Data Structures & Algorithms in C++

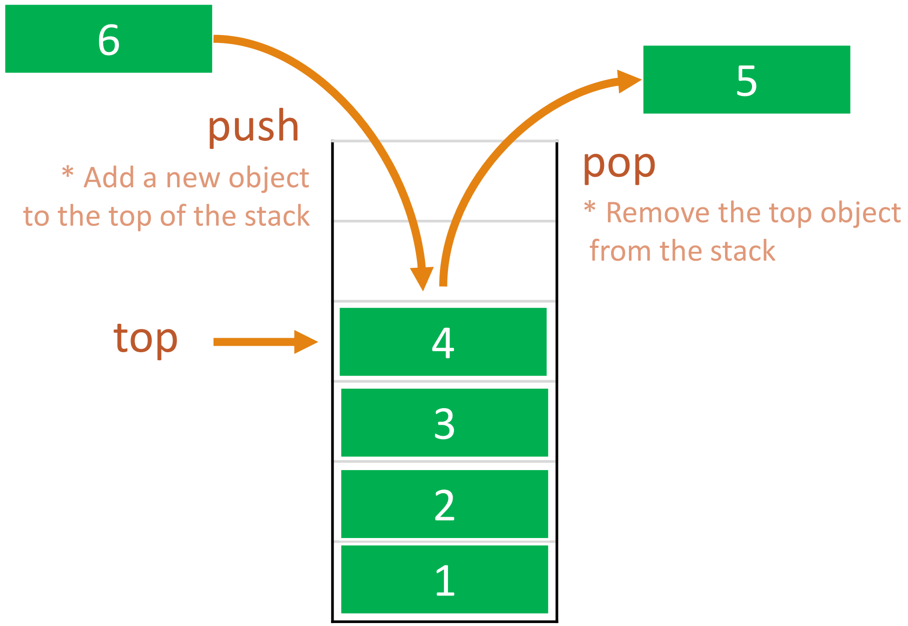
WEEK 05

Chap 5. Stacks, Queues, and Deques

- Stacks
- Queues
- Deques (Double-Ended Queues)
- A Maze Problem (not in textbook)

Stacks

- A Stack is a one-ended linear data structure which models a real world stack by having two primary operations, namely push and pop



- When and where is a Stack used?
  - Used by undo mechanism in text editors
  - Used in compiler syntax checking for matching brackets and braces
  - Can be used to model a pile of books or plates
  - Used behind the scenes to support recursion by keeping track of previous function calls
  - Can be used to do a Depth First Search (DFS) on a graph

Stacks
- Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

- Using various ADTs as building blocks to handle data efficiently
  - **Stacks** : could be used to track trading history or undo operations
  - **Queues** : might manage orders waiting to be processed
  - **Lists/Dictionaries** : could store stock information, prices, and customer portfolios

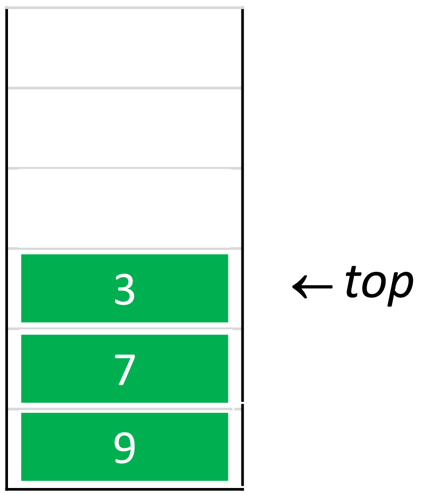
Stacks
- The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the **Last-In First-Out (LIFO)** scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(e): add element e at the top
  - pop(): remove the top element (Error : if empty)
  - top(): return the top element without removing it (Error : if empty)
- Auxiliary stack operations:
  - size(): return the number of elements in the stack
  - empty(): check if the stack is empty (return true/false)

Stacks
- Example of Stack Operation

Operation	Output	Stack Contents	Operation	Output	Stack Contents
push(5)	–	(5)	push(9)	–	(9)
push(3)	–	(5,3)	push(7)	–	(9,7)
pop()	–	(5)	push(3)	–	(9,7,3)
push(7)	–	(5,7)	push(5)	–	(9,7,3,5)
pop()	–	(5)	size()	4	(9,7,3,5)
top()	5	(5)	pop()	–	(9,7,3)
pop()	–	()	push(8)	–	(9,7,3,8)
pop()	“error”	()	pop()	–	(9,7,3)
top()	“error”	()	top()	3	(9,7,3)
empty()	true	()			

