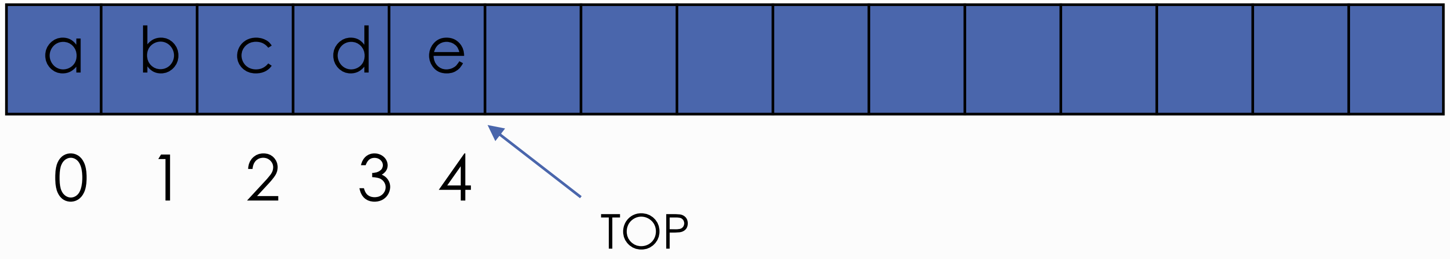


Derive From arrayList

- Push(theElement) => if full then either error or increase capacity and then add at stack[top+1]
- Suppose we increase capacity when full
- O(capacity) time when full; otherwise O(1)
- Pop() => if not empty, delete from stack[top]
- O(1) time

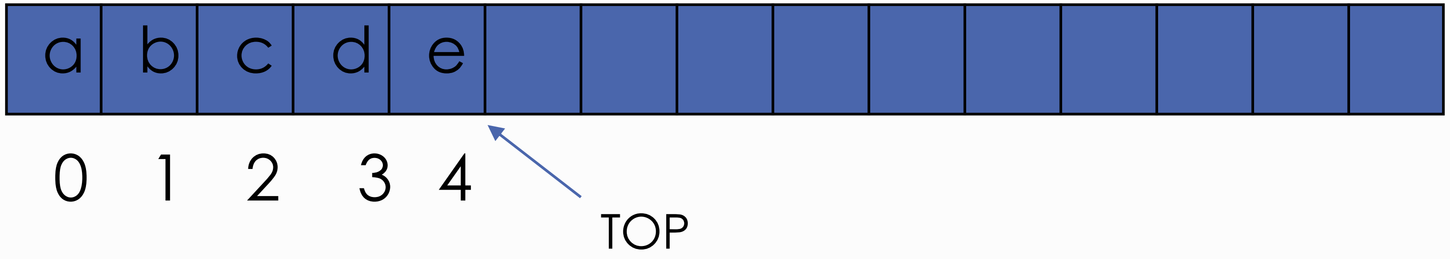


Push




```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE - 1)
       StackFull();
   /* add at stack top */
   stack[++top] = item;
}
```

Pop




```
element pop()
{
   if (top == -1)
        return StackEmpty();
   stack[top--];
}
```

StackFull()

```
StackFull()
{
    fprintf(stderr, “Stack is full, cannot add
                         element.”);
    exit(EXIT_FAILURE);
}
```

StackFull()/Dynamic Array

- Use a variable called capacity in place of MAX_STACK_SIZE
- Initialize this variable to (say) 1
- When stack is full, double the capacity using REALLOC
- This is called array doubling

re-allocation

StackFull()/Dynamic Array

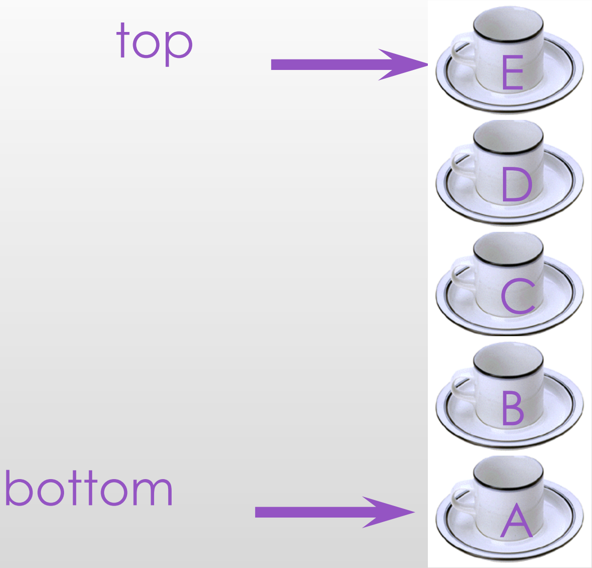
```
StackFull()
{
    REALLOC(stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
}
```

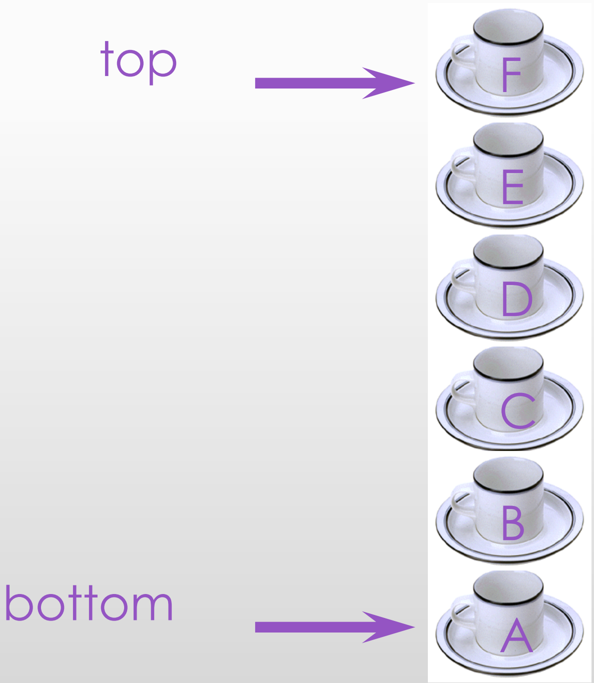
Complexity Of Array Doubling

- Let final value of capacity be 2^k
- Number of pushes is at least $2^{k-1} + 1$
- Total time spent on array doubling is $\Sigma_{1<=i=k} 2^i$
- This is $O(2^k)$
- So, although the time for an individual push is O(capacity), the time for all n pushes remains O(n)!

Stack Of Cups

- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO list.





Parentheses Matching

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - ✓ (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

Parentheses Matching

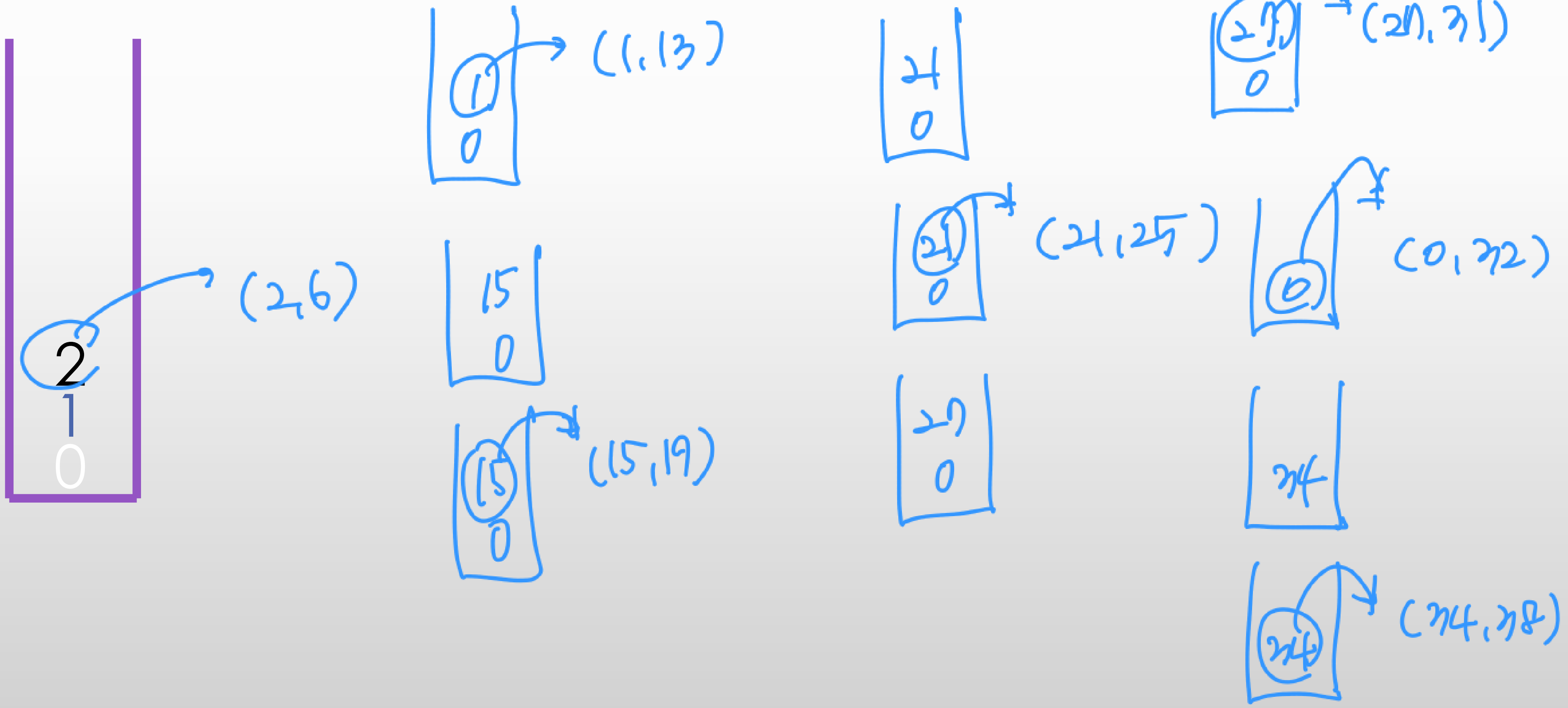
$O(n)$

★ 선형시간에 서칭해서 유효성 검증

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

Example

(((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



Example

(((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13)

Example

(((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13) (15,19)

Example

(((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13) (15,19) (21,25)

Example

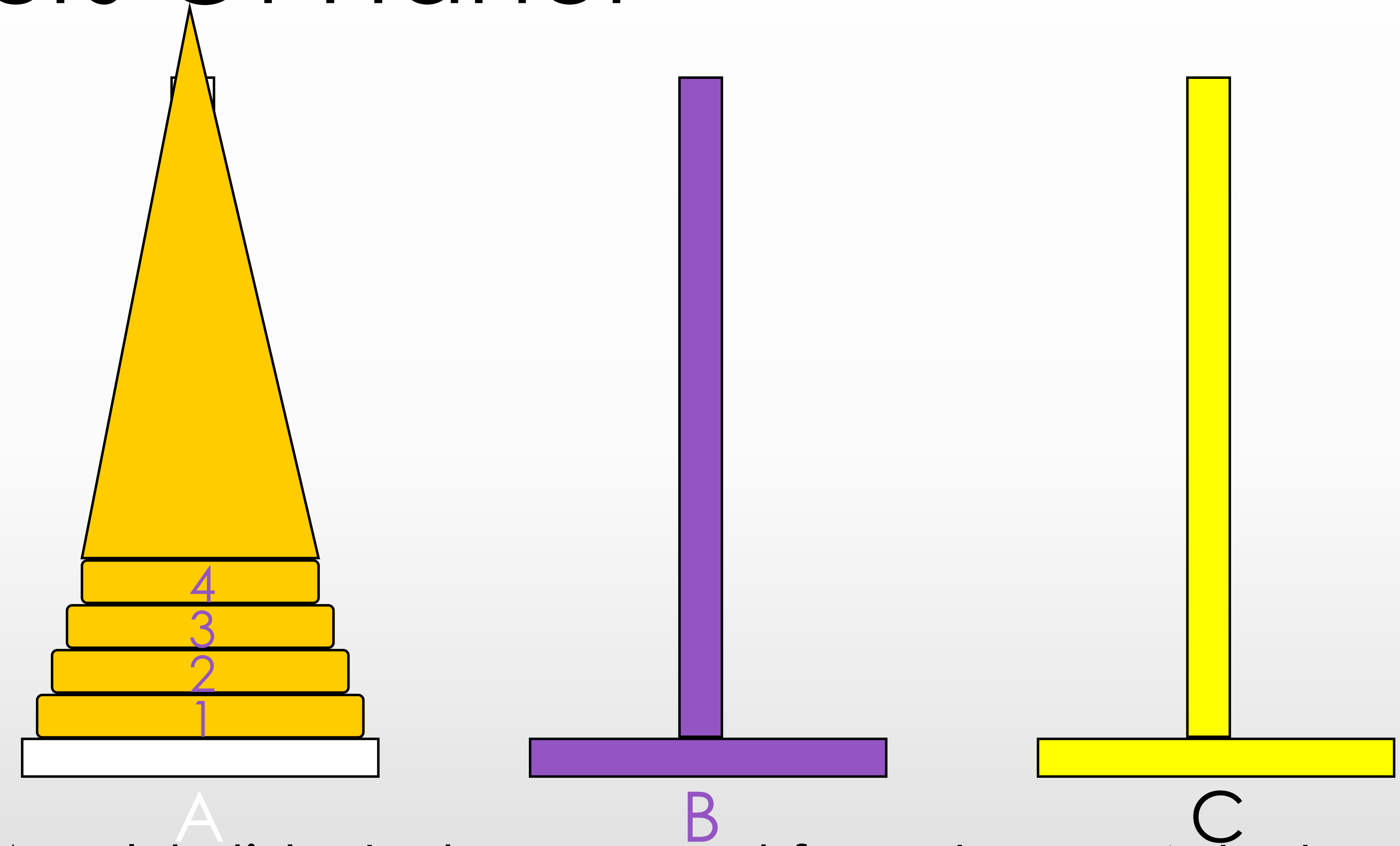
(((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13) (15,19) (21,25)(27,31) (0,32)

