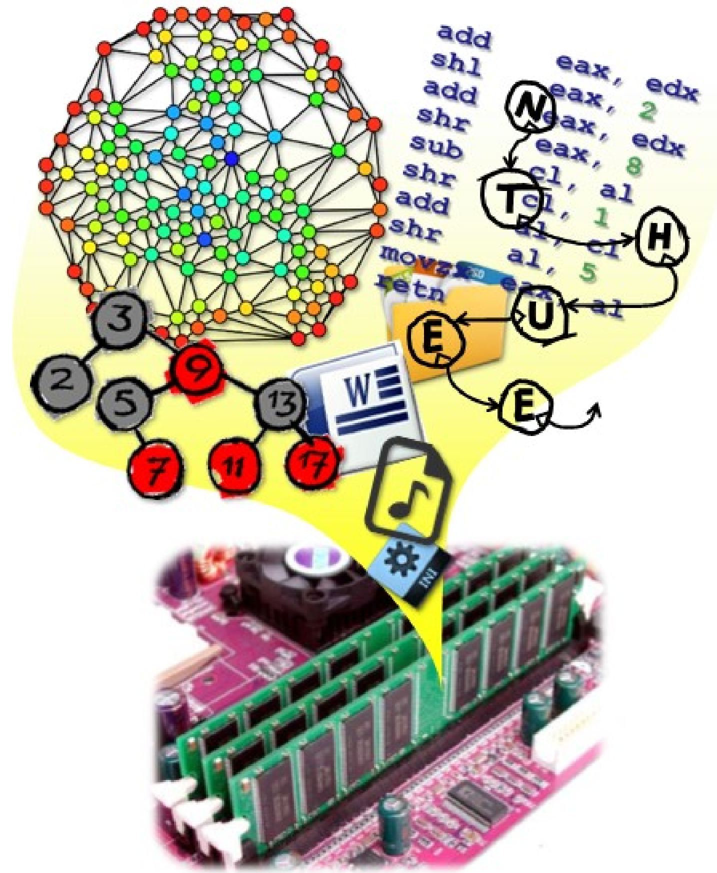
Data Structures

CH3 Stacks & Queues



Prof. Tai-Lang Jong

Office: Delta 928

Tel: 42577

email: tljong@mx.nthu.edu.tw

Spring 2021

Outline

- **3.1 Templates in C++**
- 3.2 The stack ADT
- 3.3 The queue ADT
- 3.4 Subtyping and inheritance in C++
- 3.5 A mazing problem
- 3.6 Evaluation of expressions

Observations

- Many codes look the same for different types
  - Sorting functions that handle
    - 32-bit integers
    - 64-bit integers
    - float
    - …
  - Sparse matrix classes that handle
    - 32-bit integers
    - 64-bit integers
    - float
    - …

quickSort(**int** a[], int lo, int hi);
quickSort(**float** a[], int lo, int hi);
quickSort(**double** a[], int lo, int hi);

```
class MatrixTerm {
friend SparseMatrix;
private:
  int row, col;
  int value;
};
```

```
class MatrixTerm {
friend SparseMatrix;
private:
  int row, col;
  float value;
};
```

Non-Template Solutions

- Implement the same behavior over and over
  - Hard to maintain code
  - Hard to globally modify code
- Write general code for a common base type
  - Lose the benefits of compiler's type checking
  - Incurs overhead
- Use macros (#define)
  - Sacrifice readability
  - Sacrifice debuggability

Template

- Template can be instantiated to any data type
  - So called "**parameterized types**“
  - The simple idea is to pass data type as a parameter so that we don’t need to write the same code for different data types.
  - Templates are expanded at compile time.
  - Compiler does type checking before template expansion
- C++ language supports
  - Template functions
  - Template classes

```
quickSort(int a[], int lo, int hi);
quickSort(float a[], int lo, int hi);
quickSort(double a[], int lo, int hi);

template<typename T>
quickSort(T a[], int lo, int hi){…}
quicksort<double>(a,0,100);
```

Function Template

- We write a generic function that can be used for different data types.
- Examples of function templates are sort(), max(), min(), printArray().

```
#include <iostream>
using namespace std;
// One function works for all data types. This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{   return (x > y)? x: y; }

int main(){
  cout << myMax<int>(3, 7) << endl; // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
  cout << myMax<char>('g', 'e') << endl; // call myMax for char
  return 0;
}
```

Template Function Example

```
void SelectionSort (int *a , const int n )
{
    for (int i = 0 ; i < n ; i++ )
    {
        int k = i;
        for ( int j = i + 1 ; j < n ; j++ )
            if ( a[j] < a[k] ) k = j;
                swap ( a[i], a[k] );
    }
}
```

```
template <class T>
void SelectionSort (T *a , const int n )
{
    for (int i = 0 ; i < n ; i++ )
    {
        int j = i;
        for ( int j = i + 1 ; j < n ; j++ )
            if ( a[j] < a[k] ) k = j;
                swap ( a[i], a[k] );
    }
}
SelectionSort(a, 5); //invoking
```

- template <class T> is identical to template <typename T>
- It is a convention to use "T", but one can use any other name

Template Function Example

- Can have more than one arguments to templates
- Can specify **default value** for template arguments

```
template <class T>
int anotherFunctionWithNoT(){}
void SelectionSort (T *a , const int n )
{
   for (int i = 0 ; i < n ; i++ )
   {
      int j = i;
      for ( int k = i + 1 ; k < n ; k++ )
         if ( a[k] < a[j] ) j = k;
            swap ( a[i], a[j] );
   }
} //error
```

```
template <typename T, class U>
void someFunct (T *a , U n )
{
   //.....
} //OK

template <typename T, class U=int>
void someFunct (T *a , U n )
{
   //.....
}
```

Template Function Example

- Can pass **nontype parameters** to templates

```
template <class T, int max>
int arrMin(T arr[], int n)
{
    int m = max;
    for (int i = 0; i < n; i++)
        if (arr[i] < m) m = arr[i];
    return m;
}
```

```
int main()
{
  int arr1[]  = {10, 20, 15, 12};
  int n1 = sizeof(arr1)/sizeof(arr1[0]);
  char arr2[] = {1, 2, 3};
  int n2 = sizeof(arr2)/sizeof(arr2[0]);
// Second template parameter to arrMin
// must be a constant
  cout << arrMin<int, 10000>(arr1, n1) <<
endl;
  cout << arrMin<char, 256>(arr2, n2);
  return 0;
}
```

Class Template

- Like function templates, class templates are useful when a class defines something that is independent of the data type.
- Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

```
#include <iostream>
using namespace std;

template <typename T>
class Array {
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};
```

```
template <typename T>
Array<T>::Array(T arr[], int s)
{
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}
```

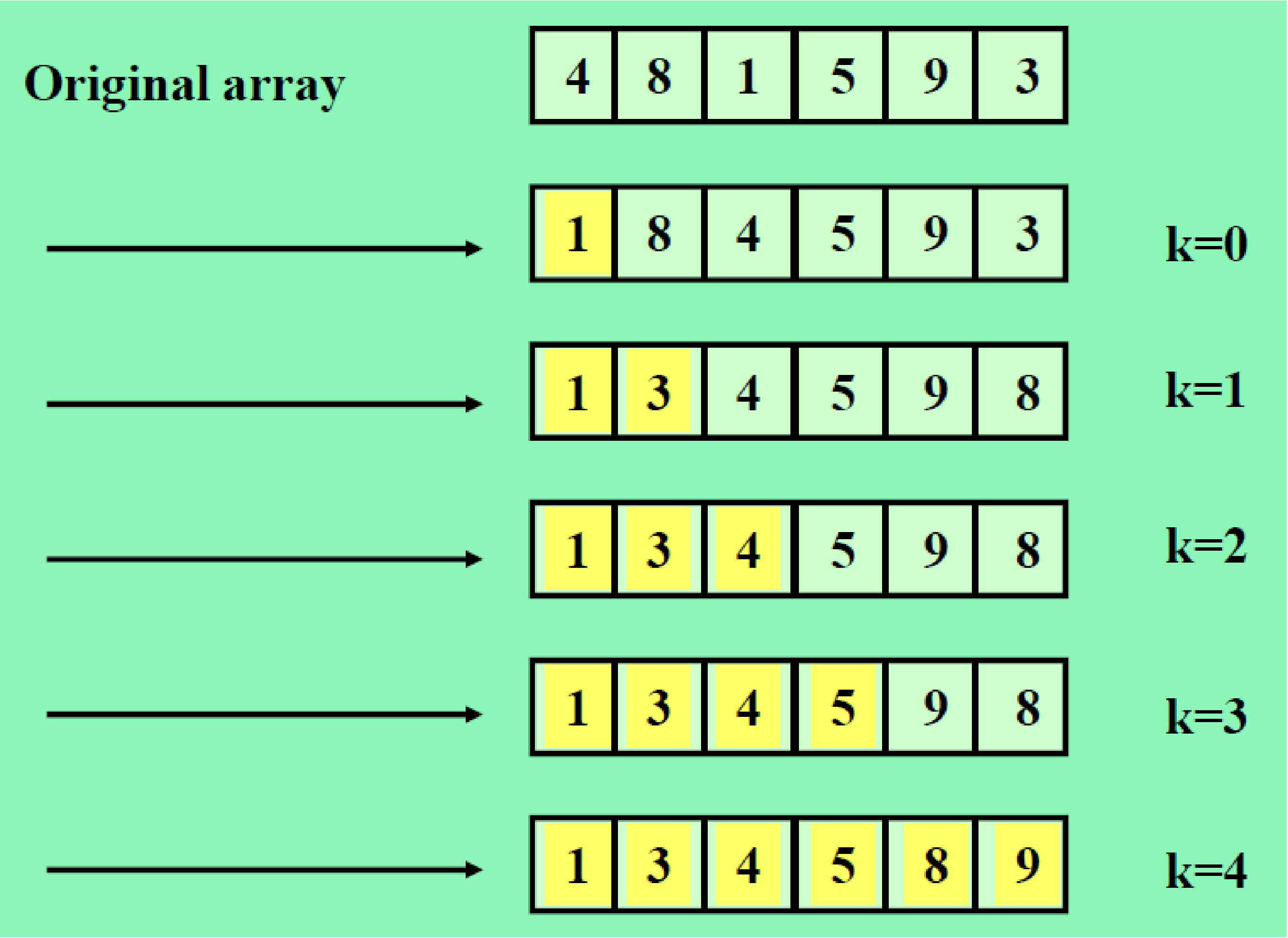
```
template <typename T>
void Array<T>::print()
{   for (int i = 0; i < size; i++)
        cout<<" "<<*(ptr + i);
    cout<<endl;
}
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

Class Template

- More than one arguments to templates
- Specify default value to template

```
#include<iostream>
using namespace std;
template<class T, class U = char>
class A {
    T x;
    U y;
public:
    A() { cout<<"Constructor Called"<<endl; }
};
int main() {
    A<char, char> a;   A<char> b;
    A<int, double> c;
    return 0;
}
```

Example of Selection Sort



Selection Sort Using Template

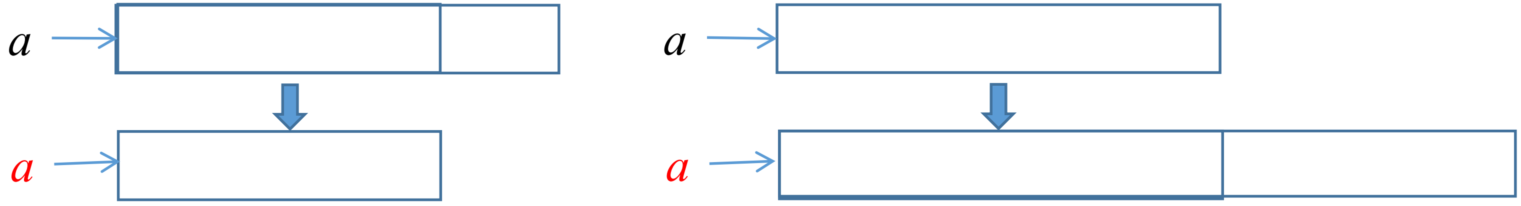
```
template <class KeyType>
void sort (KeyType *a, int n)
// sort the n KeyType a[0] to a[n-1] into nondecreasing order
{
    for(int k=0; k < n; k++){
        int smallest = k;
        // find the smallest KeyType in a[k] to a[n-1]
        for (int j = k+1; j < n; j++){
            if(a[j] < a[smallest]) smallest = j;
        }
        KeyType temp = a[k];
        a[k] = a[smallest];
        a[smallest] = temp;
    }
}
```

Selection Sort Using Template (cont.)

```
main(){
      float real_array[100];
      int int_array[250];
      . …
      // assume that the arrays have been initialized
      sort(real_array, 100);
      sort(int_array, 250);
      . …
}
```

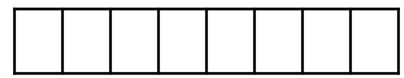
Array Resizing Using Template

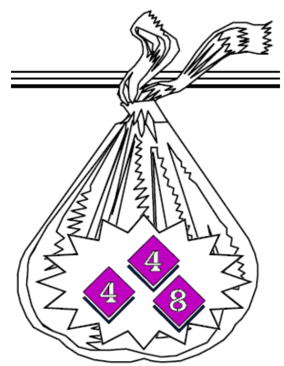
```
template <class T>
void ChangeSize1D ( T*& a, const int oldSize, const int newSize )
{
	if ( newSize < 0 ) throw "New length must be >=0";

	T* temp = new T [ newSize ];			// new array
	int number = min ( oldSize, newSize );	// number to copy
	copy (a, a + number, temp);

	delete [] a;	// deallocate old memory
	a = temp;
}
```

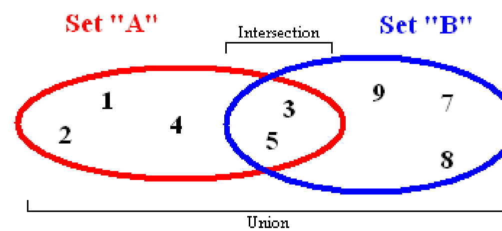


Container Class

- A container class is a class that represents a data structure that contains or stores a number of data objects.
- Container examples:
  - List
  - Bag (Multiset)
  - Set
  - Map (Dictionary)
  - Multimap

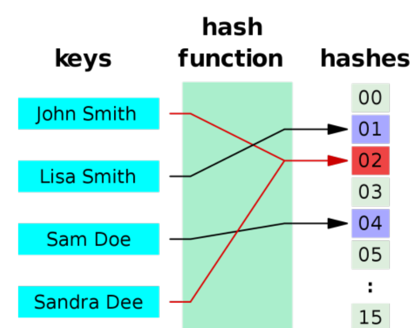






KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Aug → 37.3



Bag Class

- A Bag: unordered collection of objects that may have duplicates, where the order of insertion is completely irrelevant, e.g., a bag of marbles.
- Operations you can do with a bag include
  - **Inserting** a new value, (Push)
  - **Removing** a value, (Pop)
  - **Testing** to see if a value is held in the collection (IsIn),
  - Determining the **number of elements** in the collection.
  - Sometimes the ability to **loop over** the elements in the container

Set Class

- A Set extends the Bag in two important ways.
  - First, the elements in a set must be unique;
    - Adding an element to a set when it is already contained in the collection will have no effect.
  - Second, the set adds a number of operations that combine two sets to produce a new set:
    - Set union
    - Set intersection
    - Set difference

Bag Class Implementation

```
class Bag
{
public:
   Bag ( int bagCapacity = 10 ); // constructor
   ~Bag( );                    // destructor
   int Size( ) const;    // return number of elements in bag
   bool IsEmpty() const;
   int Element( ) const;//return an element that is in the bag
   void Push(const int);// add an integer into the bag
   void Pop();          // delete an integer in the bag

private:
   int *array; // dynamic array for Bag
   int capacity;   // capacity of array
   int top;// array position of top element
};
```

const member function

Specifies that the function does not modify the object for which it is called.

const Bag emptyBag;
emptyBag.size();   //valid
emptyBag.push(1); //error

Bag Class (for integers)

```
Bag::Bag (int bagCapacity)
:capacity ( bagCapacity )
{
    if ( capacity < 1 )
        throw "Capacity must be > 0";
    array = new int [ capacity ];
    top = -1; // empty
}

Bag::~Bag ( )
{ delete [] array; }

inline int Bag::Size( ) const
{ return top + 1; }

bool Bag::IsEmpty() const
{
    return (Size() == 0);
}
```

Initialization list

initialize member variables when they are created rather than afterwards

Bag Class (for integers)

```
inline int Bag::Element ( ) const
{
    if ( IsEmpty ( ) )
        throw “Bag is empty”;

    return array [0]; // always return 0th element
}

void Bag::Push (const int x)
{
    if (capacity == top + 1) // array is full
    {
        ChangeSize1D (array, capacity, 2 * capacity);
        capacity *= 2;
    }
    array[++top] = x;
}
```

Bag Class (for integers)

