Chapter 3

Stacks and Queues

Stacks, Queues and Templates

- Continued from last week's ordered lists

- Examples:
  - Stacks and queues
  - Will use templates to implement these data structures

- Template function in C++ makes it easier to reuse classes and functions.

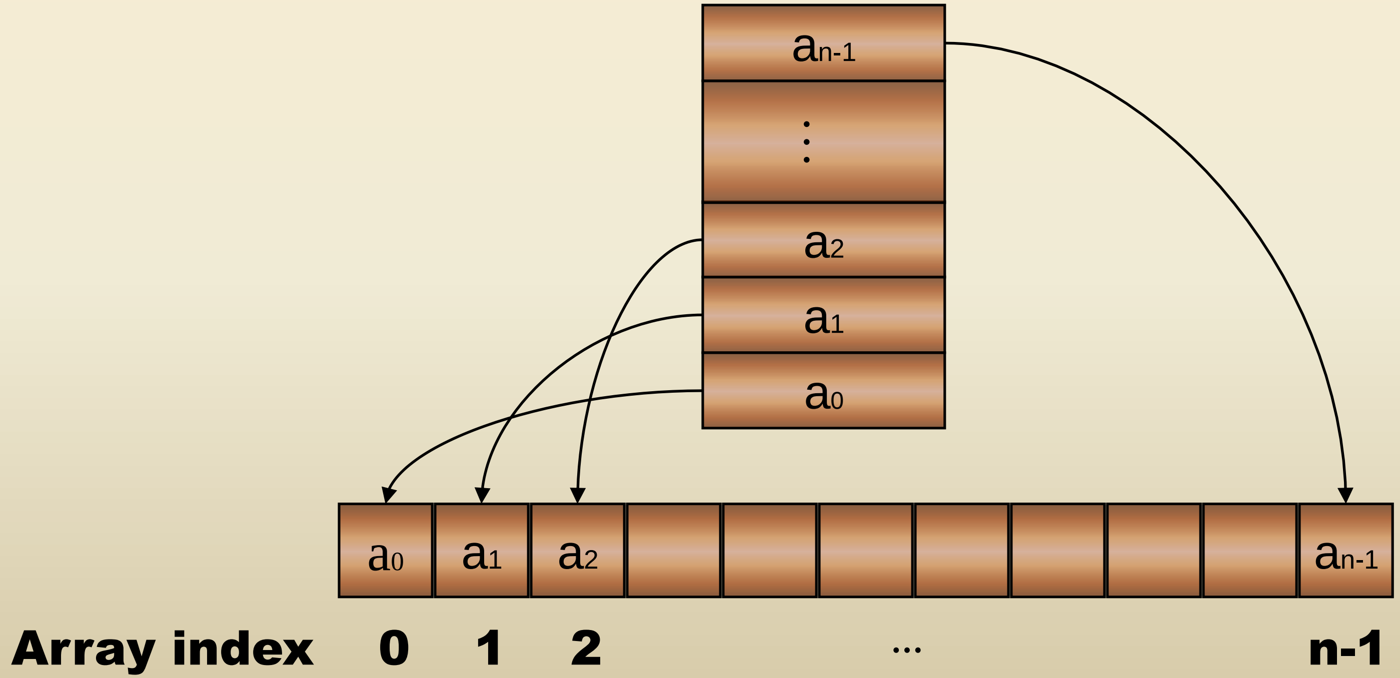
Stacks, Queues and Templates

- A template
  - can be regarded as a variable instantiated to any data type, including fundamental C++ types or user-defined types.

- See the supplementary material from
  - Chapter 22 of the OOP textbook

Stack and Its Array Implementation

- Stacks are special cases of ordered list.
  - Can use a 1D array to represent a stack.
  - Stack elements are stored in stack[0] through stack[top].



Stack: Last-In-First-Out (LIFO) List

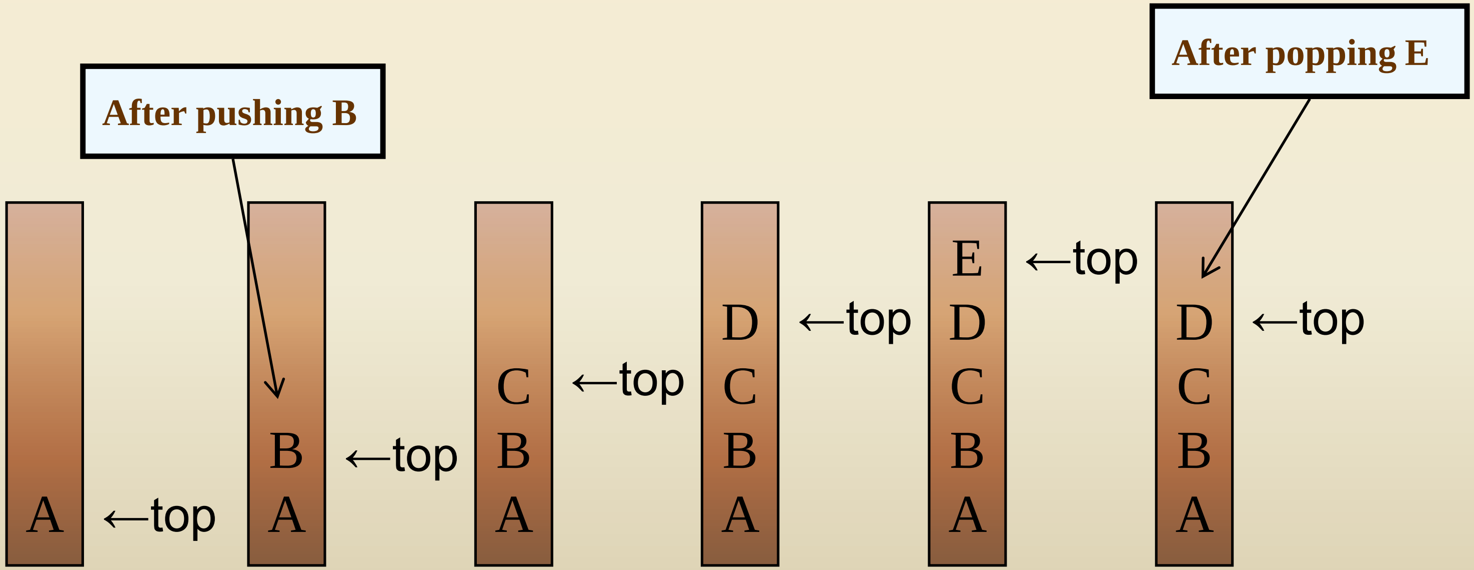
- Physical memory representations:

a	b	c	d	e										
0	1	2	3	4	5	6								

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

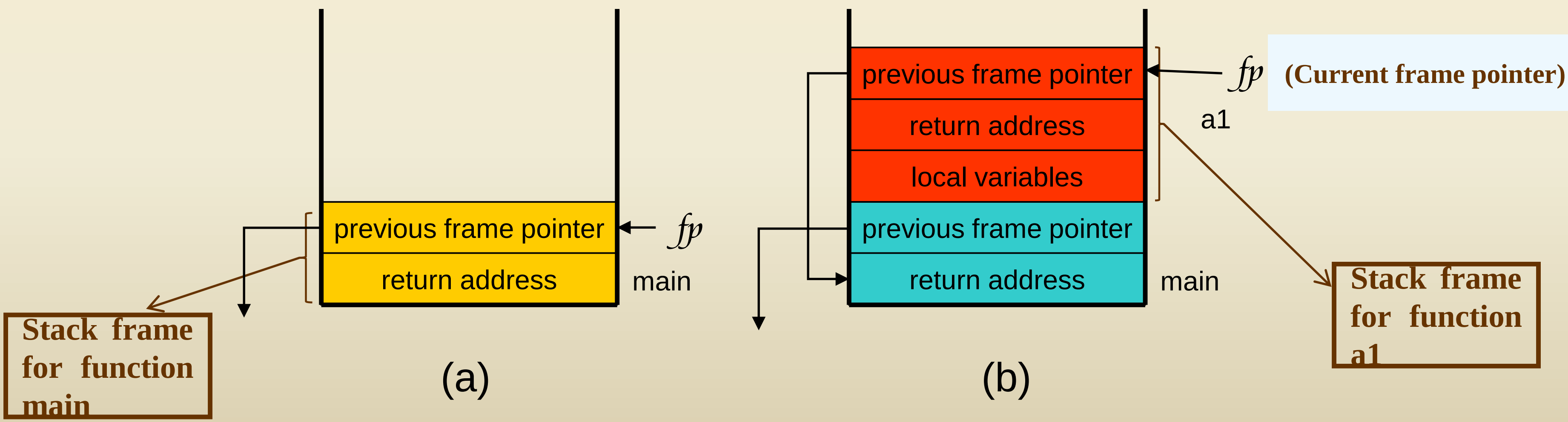
Stack: Last-In-First-Out (LIFO) List

- Elements are last-in-first-out (LIFO)
- Operations
  - Push: Add an element into a stack
  - Pop: Get and delete an element from a stack



An Example of Stack: System Stack and Stack Frame of Function Call

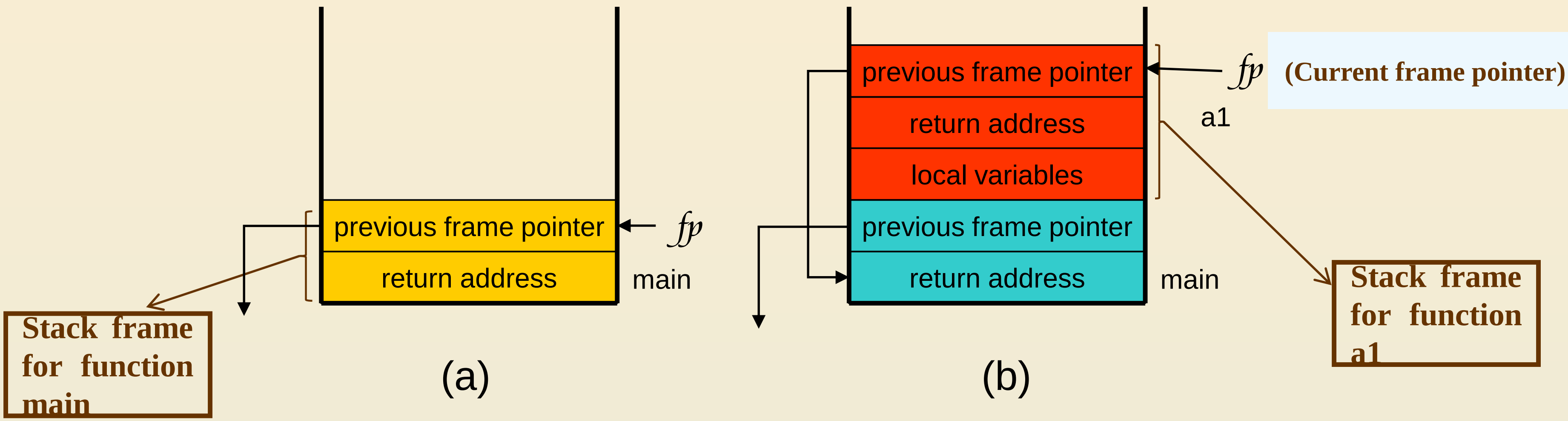
- System Stack ─ used by a program at runtime to process function calls.
- Whenever a function is invoked, the program creates a structure (named "activation record" or "stack frame") and places it on top of the system stack.
- E.g., (a) system stack before calling function a1
  (b) system stack after calling function a1



System Stack

An Example of Stack: System Stack and Stack Frame of Function Call

- From the previous figure:



- Return address: 用來記錄當程式執行權被另一個function取走後，那一個function執行後應在何處回來現在尚未執行完的程式以繼續執行剩餘的部分
  - E.g., if function a calls function b, then b 從 a 取得程式執行權以執行 b 的內容。在 b 執行完畢後透過 return address 將程式執行權退還給 a，讓a未執行完的部分繼續執行下去。
- Previous frame pointer:用來指向先前從某個function取得程式執行權以執行自己的程式後再將執行權歸還至前個function的位址。

Abstract Data Type for Stack

```
template <class T>
class Stack
{ // objects: A finite ordered list with zero or more elements
public:
    Stack (int MaxStackSize = DefaultSize);
    // Create an empty stack whose maximum size is MaxStackSize

    Boolean IsFull();
    // if number of elements in the stack is equal to the maximum size
    // of the stack, return TRUE(1) else return FALSE(0)

    Boolean IsEmpty();
    // if number of elements in the stack is 0, return TRUE(1) else return FALSE(0)

    void Push(const T& item);
    // if IsFull(), then StackFull(); else insert item into the top of the stack.

    void Pop();
    // if IsEmpty(), then StackEmpty() and return 0;
    // else remove the top element of the stack.
};
```

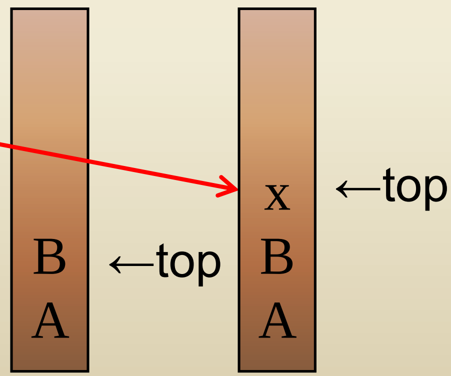
Code Snippets of Stack by Array

```
Private:
   int top;
   T *stack;
   int MaxSize;

template<class T>
Stack<T>::Stack(int MaxStackSize):MaxSize(MaxStackSize) {
         stack=new T[MaxSize];
         top=-1; //initialize this value for emptiness check
}

template<classs T>
inline Boolean Stack<T>::IsFull() {
         if (top==MaxSize-1) return TRUE;
         else return FALSE;
}

template<classs T>
inline Boolean Stack<T>::IsEmpty() {
         if (top==-1) return TRUE;
         else return FALSE;
}
```

Use the member initializer to set the values of member MaxSize

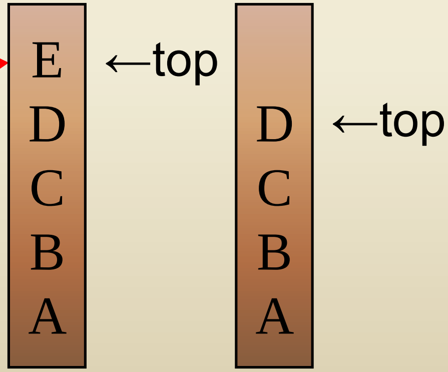
Adding (Pushing) an Element to a Stack

```
template <class T>
void Stack<T>::Push(const T& x)
{
    // add an item to the stack
    if (IsFull())
        stack_full( );
    else
        stack[++top]=x;
}
```