and so on

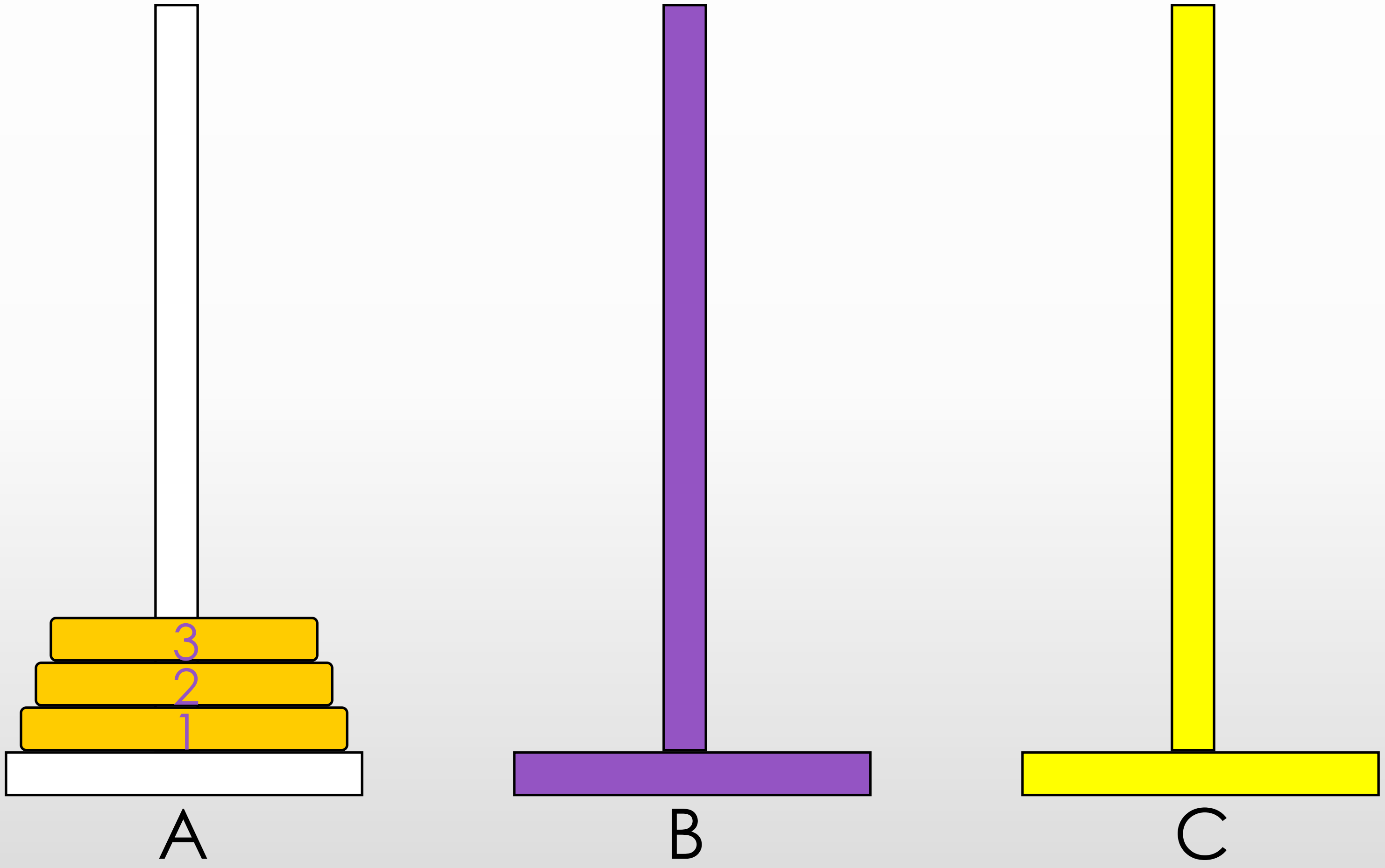
Towers Of Hanoi

recursion으로 구현 코드



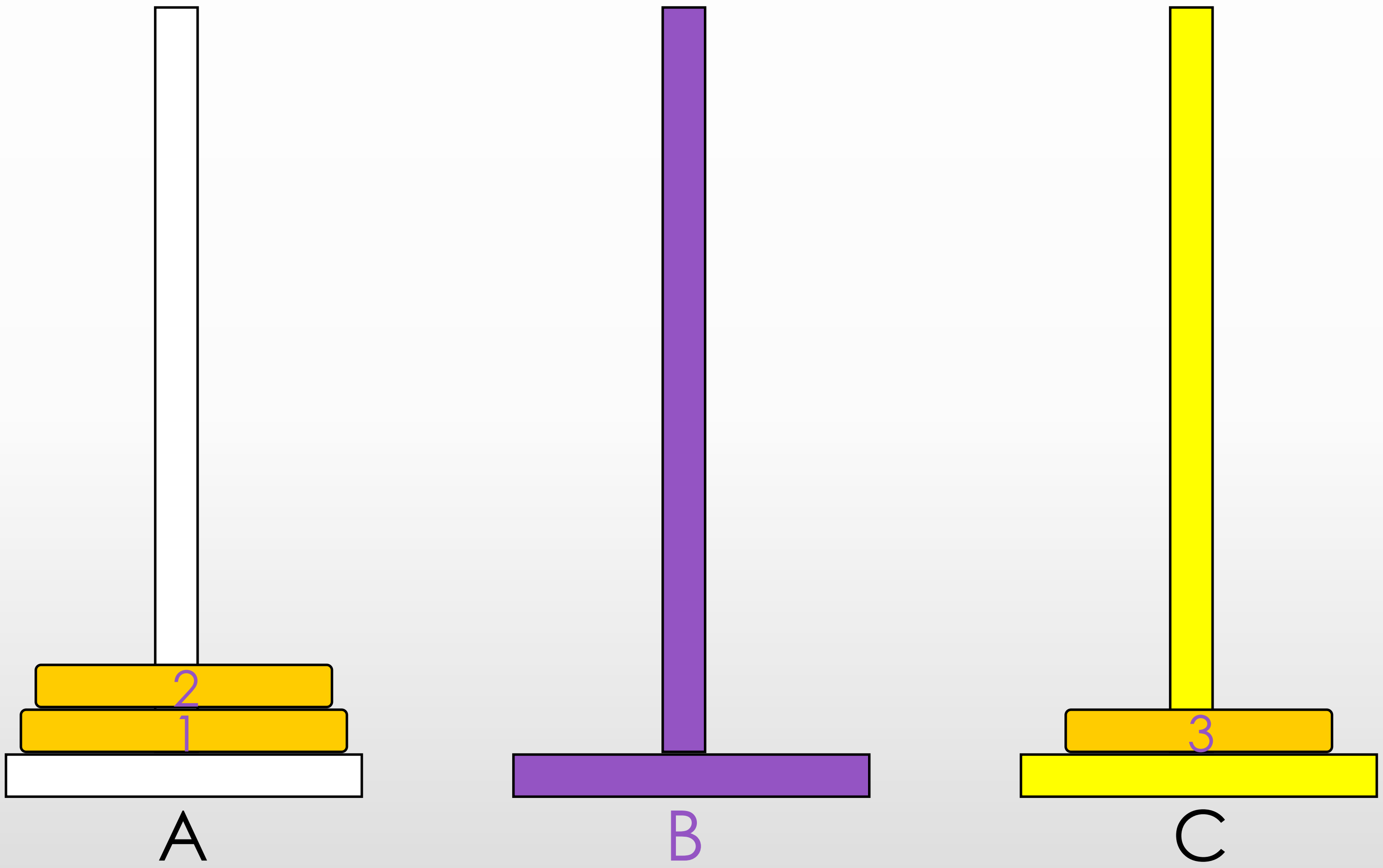
- 64 gold disks to be moved from tower A to tower C
- each tower operates as a stack
- cannot place big disk on top of a smaller one

Towers Of Hanoi



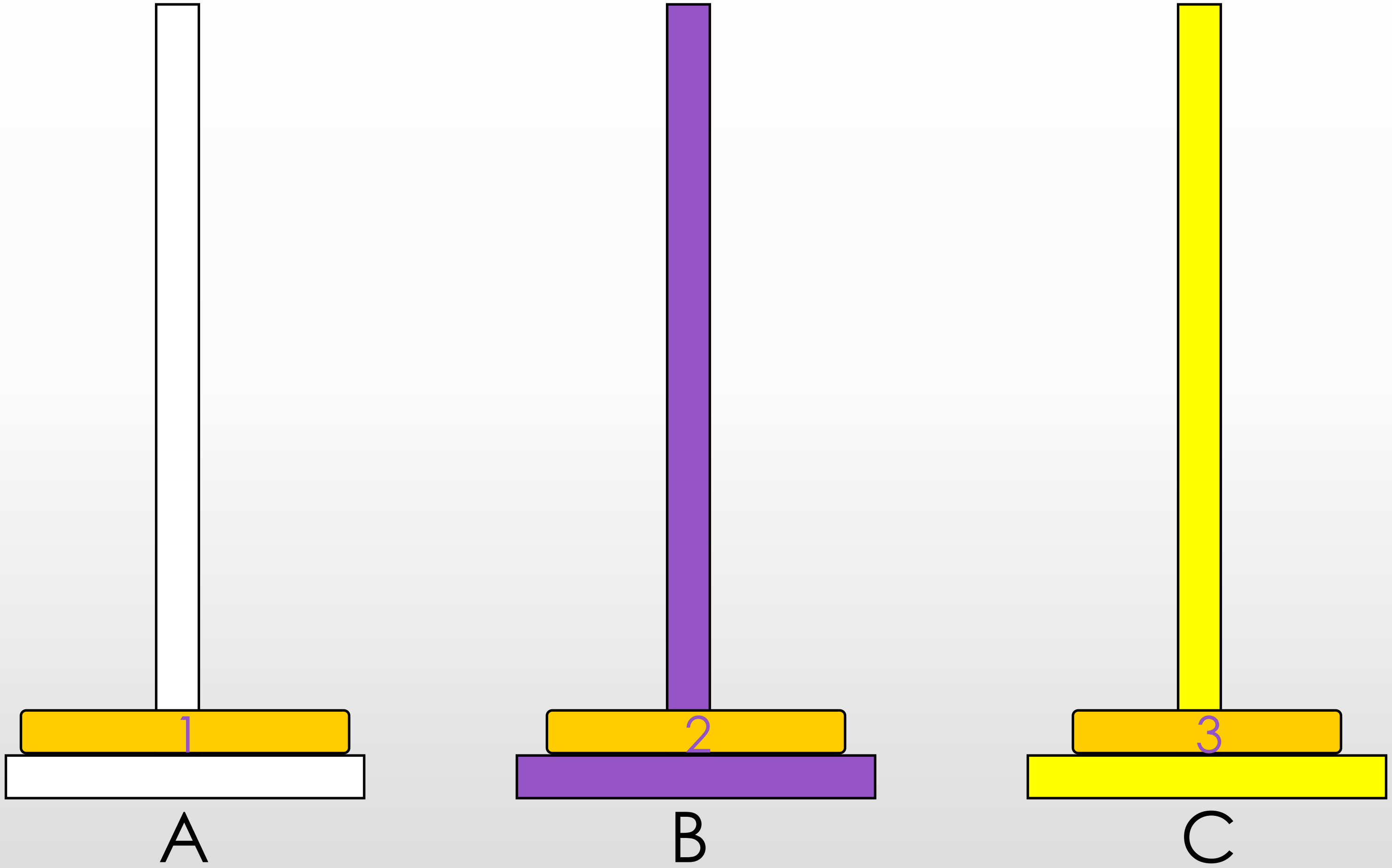
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi



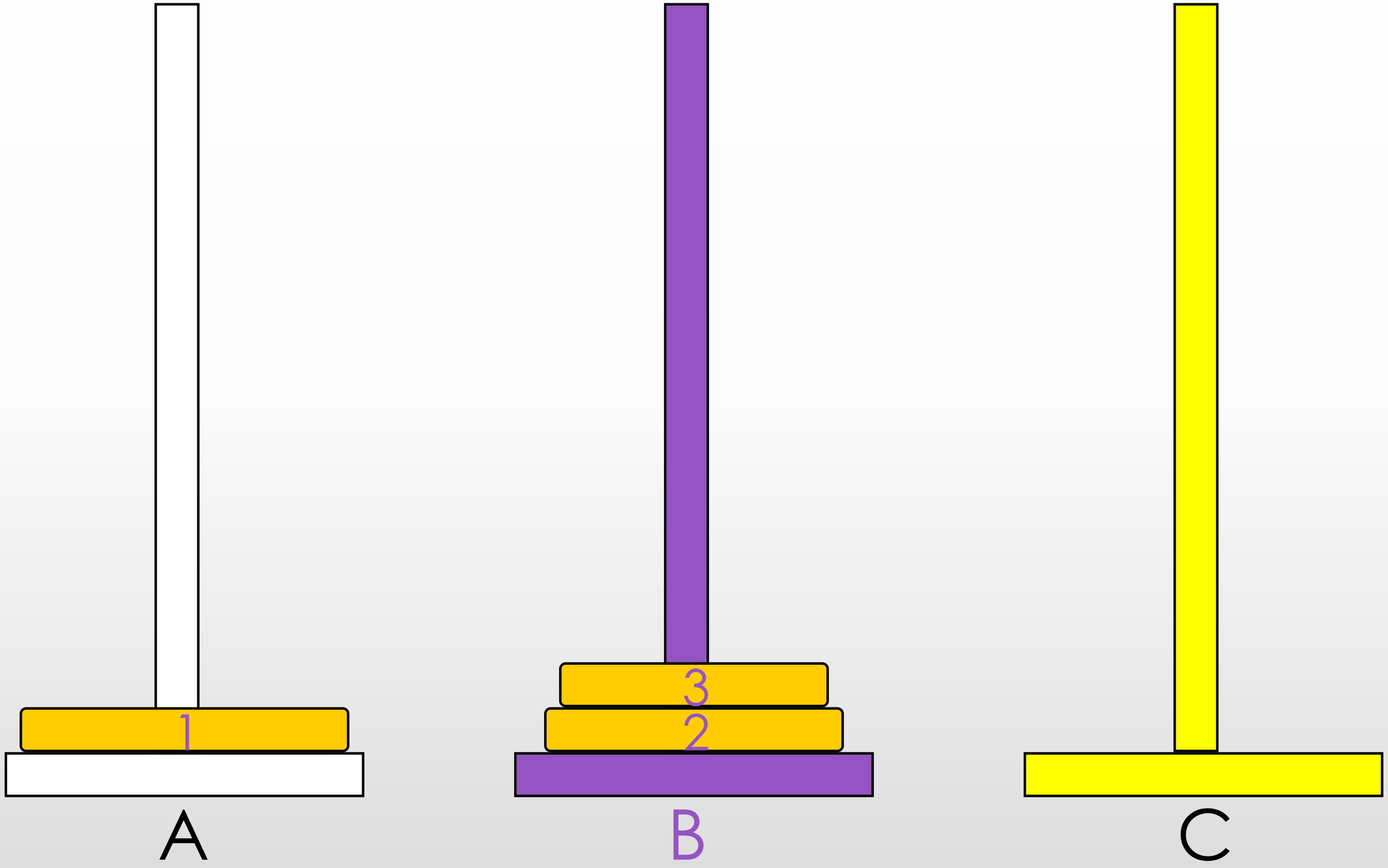
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi



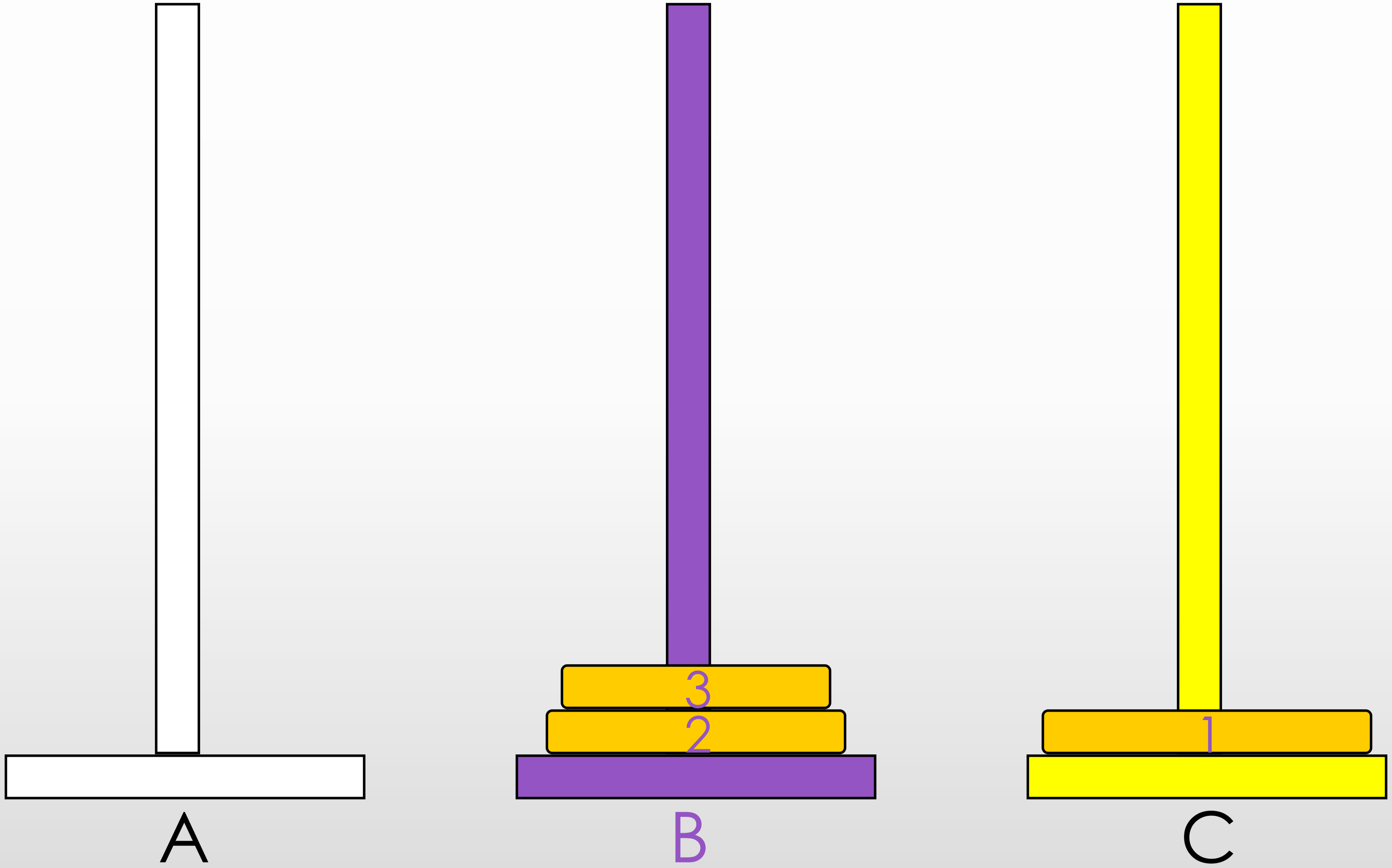
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi



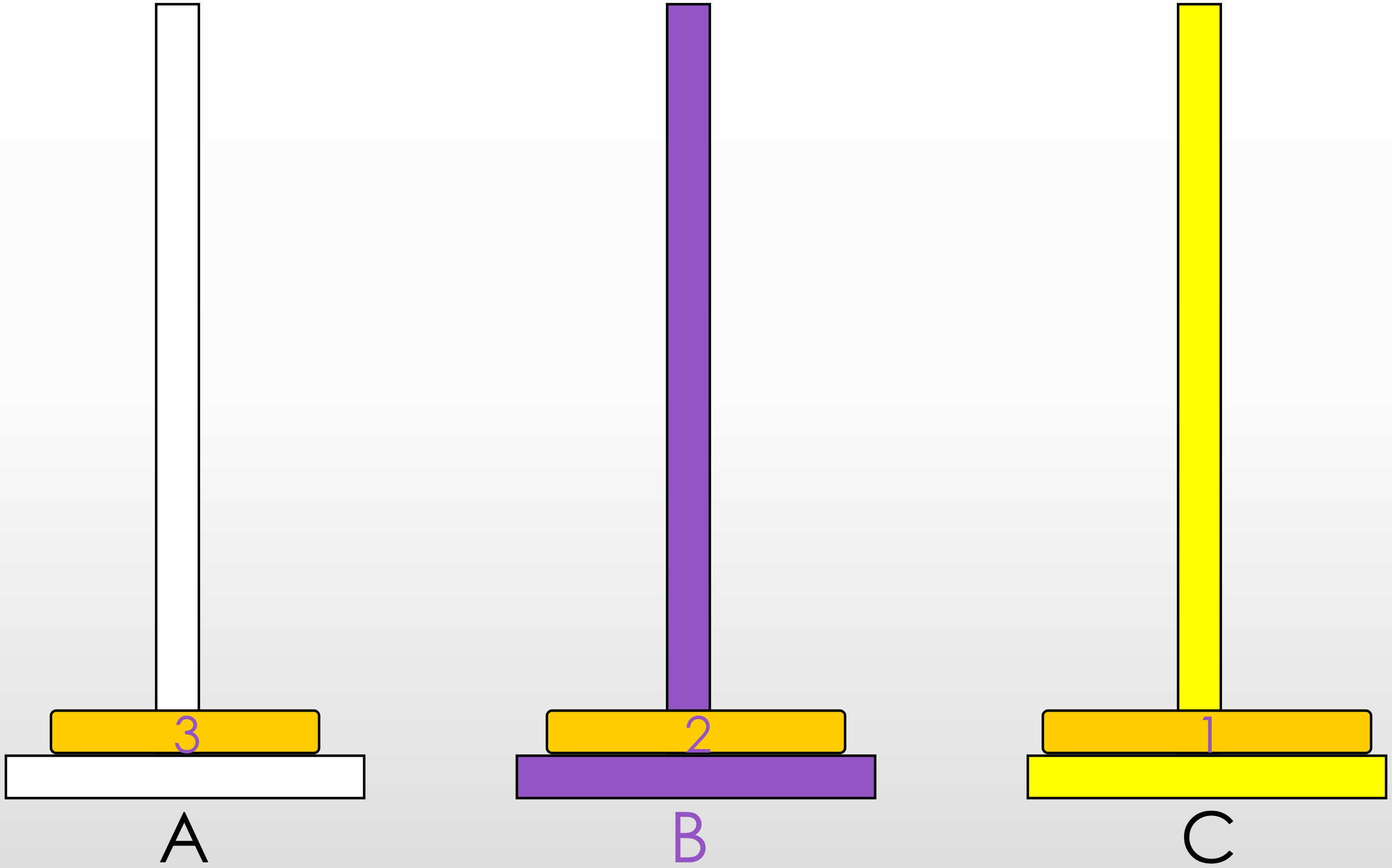
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi



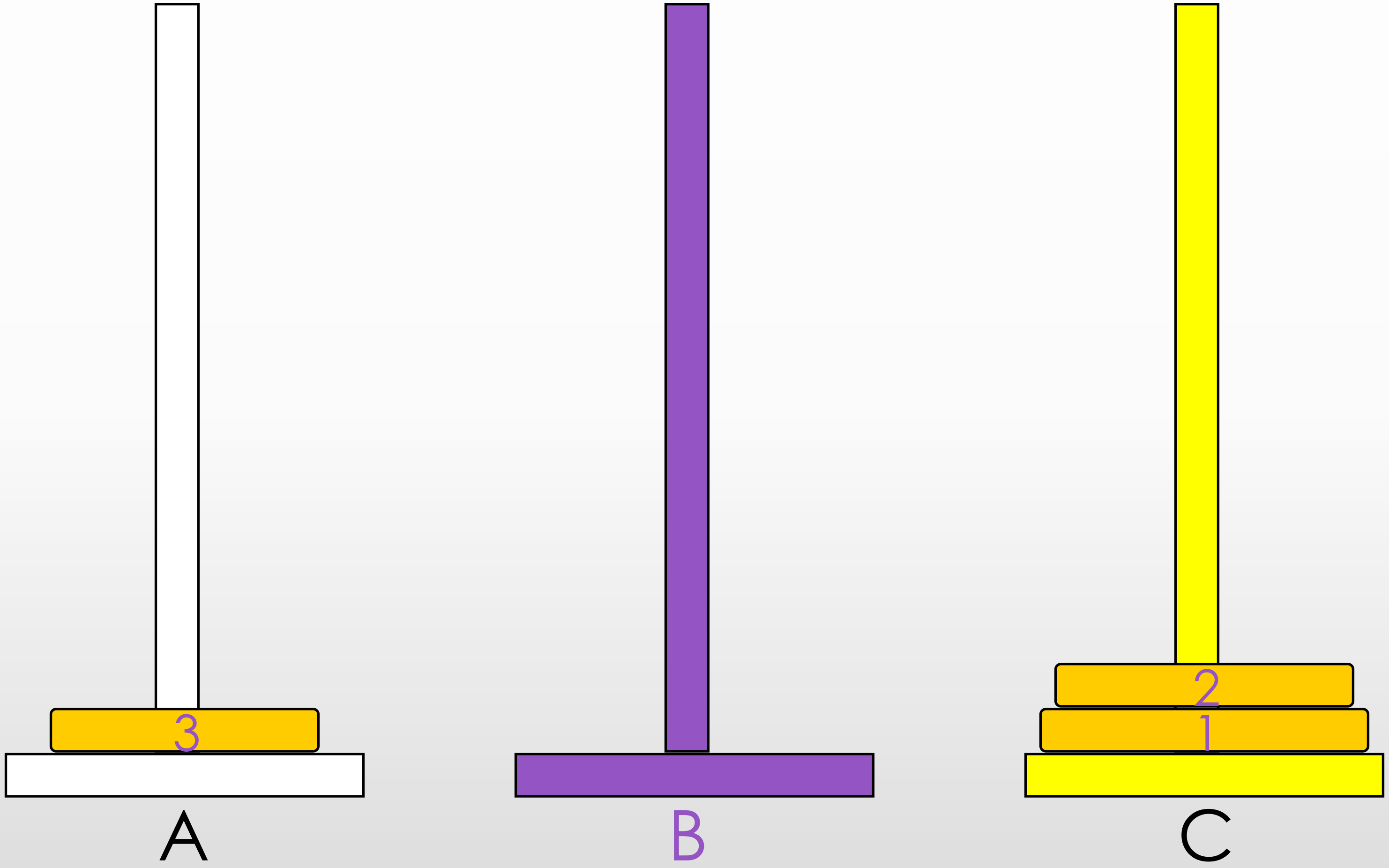
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi



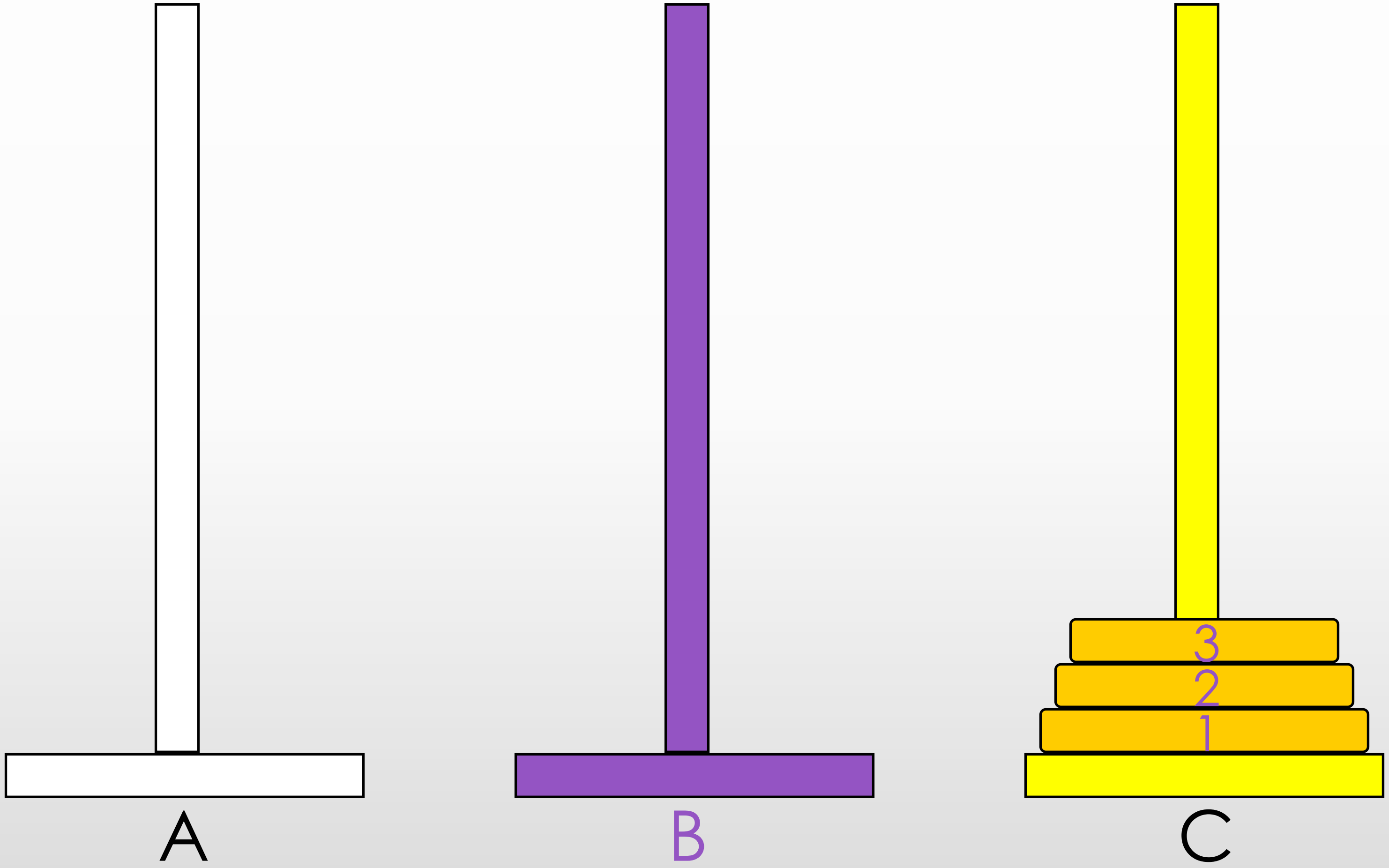
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi



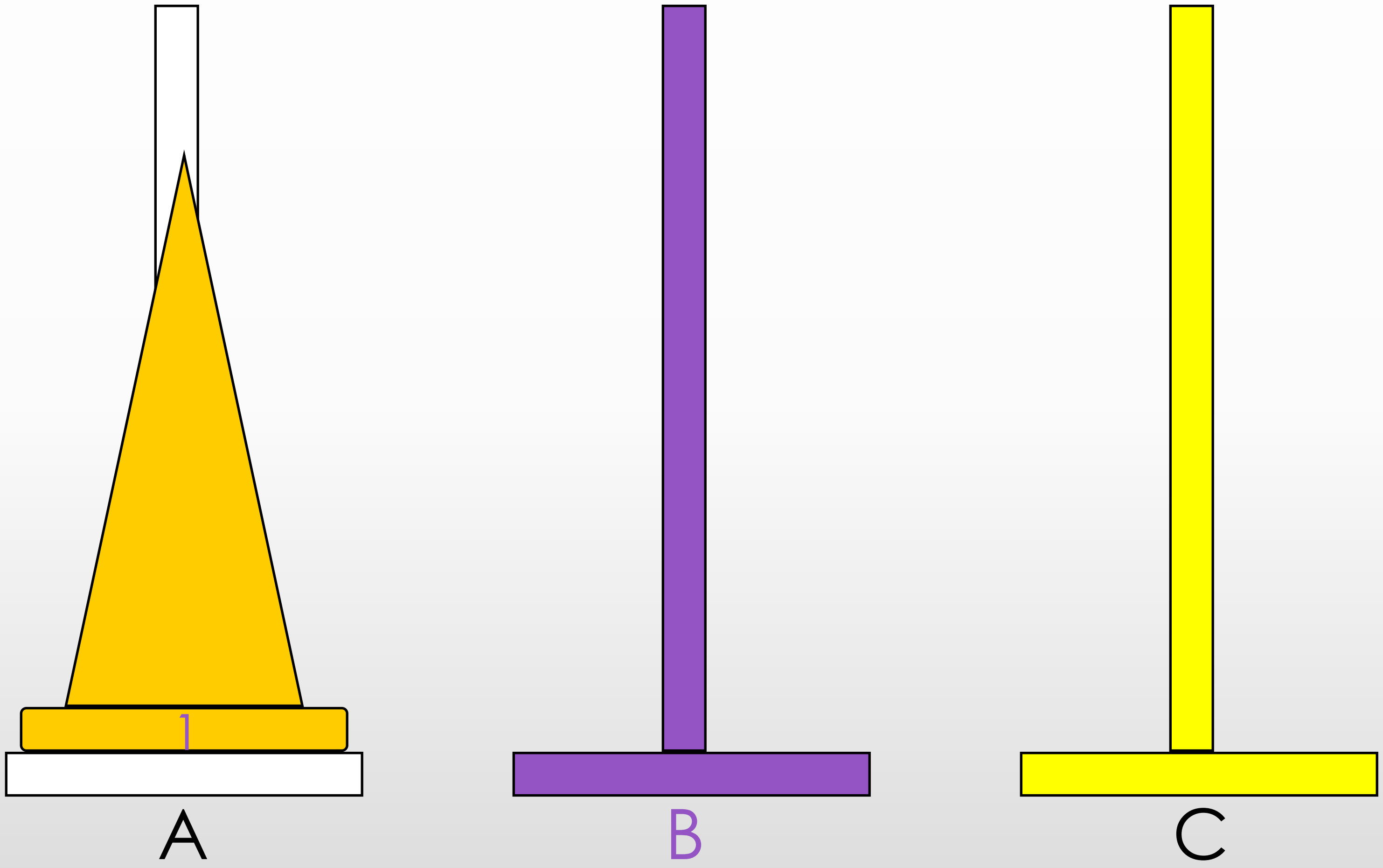
- 3-disk Towers Of Hanoi/Brahma

Towers Of Hanoi



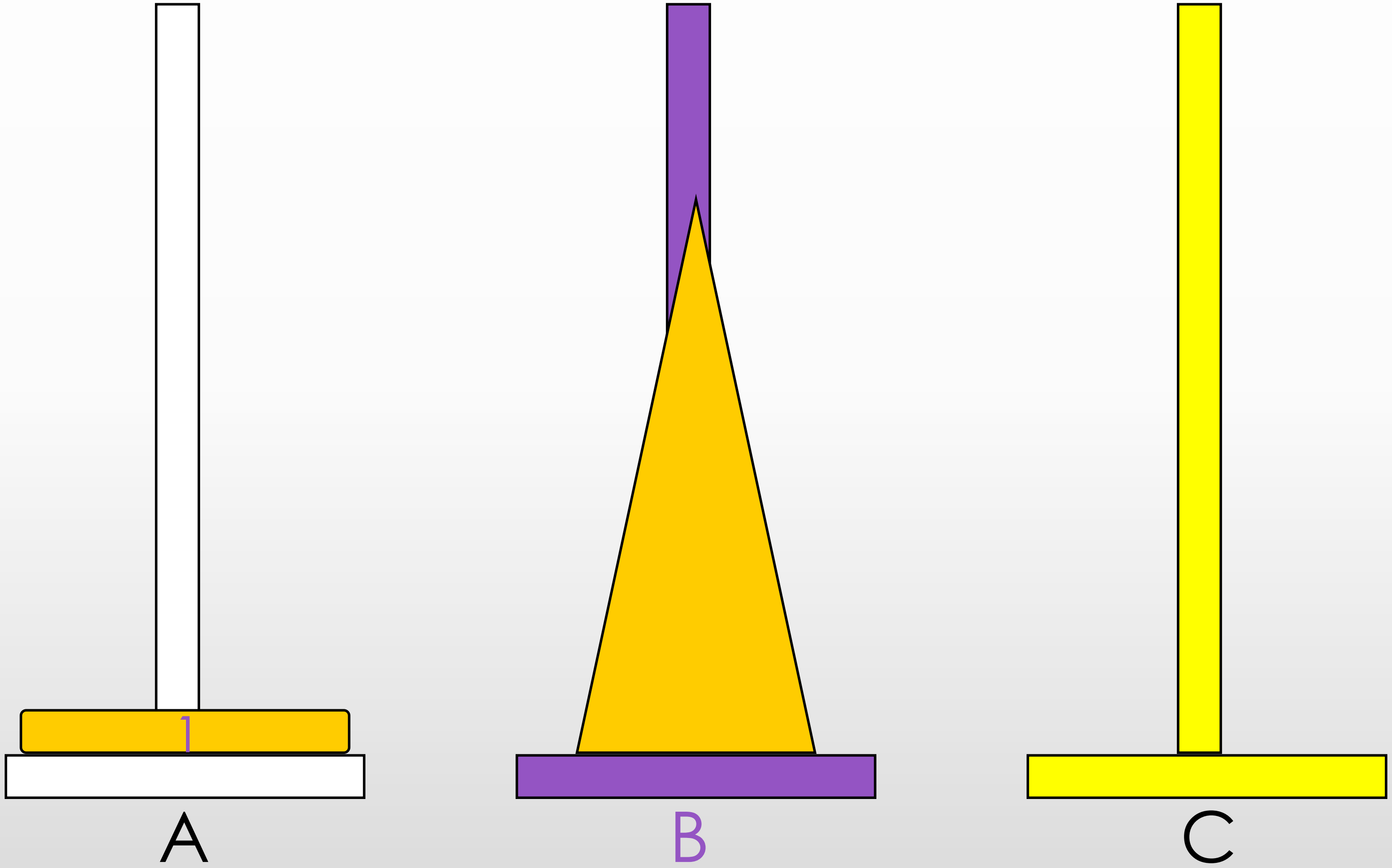
- 3-disk Towers Of Hanoi/Brahma
- 7 disk moves