```
void Bag::Pop ()
{
    if (IsEmpty()) throw “Bag is empty, cannot delete”;
    int deletePos = top/2; // always delete top/2th element
    copy(array + deletePos + 1, array + top + 1,
                         array + deletePos);
    top--;
}
```

- Container classes are particularly suitable for implementation using templates, because the algorithms for basic container class operations are usually independent of the type of objects that container class contains.

Template Class for Bag

```
template<class T>
class Bag
{
public:
    Bag( int bagCapacity = 10 );      // constructor
    ~Bag( );                           // destructor
    int Size( ) const;    // return number of elements in bag
    bool IsEmpty() const;
    T& Element( ) const; // return an element that is in the bag
    void Push(const T&); // add an integer into the bag
    void Pop();
private:
    T *array;
    int capacity;          // capacity of array
    int top;               // array position of top element
};
```

Template Bag

```
template<class T>
Bag<T>::Bag(int bagCapacity) : capacity (bagCapacity)
{
    if (capacity < 1)
        throw "Capacity must be > 0";
    array = new T [capacity];
    top = -1;
}

template <class T>
Bag<T>::~Bag( )
{delete [] array;}

template <class T>
inline int Bag<T>::Size( ) const
{ return top + 1; }
```

Template Bag

```
template <class T>
bool Bag<T>::IsEmpty() const
{
    return (Size() == 0);
}


template <class T>
T& Bag<T>::Element( ) const
{
    if ( IsEmpty() )
        throw "Bag is empty";
    return array [0];
}
```

Template Bag

```
template <class T>
void Bag<T>::Push(const T& x)
{
    if (capacity == top + 1) {
        ChangeSize1D (array, capacity, 2 * capacity);
        capacity *= 2;
    }
    array [++top] = x;
}

template <class T>
void Bag<T>::Pop( )
{
    if ( IsEmpty() ) throw "Bag is empty, cannot delete";
    int deletePos = top/2;
    copy(array+deletePos+1, array+top + 1, array + deletePos);
    array[top--].~T(); // destructor for T
}
```

Use of the Template Bag

```
int main()
{
    Bag<int> myIntBag;
    myIntBag.Push(1);
    myIntBag.Push(9);
    cout << myIntBag.Size() << endl;
    cout << myIntBag.Element();

    Bag<float>  myFloatBag;
    for(int i=0; i<10; i++)
        myFloatBag.Push(1.0/i);

    Bag<Bag<int> > myManyIntBag;

    myManyIntBag.Push(myIntBag);
    …
    return;
}
```

Outline

- 3.1 Templates in C++
- **3.2 The stack ADT**
- 3.3 The queue ADT
- 3.4 Subtyping and inheritance in C++
- 3.5 A mazing problem
- 3.6 Evaluation of expressions

A Simple Summary

- A collection of data objects organized sequentially
  - List, linear list, ordered list: $\mathbf{A = (}\boldsymbol{a_0, a_1, \ldots, a_{n-1}}\mathbf{)}$
  - Stack: Last In First Out (LIFO)
  - Queue: First In First Out (FIFO)
    - **Array** implementation
    - **Linked list** implementation
- A collection of unordered data objects
  - Bag (Multiset): can have duplicate data objects
  - Set: data object must be unique
- A collection of <key, value> pair data objects
  - Map (Dictionary): key-value mapping must be unique
  - Multimap: one key –to-multiple values mapping

List

Insert x at position 5

O(n)

front rear

1	2	3	4	5	6	7	8
a	b	c	d	e	f	g	h

1	2	3	4	5	6	7	8	9
a	b	c	d	x	e	f	g	h

Delete data at position 3

O(n)

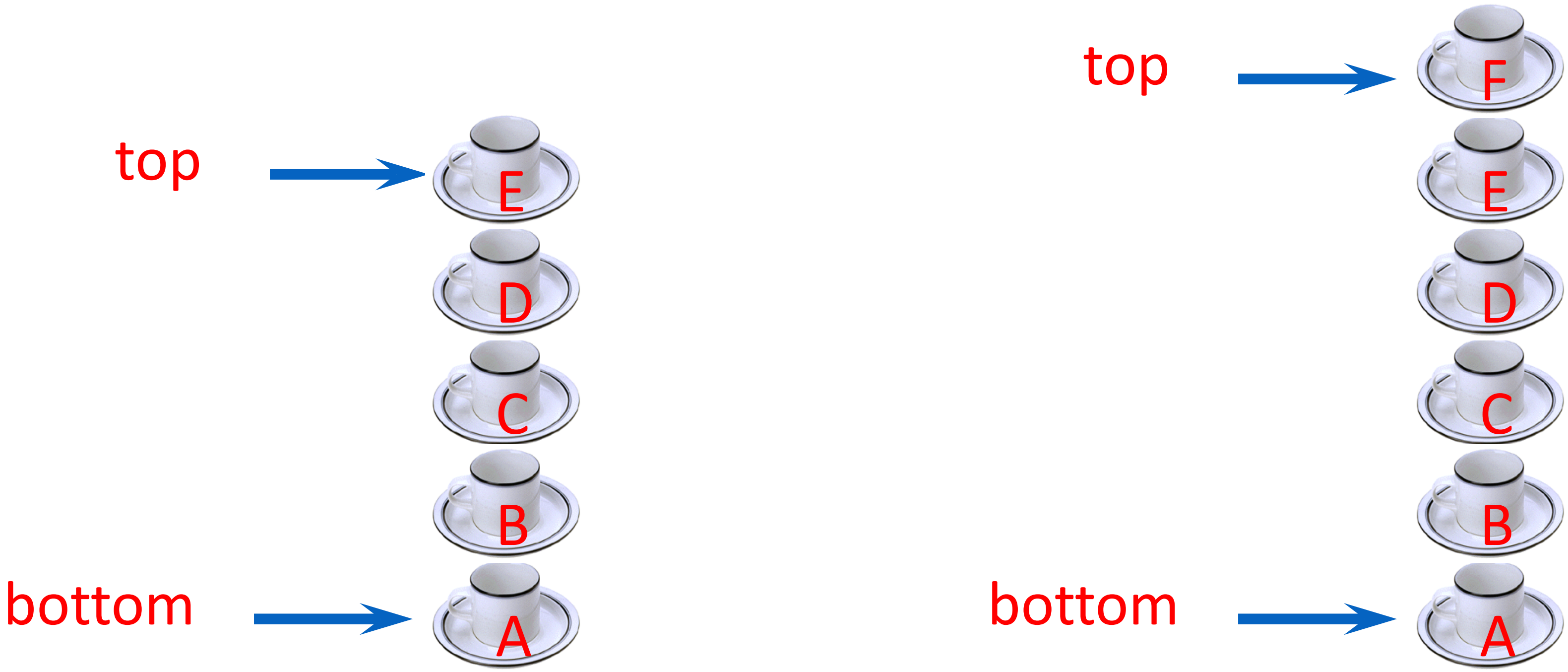
1	2	3	4	5	6	7	8	9
a	b	c	d	x	e	f	g	h

1	2	3	4	5	6	7	8	9
a	b	d	x	e	f	g	h	

Stack

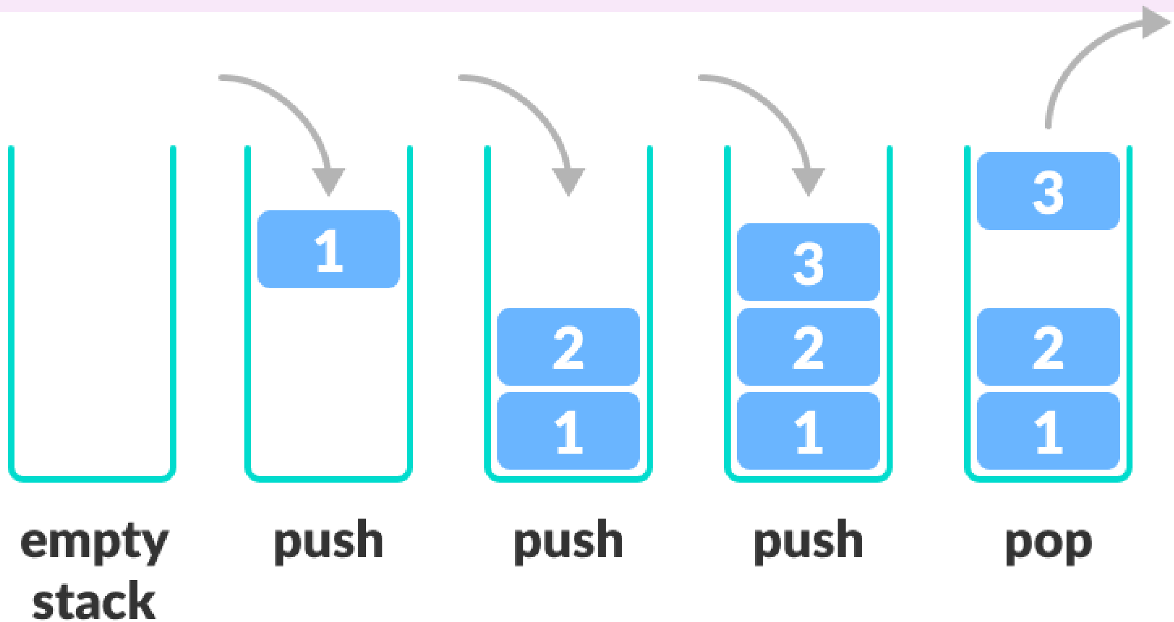
- Ordered List (linear list)
  - Suppose A = (a_0, a_1,…, a_{n-1}), where n≧0
  - a_i is called an *atom,* or an *element*
- Stack: Last-In First-Out (LIFO)
  - is a special case of ordered list
  - One end is called *top*, the other end called *bottom*
  - the **additions** and **deletions** are made at the ***top*** end only
- Example
  - Given a stack S = (a_0, a_1,…, a_{n-1})
  - a_0 is the *bottom* element
  - a_{n-1} is the *top* element

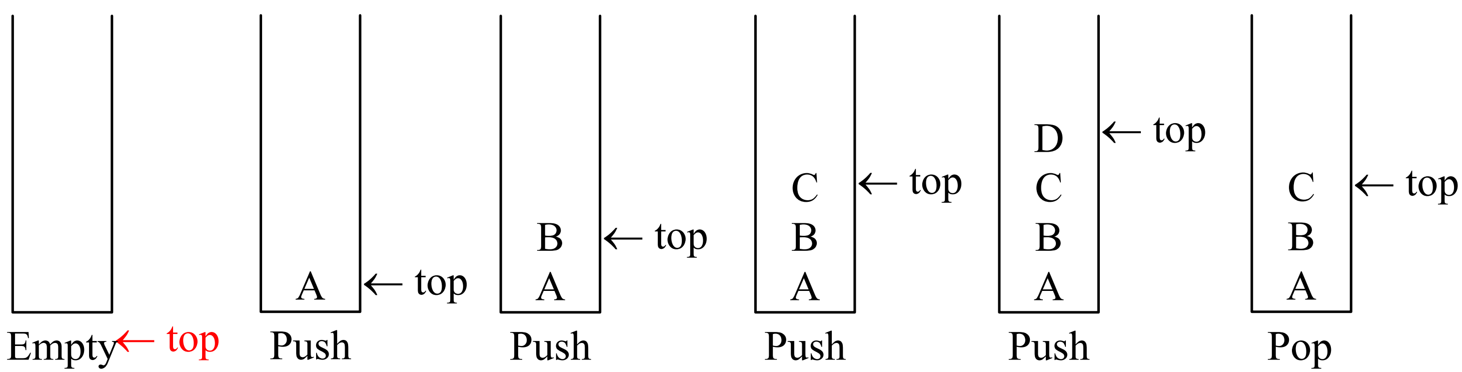
Stack of Cups



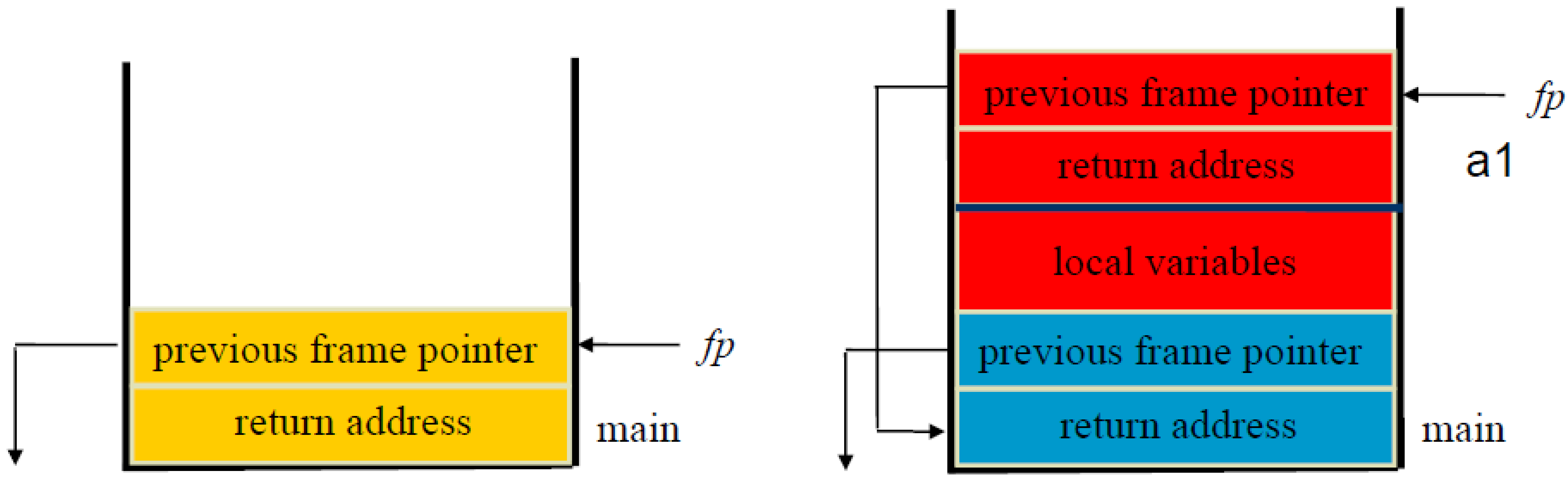
- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO list.

Stack


1
2
1
3
2
1
3
2
1
empty stack
push
push
push
pop


A
← top
B
A
← top
C
B
A
← top
D
C
B
A
← top
C
B
A
← top
Empty← top
Push
Push
Push
Push
Pop

Applications of Stack: Stack Frame of Function Call



System stack after function call

Parentheses Matching

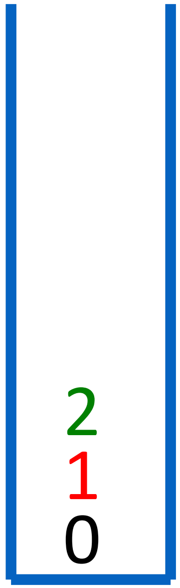
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - Right parenthesis at 5 has no matching left parenthesis
  -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

Parentheses Matching

- 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

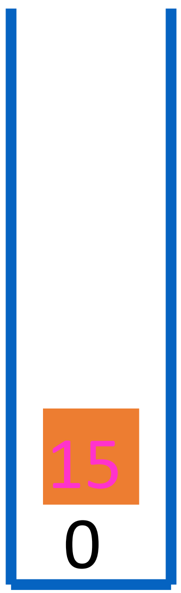
Example

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



Example

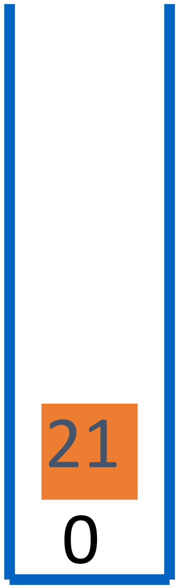
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



(2,6) (1,13)

Example

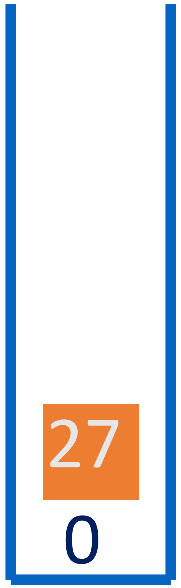
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



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

Example

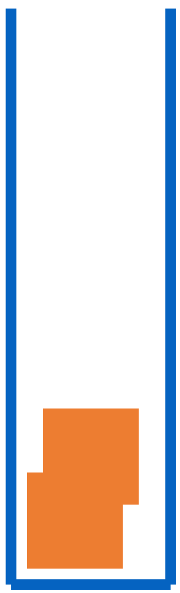
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



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

Example

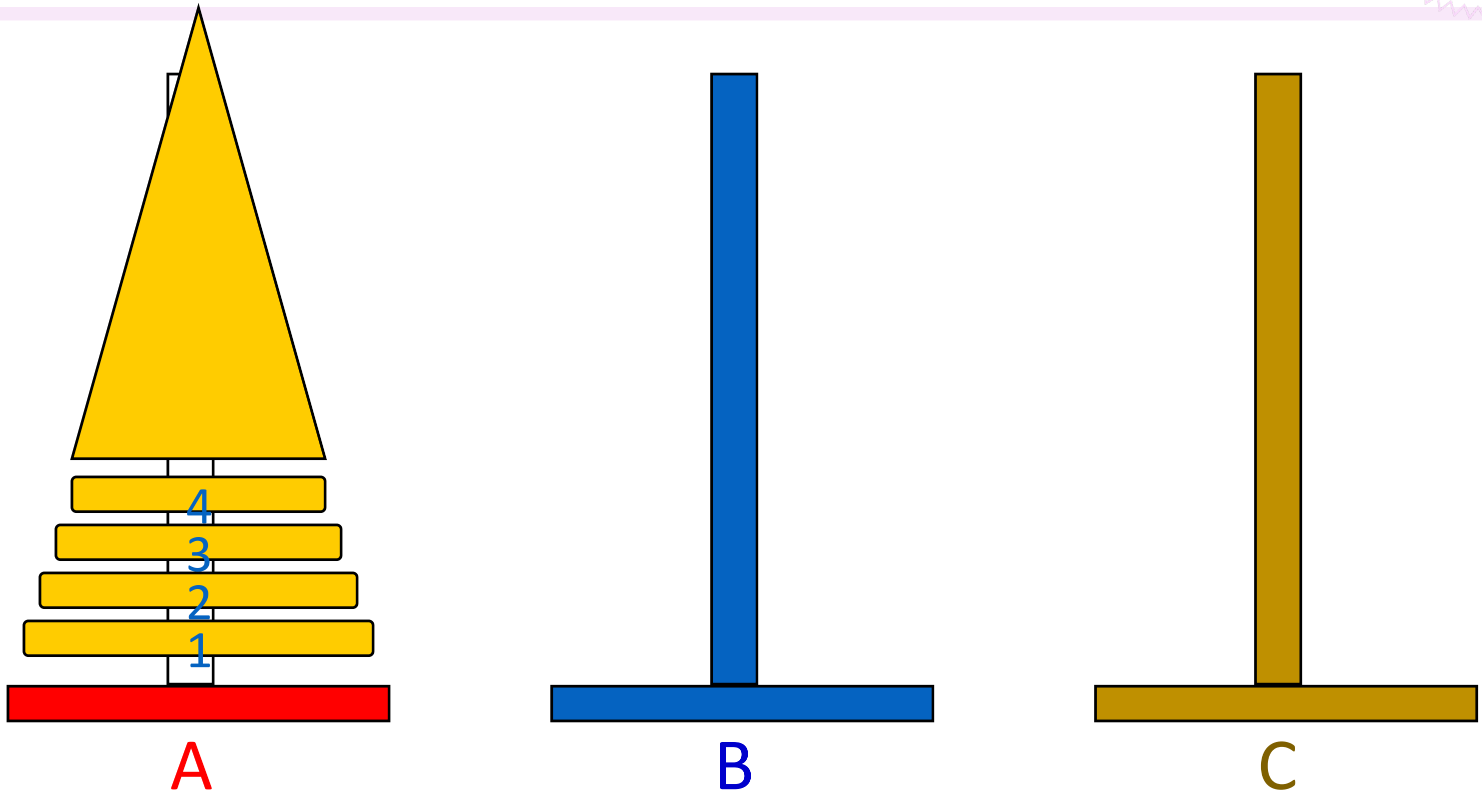
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)



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

- and so on

Towers of Hanoi/Brahma



- 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

Stacks

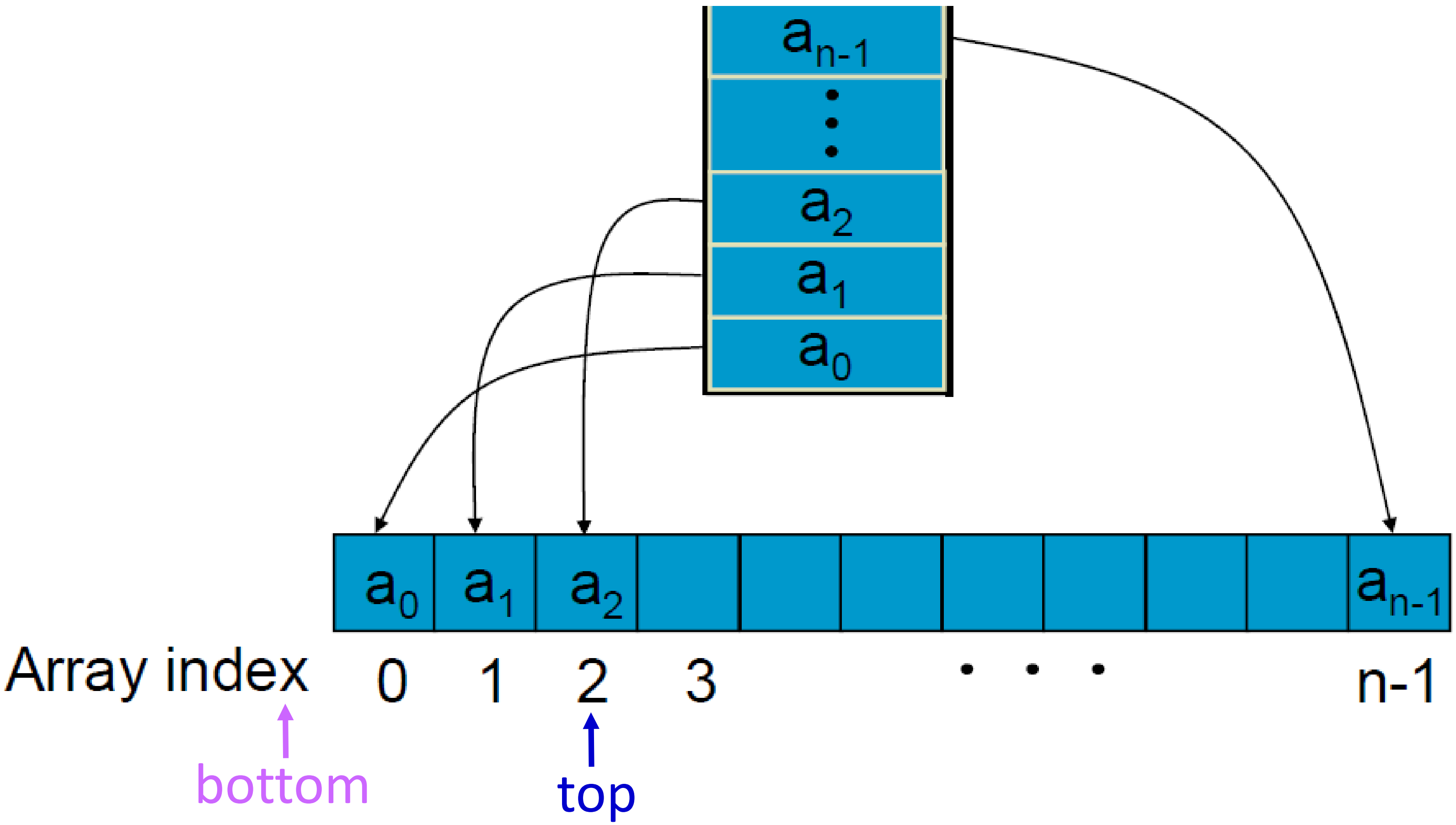
- Standard operations:
  - IsEmpty … return true iff stack is empty
  - Top … return top element of stack
  - Push … add an element to the top of the stack
  - Pop … delete the top element of the stack

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

Stack ADT

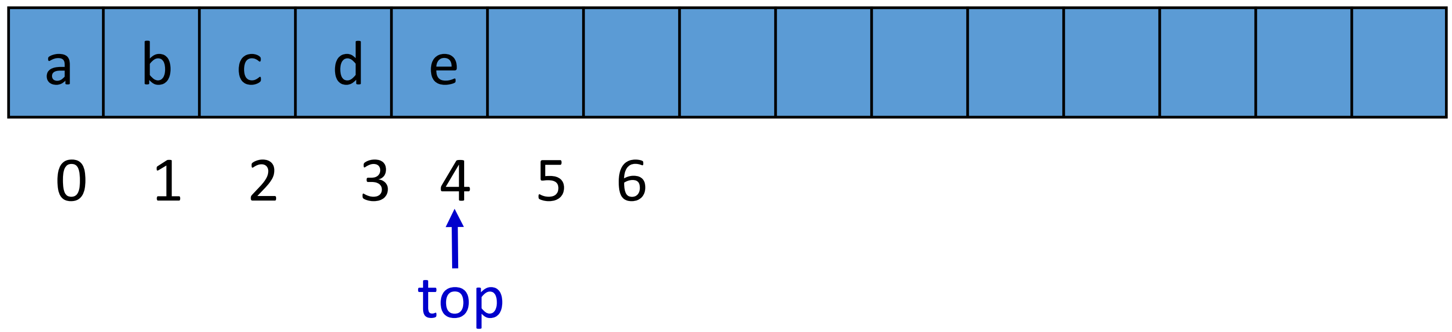
```
template < class T >
class Stack
{//a finite ordered list w. ≥0 elem
public:
    Stack (int stackCapacity = 10);