Stack



Stacks
- Stack Interface in C++

- Stack ADT (different from the built-in C++ STL class stack)
  - Focuses on the API (Application Programming Interface)
  - Describes public members and their usage but excludes private data members
  - Member functions “size”, “empty”, and “top” are declared as **const**, ensuring they don’t alter the stack’s contents

```
template <typename E>
class Stack {                                    // an interface for a stack
public:
  int size() const;                              // number of items in stack
  bool empty() const;                            // is the stack empty?
  const E& top() const throw(StackEmpty);        // the top element
  void push(const E& e);                         // push x onto the stack
  void pop() throw(StackEmpty);                  // remove the top element
};
```

Stacks
- Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting pop or top on an empty stack throws a StackEmpty exception

```
// Exception thrown on performing top or pop of an empty stack.
class StackEmpty : public RuntimeException {
public:
  StackEmpty(const string& err) : RuntimeException(err) {}
};
```

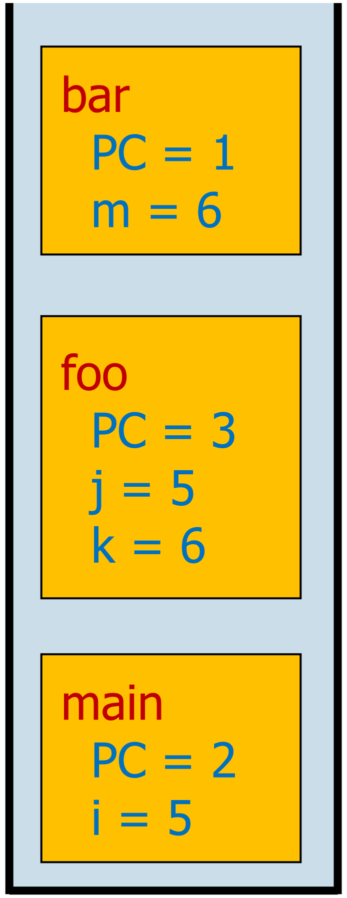
Stacks
- Applications

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the C++ run-time system
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Stacks
- C++ Run-Time Stack

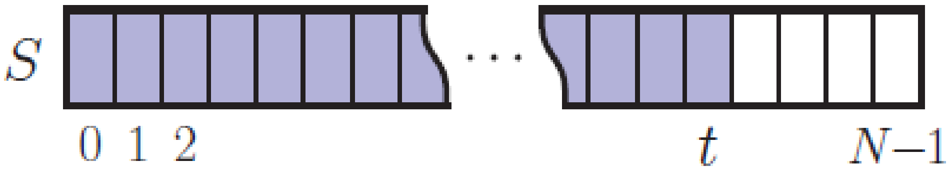
- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When the function ends, its frame is popped from the stack and control is passed to the function on top of the stack
- Allows for recursion

```
main() {
        int i = 5;
        foo(i);
        }

foo(int j) {
        int k;
        k = j+1;
        bar(k);
        }

bar(int m) {
        …
        }
```



Stacks
- Array-based Stack

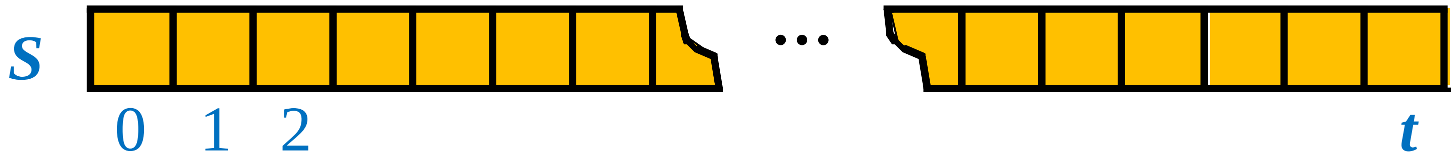
- Structures
  - Use an N**-element array (**S**)** and an integer variable t
  - t presents the index of the top element in the array
  - Initial value of t : -1 (indicating the stack is empty)
  - Size of stack : t + 1
- Algorithms
  - size() : returns t+1
  - empty(): returns true if t<0; otherwise false