Recursive Solution



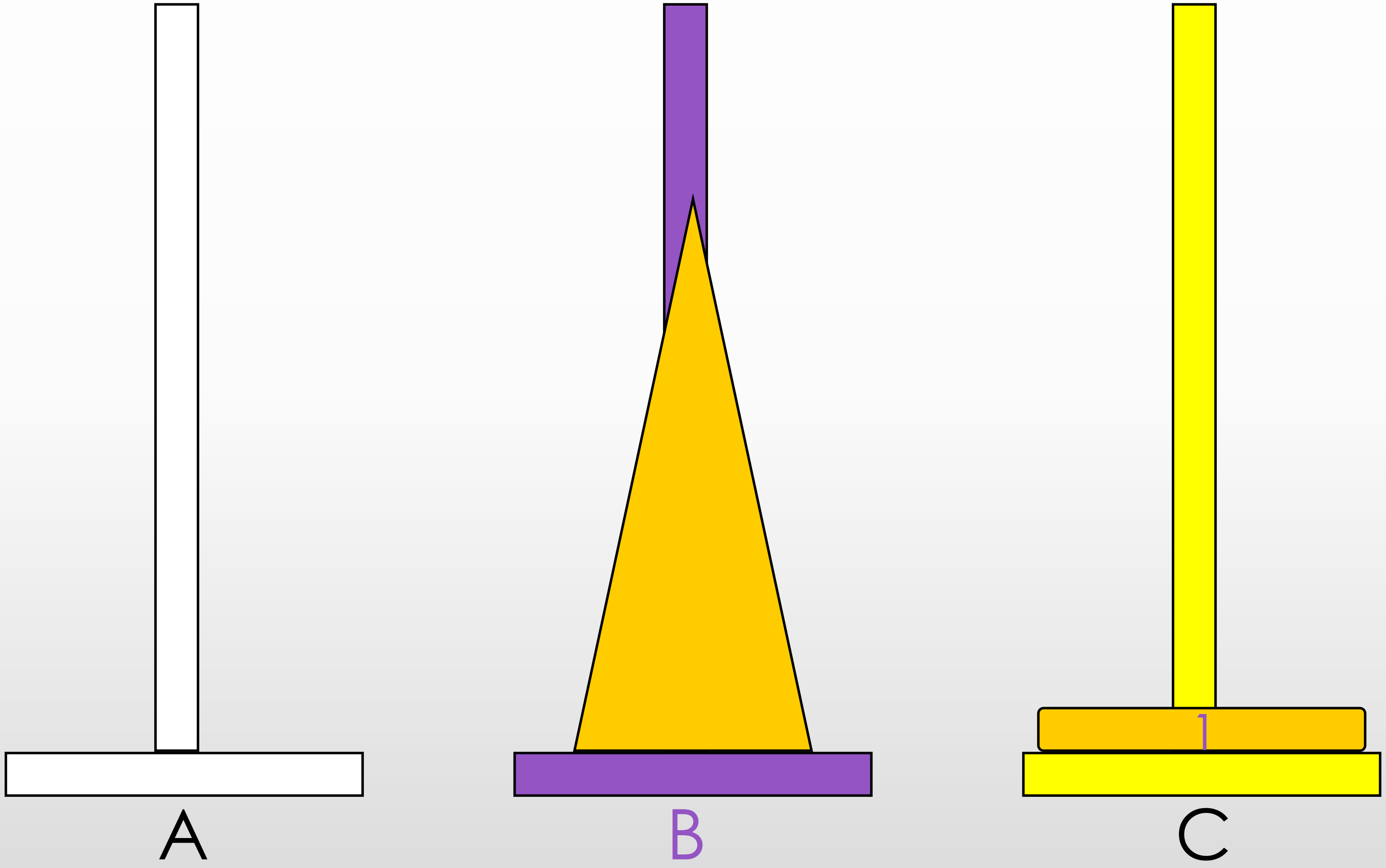
- n > 0 gold disks to be moved from A to C using B
- move top n-1 disks from A to B using C

Recursive Solution



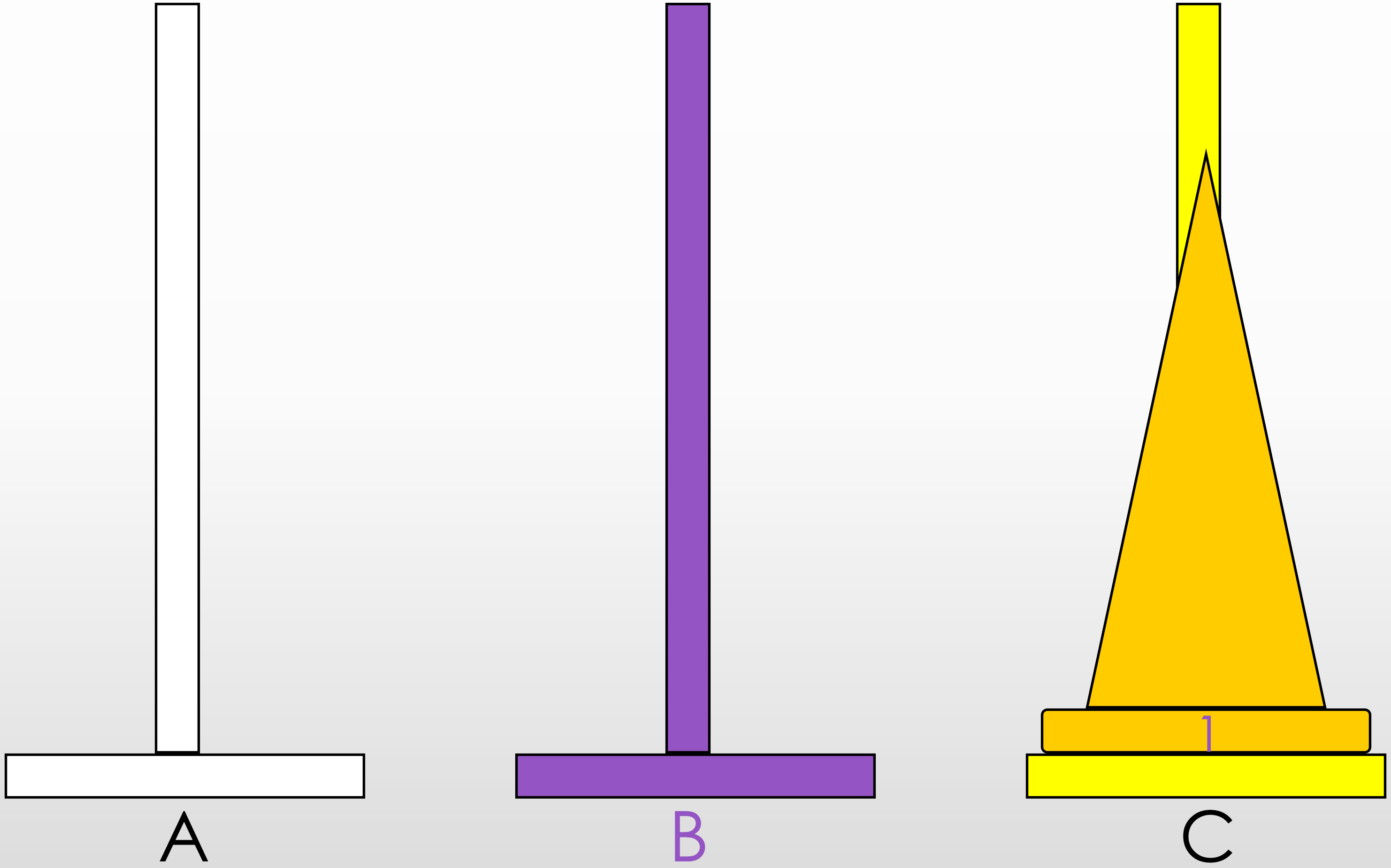
- move top disk from A to C

Recursive Solution



- move top n-1 disks from B to C using A

Recursive Solution



- moves(n) = 0 when n = 0
- moves(n) = 2*moves(n-1) + 1 = 2^n-1 when n > 0

Towers Of Hanoi

- moves(64) = $1.8 * 10^{19}$ (approximately)
- Performing 10^9 moves/second, a computer would take about 570 years to complete.
- At 1 disk move/min, the it will take about $3.4 * 10^{13}$ years.

Queue

maze 쥐 길찾기

stack → 한길 가봤다가 아니면 되돌아오기
메모리 크기가 길을 찾는것에 영향 o

queue → 방향성을 찾아서 탐색
FIFO

Queues

* linear 니어가 의미하는것?

한번 읽으면 out도 한번에 : 순서를 지킨다

⇒ buffer 필요
보통 기준순서

LIFO : Stack
FIFO : Queue

보통 여기에 사용하는 용어

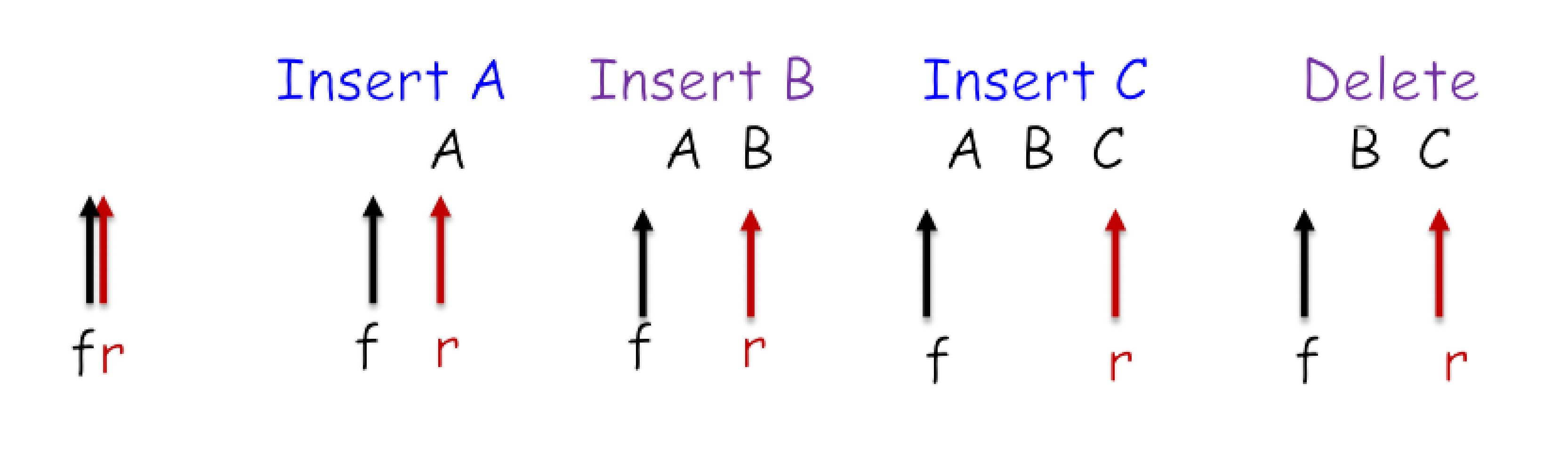
① • Linear list.
• One end is called front.
• Other end is called rear.
• Additions are done at the rear only.
• Removals are made from the front only.

Operations on a Queue

- First-In-First-Out(FIFO) order :
  - f : front pointer
  - r : rear pointer

f, r 같은공간 가리킨다 = 비어있다.

출력: f+1 후에 출력, 삭제



Revisit Of Stack Applications

- Applications in which the stack cannot be replaced with a queue.
  - Parentheses matching.
  - Towers of Hanoi.
  - Switchbox routing.
  - Method invocation and return.
  - Try-catch-throw implementation.

queue 기여하는 역할이 아님.

- Application in which the stack may be replaced with a queue.
  - Rat in a maze.
    - ✓ Results in finding shortest path to exit.

Abstract Data Type(ADT) for Queue

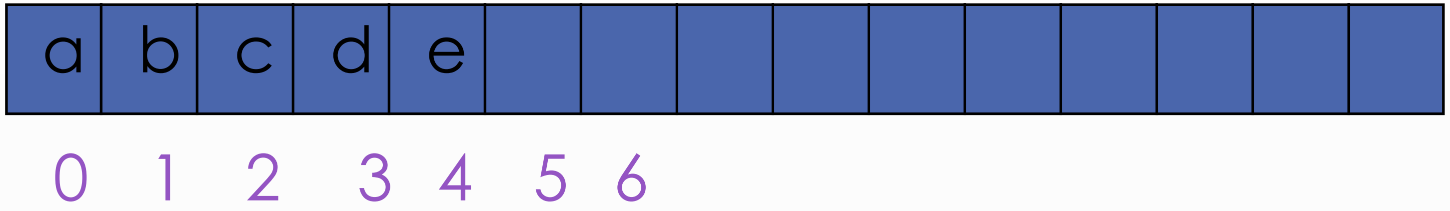
- Object : Queue
  - an ordered list in which all insertions take place at the end and all deletions take place at the other end
- Operations
  - IsFullQ … return true iff queue is full
  - IsEmptyQ … return true iff queue is empty
  - AddQ … add an element at the rear of the queue
  - DeleteQ … delete and return the front element of the queue

Queue in an Array

array를 이용해서 queue 구현

- Use a 1D array to represent a queue.
- Suppose queue elements are stored with the front element in queue[0], the next in queue[1], and so on.

Queue in an Array



- DeleteQ() => delete queue[0]
  - O(queue size) time
- AddQ(x) => if there is capacity, add at right end
  - O(1) time

- to perform each opertion in O(1) time (excluding array doubling), we use a circular representation.