    bool IsEmpty( ) const;

    void Push(const T& item);
    // add an item into the stack

    void Pop( );
    // delete an item

    T& Top() const;
    // return top element of stack
private:
    int  top, capacity;
    T*   stack;
};
```

Implementation of Stack by Array



How to check whether a stack is full or empty? – top, capacity

Stacks

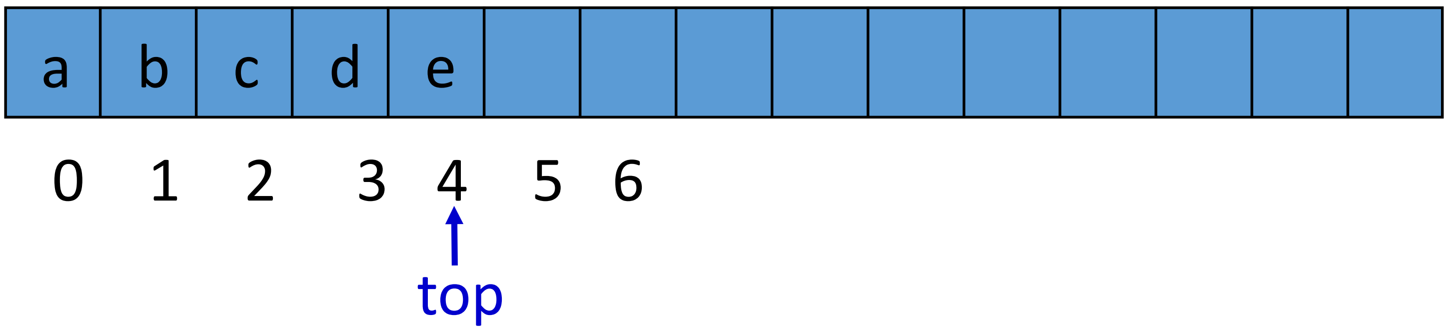


- Stack top is at element e
- IsEmpty() => check whether top >= 0

  O(1) time
- Top() => If not empty return stack[top]

  O(1) time

Stacks



- Push(theElement) =>

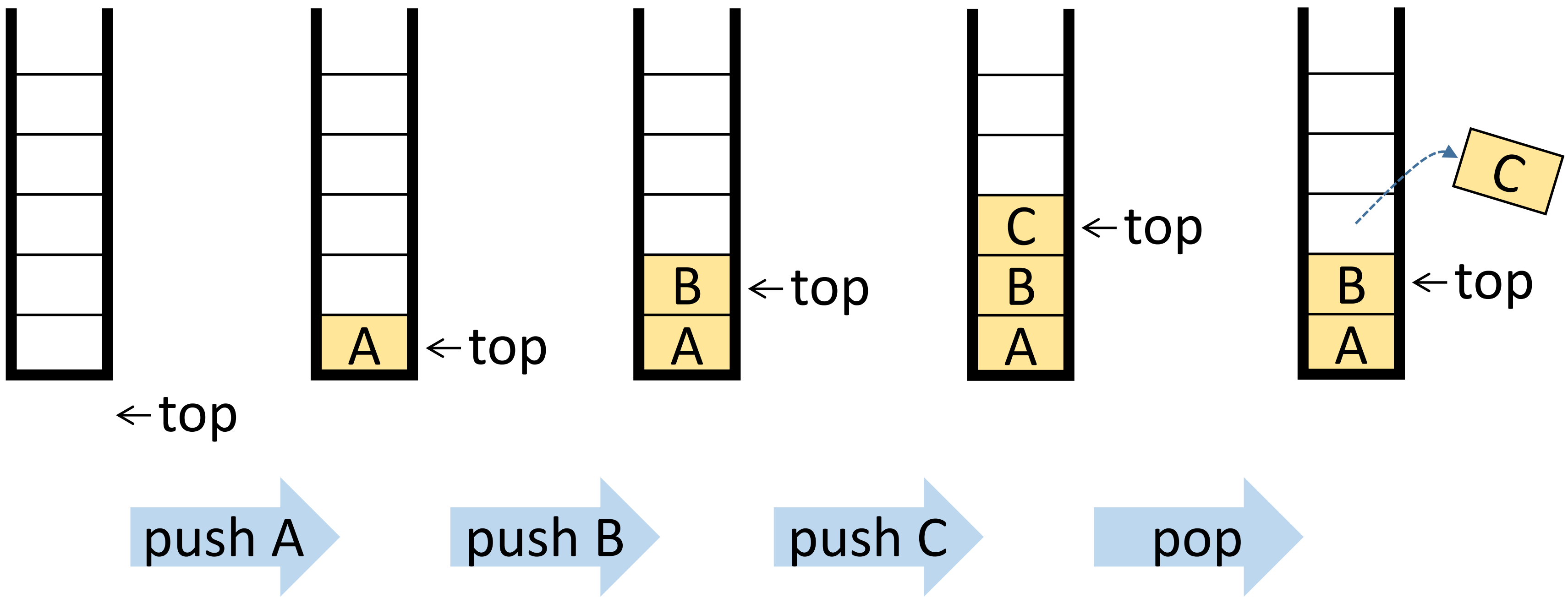
  If array is not full => O(1) time

  If array full (top == capacity – 1) increase capacity and then add at stack[top+1] => O(capacity) time when full;

- Pop() => if not empty, delete from stack[top]

  O(1) time

Stack



Time complexity

- push(): Θ(1)
- pop(): Θ(1)

Template Stack Implementation

```
template <class T>
Stack<T>::Stack(int  stackCapacity=10):capacity(stackCapacity)
{
    if (capacity < 1) throw “Stack capacity must be > 0”;
    stack = new T[capacity];
    top = -1; // indicate empty stack
}

template <class T>
inline bool Stack<T>:: IsEmpty() const
{   return top == -1;}

template <class T>
inline T& Stack<T>:: top() const
{
   if ( IsEmpty() ) throw “Stack is empty”;
   return stack[top];
}
```

Template Stack Implementation