Algorithm size():
  return $t+1$
Algorithm empty():
  return $(t<0)$
Algorithm top():
  if empty() **then**
    throw StackEmpty exception
  return $S[t]$
Algorithm push(e):
  if size() $= N$ **then**
    throw StackFull exception
  $t \leftarrow t+1$
  $S[t] \leftarrow e$
Algorithm pop():
  if empty() **then**
    throw StackEmpty exception
  $t \leftarrow t-1$

Stacks
- Array-based Stack (cont.)

- Exception Handling
  - **StackFull** : Signals an error when attempting to push onto a full stack
  - **StackEmpty** : Signals an error for top or pop operations on an empty stack
- A push operation will throw a StackFull exception
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT



Algorithm size():
    return $t+1$
Algorithm empty():
    return $(t<0)$
Algorithm top():
    if empty() **then**
        throw StackEmpty exception
    return $S[t]$
Algorithm push(e):
    if size() $= N$ **then**
        throw StackFull exception
    $t \leftarrow t+1$
    $S[t] \leftarrow e$
Algorithm pop():
    if empty() **then**
        throw StackEmpty exception
    $t \leftarrow t-1$

Stacks
- Performance and Limitations

- Performance
  - Let $\boldsymbol{n}$ be the number of elements in the stack
  - The space used is $\boldsymbol{O}(\boldsymbol{n})$
  - Each operation (size, empty, top, push, pop) runs in time $\boldsymbol{O}(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Stacks
- Array-based Stack in C++

- Class Definition
  - A constructor with a default capacity (DEF_CAPACITY)
  - Utilize default arguments and an enumeration for simplicity
  - Storage is a dynamically allocated array of type E

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };           // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);    // constructor from capacity
  int size() const;                      // number of items in the stack
  bool empty() const;                    // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);          // pop the stack
  // ...housekeeping functions omitted
private:                                 // member data
  E* S;                                  // array of stack elements
  int capacity;                          // stack capacity
  int t;                                 // index of the top of the stack
};
```

Stacks
- Array-based Stack in C++ (cont.)

- Member Functions
  - top() and pop():
    - Check if the stack is empty before proceeding
    - Throw "StackEmpty" exception if empty
  - push(e):
    - Check if the stack is full before adding elements
    - Throw "StackFull" exception if full
- Implementation Notes
  - Constructor initialize array storage, capacity, and the top index (***t***)

```
template <typename E> ArrayStack<E>::ArrayStack(int cap)
  : S(new E[cap]), capacity(cap), t(-1) { } // constructor from capacity

template <typename E> int ArrayStack<E>::size() const
  { return (t + 1); }                       // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
  { return (t < 0); }                       // is the stack empty?

template <typename E>                       // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S[t];
}

template <typename E>                       // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
  if (size() == capacity) throw StackFull("Push to full stack");
  S[++t] = e;
}

template <typename E>                       // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --t;
}
```

Stacks
- Example use in C++

- Limitations of ArrayStack
  - Default capacity "**N**=100" is arbitrary but adjustable in the constructor
  - Waste of memory if actual usage is less than **N**
  - Stack Overflow if more elements are pushed than the capacity
- Enhancements : using STL (std::vector)
  - Automatically expands when stack overflows
  - Offers better flexibility for dynamic sizing
- Practical Considerations
  - Array-based implementation is fast and simple with good estimate of the required stack size

```
ArrayStack<int> A;                       // A = [], size = 0
A.push(7);                               // A = [7*], size = 1
A.push(13);                              // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();        // A = [7*], outputs: 13
A.push(9);                               // A = [7, 9*], size = 2
cout << A.top() << endl;                 // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();        // A = [7*], outputs: 9
ArrayStack<string> B(10);                // B = [], size = 0
B.push("Bob");                           // B = [Bob*], size = 1
B.push("Alice");                         // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();        // B = [Bob*], outputs: Alice
B.push("Eve");                           // B = [Bob, Eve*], size = 2
```

Stacks

- Implementing a Stack with Generic Linked List