Operations : Problem = 문제점

- Problem: a Job queue by operating system

Q[1]	Q[2]	Q[3]...			
J_1					Insert J_1
J_1	J_2	J_3			Insert J_2, J_3
	J_2	J_3			Delete
	J_2	J_3	J_4	J_5	Insert J_4, J_5
		J_3	J_4	J_5	Delete
			J_4	J_5	Delete

비어있어서 비효율적

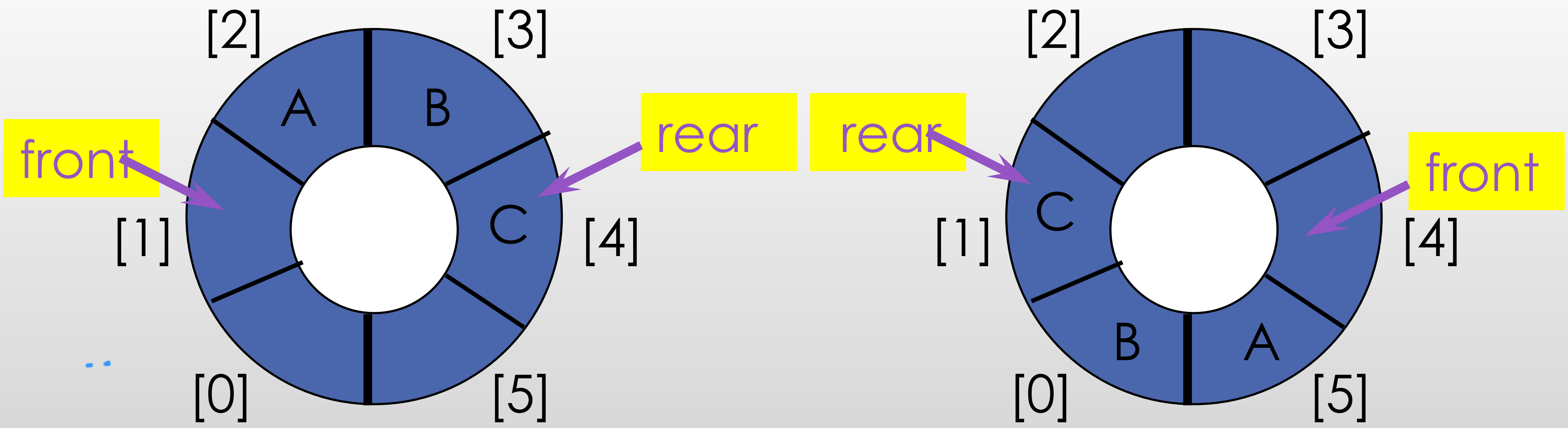
- The queue shifts to the right
- When queue is full, we need to move the entire queue left → time consuming

Circular Array

나머지연산 이용

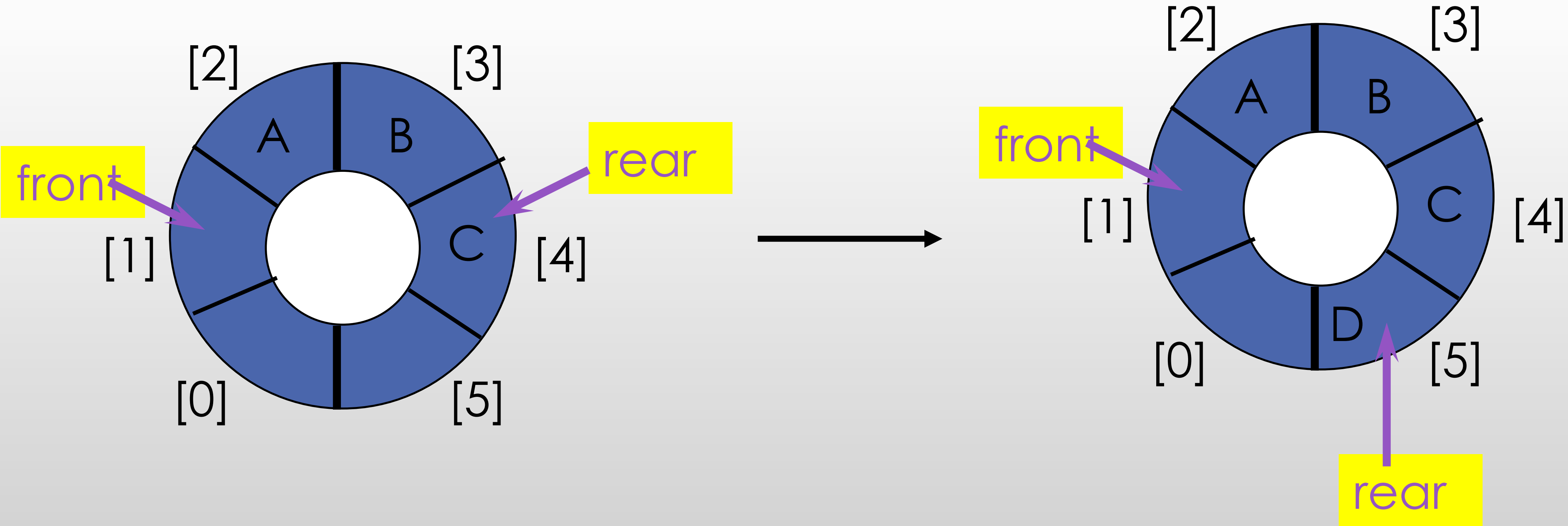
=해결법

- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element



Add an Element

- Move rear one clockwise.
- Then put into queue[rear].



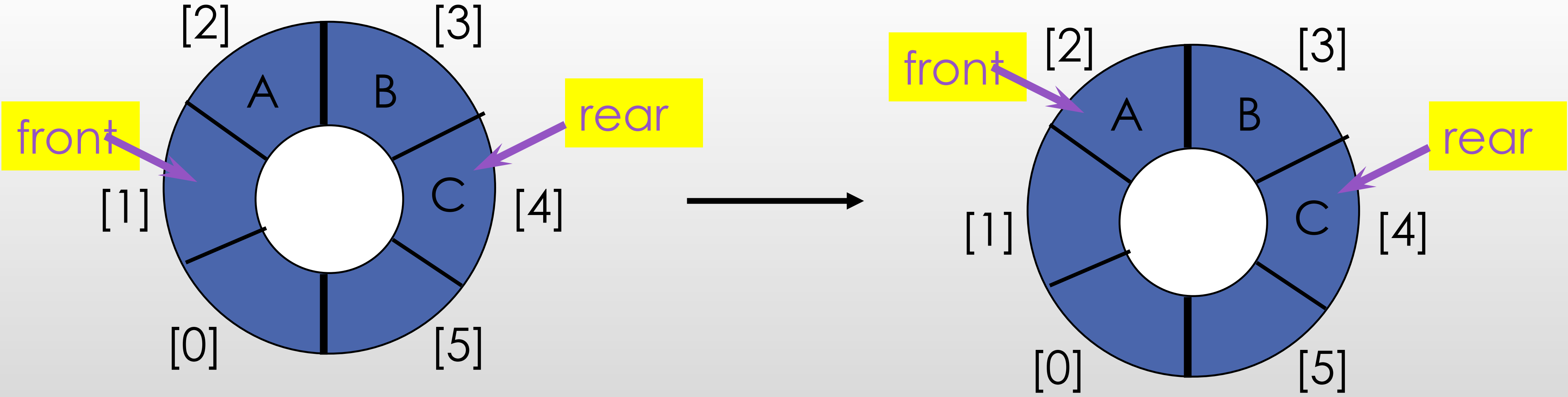
Operations - AddQ

가득참

```
AddQ(int *rear, element item)
{
    if (*rear == MAX_Queue_SIZE - 1){
        QueueFull();
        return;
    }
    *rear = *rear +1;
    queue[*rear] = item;
}
```

Delete An Element

- Move front one clockwise.
- Then extract from queue[front].



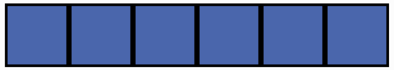
Operations - DeleteQ

```
element DeleteQ(int *front, int rear)
{
	if (*front == rear)
		return QueueEmpty();
	*front = *front + 1;
	return queue[*front];
}
```

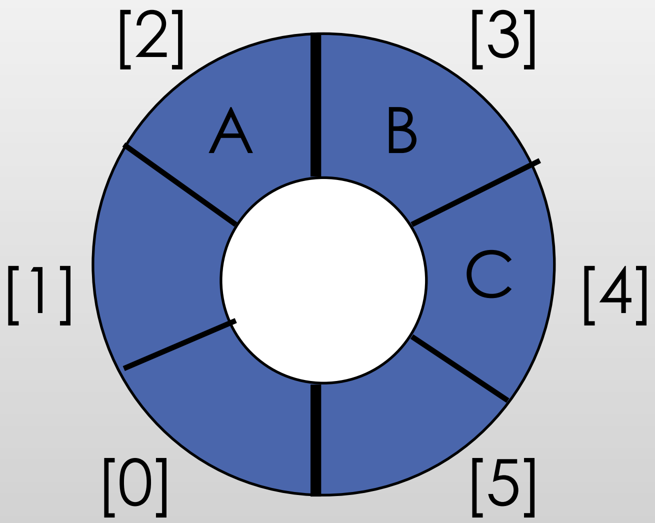
Circular Array

- Use a 1D array queue.

queue[]

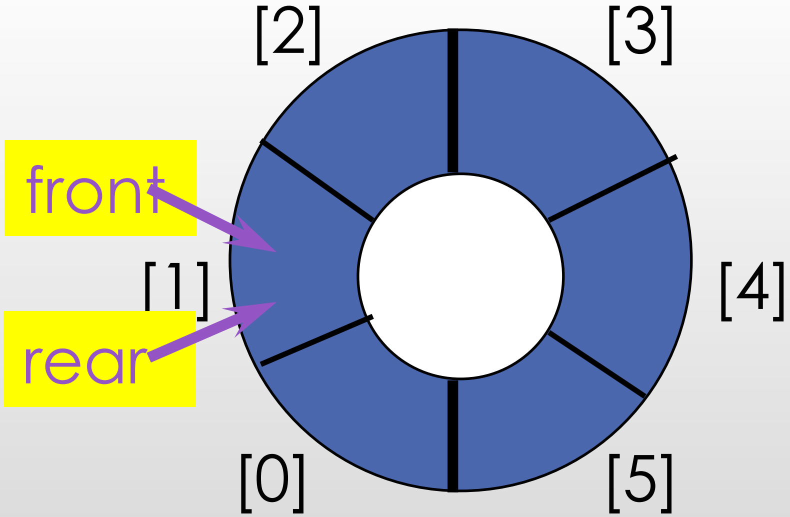


- Circular view of array.
    - Possible configuration with 3 elements.



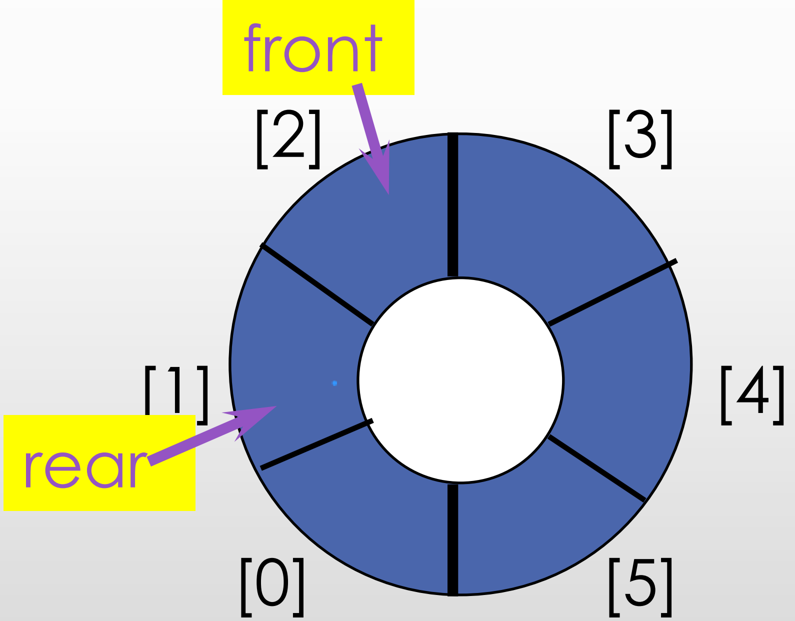
How to test “Empty” & “Full”?

- Empty <=> front == rear



How to test "Empty" & "Full"?

- Full <=> rear + 1 == front
  - use only n-1 element
  - use n elements → will not be able to distinguish full or empty



AddQ using Circular Array

```
AddQ( int front, int *rear, element item)
{
        *rear = (*rear +1) % MAX_QUEUE_SIZE;
        if (front == *rear){
                QueueFull();
                return;
        }
        Queue[*rear] = item;
}
```

나머지계산 연산

⇒ 이거 왜 쓰는건지?

아 원래는 6+1 하면 7

근데 우리는 6칸 원형큐로

생각 해야하니까 6다음이 0이

되게해야함.

∴ 만약 rear=5, front=0이면

$(5+1) \% 6 = 0$

∴ *rear =0 이되게 다시 셋팅

DeleteQ using Circular Array

```
element DeleteQ(int *front, int rear)
{
	if (*front == rear)
		return QueueEmpty();
	*front = (*front +1) % MAX_QUEUE_SIZE;
	return queue[*front];
}
```

Evaluation of An Expression

- x = A/B-C
  How to evaluate an expression ?

- let A=6, B=3, C=2

  (A / B) - C
  =(6/3)-2
  =2-2
  =0

  (A / (B - C))
  =(6/(3-2))
  =6/1
  =6

식을 쭉 읽어보고 빼면서 연산

A / B - C

operator가 들어오면 operation (피연산자)을 기다려야.

(A / (B - C))

operand

operation 아니므로 연산기다려

→ 연산해

결과 operation 만났으니까 계산해

'(' 의 우선순위가 '/' 보다 높으므로 그다음 연산부터 시행

queue 쓰면 O(n) 걸림

즉, 선형시간이용

* 우선순위 찾아서 계산하면 $O(n^2)$ ⇒ 괄호(우선순위)를 찾으러 for문 n번돌려야 하니까

infix: 선형으로 처리 불가.
postfix : 선형연산

A / B - C ⇒ A B / C -

operator 위한 stack 하나 필요

operander 2개 + operator 1개 만나면 연산.

if A B C / -

(x)

(o) B / C

(o) A - (B / C)

queue는 순서를 지키기 위한 것. (FIFO)

stack은 우선순위를 정하기 위한 것.

Rules to evaluate an expression

- Assign precedence to operator, and
  Evaluate first the operator with higher precedence
- Left to right for the operator with the same precedence (in general),
- Right to left for prefix unary operators

Rules to evaluate an expression

Token	Operator	Precedence	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement2	16	left-to-right
-- ++ !- +	decrement, increment3 logical not unary minus or plus	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	left-to-right
+ -	Add, substract	12	left-to-right

Data Structure

5rd Week

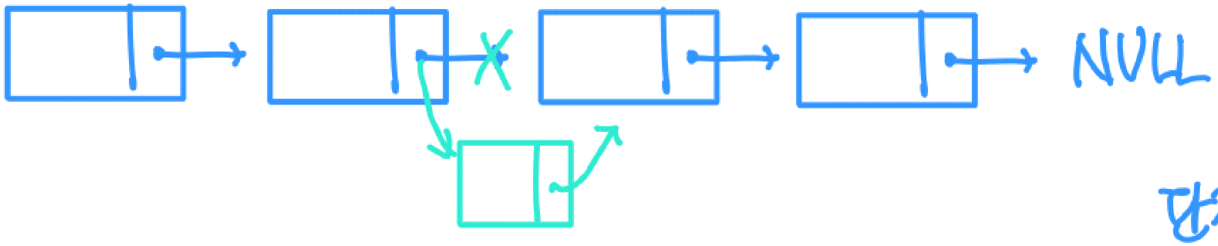
Linked Lists

Mucheol Kim, CAU

Linked Lists

Definition of Linked List

- Referenced from Wikipedia
  - A linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next.
  - collection of nodes which together represent a sequence.
    - each node contains data, and a reference to the next node in the sequence.
  - This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.
  - A drawback of linked lists is that data access time is linear in respect to the number of nodes in the list.



지속적인 추가 / 삭제가 매우 효율적임

단점 : 개별리스트 전체를 탐색해야 접근가능

Linked Lists

이름을 몇개 지어야할지 모르겠으니까
주소만 한당
단점 : 이름을 모르니까 탐색할때 리스트 전체를 돌아야한다.