```
template <class T>
void Stack<T>::Push(const T& x)
{// add x to stack
    if (top == capacity - 1) {
        ChangeSize1D (stack, capacity, 2 * capacity);
        capacity *= 2;
    }
    stack[++top] = x;
}

template <class T>
void Stack<T>::Pop( )
{
    if ( IsEmpty() ) throw “Bag is empty, cannot delete”;
    int deletePos = top/2;
    stack[top--].~T(); // destructor for T
}
```

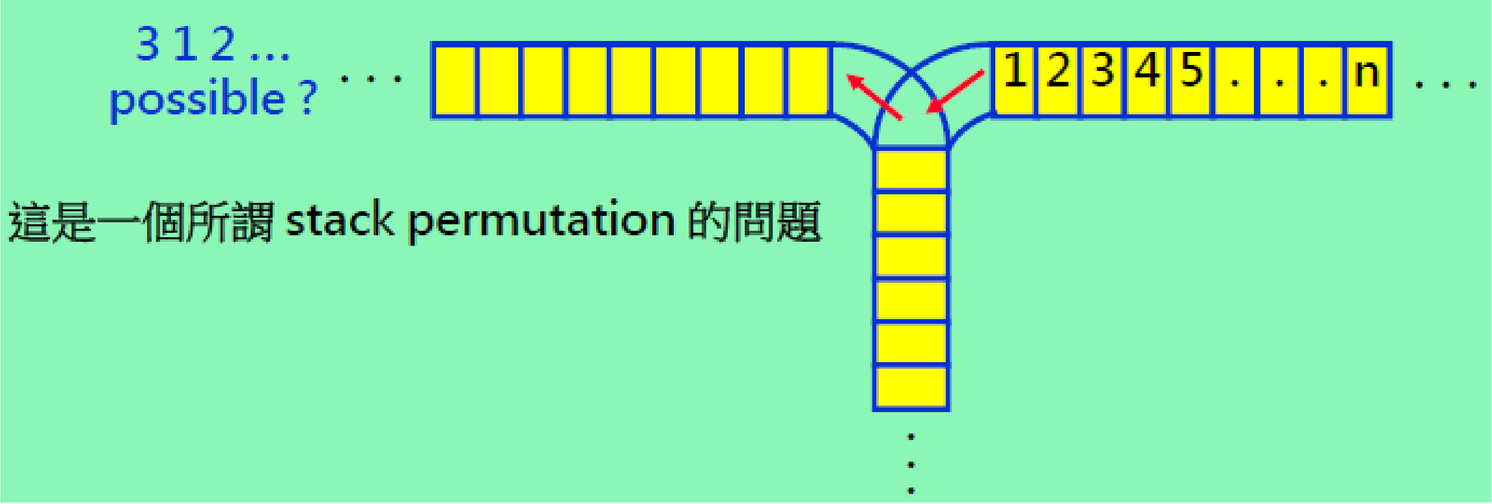
Railroad Switching System

- Switching Rule
  - Initial: train 1, 2, …, *n in the top right track segment*
  - Movement:

    (1) from top-right to the vertical segment one at a time

    (2) from the vertical to the top-left segment one at a time

    (3) The vertical segment operates like a stack
- Question: What output permutations are not possible?



Outline

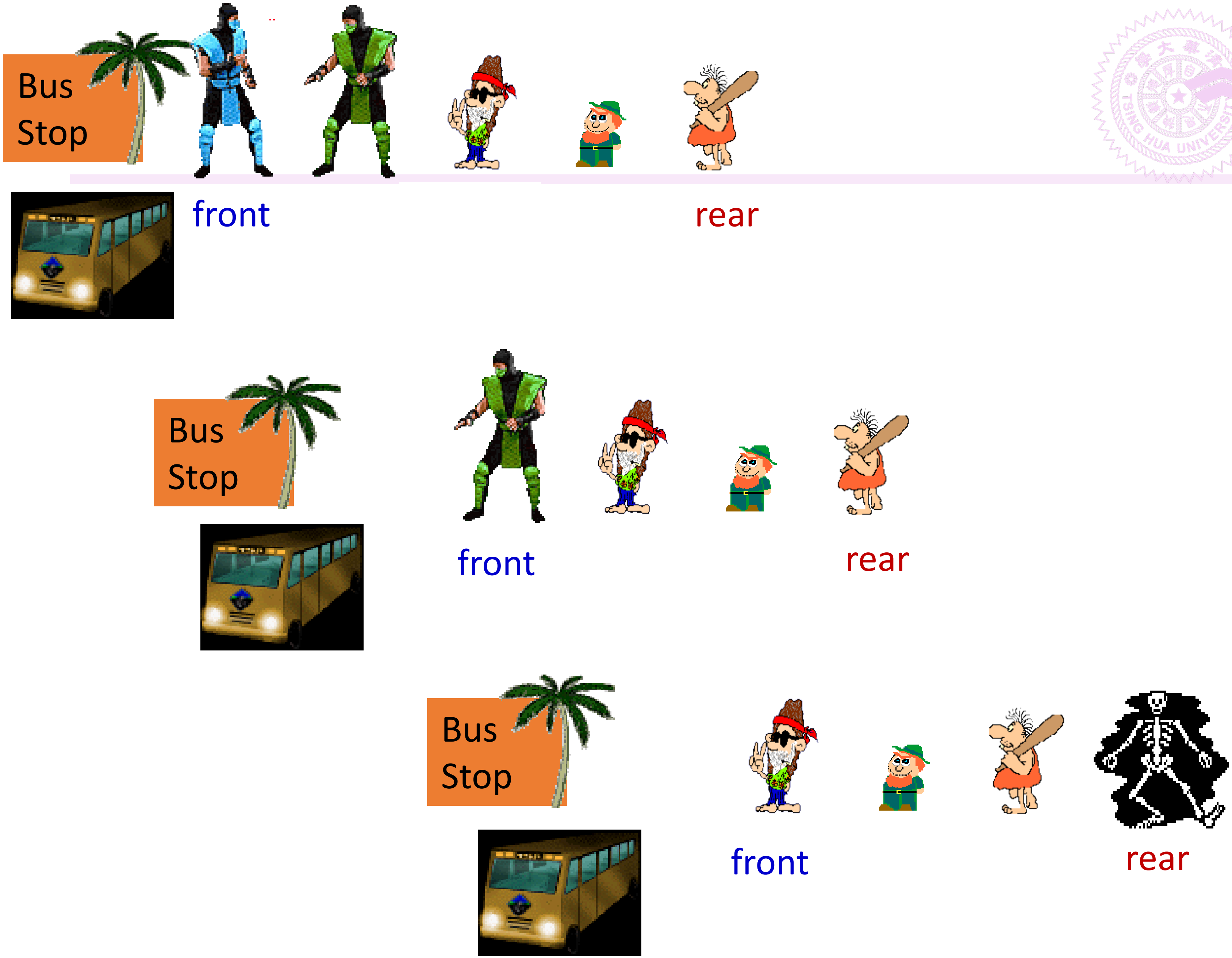
- 3.1 Templates in C++
- 3.2 The stack ADT
- **3.3 The queue ADT**
- 3.4 Subtyping and inheritance in C++
- 3.5 A mazing problem
- 3.6 Evaluation of expressions

Queue

- A Queue
  - Is an ordered (linear) list
  - One end called ***front***
  - The opposite end called ***rear***
  - **Insertions** take place at the ***rear*** **only**
  - **Deletions** take place from the ***front*** **only**
  - Is also called First-In First-Out (FIFO)
- Example
  - Given a queue Q = (a_0, a_1,…, a_{n-1})
  - a_0 is the *front* element, a_{n-1} is the *rear* element
  - a_i is *behind* a_{i-1} for $1 \leqq i \leqq n$
  - Delete at *front*, insert at *rear*

Queue

- Bus Stop Queue

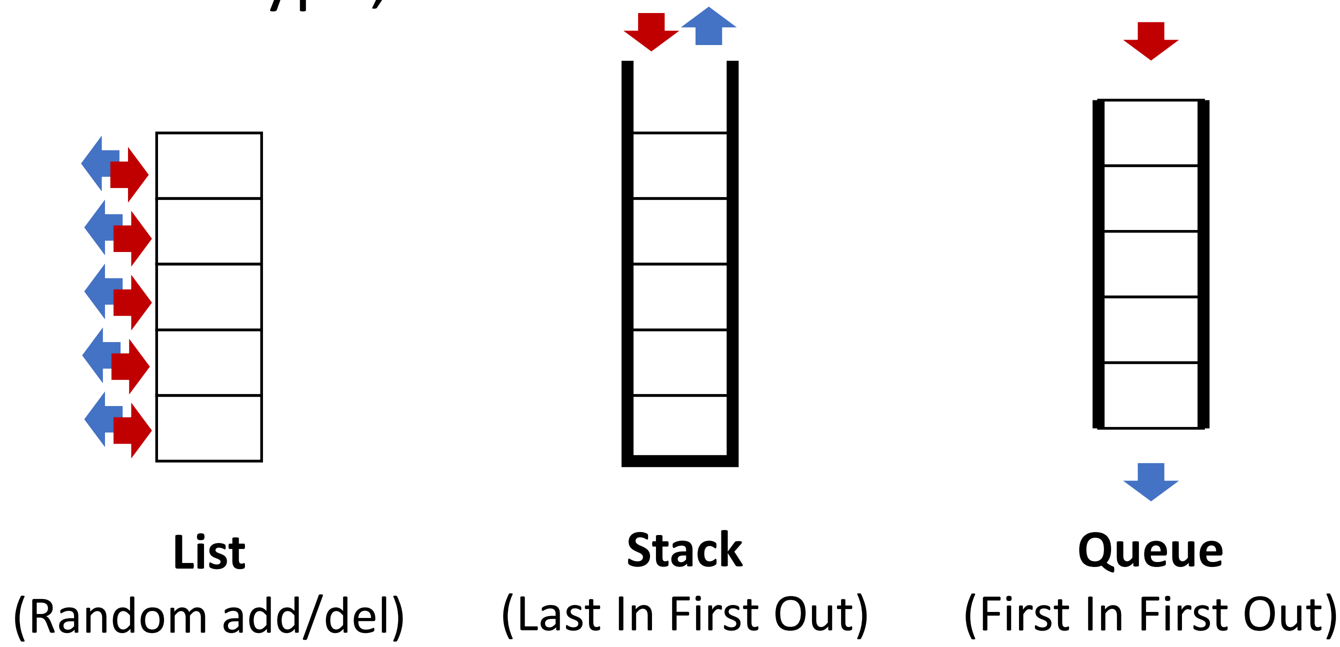

Bus
Stop
front
rear
Bus
Stop
front
rear
Bus
Stop
front
rear

Queue Operations

- IsEmpty … return true iff queue is empty
- Front … return front element of queue
- Rear … return rear element of queue
- Push … add an element at the rear of the queue
- Pop … delete the front element of the queue

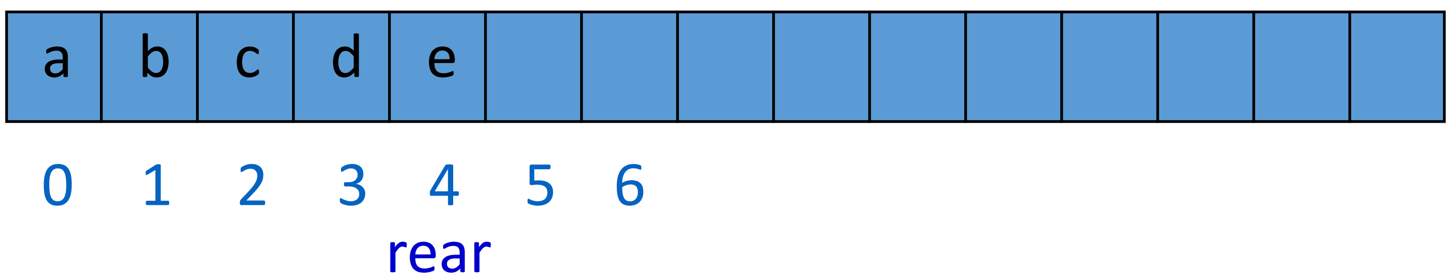
List, Stack, and Queue

- Stack & Queue are two frequently used data structures
- They are special cases of the more general data structure type, **lists**



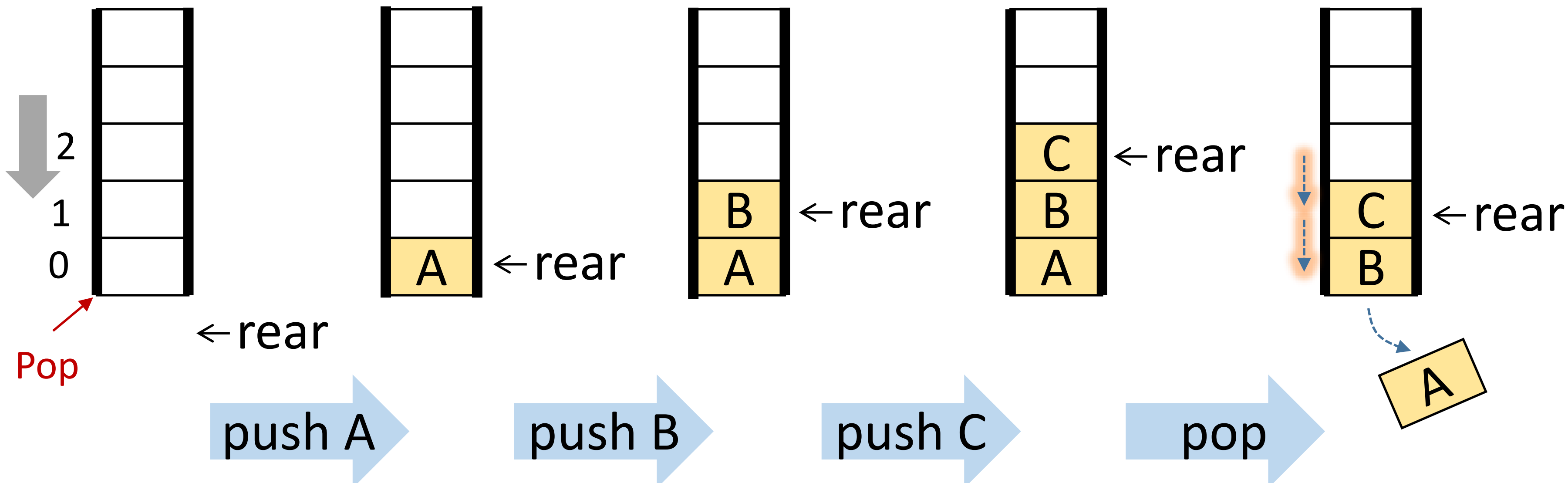
Queue in an Array

- 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.



- Pop => remove queue[0], shift left queue one place.
  - O(queue size) time
- Push => if there is capacity, add at right end
  - O(1) time

Queue (Single Pointer)

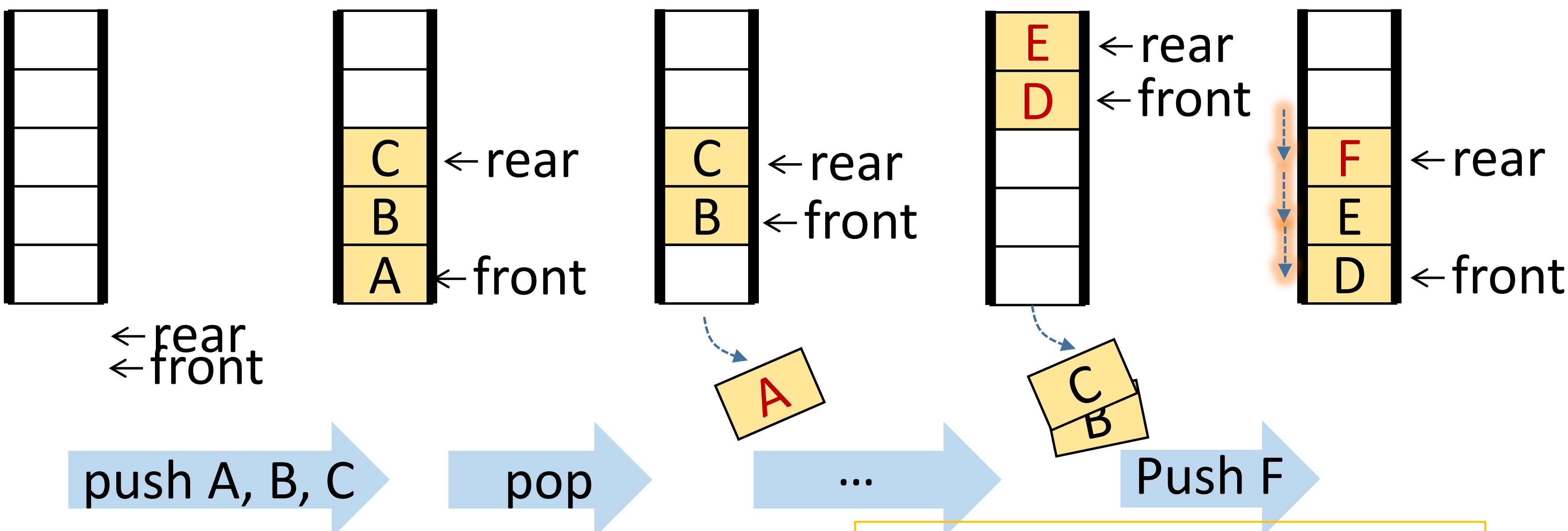


rear = -1 => queue empty
the last element in queue[rear]
The front element in queue[0]

Time complexity

- push(): Θ(1) (exclusive of array resizing time)
- pop(): Θ(size)

Queue (Dual Pointers)

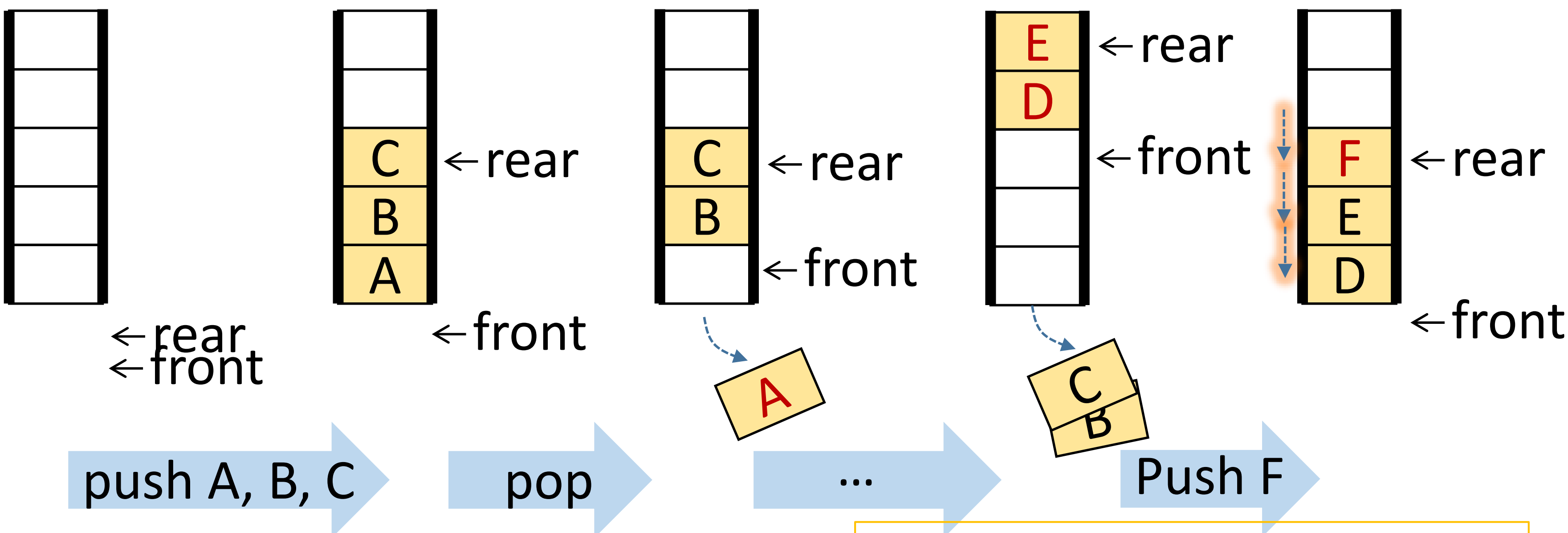


The last element in queue[rear]
The front element in queue[front]

Time complexity

- push(): O(size) (when no array resizing needed)
  - When the rear pointer reaches the boundary and a push occurs, data need to be moved
- pop(): Θ(1)

Queue (Dual Pointers)



The last element in queue[rear]
The front element in queue[front+1]

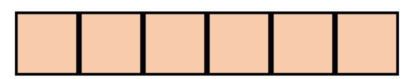
Time complexity

- push(): O(size)   (when no array resizing needed)