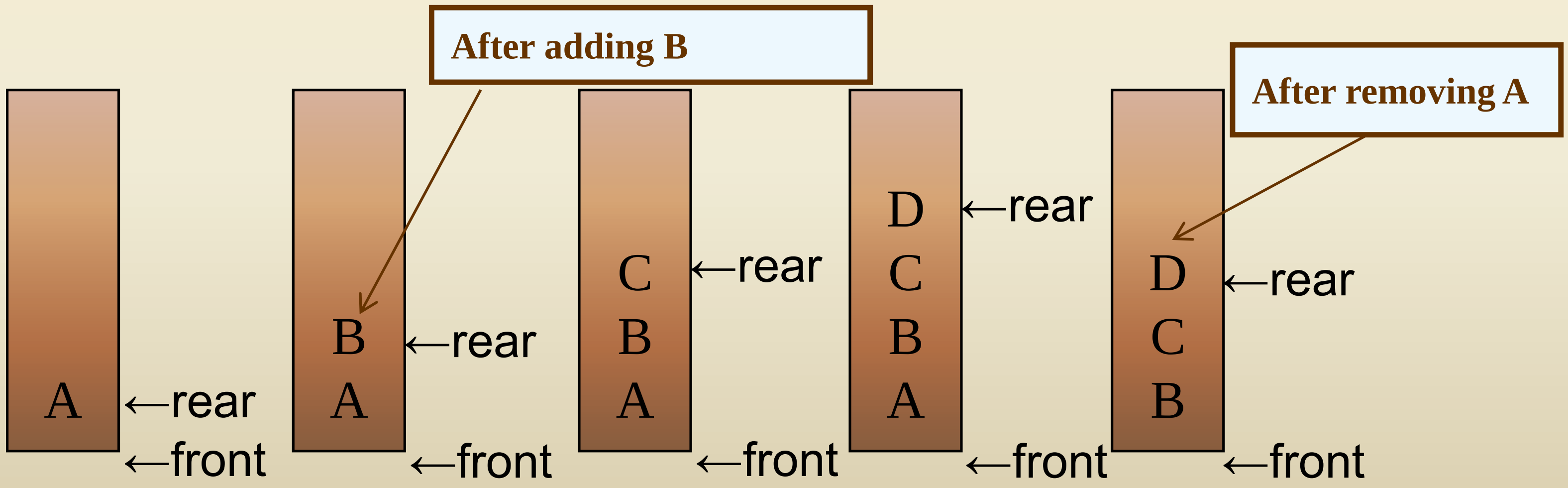
Deleting (Popping) an Element from a Stack

```
template <class T>
void Stack<T>::Pop()
{
    // return the top element from the stack
    if (IsEmpty())
    {
        stack_empty( );
        return 0;
     }
    stack[top--] .~T(); //destructor for T;
}
```



Queue: First-In-First-Out (FIFO) List

- Data structures are similar to stacks, but elements are first-in-first-out (FIFO)
- Operations:
  - Push: Add an element into a queue
  - Pop: Get and delete the first element from a queue



Application: Job Scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				J1 is added
-1	1	J1	J2			J2 is added
-1	2	J1	J2	J3		J3 is added
0	2		J2	J3		J1 is deleted
1	2			J3		J2 is deleted

Insertion and deletion from a sequential queue

Abstract Data Type of Queue

```
template <class T>
class Queue {
// objects: A finite ordered list with zero or more elements
public:
     Queue(int MaxQueueSize = DefaultSize);

    // Create an empty queue whose maximum size is MaxQueueSize
    Boolean IsFull();
    /* if number of elements in the queue is equal to the maximum size of
    the queue, return TRUE(1); otherwise, return FALSE(0) */

    Boolean IsEmpty();
    // if number of elements in the queue is equal to 0, return TRUE(1)
    // else return FALSE(0)

    void Push(const T& item);
    // if IsFull(), then QueueFull(); else insert item at the rear of the queue

    void Pop();
    // if IsEmpty(), then QueueEmpty() and return 0;
    // else remove the item at the front of the queue
};
```

Implementation 1: Using Array

```
Private:
    int front, rear;
    T *queue;
    int MaxSize;

template<class T>
Queue<T>::Queue(int MaxQueueSize):MaxSize(MaxQueueSize) {
    queue=new T[MaxSize];
    front=rear= -1;
}

template<classs T>
inline Boolean Queue<T>::IsFull() {
    if (rear==MaxSize-1) return TRUE;
    else return FALSE;
}

template<classs T>
inline Boolean Stack<T>::IsEmpty() {
    if (front==rear) return TRUE;
    else return FALSE;
}
```

Add an Element to a Queue

```
template <class T>
void Queue<T>::Push(const T& x)
{
    // add an item to the queue
    if (IsFull())
            QueueFull( );
    else
            queue[++rear]=x;
}
```

Delete an Element from a Queue

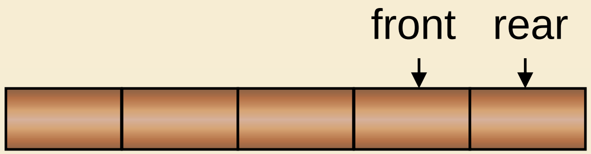
```
template <class T>
void Queue<T>::Pop()
{
    // return the top element from the queue
    if (IsEmpty())
    {
            QueueEmpty( );
            return 0;
    }
    queue[++front].~T(); //destructor for T
}
```

Problem

- As the elements enter and leave the queue, the queue gradually shifts to the right. (See p.14 for example.)
  - Eventually,
    - the index of rear = *MaxSize-1*
    - This suggests that the queue is full even though the underlying array is not full

Problem

- Solution:
  - Use a function to move the entire queue to the left so that *front=-1*

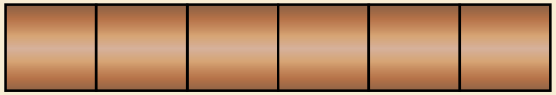


  - It is time-consuming. Time complexity = O(MaxSize)
- A clever solution:
  - Use a circular representation to replace the linear representation.
  - Then the computing time of each operation is O(1).