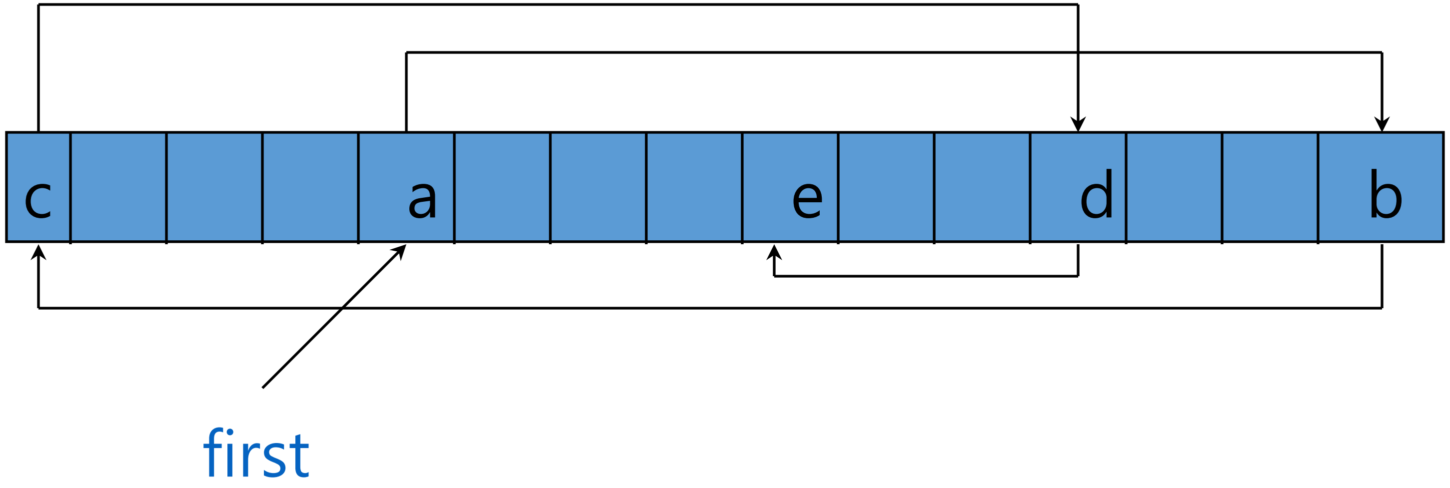
- list elements are stored, in memory, in an arbitrary order
- explicit information (called a link) is used to go from one element to the next
- Layout of L = (a,b,c,d,e) using an array representation.

a	b	c	d	e										

A linked representation uses an arbitrary layout.

c				a				e			d			b

Linked Representation



- pointer (or link) in e is NULL

  use a variable first to get to the first element a

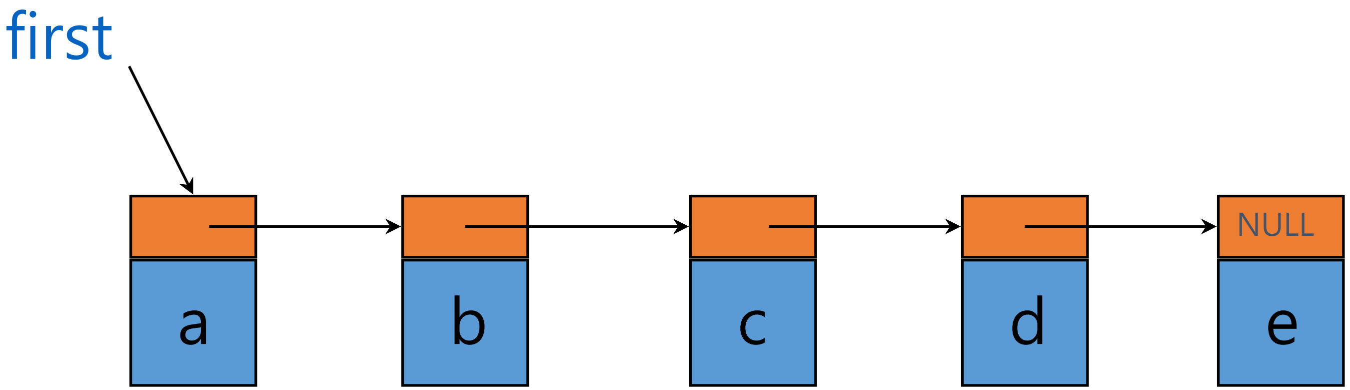
요청할 때마다
메모리주소 탐색 요구

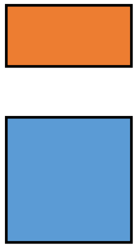
ADT of Linked List

- Object :
  - An ordered list of elements

- Operations:
  - Insertion of an element
  - Deletion of an element

Normal Way to Draw A Linked List



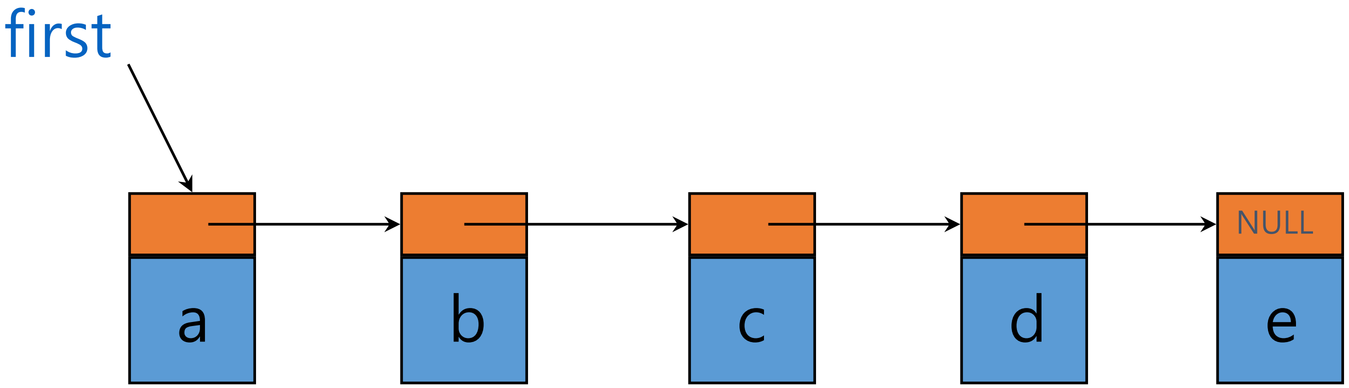


- link or pointer field of node

data field of node

Chain

- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer.



Node Representation

구조체

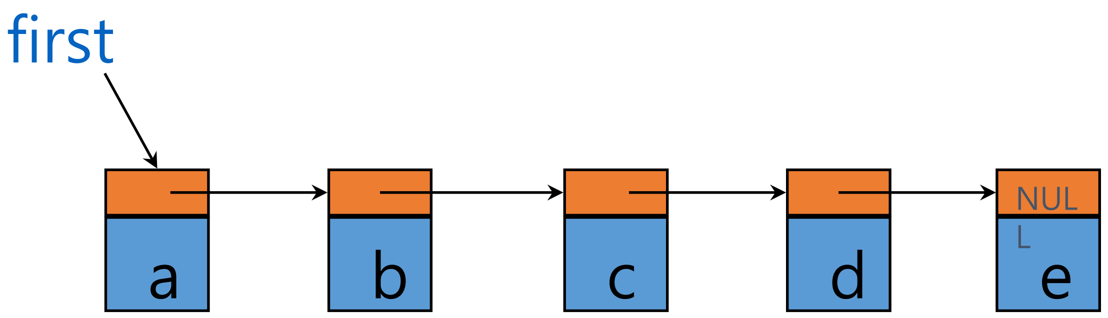
```
listNode *listPointer;
{
char data;
    listPointer link;
} listNode;
```

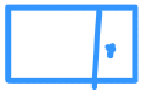
listPointer → listNode 를

데이터타입으로 가짐

listNode 라는 구조체를 listPointer가 참조

Get nodes





desiredNode = first; // gets you to first node
return desiredNode–>data; a

desiredNode = first–>link; // gets you to second node
return desiredNode–>data; b

desiredNode = first–>link–>link; // gets you to third node
return desiredNode–>data; c

desiredNode = first–>link–>link–>link–>link–>link;
// desiredNode = NULL
return desiredNode–>data; // NULL.element
NULL

* desiredNode = desiredNode → link; 로 해도 동일

* d 찾고 b 찾으려면?

d → first부터 다시 탐색 → b ⊕ (정규화
애초에 전 data 기억하지 X

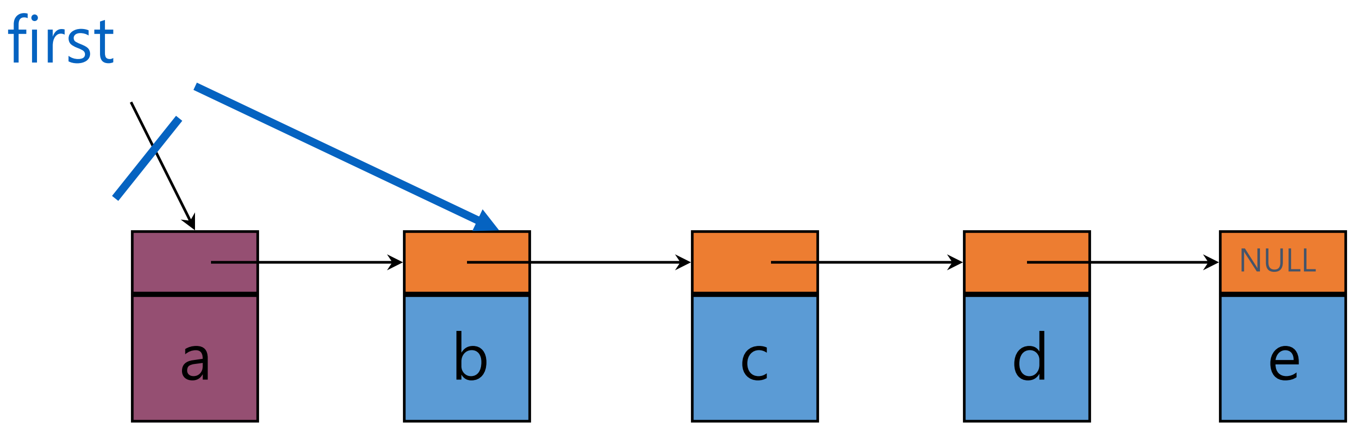
int *a; → 주소를 가져옴. 즉, 있는거에 이름을 붙이는것.

int *a=malloc(~);) 둘의 차이알기

↳ 이주소공간의 자료형 size만큼의 메모리를 점유, 저장.

즉, 새로만드는것. ⇒ insert 할 때 이용

Delete An Element



delete(0)

deleteNode = first;

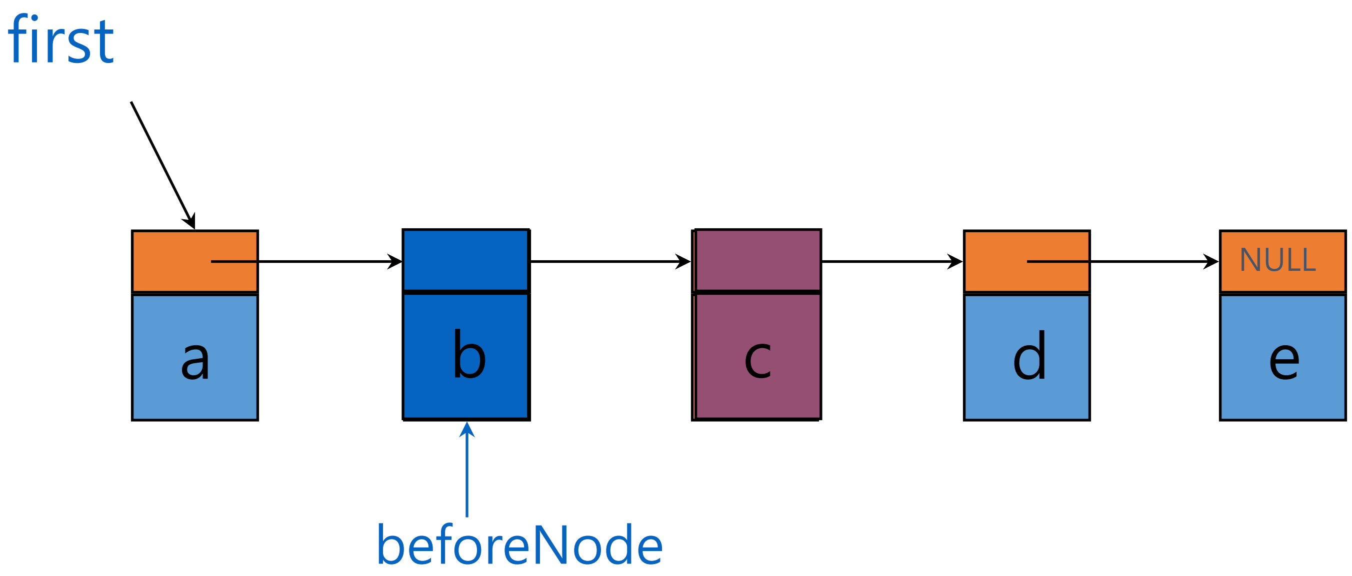
first = first–>link;

free(deleteNode);

first를 할당 없이 지우는 것 안됨.
그뒤의 node들 다 잃어버리는것임
(first는 a의 주소만 가지고있기때문)

Delete An Element

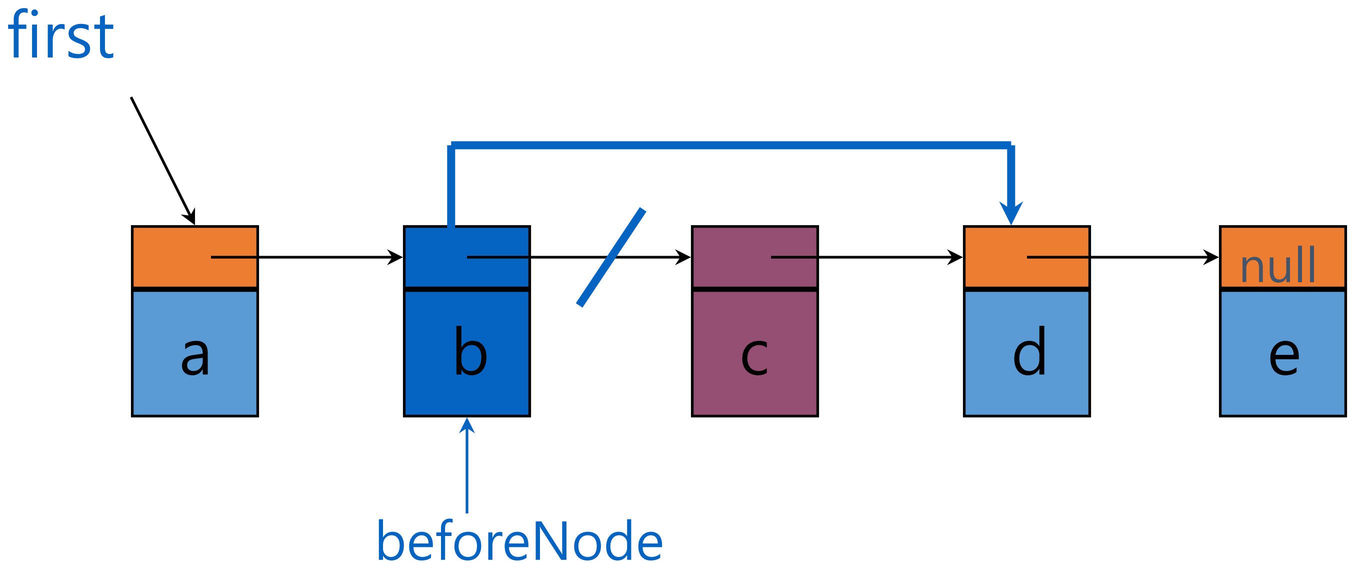
3번째 노드 삭제할때



first get to node just before node to be removed

beforeNode = first–>link;

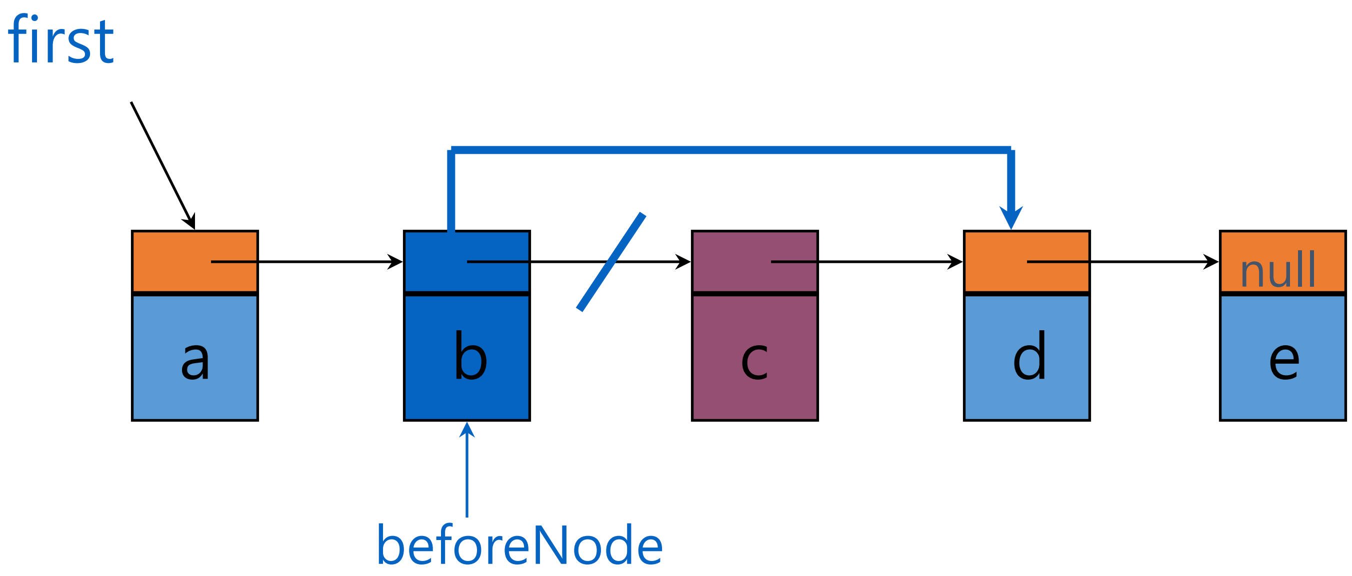
Delete An Element



save pointer to node that will be deleted

deleteNode = beforeNode–>link;