```
typedef string Elem;                            // stack element type
class LinkedStack {                             // stack as a linked list
public:
  LinkedStack();                                // constructor
  int size() const;                             // number of items in the stack
  bool empty() const;                           // is the stack empty?
  const Elem& top() const throw(StackEmpty);   // the top element
  void push(const Elem& e);                     // push element onto stack
  void pop() throw(StackEmpty);                 // pop the stack
private:                                        // member data
  SLinkedList<Elem> S;                          // linked list of elements
  int n;                                        // number of elements
};
```

```
LinkedStack::LinkedStack()
  : S(), n(0) { }                               // constructor

int LinkedStack::size() const
  { return n; }                                 // number of items in the stack

bool LinkedStack::empty() const
  { return n == 0; }                            // is the stack empty?

                                                // get the top element
const Elem& LinkedStack::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S.front();
}
void LinkedStack::push(const Elem& e) {         // push element onto stack
  ++n;
  S.addFront(e);
}
                                                // pop the stack
void LinkedStack::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --n;
  S.removeFront();
}
```

Stacks

- Reversing a Vector Using a Stack

- A stack can reverse the elements in a vector through a **non-recursive algorithm**
  - Utilize the **array-reversal problem** approach
- Insert all elements of the vector into the stack in order (push), and remove elements from the stack, filling the vector back up in reverse order (pop)
- Advantage : avoids recursion, simplifying implementation

```
template <typename E>
void reverse(vector<E>& V) {                 // reverse a vector
  ArrayStack<E> S(V.size());
  for (int i = 0; i < V.size(); i++)         // push elements onto stack
    S.push(V[i]);
  for (int i = 0; i < V.size(); i++) {       // pop them in reverse order
    V[i] = S.top();  S.pop();
  }
}
```

Stacks
- Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Stacks
- Parentheses Matching Algorithm

Algorithm ParenMatch(X,n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i ← 0 to n − 1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.empty() then
            return false          {nothing to match with}
        if S.top() does not match the type of X[i] then
            return false          {wrong type}
        S.pop()
if S.empty() then
    return true          {every symbol matched}
else
    return false          {some symbols were never matched}
```

Stacks
- Evaluating Arithmetic Expressions

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group
evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Stacks
- Algorithms for Evaluating Expressions