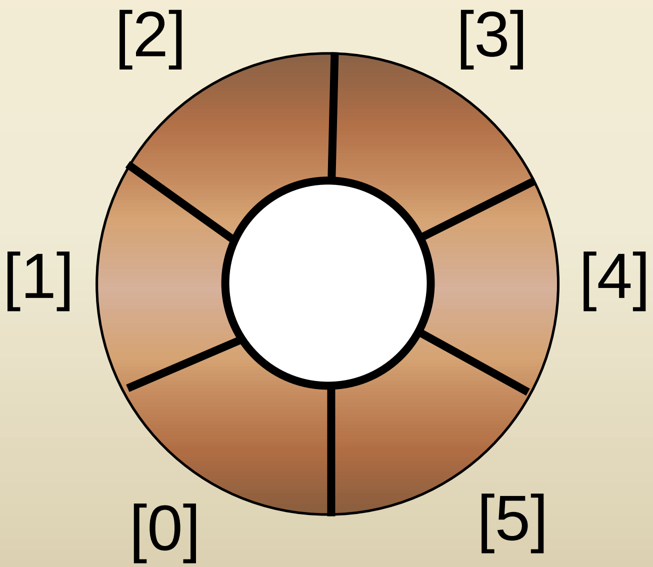
Circular Representation

- Use a 1D array queue.

queue[]

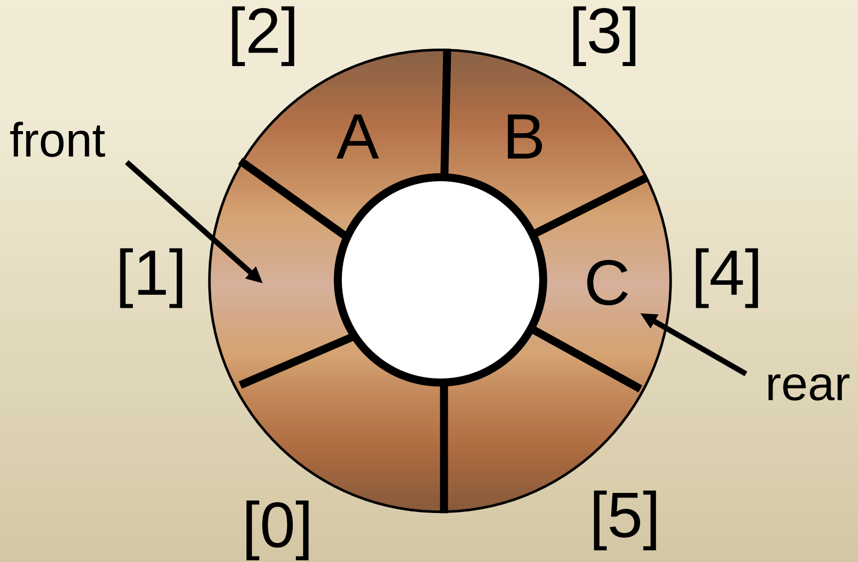


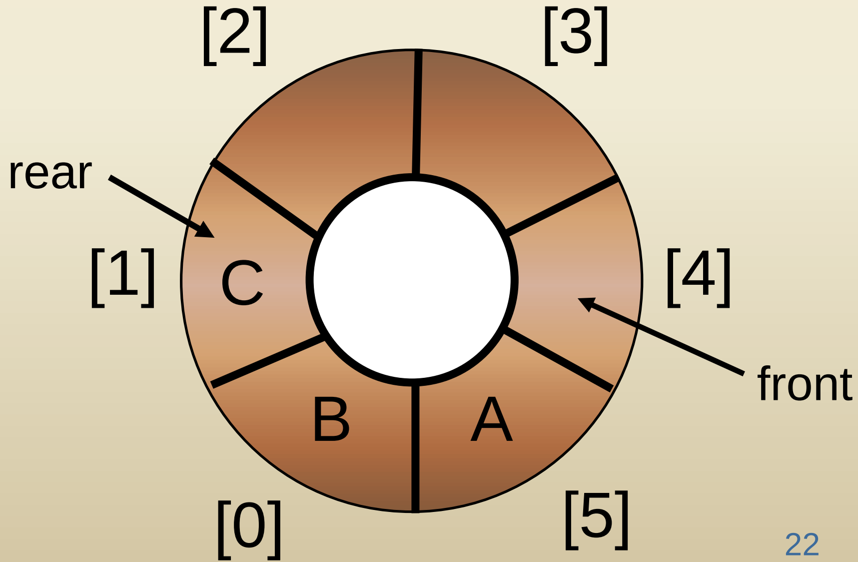
- Circular view of array.



Circular Representation

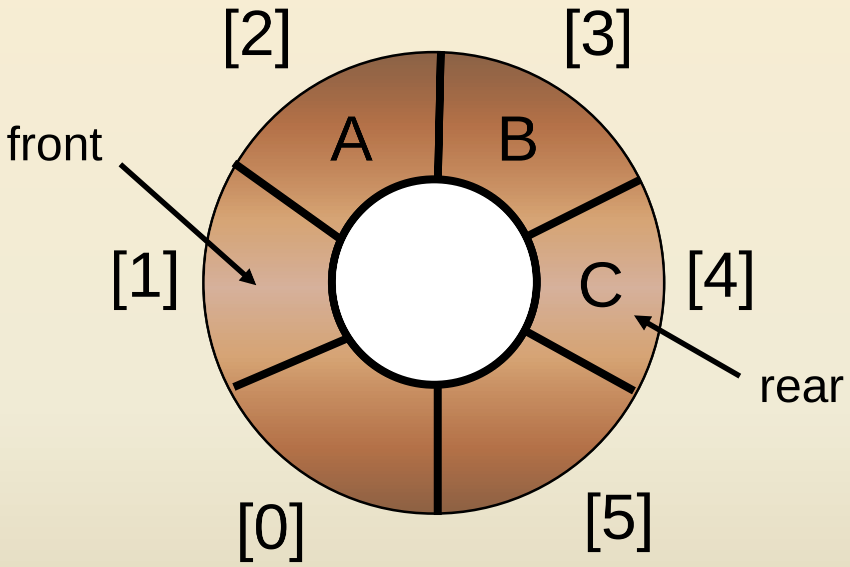
- Use integer variables front and rear.
  - front is one position counterclockwise from the first element
  - rear gives the position of the last element





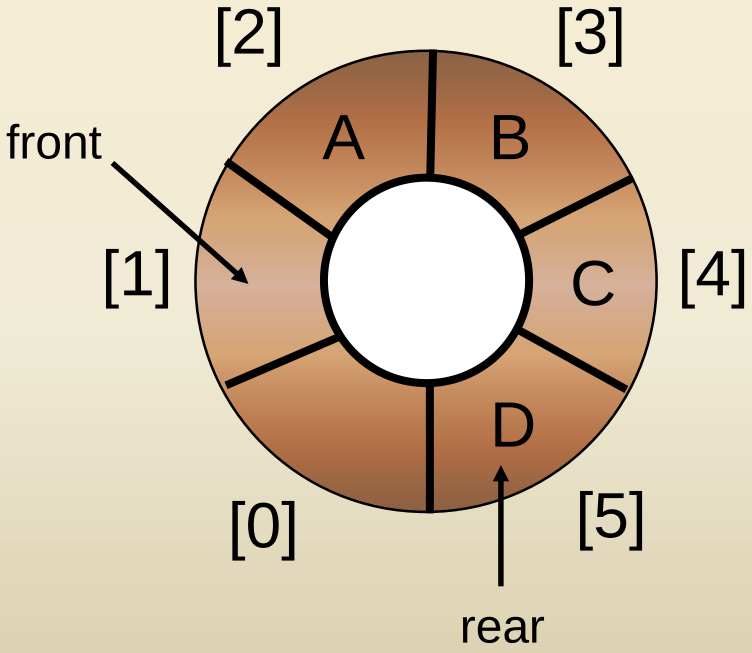
Push an Element

- Move rear one clockwise.