Delete An Element

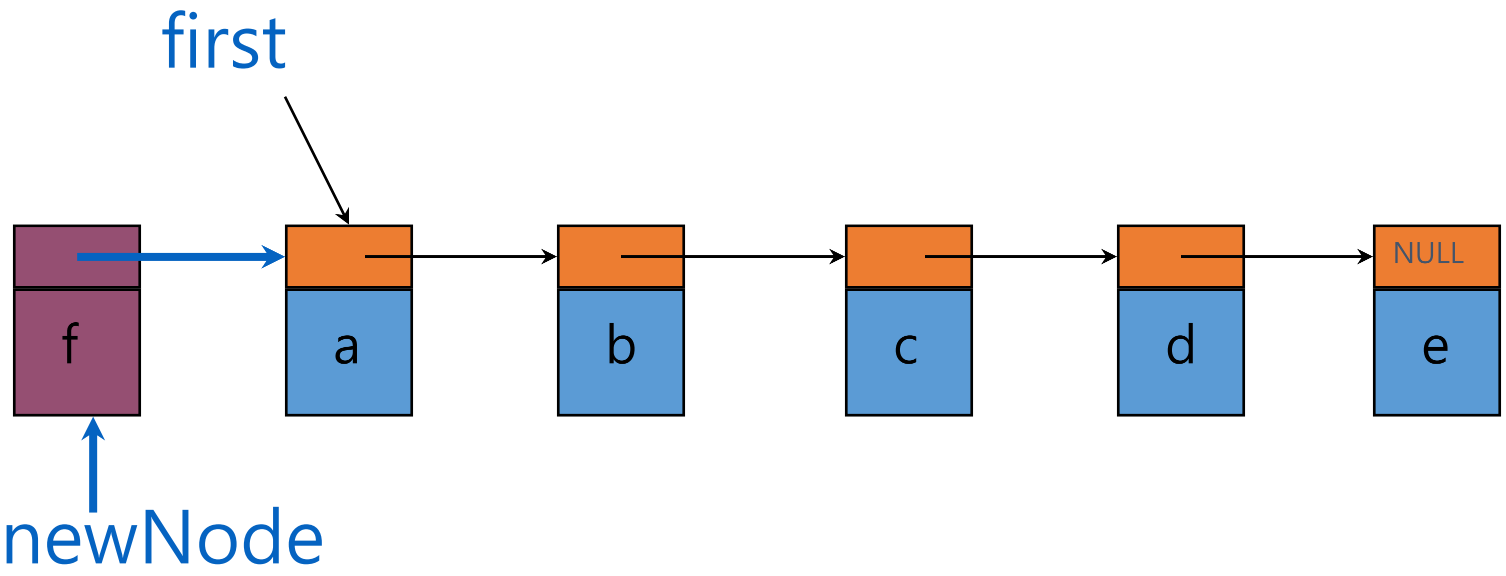


now change pointer in beforeNode

```
beforeNode–>link = beforeNode–>link–>link;
free(deleteNode);
```

이과정을 해주지않고 일단 지우면
d,e는 사라지는것임.

insert an Element

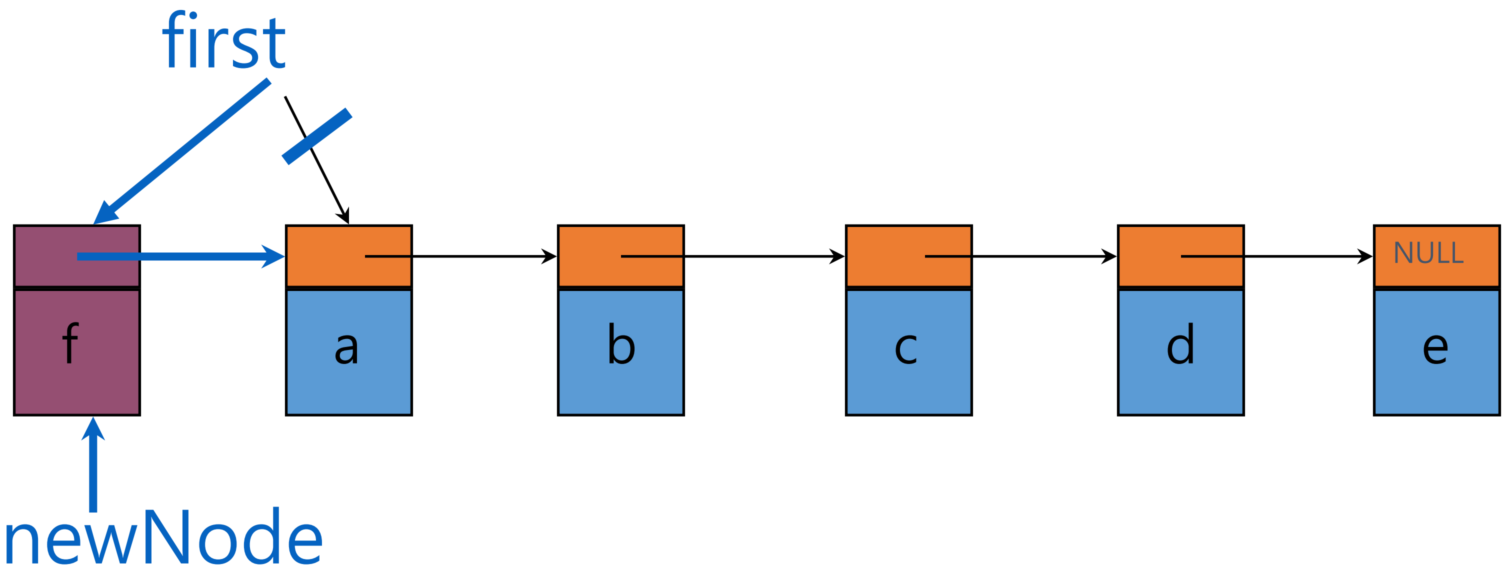


Step 1: get a node, set its data and link fields

```
MALLOC( newNode, sizeof(*newNode));
newNode->data = 'f';
newNode->link = NULL;
```

생성할 때 필수적 malloc (자료형. 자료형 size)

Insert an element

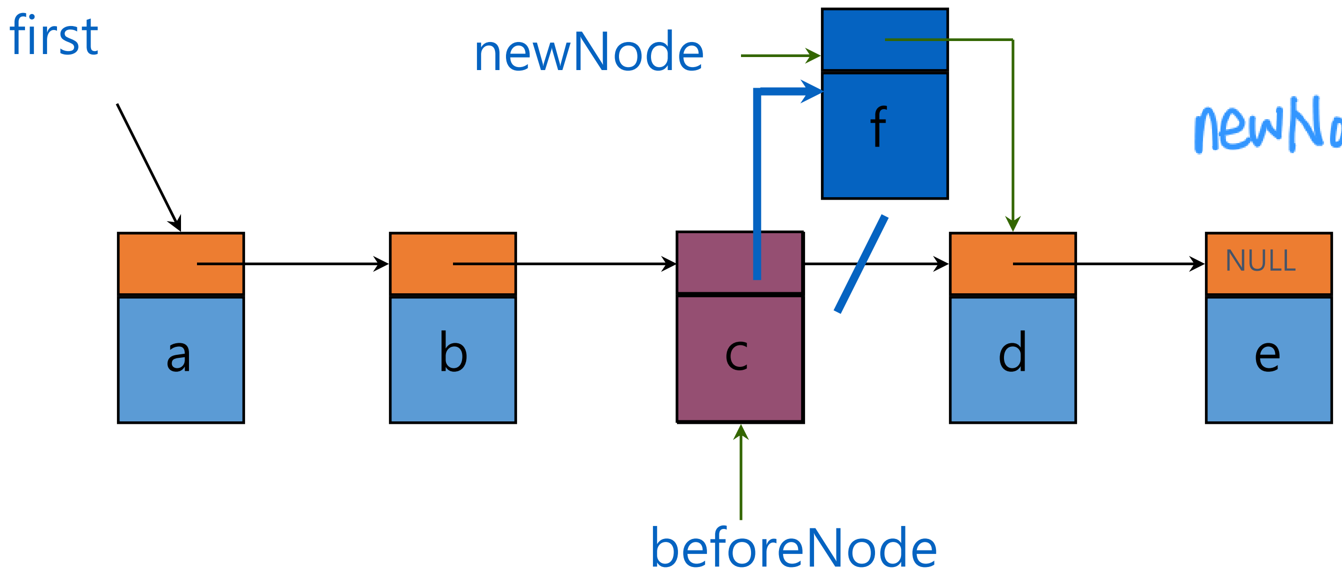


Step 2: update first

newNode → link = first;

first = newNode;

Insert an element

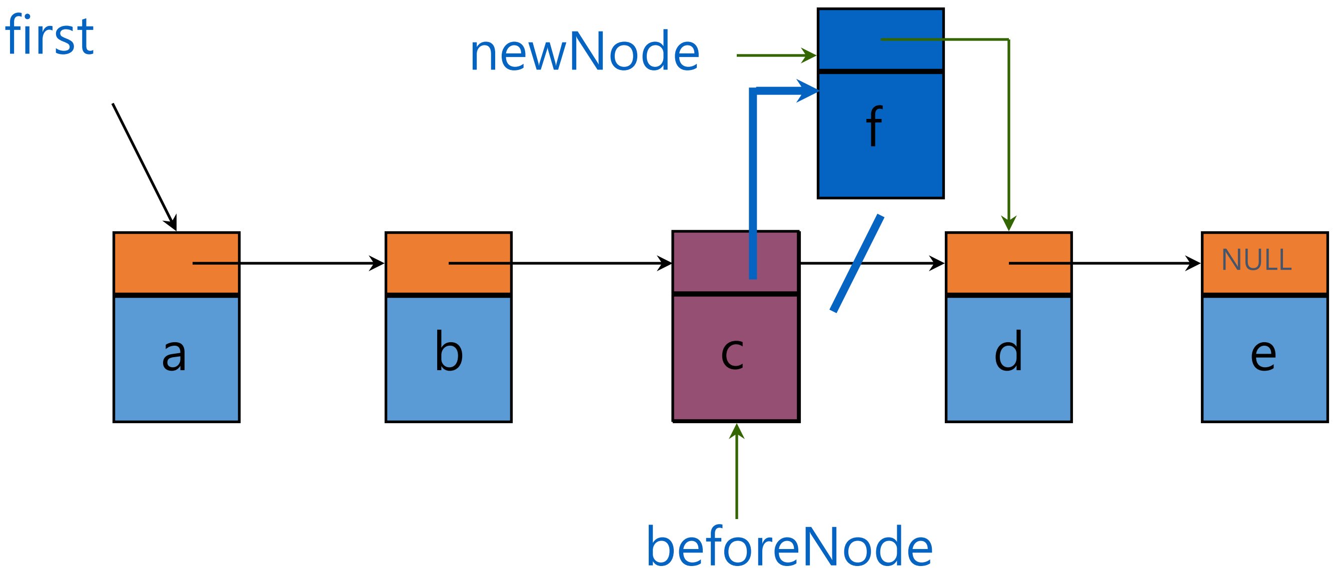


newNode→link = beforeNode→link;

beforeNode→link = newNode;

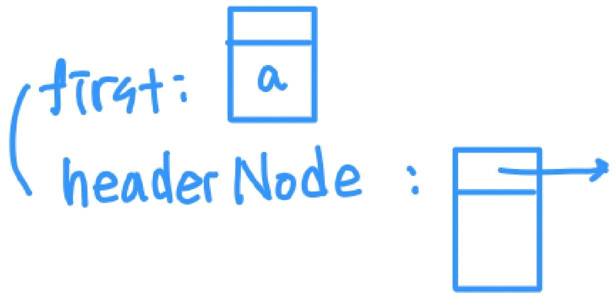
- first find node whose index is 2
- next create a new node and set its data and link fields
- finally link beforeNode to newNode

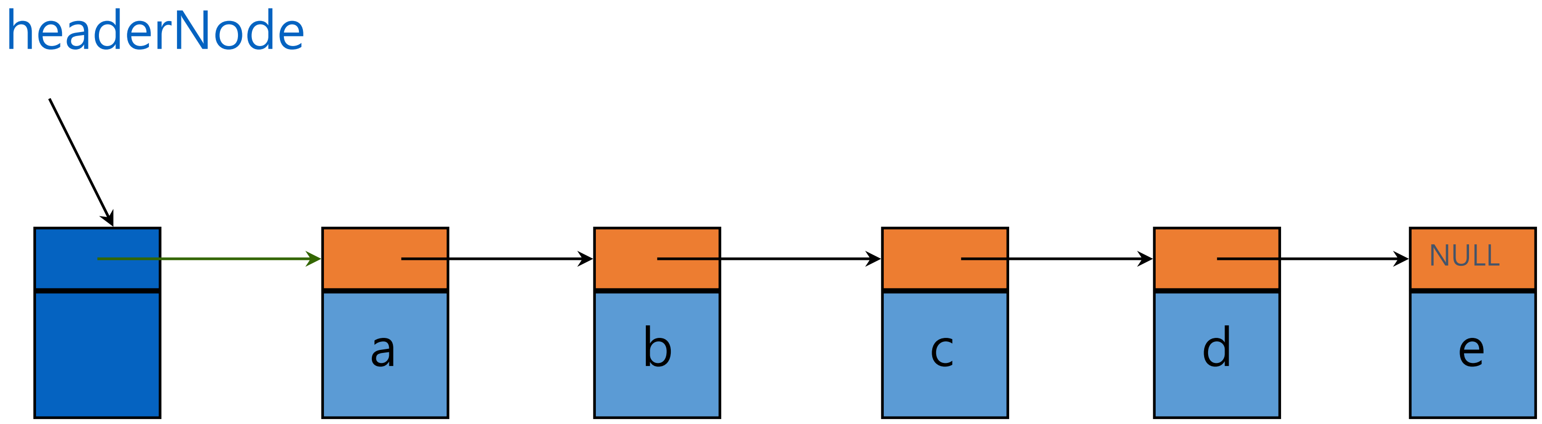
Insert an element



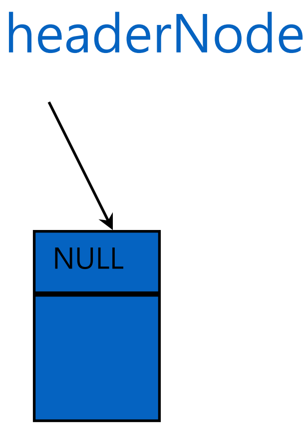
beforeNode = first–>link–>link;
MALLOC(newNode, sizeof(*newNode)); → 새 노드 생성
newNode->data = 'f'; → 생성된 Node에 data 추가
newNode->link = beforeNode->link;
beforeNode–>link = newNode;