    - When the rear pointer reaches the boundary and a push occurs, data need to be moved
- pop(): Θ(1)

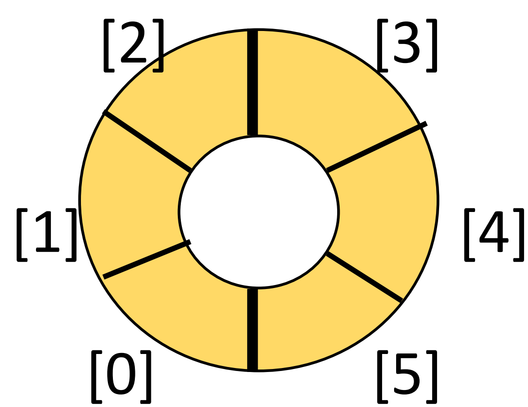
O(1) Pop and Push

- To perform each operation in O(1) time (excluding array doubling), we use a circular representation – circular queue
- Use a 1D array queue

queue[]

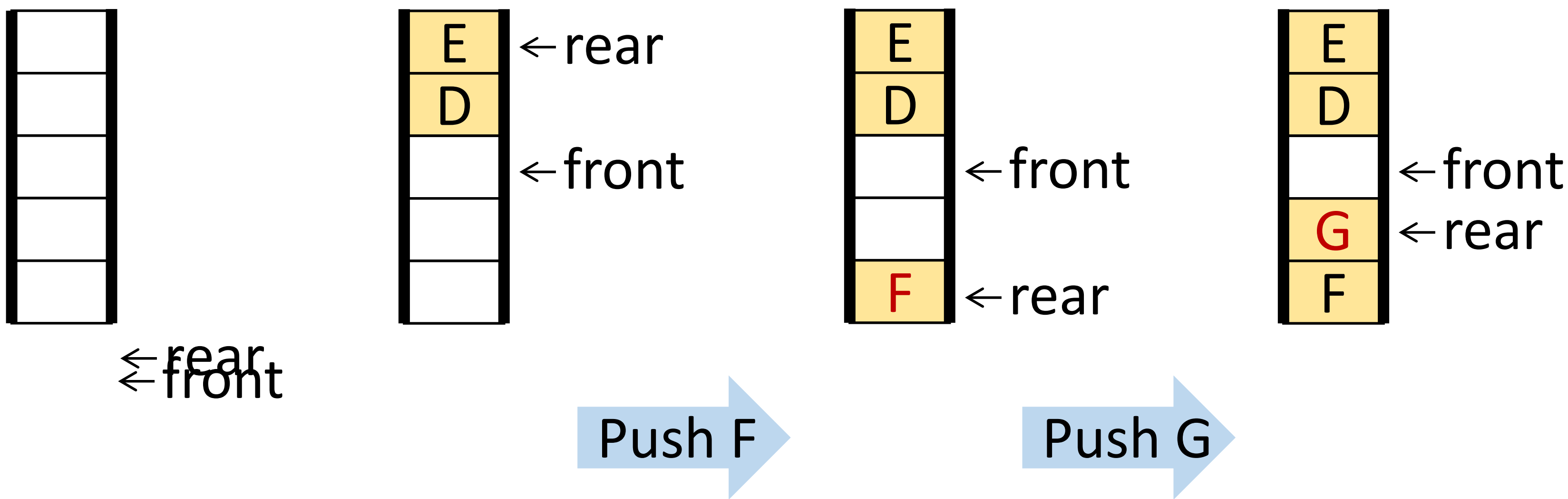


- Circular view of array



Circular Queue

- Permit the queue to wrap around the end space



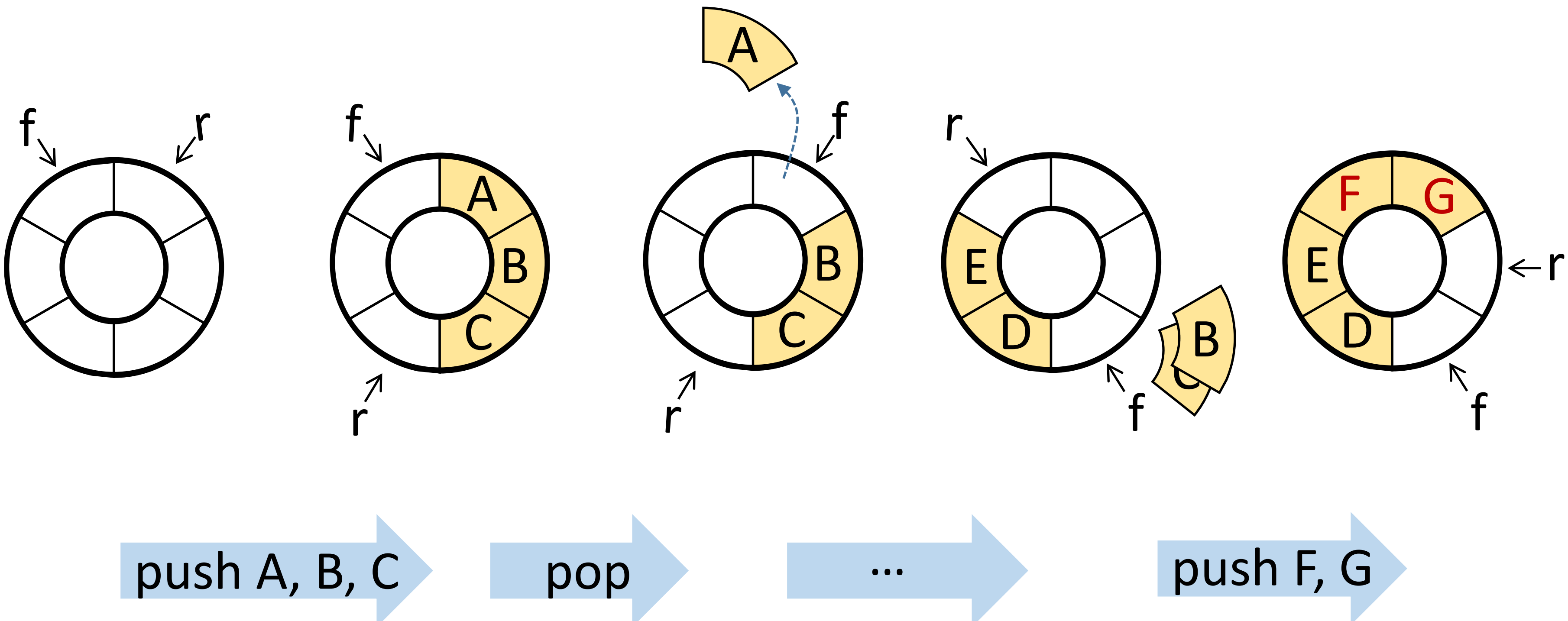
Time complexity

- push(): $\Theta(1)$
- pop(): $\Theta(1)$

Note that in this version of circular buffer, the position that the **front** pointer points to is a **dead space**. A slot is deliberately unused.

- Otherwise, we cannot determine whether the queue is empty or full.

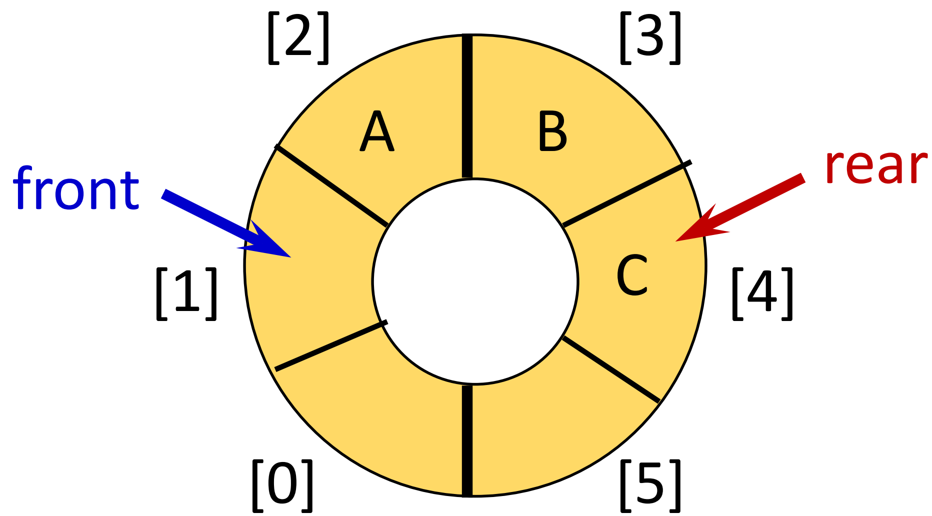
Circular Queue (Circular Illustration)



- Use integer variables front (f) and rear (r)
- front is one position counterclockwise from first element
- rear gives position of last element

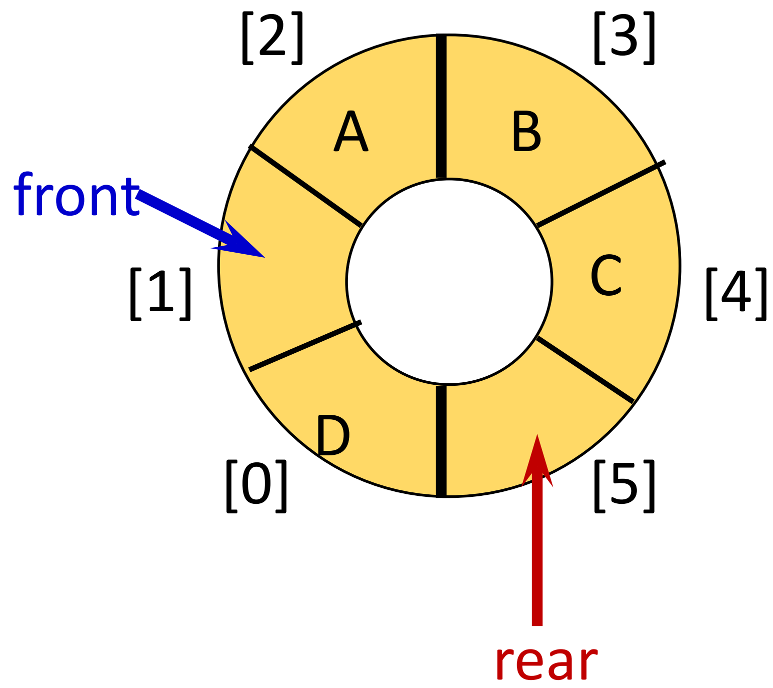
Push An Element (1/2)

- Move rear one clockwise.



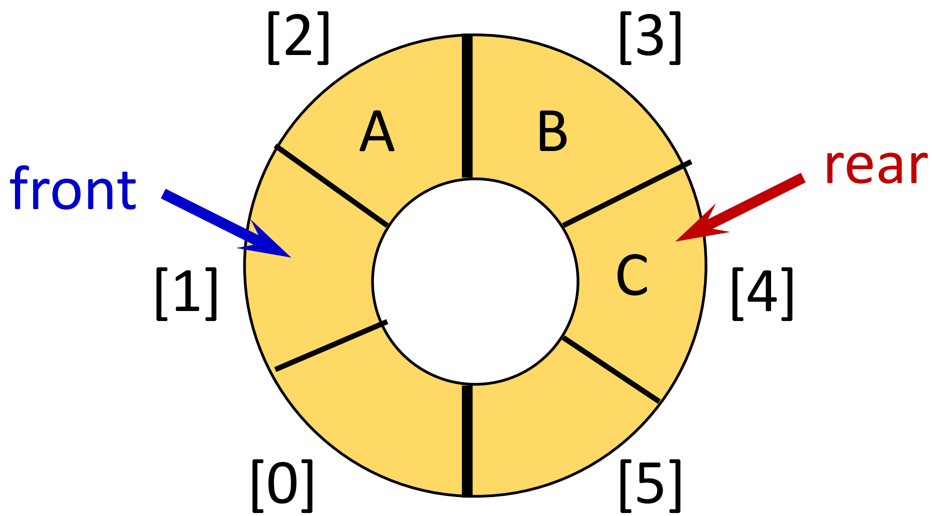
Push An Element (2/2)

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



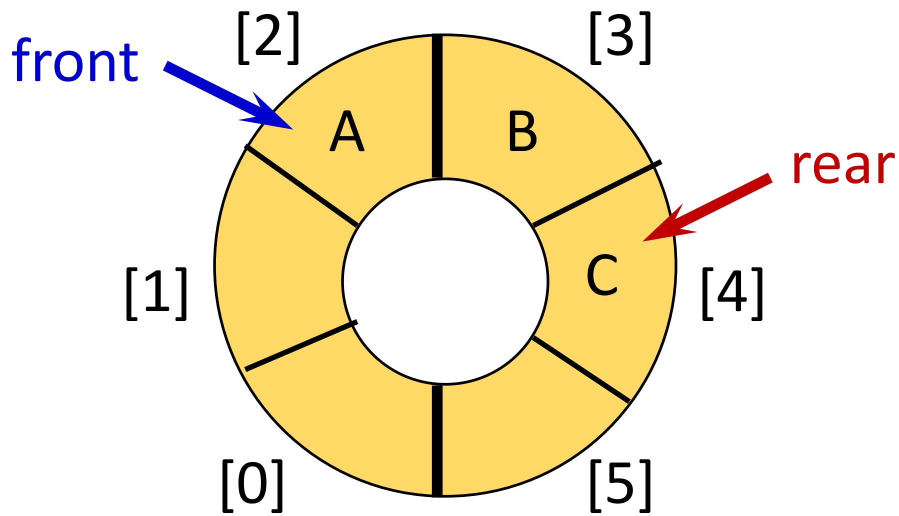
Pop An Element (1/2)

- Move front one clockwise.



Pop An Element (2/2)

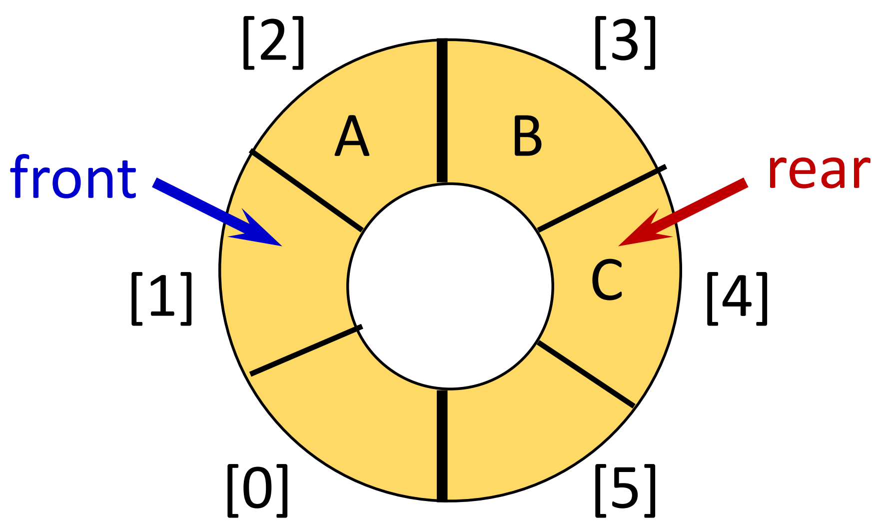
- Move front one clockwise.
- Then extract from queue[front].



Moving rear Clockwise

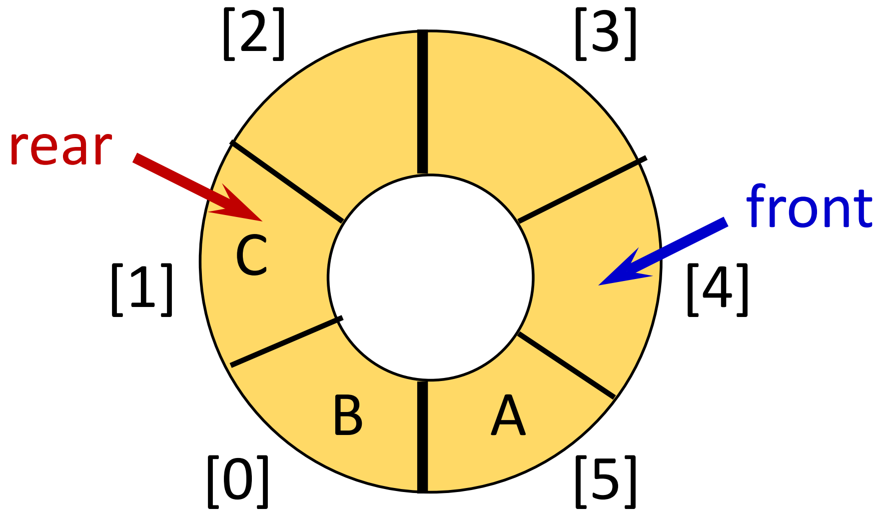
- rear++;

  if (rear = = capacity) rear = 0;

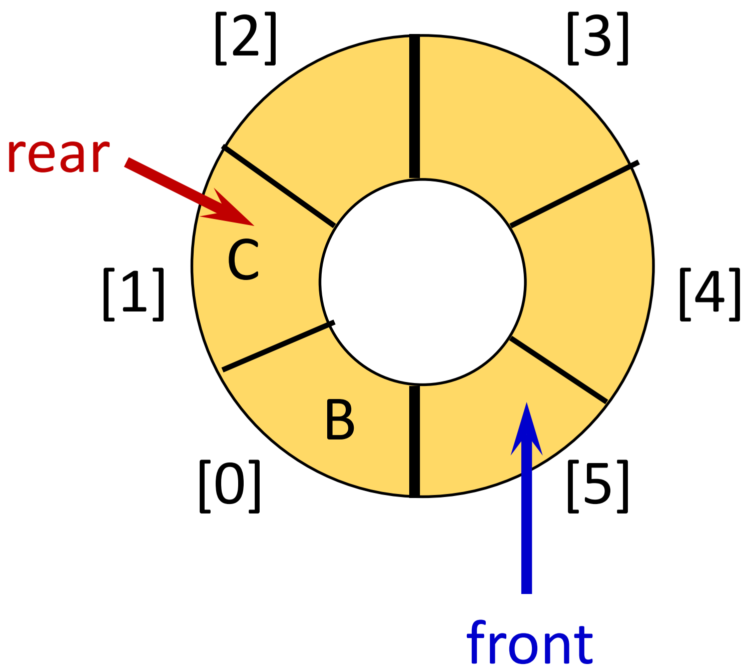


- rear = (rear + 1) % capacity;

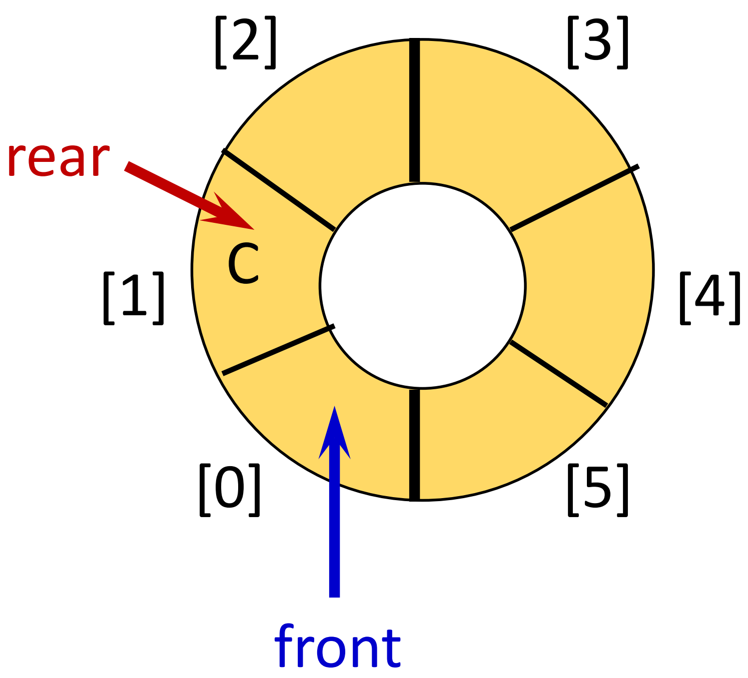
Empty That Queue (1/4)



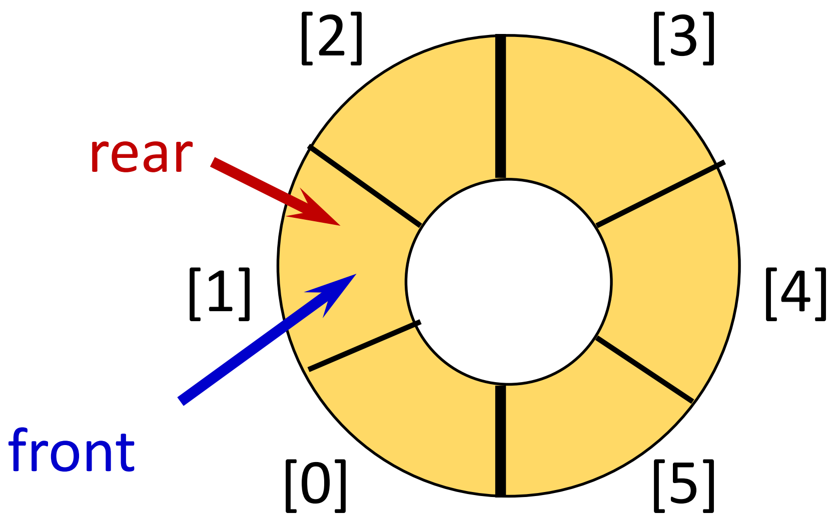
Empty That Queue (2/4)



Empty That Queue (3/4)

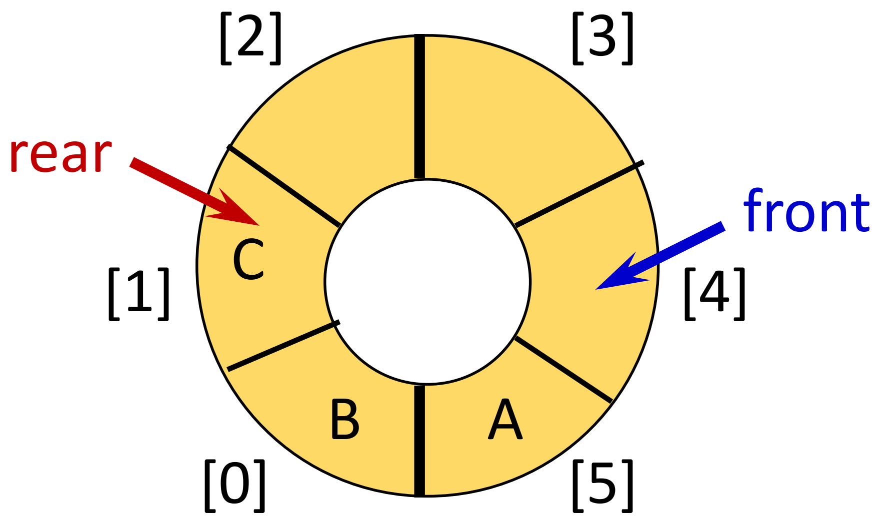


Empty That Queue (4/4)

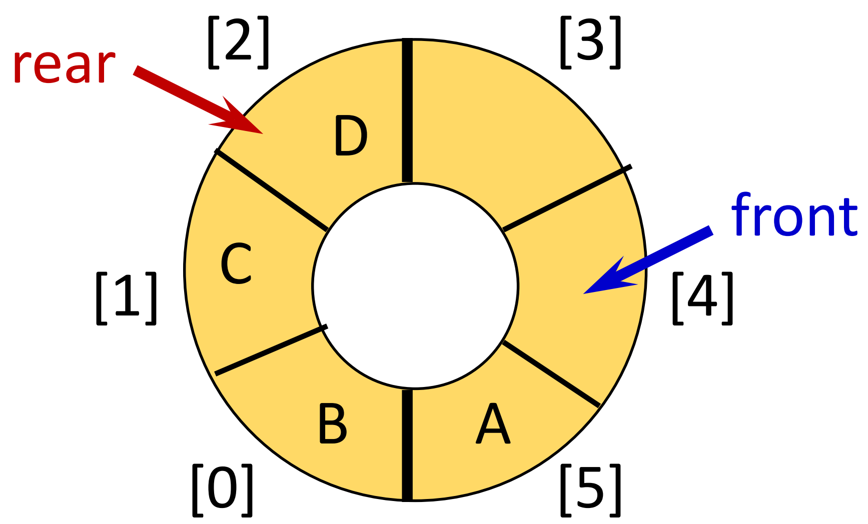


- When a series of removes causes the queue to become empty, front = rear.
- When a queue is constructed, it is empty.
- So initialize front = rear = 0.

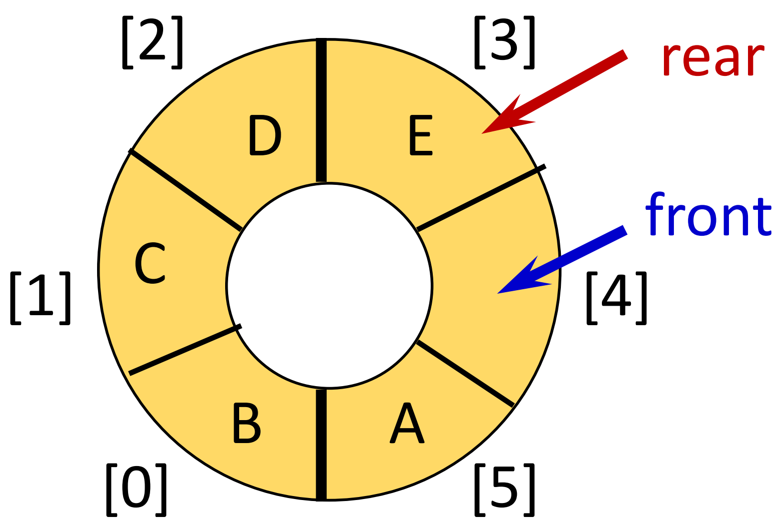
A Full Tank Please (1/4)



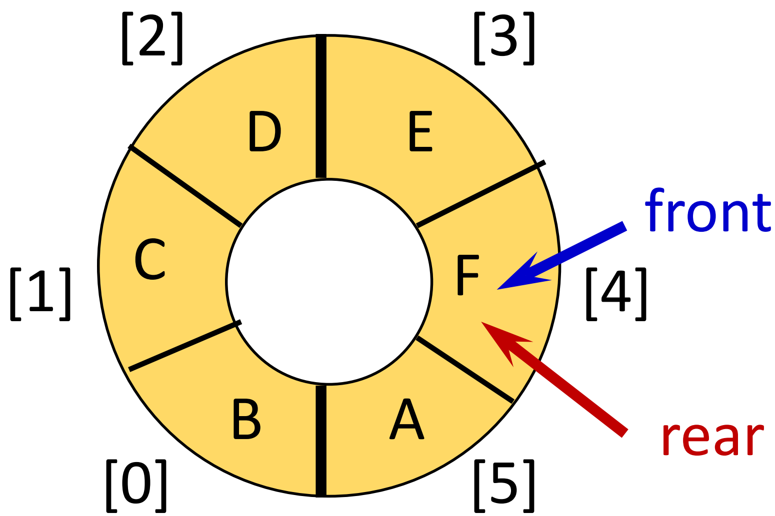
A Full Tank Please (2/4)



A Full Tank Please (3/4)



A Full Tank Please (4/4)



- When a series of adds causes the queue to become full, front = rear.
- So we cannot distinguish between a full queue and an empty queue!

Ouch!!!!! Remedies

1. Don't let the queue get full.
   - When the addition of an element will cause the queue to be full, **increase array size**.
   - This is what the text does.
2. Define a boolean variable lastOperationIsPush.
   - Following each push set this variable to true.
   - Following each pop set to false.
   - Queue is empty iff (front == rear) && !lastOperationIsPush
   - Queue is full iff (front == rear) && lastOperationIsPush
3. Define an integer variable size.
   - Following each push do size++.
   - Following each pop do size--.
   - Queue is empty iff (size == 0)
   - Queue is full iff (size == arrayLength)

- Performance is slightly better when first strategy is used.

Queue ADT