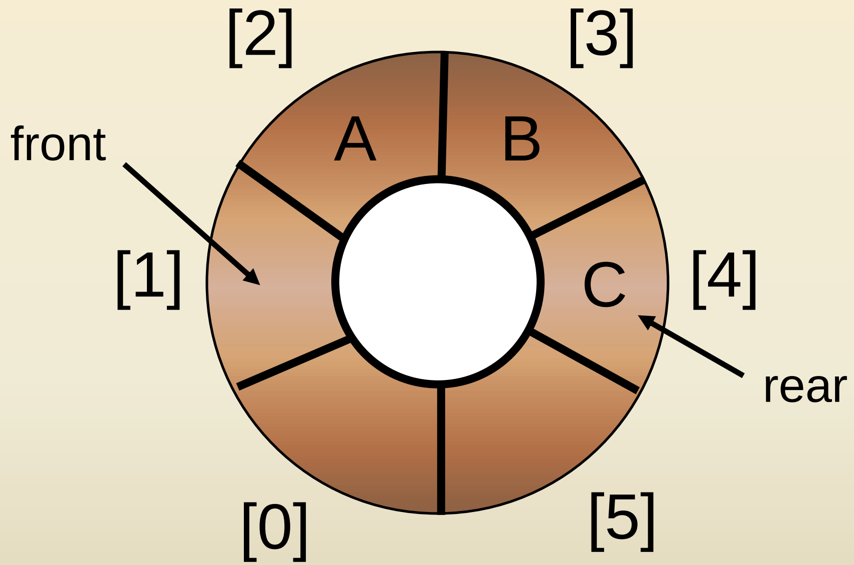
Push an Element

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



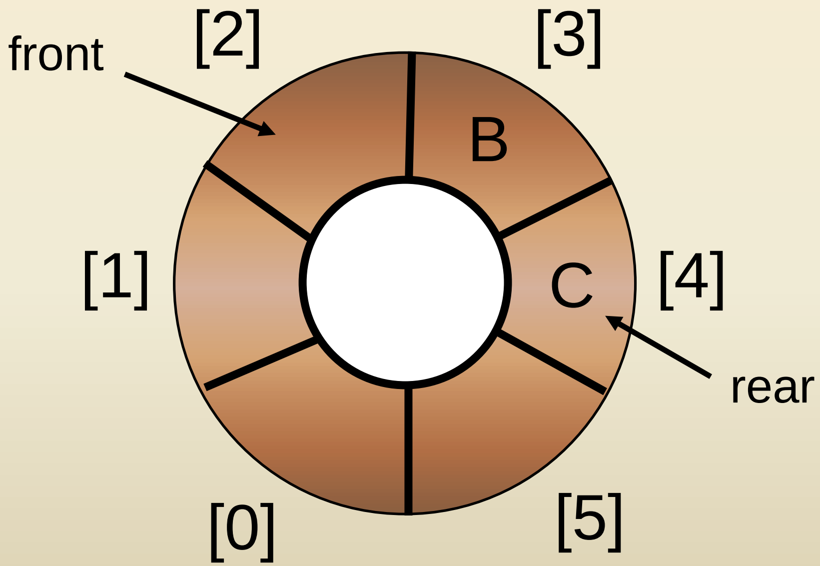
Pop an Element

- Move front one clockwise.



Pop an Element

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

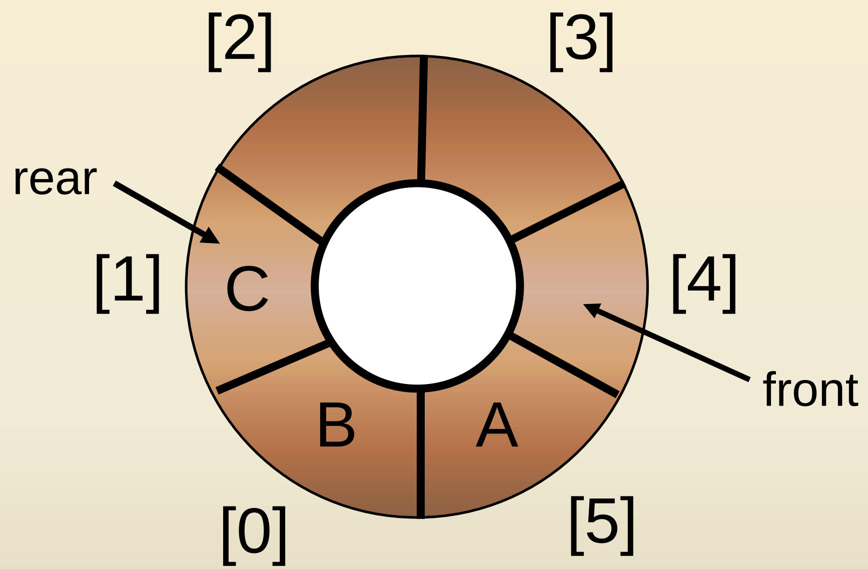


Recap: Regarding an Array as a Circular Queue

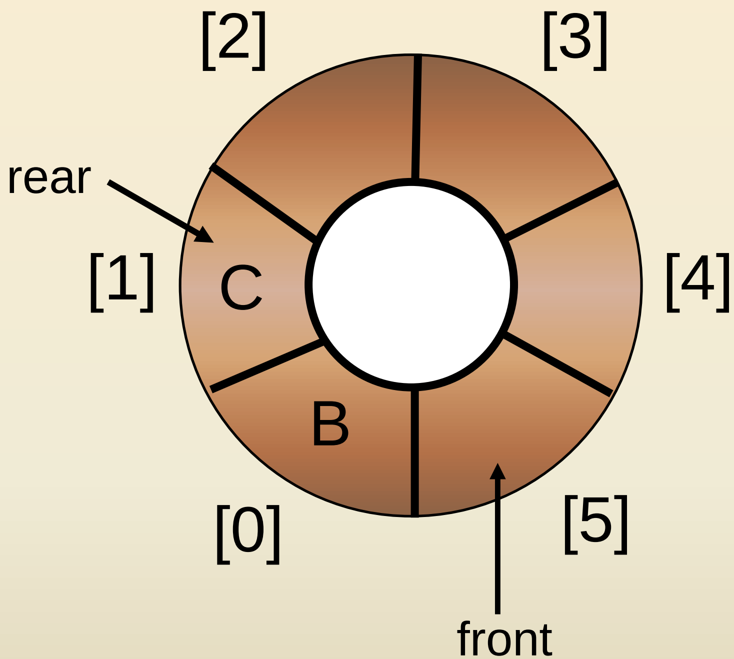
- Two indices
  - front: one position counterclockwise from the first element
  - rear: current end

- Problem with this representation:
  - when rear = front, we cannot distinguish whether a circular queue is full or empty.
  - Discuss this further now.