Chain With Header Node

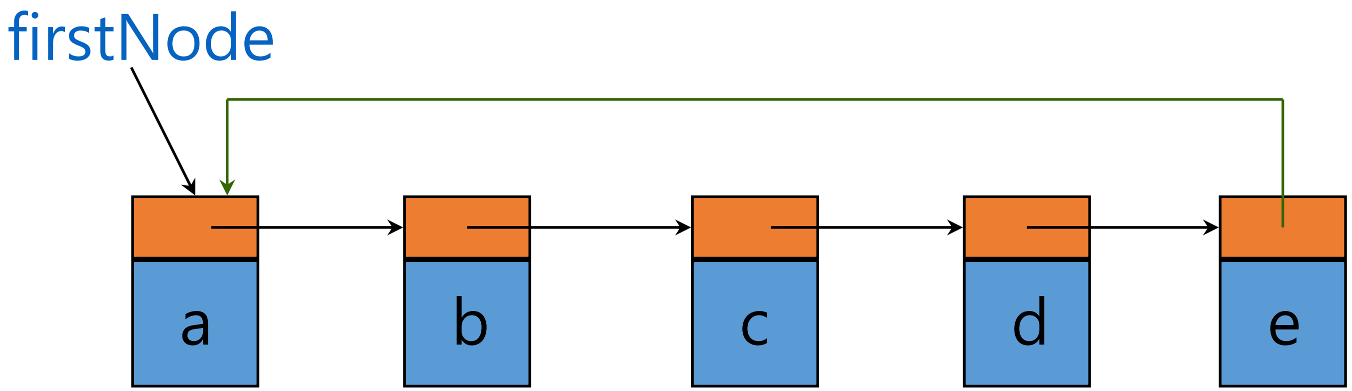




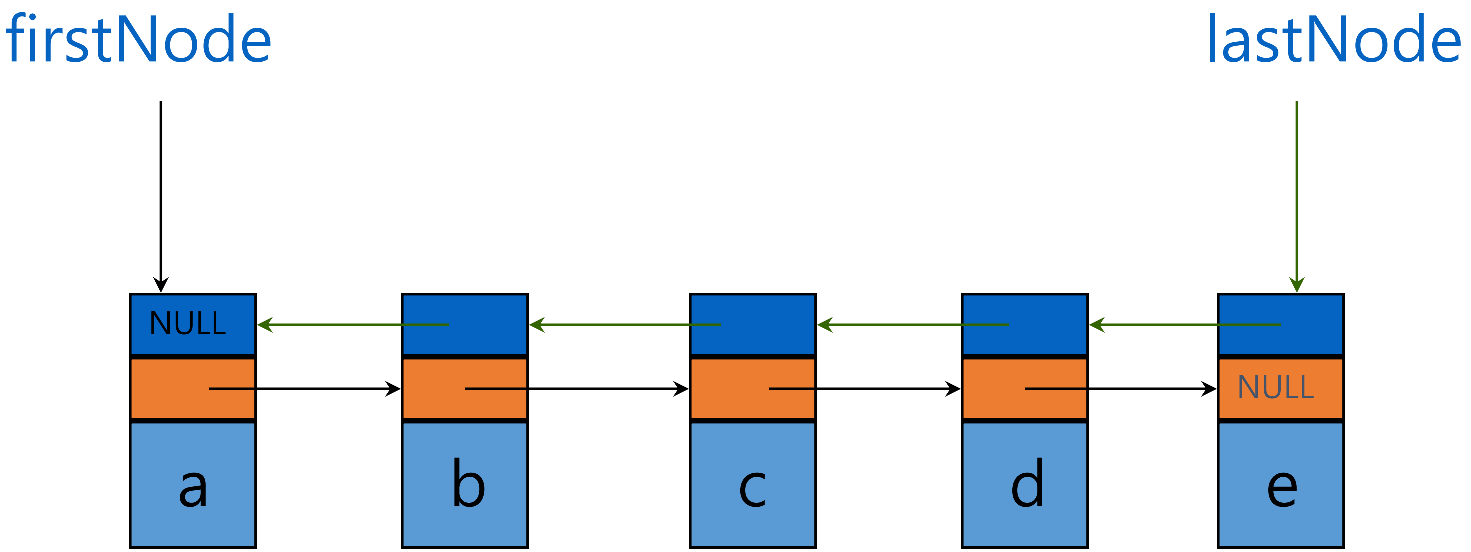
Empty Chain With Header Node



Circular List



Doubly Linked List

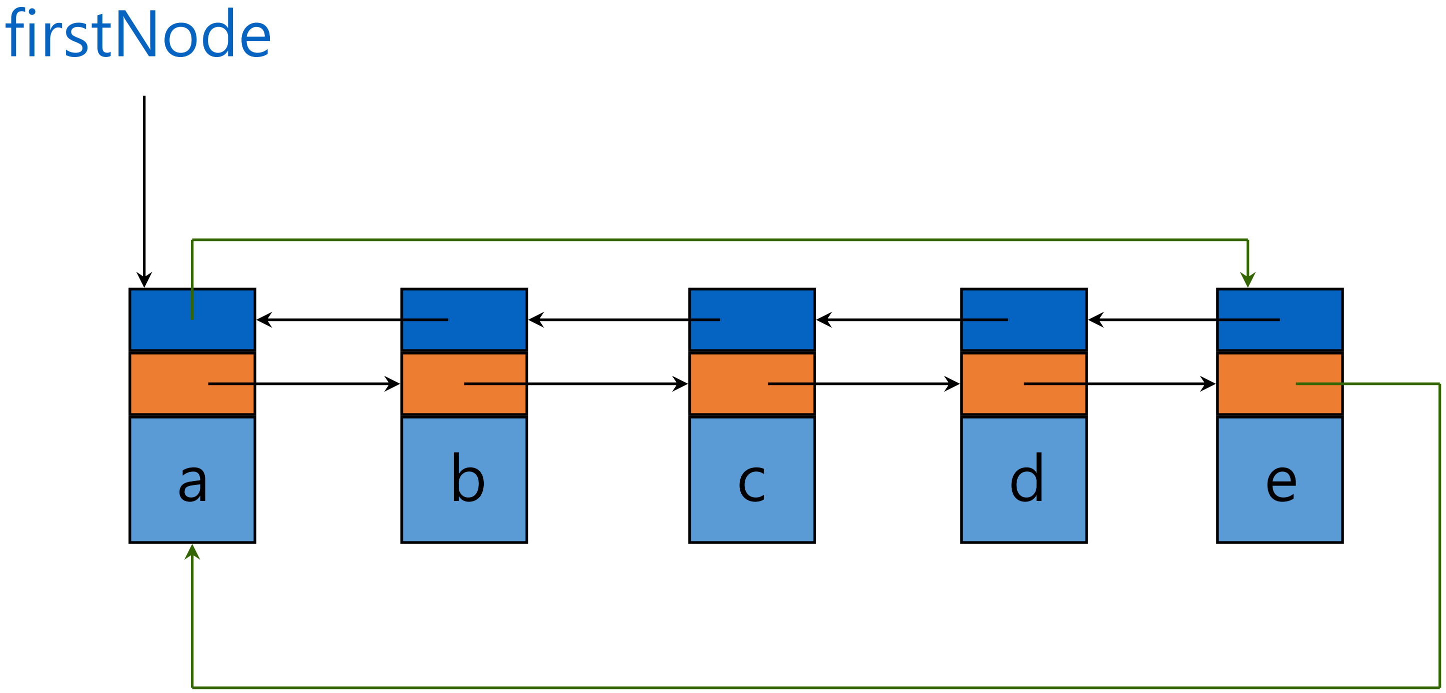


Doubly Linked Circular List

* 구현해보기

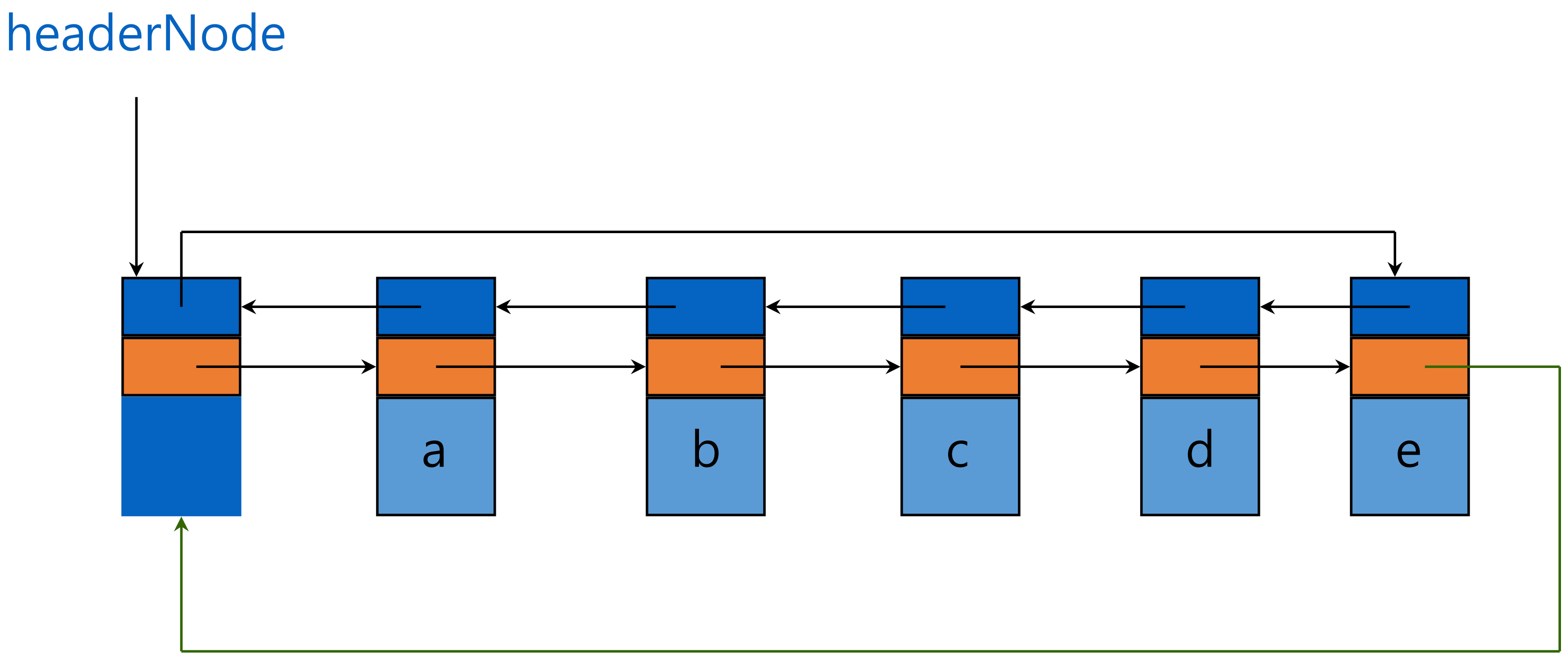
polynomials ⇒ 다항식의 연산
linked list로 구현

ex.) $(2x^2-3x-2)(2x+1)$

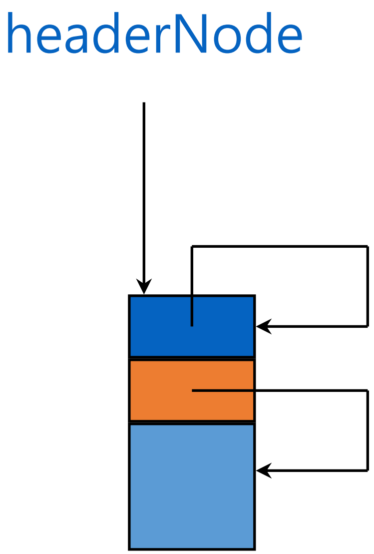


Doubly Linked Circular List With Header Node

장점: headerNode 라는 기준 덕분에
Node 추가/삭제 일반화와 쉬워짐



Empty Doubly Linked Circular List With Header Node



(linked list 구현해보기)

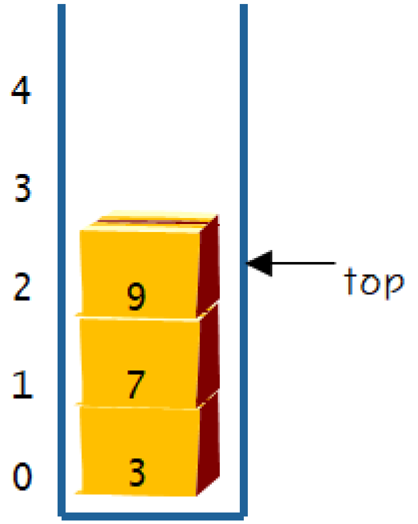
Linked Stack & Linked Queue

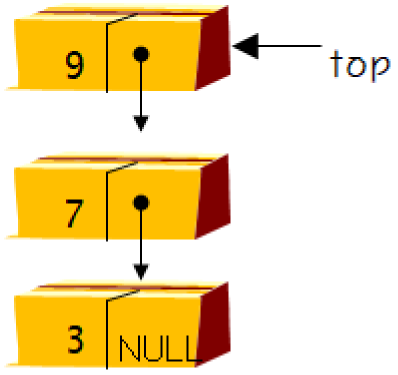
Array vs Linked list for Stack

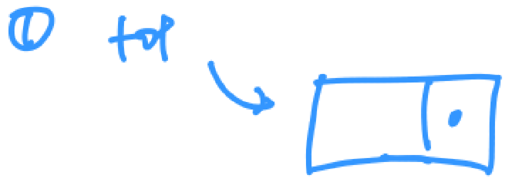
- The major problem when the stack implemented using an array
  - It works only for a fixed number of data values.
- Stack using a linked list
  - An unlimited number of values
  - No need to fix the size at the beginning of the implementation

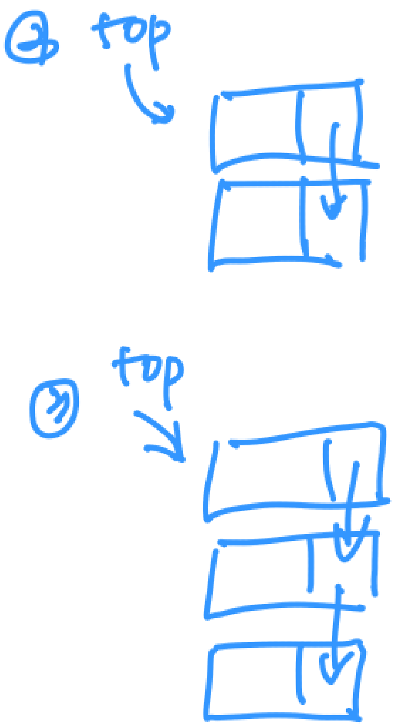
Stack using linked list

- Linked stack
  - Every new element is inserted as '**top**' element.
  - remove an element from the stack pointed by '**top**'
    - moving '**top**' to its previous node in the list.

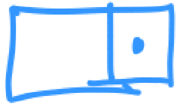








Stack using linked list



```
typedef int element;

typedef struct StackNode {
      element item;
      struct StackNode *link;
} StackeNode;

typedef struct {
      StackNode *top;
} LinkedStackType;
```

Stack using linked list – push Operation

```
                stack의주소
void push(LinkedStackType *s, element item)
{
    StackNode *temp=(StackNode *)malloc(sizeof(StackNode));
    if( temp == NULL ){
        fprintf(stderr, "error₩n");
        return;
    }
    else{
        temp->item = item;
        temp->link = s->top;   stack의 top을 temp(new)의 link에 전달
        s->top = temp;
                         temp가 스택의 탑이됨.
    }
}
```

Stack using linked list – pop Operation

```
element pop(LinkedStackType *s)
{
     if( is_empty(s) ) {
               fprintf(stderr, "error₩n");
               exit(1);
      }
      else{
               StackNode *temp=s->top;
               int item = temp->item;
               s->top = s->top->link;
               free(temp);
               return item;
      }
}
```

탑의 주소를 temp에 저장

탑(temp)의 아이템 저장

top에 top의 link 전달

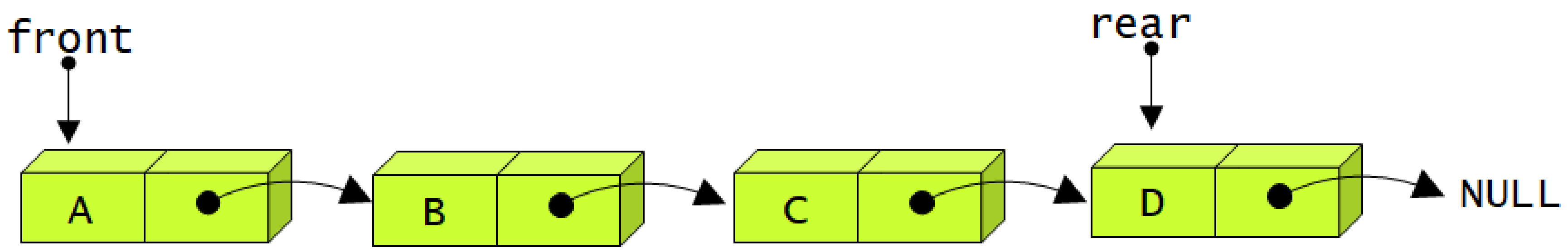
temp 삭제

Array vs Linked list for Queue

- The major problem with the queue implemented using an array is
  - It will work for an only fixed number of data values.
- Queue using a linked list
  - It works for an unlimited number of values.

Queue Using Linked List

- In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



Queue using linked list

```
typedef int element;

typedef struct QueueNode {
      element item;
      struct QueueNode *link;
} QueueNode ;

typedef struct {
      int count;
      QueueNode *front;
      QueueNode *rear;
} LinkedQueueType;
```

Queue using linkedlist – enqueue Operation

queue 추가

```
void enqueue(queue *q, int value)
{
    node *tmp;
    tmp = malloc(sizeof(node));
    tmp->data = value;
    tmp->next = NULL;
    if(!isempty(q)) {
        q->rear->next = tmp;
        q->rear = tmp;
    }
    else {
        q->front = q->rear = tmp;
    }
    q->count++;  queue의 element 수 ++
}
```

Queue using linkedlist – dequeue Operation

queue 삭제

```
int dequeue(queue *q)
{
    node *tmp;
    int n = q->front->data;
    tmp = q->front;
    q->front = q->front->next;
    q->count--;
    free(tmp);
    return(n);
}
```

linked list : 선형리스트를 동적으로 표현하는 방식

linked stack → 우선순위 판단
linked queue → 순서 판단
→ 연산하는 시간을 선형적으로 줄이도록 해준다.

* 과제 : C로 작성 필수