```
template < class T >
class Queue
{
public:
    Queue (int queueCapacity = 0);

    bool IsEmpty( ) const;

    void Push(const T& item);
    // add an item into the queue

    void Pop( );
    // delete an item

    T& Front() const;
    // return top element of stack

    T& Rear() const;
    // return top element of stack
} ;
```

Template Queue Implementation

```
template < class T >
class Queue{
public:
    Queue (int queueCapacity = 0);
    bool IsEmpty( ) const;
    void Push(const T& item);
    // add an item into the queue
    void Pop( );
    // delete an item
    T& Front() const;
    // return top element of stack
    T& Rear() const;
    // return top element of stack
private:
    T* queue;
    int  front,
         rear,
         capacity;
} ;
```

Template Queue Implementation

```
template < class T >
Queue<T>::Queue(int queueCapacity = 10):capacity(queueCapacity)
{
    if (capacity < 1) throw "Queue capacity must be > 0";
    queue= new T[capacity];
    front = rear = 0; // indicate empty stack
}

template <class T>
inline bool Queue<T>:: IsEmpty() const
{
    return front == rear;
}
```

Template Queue Implementation

```
template <class T>
inline T& Queue<T>:: Front() const
{
   if ( IsEmpty() ) throw “Queue is empty. No front element.”;
   return queue[(front + 1) % capacity];
}

template <class T>
inline T& Queue<T>:: Rear() const
{
   if ( IsEmpty() ) throw “Queue is empty. No rear element.”;
   return queue[rear];
}
```

Template Queue Implementation

```
template <class T>
void Queue<T>::Push(const T& x)
{// add x to stack
    if ((rear + 1) % capacity == front)
    {   T* newQu = new T[2*capacity];
        int start = (front+1) % capacity;
        if(start<2)
           copy(queue +start, queue+start+capacity-1, newQu);
        else{
           copy(queue +start, queue+capacity, newQu);
           copy(queue, queue+rear+1,newQu+capacity-start);
        }
        front = 2*capacity – 1;
        rear = capacity -2;
        delete[] queue;
        queue = newQu;
    }
    rear = (rear+1)%capacity;  queue[rear] = x;
}
```

Template Queue Implementation