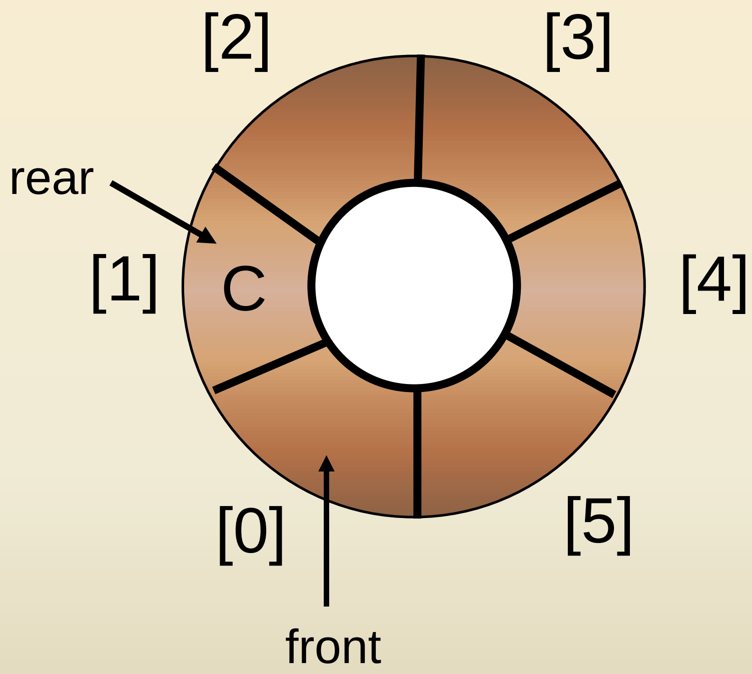
Empty the Queue



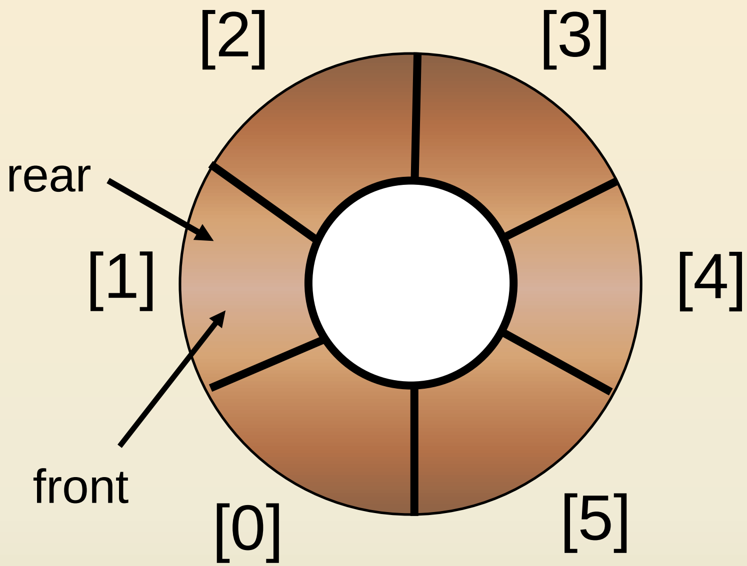
Empty the Queue



Empty the Queue

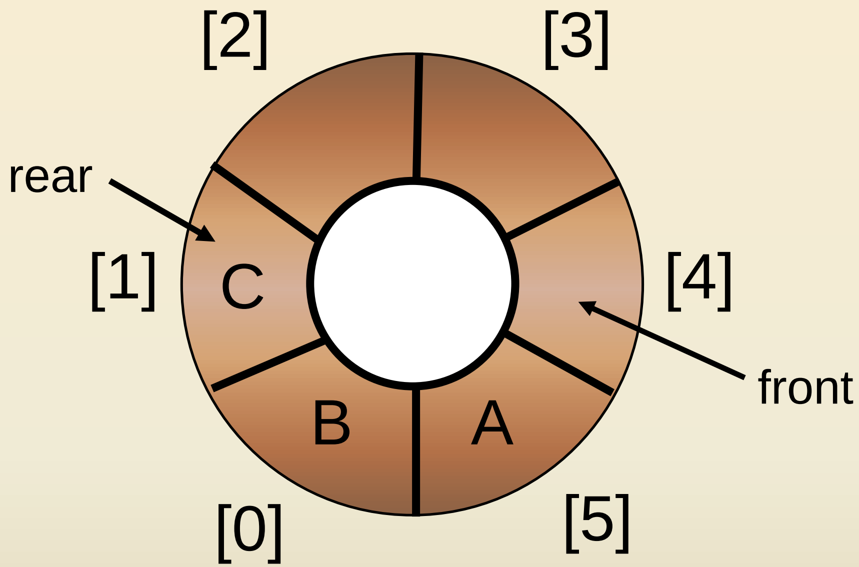


Empty the Queue

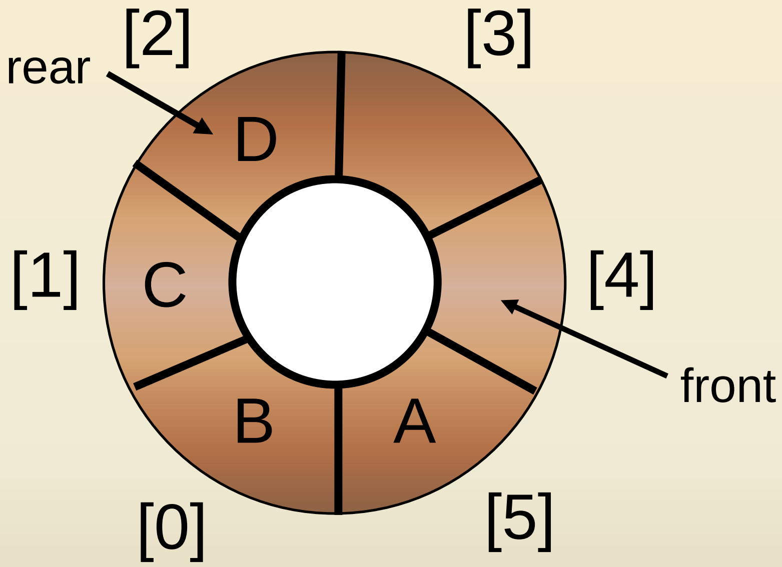


- So after a bunch of removes, the queue is empty ➔ front = rear.