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤ prec(opStk.top())
doOp()
```

Algorithm EvalExp()

Input: a stream of tokens representing an arithmetic expression (with numbers)

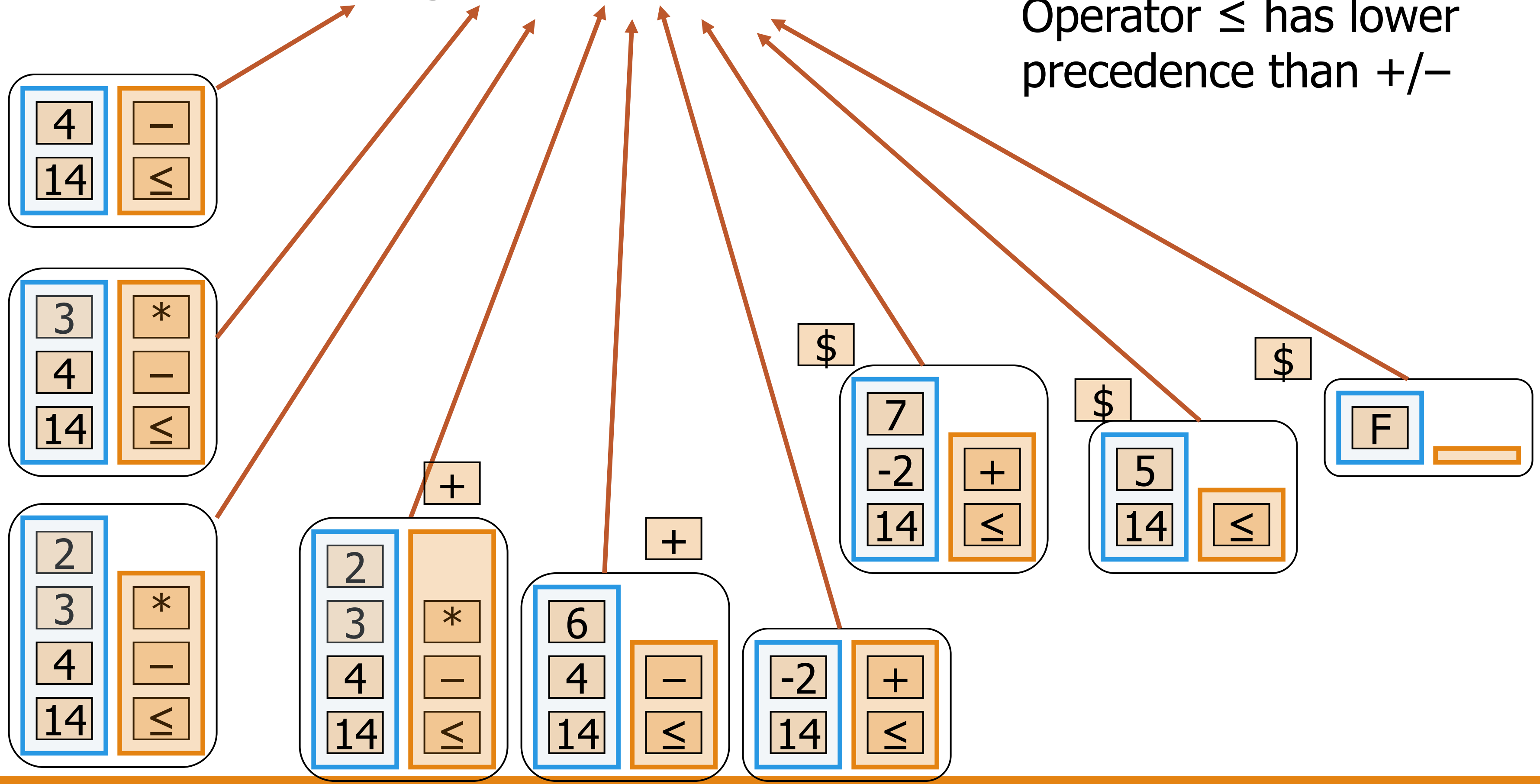
Output: the value of the expression

```
while there's another token z
    If isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Stacks
- Algorithms on an Example Expression

14 ≤ 4 – 3 * 2 + 7

Operator ≤ has lower precedence than +/–



Stacks
- Tags Matching in HTML

- HTML tags : an HTML document and its rendering

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even
as a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

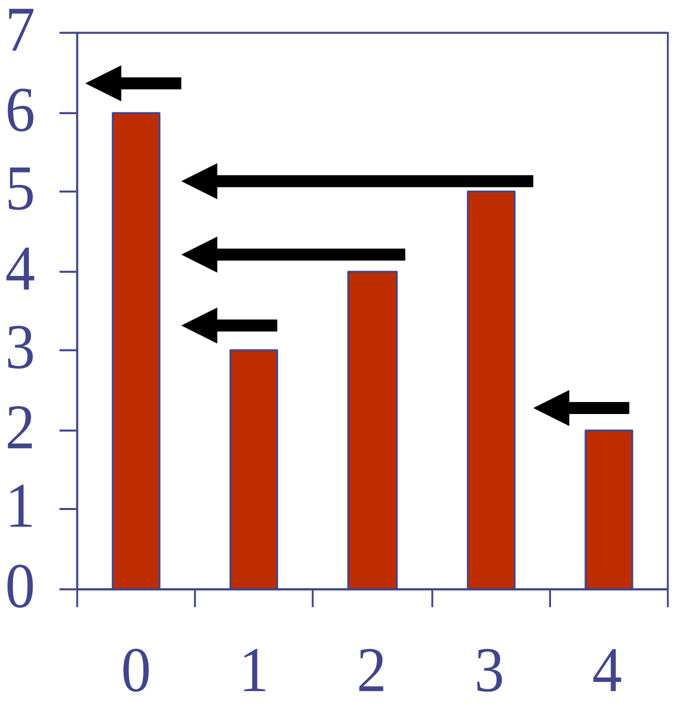
Stacks
- Tags Matching in HTML (C++ code)

```
vector<string> getHtmlTags() {                    // store tags in a vector
  vector<string> tags;                            // vector of html tags
  while (cin) {                                   // read until end of file
    string line;
    getline(cin, line);                           // input a full line of text
    int pos = 0;                                  // current scan position
    int ts = line.find("<", pos);                 // possible tag start
    while (ts != string::npos) {                  // repeat until end of string
      int te = line.find(">", ts+1);              // scan for tag end
      tags.push_back(line.substr(ts, te−ts+1));   // append tag to the vector
      pos = te + 1;                               // advance our position
      ts = line.find("<", pos);
    }
  }
  return tags;                                    // return vector of tags
}

int main() {                                      // main HTML tester
  if (isHtmlMatched(getHtmlTags()))               // get tags and test them
    cout << "The input file is a matched HTML document." << endl;
  else
    cout << "The input file is not a matched HTML document." << endl;
}
```

```
                                                  // check for matching tags
bool isHtmlMatched(const vector<string>& tags) {
  LinkedStack S;                                  // stack for opening tags
  typedef vector<string>::const_iterator Iter;   // iterator type
                                                  // iterate through vector
  for (Iter p = tags.begin(); p != tags.end(); ++p) {
    if (p−>at(1) != '/')                          // opening tag?
      S.push(*p);                                 // push it on the stack
    else {                                        // else must be closing tag
      if (S.empty()) return false;                // nothing to match - failure
      string open = S.top().substr(1);            // opening tag excluding '<'
      string close = p−>substr(2);                // closing tag excluding '</'
      if (open.compare(close) != 0) return false; // fail to match
      else S.pop();                               // pop matched element
    }
  }
  if (S.empty()) return true;                     // everything matched - good
  else return false;                              // some unmatched - bad
}
```