```
template <class T>
void Queue<T>::Pop( )
{
    if ( IsEmpty() ) throw “Queue is empty, cannot delete”;

    front = (front + 1) % capcity;
    front].~T(); // destructor for T
}
```

Example: Job Scheduling

- In Operating System
  - jobs are processed in the order they enter the system if no priority is set on jobs

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	comments
-1	-1							queue is empty
-1	0	J1						Job 1 joins Q
-1	1	J1	J2					Job 2 joins Q
-1	2	J1	J2	J3				Job 3 joins Q
0	2		J2	J3				Job 1 leaves Q
0	3		J2	J3	J4			Job 4 joins Q
1	3			J3	J4			Job 2 leaves Q

Worst-Case Scenario

front	rear	Q[0]	Q[1]	Q[2]	…	Q[n-1]	Next Operation
-1	n-1	J1	J2	J3	…	J_n	initial state
0	n-1		J2	J3	…	J_n	delete J1
-1	n-1	J2	J3	J4	…	J_{n+1}	add J_{n+1} (J2 to Jn are moved)
0	n-1		J3	J4	…	J_{n+1}	delete J2
-1	n-1	J3	J4	J5	…	J_{n+2}	add J_{n+2}

- In the above job scheduling, it takes n-1 steps to add a new job

DeQue

- Definition
  - A double-ended queue (Deque) is a linear list in which additions and deletions may be made at either end

- Practices
  - Design a data representation that maps a deque into a one-dimensional array
  - Write algorithms to add and delete elements from either end of the queue

Outline

- 3.1 Templates in C++
- 3.2 The stack ADT
- 3.3 The queue ADT
- **3.4 Subtyping and inheritance in C++**
- 3.6 Evaluation of expressions
- 3.5 A mazing problem

Relationships Between Things

- We abstract things on two key dimensions
  - IS-A relationship
  - HAS-A relationship

- Real world examples
  - iPhone is a smartphone. iPhone has a battery
  - NTHU is a university. NTHU has a Math department

- ADT examples
  - Rectangle is a Polygon. Rectangle has a *height* dimension
  - Stack is a Bag. Stack has a *top* pointer
    - Stack is a specialized bag that requires elements to be deleted in the LIFO order

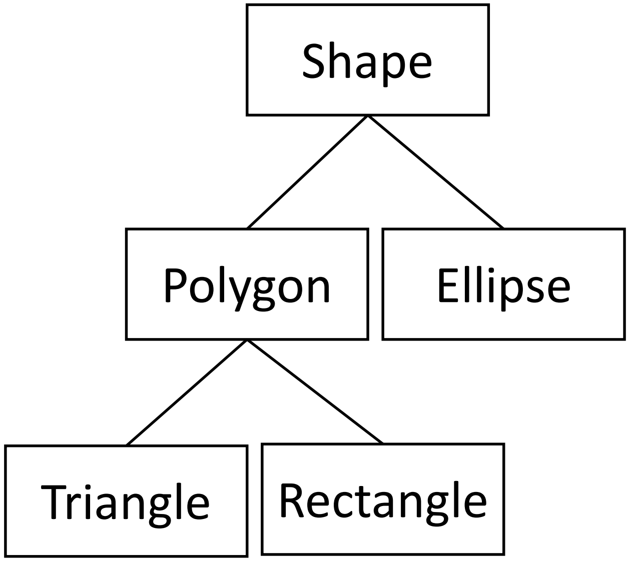
Subtype / IS-A / Subclass

- Subtype
  - Equivalent concept to the IS-A relationship
    - Rectangle is a subtype of Polygon
  - Since C++ use *classes* to denote data types, subtypes are also widely referred to as subclasses

- Subtype is conceptual relationship between ADT specifications
  - "Stack IS A Bag" is true regardless of the implementation

Inheritance

- Use
  - Express IS-A relationships between classes
  - Derive a new class (derived class / sub type / sub class) from an existing class (base class)
- Objective
  - Eliminate redundant implementation
    - Members (data and functions) are by default inherited from a base class to a derived class
- Different inheritance styles
  - Public inheritance
    - Access levels (public/protected/private) of the members are also inherited
  - Protected inheritance
  - Private inheritance



Effects of Inheritance

- Stack inherits from Bag
  - Stack must redefine its constructors and destructors
  - Stack can redefine its unique data and functions (pop and top)
  - Stack inherits all the other data and functions of Bag

```
Class Bag
{
public:
    Bag (int bagCapacity = 10);
    virtual ˜Bag( );
    virtual int Size( ) const;
    virtual bool IsEmpty( ) const;
    virtual int Element( ) const;
    virtual void Push(const int);
    virtual void Pop( );
protected:
    int *array;
    int top;
};
```

```
class Stack : public Bag
{
public:
    Stack (int stackCapacity = 10);
    ~Stack( );
    int Top( ) const;
    void Pop( );
    {
protected:
    {
};
```

Usage Example of Derived Classes

```
Bag b(4);      // invoke Bag constructor
Stack s(7);  // invoke Stack constructor, which also invokes Bag constructor
b.Push(2017);   // use Bag::Push()
s.Push(330);      // Stack does not contains a specialized Push(), so use Bag::Push
b.Pop();       // use Bag::Pop()
s.Pop();       // Stack contains a specialized Pop() overriding Bag::Pop(), so use Stack::Pop()
```

```
Class Bag
{
public:
    Bag (int bagCapacity = 10);
    virtual ~Bag( );
    virtual int Size( ) const;
    virtual bool IsEmpty( ) const;
    virtual int Element( ) const;
    virtual void Push(const int);
    virtual void Pop( );
protected:
    int *array;
    int top;
};
```

```
class Stack : public Bag
{
public:
    Stack (int stackCapacity = 10);
    ~Stack( );
    int Top( ) const;
    void Pop( );
    {
protected:
    {
};
```

Syntax of Implementing Derived Classes

```
Stack::Stack(int stackCapacity)
: Bag(stackCapacity)
// explicitly call to the Bag constructor that has arguments
{
     // here is code specifically for creating a stack, if any
}
int Stack::Stack( )
{      // here is code specifically for destroying a stack, if any
}
//Bag destructor is automatically called when a stack is destroyed

int Stack::Top( ) const
{
        if (IsEmpty( )) throw "Stack is empty.";
        return array[top];
}
void Stack::Pop( )
{
        if (IsEmpty( )) throw "Stack is empty. Cannot delete.";
        top--;
}
```

Outline

- 3.1 Templates in C++
- 3.2 The stack ADT
- 3.3 The queue ADT
- 3.4 Subtyping and inheritance in C++
- **3.5 A mazing problem**
- 3.6 Evaluation of expressions

An Example Maze

entrance

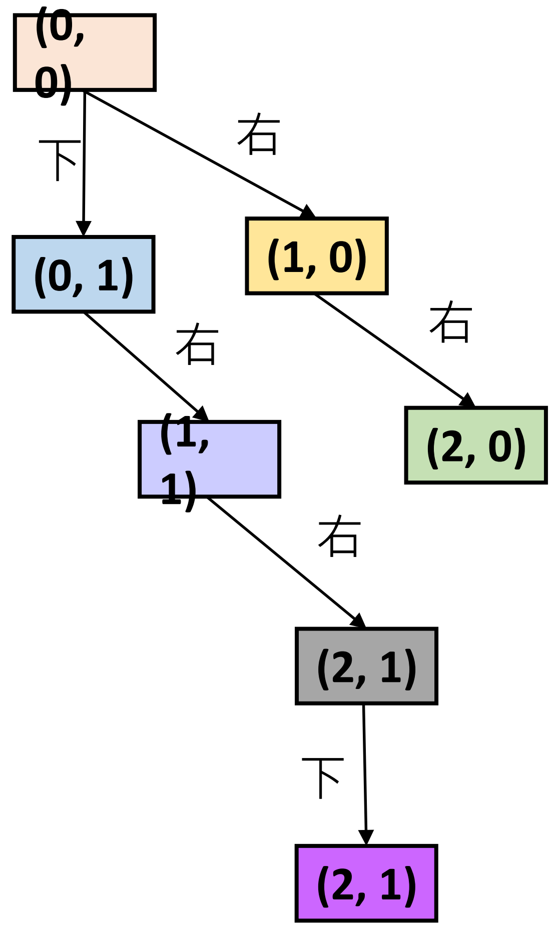
0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

exit

15 x 12 maze

How a Computer Traverses a Maze

右 > 下 > 上 > 左



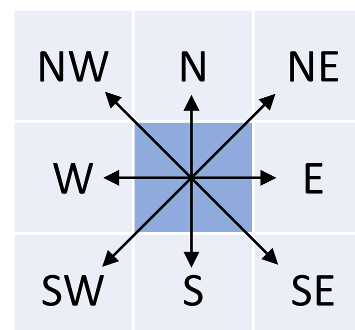
A Mazing Problem

- Maze
  - Is represented as a two-dimensional array maze[i][j]
  - maze[i][j]=0: location that can be passed through
  - maze[i][j]=1: blocked location
  - Entrance: **maze[1][1]**
  - Exit: **maze[m][p]**
- To model border condition
  - The array is declared as **maze[m+2][p+2]**
  - i.e., the original maze array is surrounded by a border of ones

j

i

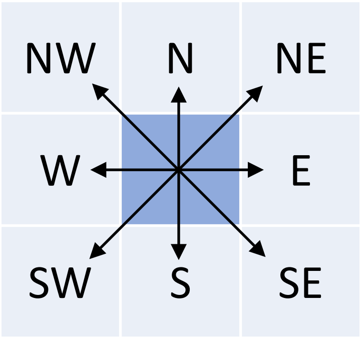
1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	0	1	1	1
1	1	1	0	1	0	0	0	1	0	1	1	1
1	1	0	1	0	1	0	1	0	1	0	1	1
1	1	1	1	0	0	0	1	0	1	1	1	1
1	1	0	1	1	0	1	1	1	0	1	1	1
1	1	1	0	1	0	1	1	1	0	1	1	1
1	1	0	0	1	1	0	0	1	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1



A Mazing Problem

- Allowable moves
  - Non-blocked squares of the eight neighboring squares
- How can a program get through the maze?

1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	0	1	1	1
1	1	1	0	1	0	0	0	1	0	1	1	1
1	1	0	1	0	1	0	1	0	1	0	1	1
1	1	1	1	0	0	0	1	0	1	1	1	1
1	1	0	1	1	0	1	1	1	0	1	1	1
1	1	1	0	1	0	1	1	1	0	1	1	1
1	1	0	0	1	1	0	0	1	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1



Strategy of Searching

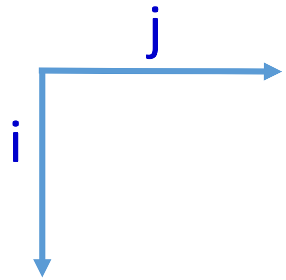
- As a rat walks through the maze

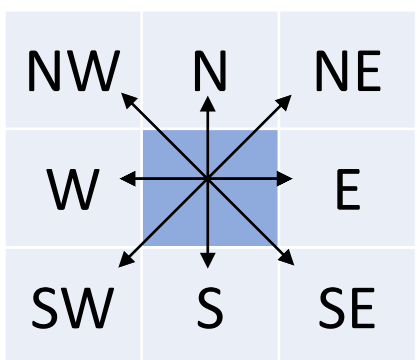
  (1) He picks a valid move from the current position
    - e.g., starting from north and looking clockwise

  (2) Put the selected move into a stack
    - So that he can return from a dead path

  (3) He learns not to make the same mistake twice
    - Avoid getting into a cell visited before
    - A 2-dimensional array, mark[m+2][p+2] is used
    - The mark array records the cells visited before

Allowable Moves





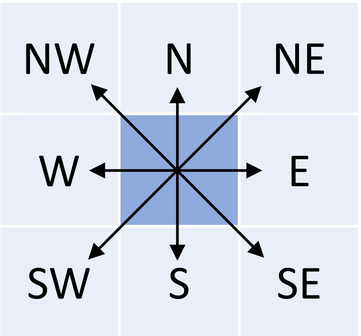
q		move[q].di	move[q].dj
0	N	−1	0
1	NE	−1	1
2	E	0	1
3	SE	1	1
4	S	1	0
5	SW	1	−1
6	W	0	−1
7	NW	−1	−1

Algorithm (Pseudo Code)

- The coordinates of the next move is computed by the following data structure

```
struct Offsets
{
    int di, dj;
}

enum directions {N, NE, E, SE,
S, SW, W, NW};

Offsets move[8];
struct Items
{
    int x, y, dir;
}
```

q	move[q].di	move[q].dj
N	−1	0
NE	−1	1
E	0	1
SE	1	1
S	1	0
SW	1	−1
W	0	−1
NW	−1	−1



Algorithm ()

```
initialize a stack // remember the point to retract
add the starting point, (0, 1, E), to the stack

while (the stack is not empty) { // there are still unexplored points
  (i, j, dir) = the top of the stack;
  remove the top of the stack;
  while(there are more move from (i, j)){
    (g, h) = nextPoint((i, j), dir);
    if ((g == m) && (h == p)) return success;

    if ((!maze [g][h]) && (!mark [g][h])) {
        dir = Next(dir);
        add (i, j, dir) to the stack;  // prepare for a dead end

        (i, j, dir) = (g, h, N); // move to (g, h), start from dir N
        mark[i][j] = 1;
    }
  }
}
cout << "No path in maze." << endl;
```

- Each position can be visited at most once.
- At most eight valid moves from each position

→ O(size of the array) time

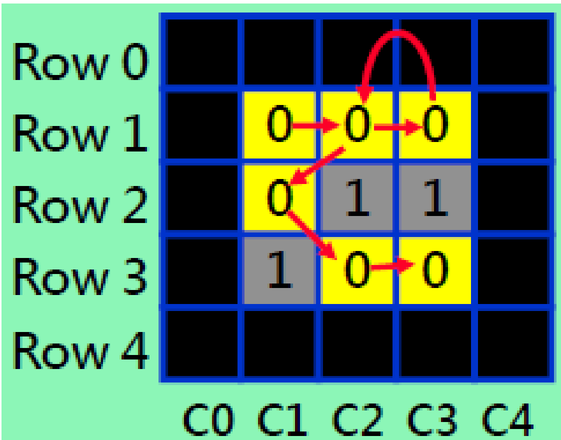
Use a Stack to Keep Pass History

- What is the maximal size of the stack?
  - A maze is represented by a two dimensional array maze[m][p]
  - Since each position is visited at most once, at most m×p elements can be placed in the stack

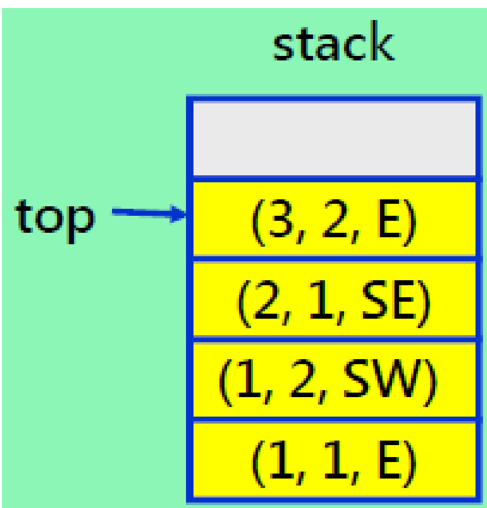
```
typedef struct {
   int x;
   int y;
   int dir;
} Item;
Item mazestack[m*p];
```

```
#include <stack>
typedef struct {
   int x;
   int y;
   int dir;
} Item;
stack<Item> mazestack;
```

Example: A Mazing Problem



Current Position	Next Legal Move	Stack operation
(1, 1)	(1, 2, E)	Push (1, 1, E)
(1, 2)	(1, 3, E)	Push (1, 2, E)
(1, 3)	No legal move	Pop to backtrack
(1, 2)	(2, 1, SW)	Push (1, 2, SW)
(2, 1)	(3, 2, SE)	Push (2, 1, SE)
(3, 2)	(3, 3, E) success!	Pop out the entire stack



stack right before success

Complete path:
(3, 3) ← (3, 2, E) ← (2, 1, SE) ← (1, 2, SW) ← (1, 1, E)

Algorithm (Pseudo Code)

```
void driver()
{
   if (findPath(0, 1))
       cout << "Success" << endl;
   else
       cout << "No path in maze." << endl;
   return;
}
bool findPath(int i, int j) // find a path starting from (i, j)
{
    for (all eight directions) {  // explore all directions
        (g, h) = (i, j) + direction;

        if ((g == m) && (h == p)) return true;

        if ((!maze [g][h]) && (!mark [g][h])) {
            findPath(g, h); // keep finding a path…
        }
    }
    return false;
}
```

Program 3.16

```
void Path(const int m, const int p)
{// 輸出迷宮的一個路徑（如果有的話）； maze[0][i] = maze[m+1][i] =
 // maze[j][0] = maze[j][p+1] = 1, 0 ≤ i ≤ p+1, 0 ≤ j ≤ m+1 。
    // 從 (1, 1) 開始
    mark[1][1] = 1;
    Stack<Items> stack(m*p);
    Items temp(1, 1, E);
          // 設定 temp.x、temp.y、與temp.dir
   Stack.Push(temp);
   while (!stack.IsEmpty( ))
   { // 堆疊不是空的
          temp = stack.Top( );
          stack.Pop( ); // 彈出
          int i = temp.x; int j = temp.y; int d = temp.dir;
```

```
        while (d < 8)  // 往前移動
        {
            int g = i + move[d].di; int h = j + move[d].dj;
            if ((g = = m) && (h = = p)) { // 抵達出口
            {
                    cout << stack; // 輸出路徑
                    cout << i << " " << j << endl; // 路徑上的上兩個方塊
                    cout << m << " " << p << endl;
                    return;
            }
            if ((!maze [g ][h]) && (!mark [g ][h]))   // 新位置
            {       mark[g][h] = 1;
                    temp.x = i; temp.y = j; temp.dir = d+1;//try new direction
                    stack.Push(temp); // 加入堆疊
                    i = g; j = h; d = N; // 移到 (g, h)
            }
            else d++; // 試下一個方向
        }
    }
  cout << "No path in maze." << endl;
}
```

Stack Provided by C++ Library

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> s;
    for(int i=0; i < 5; i++){
        s.push(i);
    }
    while(!s.empty())
    {
        cout << s.size() << " ";
        cout << s.top() << endl;
        s.pop();
    }
}
```

output

```
5 4
4 3
3 2
2 1
1 0
```

Reference of STL's Stack

http://en.cppreference.com/w/cpp/container/stack

Outline

- 3.1 Templates in C++
- 3.2 The stack ADT
- 3.3 The queue ADT
- 3.4 Subtyping and inheritance in C++
- 3.5 A mazing problem
- **3.6 Evaluation of expressions**

Types of Expression

- Arithmetic Expression
  - For example: X = A/B – (C + D * E – A * C)
  - The evaluation of this expression is critical in enabling high level programming
  - An expression consists of
    1. **Operands**: A, B, C, D, E
    2. **Operator**: plus, minus, multiply, and divide
    3. **Delimiter**: like parenthesis “(“, “)”
- Boolean Expression (relational + logical + compound)
  - The result is TRUE or FALSE
  - Use relational and logical operators
    - **Relational operator**: <, <=, >, >=, ==, !=,
    - **Logical operator**: &&, ||, !