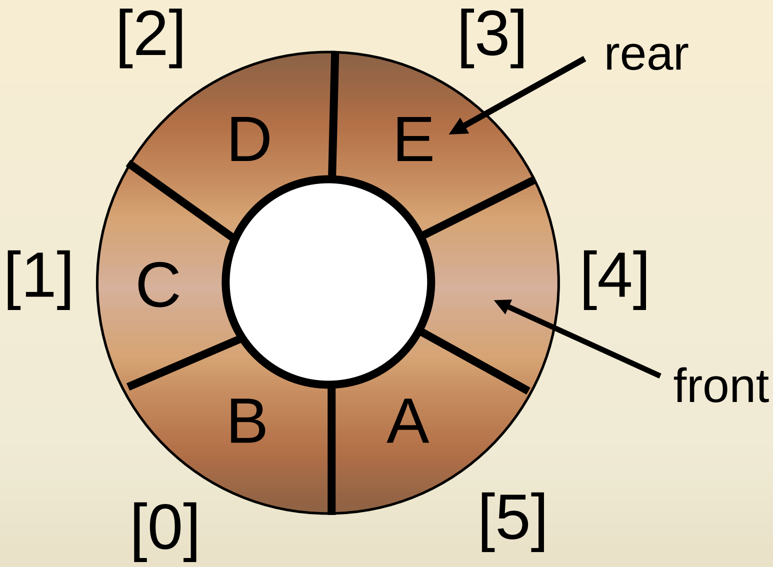
Reverse Operation – Stuff the Tank



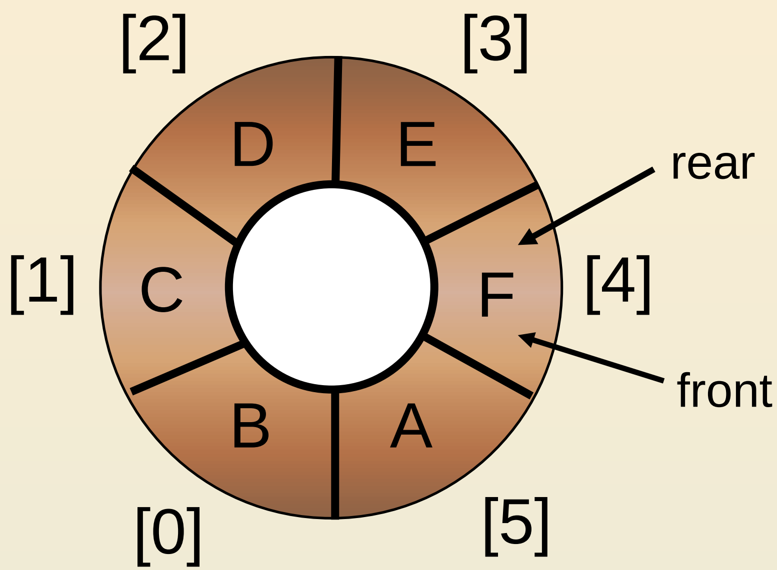
Stuff the Tank



Stuff the Tank



Stuff the Tank Please



- After a bunch of “pushes,” the queue is full ➔ front = rear.
- Since front = rear is also true for empty queues
  - So we cannot distinguish between a full queue and an empty queue!

Remedies

- The 1st strategy
  - Define a boolean variable "lastOperationIsPush"
    - Following each push set this variable to true.
    - Following each pop set to false.
    - Queue is full iff (front == rear) && lastOperationIsPush
    - Queue is empty iff (front == rear) && !lastOperationIsPush

Remedies

- Alternatively,
  - Define an integer variable size to store #elements
    - 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.

Example – A Mazing Problem

- Mazing problem
  - a rat finding a path out of a maze
  - A classical problem in experimental psychology
  - At the end of the maze is a nice chunk of cheese to tempt the rat to find it
  - Idea: run the experiment repeatedly until the rat zips through the maze without taking a single false path
  - Record the time consumed by the rat, taking averages and other statistics to yield its learning curve

Example – A Mazing Problem

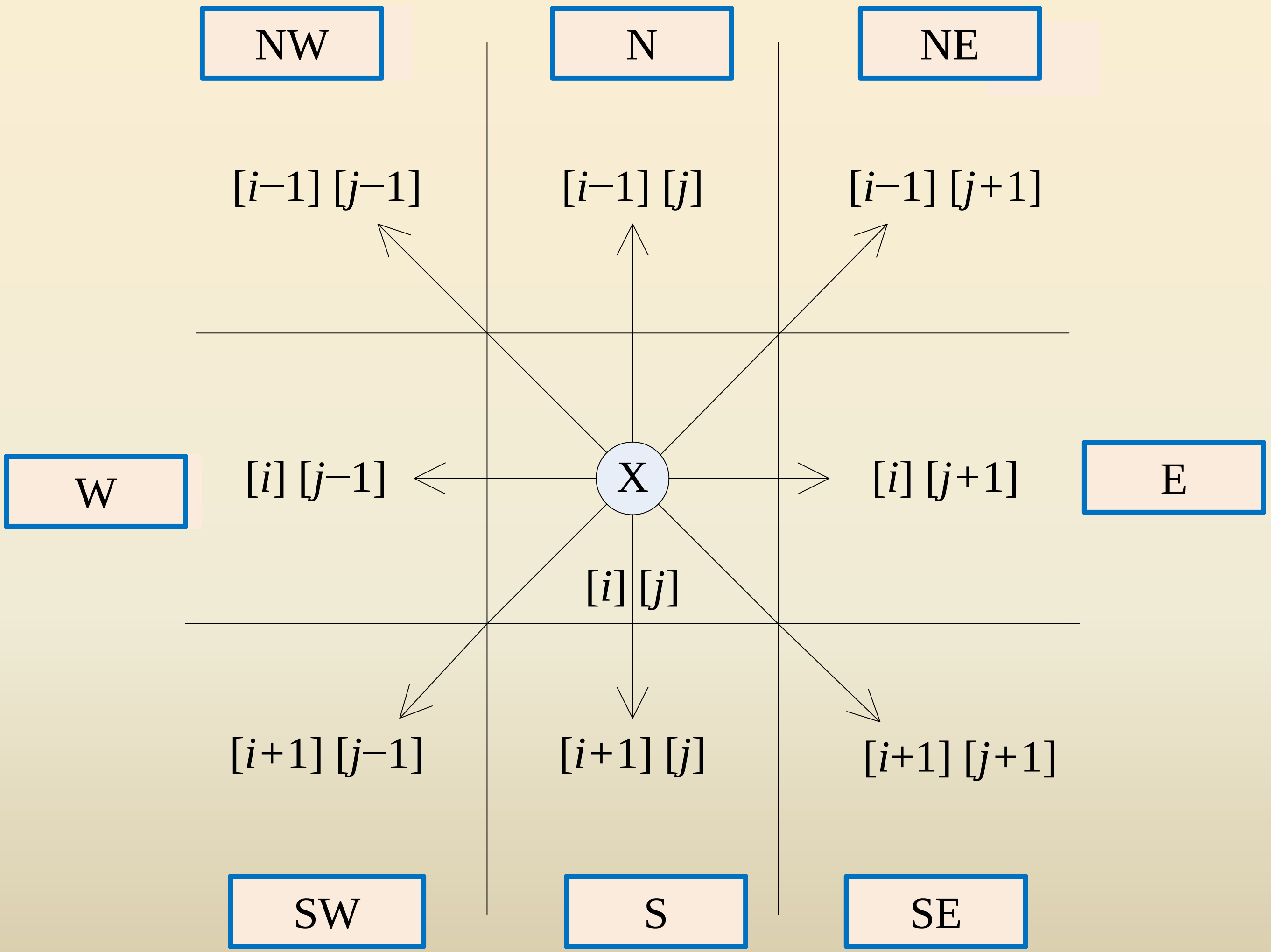
Entrance
maze[1][1]

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	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
maze[m][p]

maze[i][j] = 1 → blocked path;
= 0 → available path

Allowable Moves



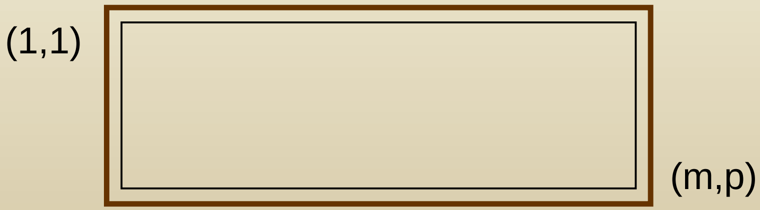
What Data Structures Are Needed First?

- For availability of position (i, j)
  - Using an array "*maze*"
  - maze[i][j] = 1 → blocked path;
    = 0 → available path

- Need to store the rat's current coordinate (x, y) and moving directions (using array "move")

Data Structures Needed First

- To prevent the rat from going down the same path twice
  - Using array "mark"
    - mark[i][j] initially set to 0,
      - will be set to 1 once visiting that position
    - To confine the rat in the maze,
      - the size of mark is increased to (m+2)*(p+2) for borders,
      - which are set to 1 so the rat cannot move there.