Stacks
- Computing Spans (not in textbook)

- Using a stack as an auxiliary data structure in an algorithm
- Given an an array ***X***, the span ***S***[***i***] of ***X***[***i***] is the maximum number of consecutive elements ***X***[***j***] immediately preceding ***X***[***i***] and such that ***X***[***j***] $\leq$ ***X***[***i***]
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

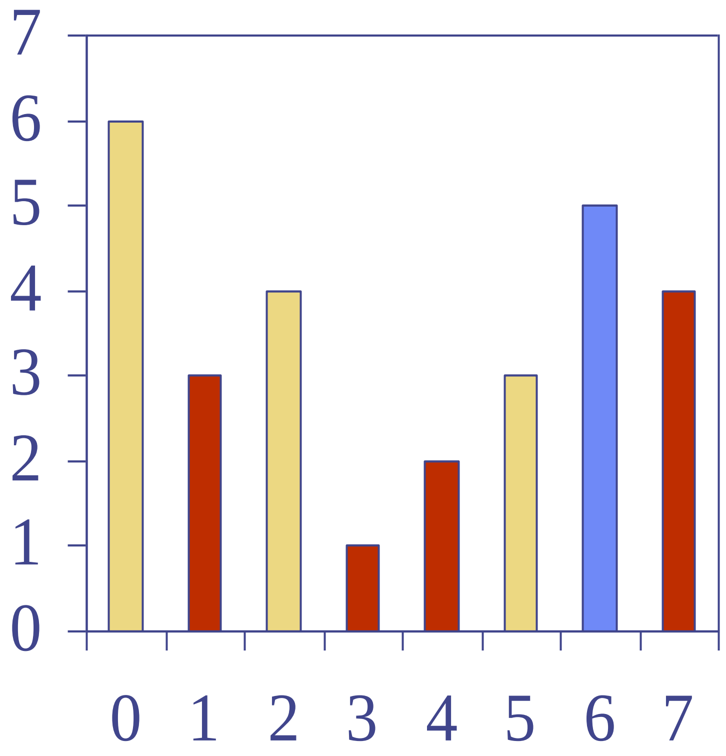
Stacks
- Quadratic Algorithm

Algorithm ***spans1*** (X, n)	
Input array X of n integers	
Output array S of spans of X	**#**
$S \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow 1$	n
while $s \leq i \wedge X[i - s] \leq X[i]$	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + 1$	$1 + 2 + \ldots + (n - 1)$
$S[i] \leftarrow s$	n
return S	1