Evaluation of Expressions

- Arithmetic expressions
  - X = (A / B) – C + D * E – A * C
- Boolean expressions
  - X = (A == B) || !(C>D)
- Expressions are made up of
  - Operands: A, B, C, D, E
  - Operators:
    - Binary arithmatic operators: +, -, *, /, %
    - Unary arithmatic operators: -
    - Relational operators: <, <=, ==, !=, >=, >
    - Binary logical operators: &&, ||
    - Unary logical operators: !
  - Delimiters: (,)

Evaluation of Expressions

- Let's focus on an arithmetic expression
  - X = A / B – C + D * E – A * C

- Order of evaluation matters
  - Let A = 4, B = C = 2, D = E = 3
  - Interpretation 1:

    ((4/2)–2)+(3*3)–(4*2) = 0 + 9 – 8 = 1
  - Interpretation 2:

    (4/(2–2+3))*(3–4)*2 = (4/3)*(–1)*2 = -2.666…

- How can computers uniquely define the order of evaluation of an expression?

  operator precedence (priority) rule + associative rule

Priority of Operators

priority	operator
1	Unary minus, !
2	*, /, %
3	+, −
4	<, <=, >, >=
5	==, !=
6	&&
7	||

Evaluation of operators of the same priority will proceed from left to right, e.g., A/B*C → (A/B) * C

Priority of Operators (cont.)

- Priority is introduced to help defining the order
  - Tie break rule: left to right

- Example

  Two operators compete for one operand

  '/' and '*' win

  - A / B – C + D * E – A * C → (A/B) – C + (D*E) – (A*C)
  - A/B*C/D → ((A/B)*C)/D *Tie-break rule*

Infix, Prefix, and Postfix Notations

- Infix
  - Each Binary operators come in-between their operands
  - e.g., 2*3,   A*B/C
- Postfix
  - Binary operators appear after their operands
  - e.g., 23*, AB*C/
- Prefix
  - Binary operators appear before their operands
  - e.g., *23,   /*ABC
- Compiler
  - Translates an expression into a sequence of machine codes
  - It first re-writes the expression into a form called **postfix notation**, and then **evaluate** postfix notation.

Evaluation of Expressions

user	computer
Infix	**Postfix**
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea *c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

Postfix & prefix: no parentheses, no precedence

Two Essential Algorithms

- Combining two algorithms enables computers to handle human-written expressions

  Phase 1: Infix-to-Postfix conversion

  Phase 2: Postfix evaluation (just mentioned)

Infix

A/B-C+D*E-A*C

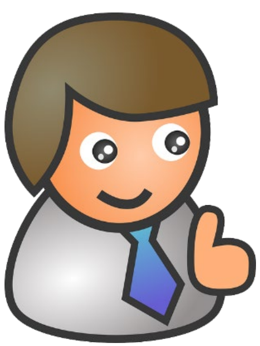
Conversion algorithm

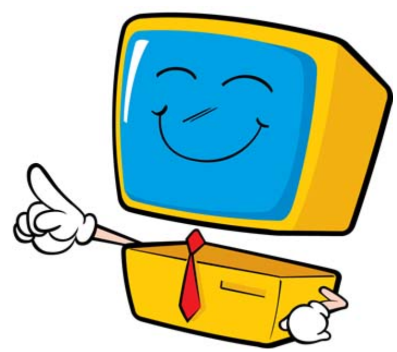
Postfix

AB/C-DE*+AC*-

Evaluation

Expressions





Evaluation of Expression Example

- Phase 1: Infix to postfix conversion

  6/2-3+4*2 → 6 2 / 3 – 4 2 * +

- Phase 2: Postfix expression evaluation

  6 2 / 3 – 4 2 * + → 6 2 / → 3

  3 3 - → 0

  0 4 2 * → 4 2 * → 8

  0 8 + → 8

Phase 2: Postfix expression evaluation

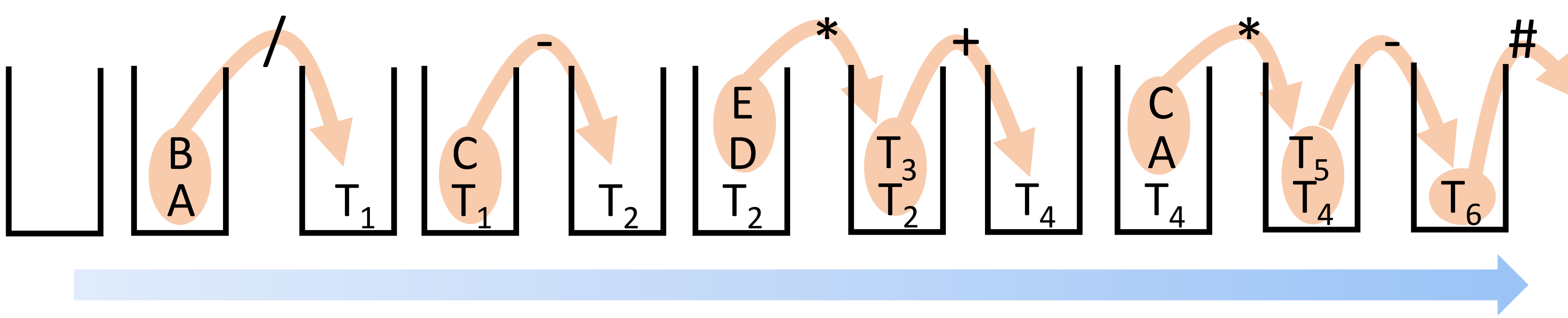
6 2 / 3 – 4 2 * +

Token	Stack [0]	Stack [1]	Stack [2]	Top
6	6			0
2	6	2		1
/	3			0
3	3	3		1
–	0			0
4	0	4		1
2	0	4	2	2
*	0	8		1
+	8			0

Postfix Evaluation

- Rules
  - Left to right scan
  - Push operands onto a stack
  - Evaluate operators using the required number of operands from the stack
  - Push the evaluating results onto the stack again

- AB/C-DE*+AC*-#   *(# denotes the end of an expression)*



Advantages of Postfix Notation

- Evaluation is simpler than infix notation
  - The need for parenthesis is gone
  - The need for operator priority is gone

```
void Eval(Expression e)
{
   Stack<Token> stack;   // initialize a stack
   for (Token x = NextToken(e); x!= end of expression; x=NextToken(e))
   {
      if (x is an operand) {
         stak.Push(x)
      } else {// x is an operator
         pop from the stack the correct number of operands for the operator;
         perform the operation x and store the result (if any) onto the stack;
   }
}
```

Two Essential Algorithms

- Combining two algorithms enables computers to handle human-written expressions
  - Infix-to-Postfix conversion
  - Postfix evaluation (just mentioned)

Infix

A/B-C+D*E-A*C

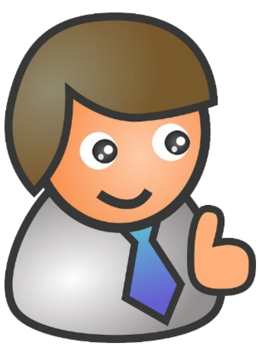
Conversion algorithm

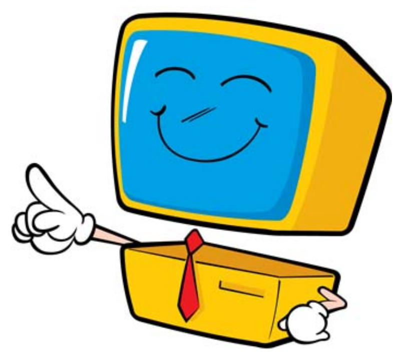
Postfix

AB/C-DE*+AC*-

Evaluation

Expressions





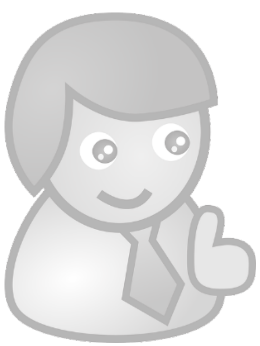
Infix to Postfix Conversion

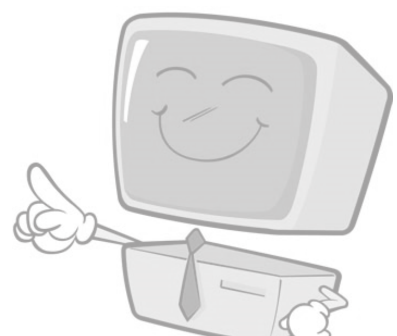
- Observations
  - Number of operands and operators do not change
  - Order of operands (A, B, C…) do not change

Infix
A/B-C+D*E-A*C

Conversion algorithm

Postfix
AB/C-DE*+AC*-





Infix to Postfix Conversion

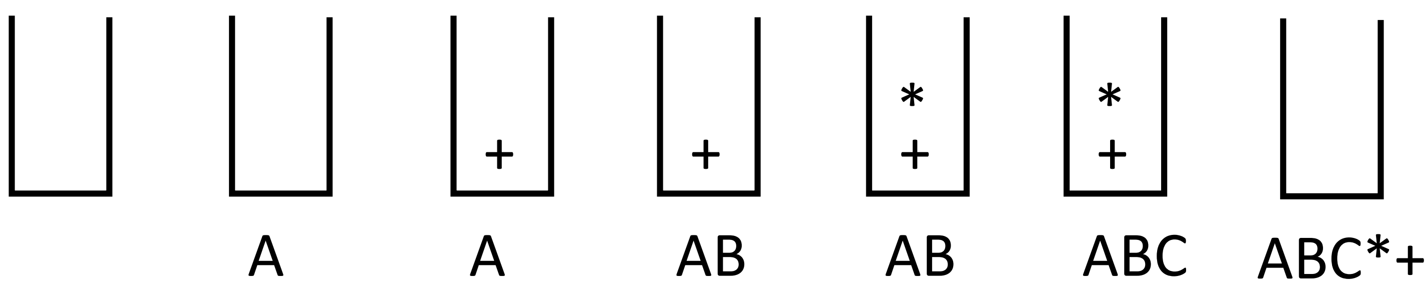
- Method 1
  - Fully parenthesize the expression (based on the operator priorities)
  - Move all operators so that they replace their corresponding right parentheses
  - Delete all parentheses

A/B-C+D*E-A*C ➡ ((((A/B)-C)+(D*E))-(A*C))

((((AB/C-(DE* +(AC* - ➡ AB/C-DE*+AC* -

Infix to Postfix Conversion

- Stack-based algorithm
  - Create a stack
  - Scan the input infix expression left to right
  - Bypass each incoming operand to the output
  - For each incoming operator
    - First, continuously pop from the stack an operator (the top) if the top has equal or lower priority than the incoming operator
    - Then, push the incoming operator onto the stack
  - Pop all operators upon the end of an expression
- Example: A + B * C



Parentheses Handling

- We want the stack algorithm to handle parentheses similarly to handling operators
- Specialized rules for left parenthesis
  - Incoming left parenthesis has the highest priority (i.e., always gets pushed onto the stack)
    - **In-coming priority (ICP) = 0**
  - Only gets popped from the stack upon a matched right parenthesis
  - Otherwise, behaves as one with the lowest priority
    - **In-stack priority (ISP) = 8**

Priority	Operator
0	In-coming (
1	Unary minus (負號), !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	= =, !=
6	&&
7	| |
8	In-stack (

Example

- A*(B+C)/D

Incoming token	Stack	Output	Note
Empty	Empty	Empty	
A			
*			
(			
B			
+			
C			
)			
/			
D			
Done			

Example

- A*(B+C)/D

Incoming token	Stack	Output	Note
Empty	Empty	Empty	
A	Empty	A	Bypass operands
*	*		
(	*(		ICP('(') higher than ISP('*')
B	*(	AB	Bypass operands
+	*(+		ICP('+') higher than ISP('(')
C	*(+	ABC	Bypass operands
)	*	ABC+	Pop until a left parenthesis
/	/	ABC+*	ICP('/') == ISP('*')
D	/	ABC+*D	Bypass operands
Done	Empty	ABC+*D/	Pop all operators

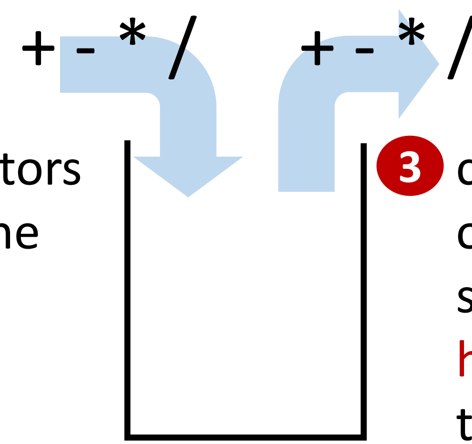
Recap Infix to Postfix Conversion

A B C

2 incoming operands always bypasses the stack

A + B * C

1 left to right scan

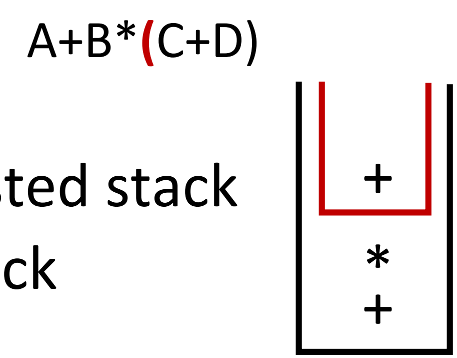


4 incoming operators always enters the stack

3 continuously pop top operator from the stack if it has equal or higher priority than the incoming ones

Recap Parenthesis Handling

- Incoming left parenthesis has the highest priority
  - It always enters the stack without popping any stacked operator
- In-stack left parenthesis has the lowest priority
  - It never gets popped from the stack until the right parenthesis appears
- Different perspective [1]
  - Left parenthesis creates an isolated, nested stack
  - Right parenthesis cleans up a nested stack



1. Contributed by Mr. 陳德暉 (101061132) on April 2, 2015

Infix to Postfix Algorithm

```
void Postfix(Expression e)
{
    Stack<Token>stack; // initialize the stack
    stack.Push('#');
    for (Token x = NextToken(e); x != '#'; x = NextToken(e))
        if (x is an operand) cout << x;
        else if (x == ')' ) { // pop until a left parenthesis
            for (;stack.Top( ) != '('; stack.Pop( ))
                cout << stack.Top( );
            stack.Pop( ); // remove the left parenthesis
        } else { // x is a operator
            for (; isp(stack.Top( )) <= icp(x); stack.Pop( ))
                cout << stack.Top( );
                                              // higher or equal priority
            stack.Push(x);
        }
    // end of expression; empty the stack
    for ( ; !stack.IsEmpty( ); cout << stack.Top( ), stack.Pop( ));
    cout << endl;
}
```

Limitations of the Current Algorithm

- Characters to tokens conversion (parser)
  - Energy = Mass * LightSpeed * LightSpeed
  - Area = 3.14*radius1*radius2
- Grammar
  - X = A - B + -A
    computers need rules to differentiate the two minus symbols; Otherwise, the aforementioned postfix algorithm cannot work correctly.
- More techniques are available in a compiler course