(m+2)*(p+2) to build borders

Table of Moves

Direction(q)	move[q].a	move[q].b
N(0)	-1	0
NE(1)	-1	1
E(2)	0	1
SE(3)	1	1
S(4)	1	0
SW(5)	1	-1
W(6)	0	-1
NW(7)	-1	-1

Data Structures for Next Moves

```
struct offsets {
        int a; // to be used in move[q].a
               // (表示列的位移量)
        int b; // to be used in move[q].b
               // (表示行的位移量)
};

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

offsets move[8];
/*array of moves for each direction*/
```

Data Structures for Setting Next Moves

- If the rat is currently at position (i, j), and
  - wish to move to the southwest location (g, h),
  - then set
  - g = i + move[SW].a;
  - h = j + move[SW].b;

SW = 5 (see p. 43)

- We will show how to use the "stack" data structure to solve this problem, where an element in the stack is (*x*, *y*, *q*)

"x, y" are the coordinates

"q" is the moving direction

First Glance on Stack

entrance →	0	0	0	0	
	0	1	0	0	
	0	1	1	1	
	0	0	0	0	→ exit

Position = (1,1)

x y

Initially at (1,1) Mark (1,1)

x y q

stack: ***(1,1,2)*** — ***(1,2) Mark (1,2)***

stack: ***(1,2,2) (1,1,2)*** — ***(1,3) Mark (1,3)***

stack: ***(1,3,2) (1,2,2) (1,1,2)*** — ***(1,4) Mark (1,4)***

stack: ***(1,4,4) (1,3,2) (1,2,2) (1,1,2)*** — ***(2,4) Mark (2,4)***

(2,4,6) (1,4,4) (1,3,2) (1,2,2) (1,1,2) — stack — ***(2,3) Mark (2,3)***

Deadend, so move back and pop (2,4,6) out of the stack

(1,4,4) (1,3,2) (1,2,2) (1,1,2) — stack — ***(2,4)***

Move back again, and pop (1,4,4) out of the stack since (1,4) has been visited

(1,3,2) (1,2,2) (1,1,2) — stack — ***(1,4)***

…

First Pass at Finding a Path through a Maze

Program 3.15 (will be getting clearer if you can follow)

```
initialize list to the maze entrance coordinates and direction east;
While (list is not empty)
{
    (i, j, dir) = coordinates and direction from end of list;
    …
    while (there are more moves from (i,j))
    {
        (g, h) = coordinates of next move;
        if ((g == m) && (h == p)) success;
        if ((!maze[g][h]) // legal move
                && (!mark[g][h]) //haven't been here before
        {
            mark[g][h] = 1;
            dir = next direction to try;
            add (i, j, dir) to end of list;
            …
        }
    }
}
Cout << "No path in maze." << endl;
```

Will be used as a stack

If a path really exists, the stack will store the solution (see example later on); o.w., the stack will be empty eventually and no solution is available.

Recap: Using Stack to Keep Past History

- Save the past positions and directions in a stack, in order to
  - return back to the previous positions and try new directions if the rat took the wrong path.
  - The coming example offers a good understanding, but before this…
  - What is the maximal size of the stack?
    - A maze is represented by a 2-dimensional array *maze*[*m*][*p*]
    - Since each position is visited at most once, at most $m \times p$ elements can be placed in the stack
    - Need this info to declare the size of the stack

Data Structures for the Stack

- ***stack*** can be defined as a stack of ***Items***, where
  - the struct ***Items*** is defined as

    struct *Items* {

    int x, y, dir; //coordinates and moving directions

    }
  - The updating process of this data structure can be built from 3 arrays:
    - maze (storing the legal positions)
    - move (storing the moving directions)
    - mark (storing the positions visited)

Stack Revisited

entrance →

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

→ exit

x y

Initially at ***(1,1)*** ***Mark*** ***(1,1)***

x y q

stack

(1,1,2) ***(1,2)*** ***Mark*** ***(1,2)***

stack

(1,2,2) ***(1,3)*** ***Mark*** ***(1,3)***
(1,1,2)

stack

(1,3,2) ***(1,4)*** ***Mark*** ***(1,4)***
(1,2,2)
(1,1,2)

stack

(1,4,4) ***(2,4)*** ***Mark*** ***(2,4)***
(1,3,2)
(1,2,2)
(1,1,2)

(2,4,6) ***(2,3)*** ***Mark*** ***(2,3)***
(1,4,4)
(1,3,2)
(1,2,2)
(1,1,2)

stack

> Deadend, so move back and pop (2,4,6) out of the stack

(1,4,4) ***(2,4)***
(1,3,2)
(1,2,2)
(1,1,2)

stack

> Move back again, and pop (1,4,4) out of the stack since (1,4) has been visited

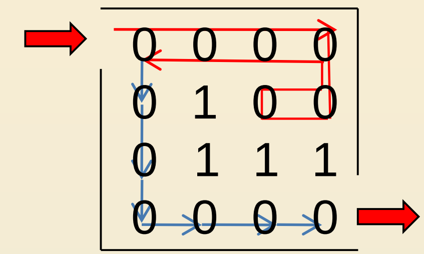
(1,3,2) ***(1,4)***
(1,2,2)
(1,1,2)

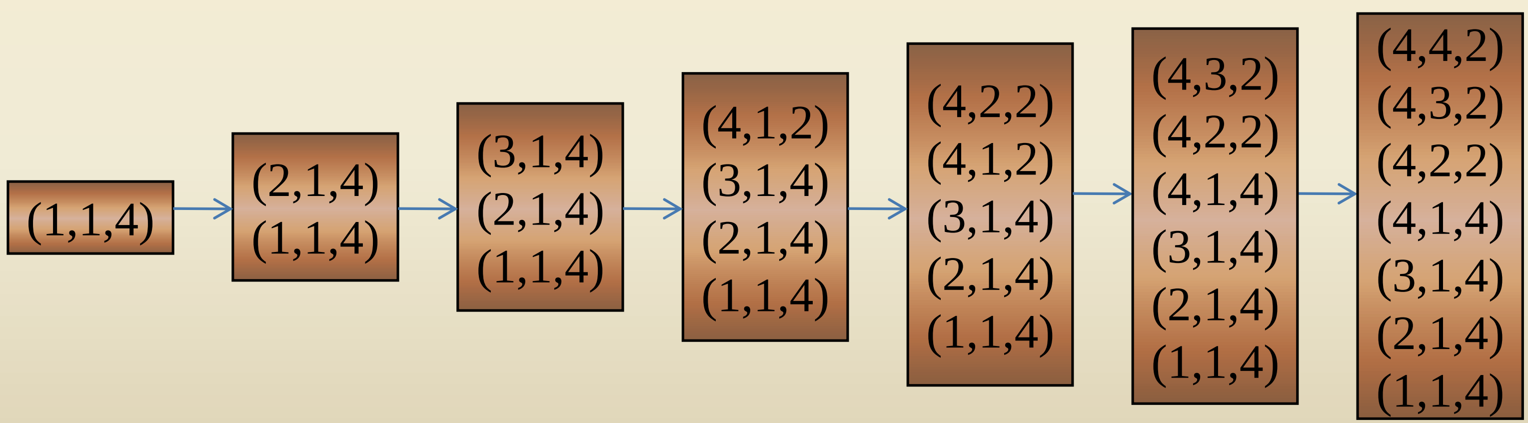
stack

…

Stack Revisited

- Moving back again and again, and unstack until position (1,1)
- Then start exploring the south direction and re-build the stack





- Out. Good job!!!

Homework

- Available on E-learning
- Due May/4/2018