- Algorithm ***spans1*** runs in $O(n^2)$ time

Stacks
- Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let $\boldsymbol{i}$ be the current index
  - We pop indices from the stack until we find index $\boldsymbol{j}$ such that $\boldsymbol{X}[\boldsymbol{i}] < \boldsymbol{X}[\boldsymbol{j}]$
  - We set $\boldsymbol{S}[\boldsymbol{i}] \leftarrow \boldsymbol{i} - \boldsymbol{j}$
  - We push $\boldsymbol{x}$ onto the stack



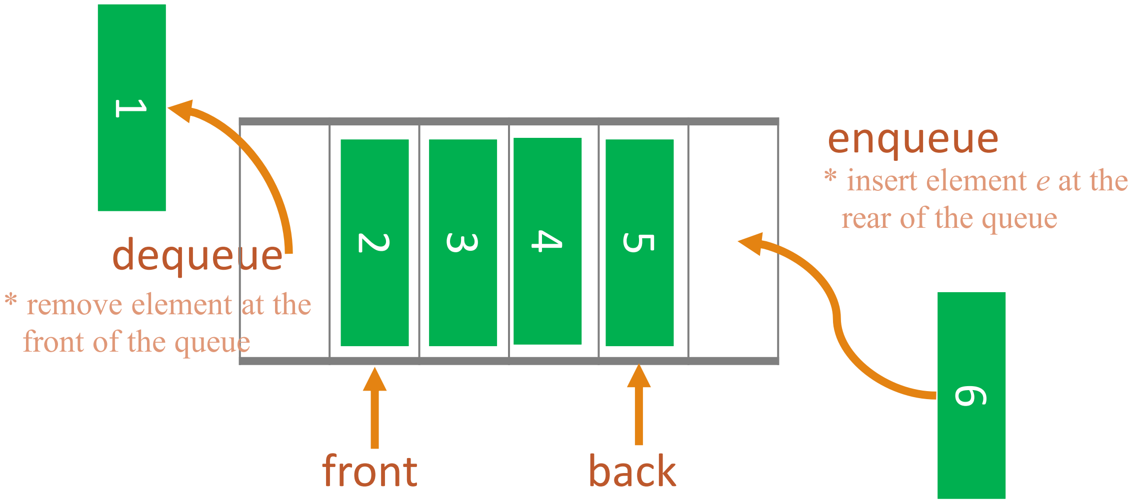
Stacks
- Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

Algorithm *spans2*(X, n)	#
$S \leftarrow$ new array of n integers	n
$A \leftarrow$ new empty stack	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
while $(\neg A.empty() \land$	
$X[A.top()] \leq X[i]$) **do**	n
$A.pop()$	n
if $A.empty()$ **then**	n
$S[i] \leftarrow i + 1$	n
else	
$S[i] \leftarrow i - A.top()$	n
$A.push(i)$	n
return S	1

Queues

- A Queue is a linear data structure which models real world queues by having two primary operations, namely enqueue and dequeue



- When and where is a Queue used?
  - Any waiting line models a queue, for example a lineup at a movie theater or restaurant
  - Can be used to efficiently keep track of the x most recently added elements
  - Web server request management where you want first come first service
  - Can be used to do a Breadth First Search (BFS) on a graph

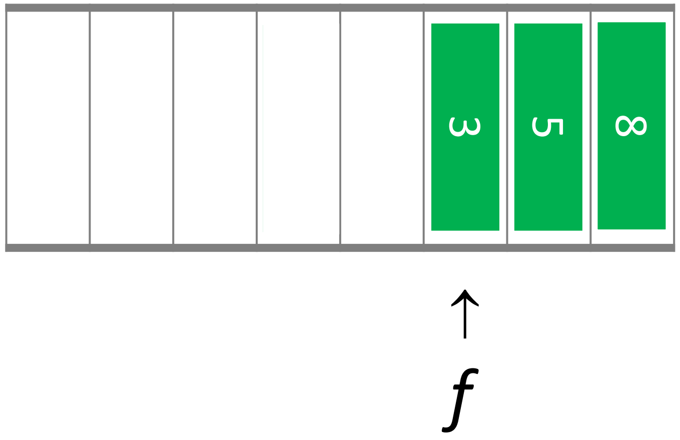
Queues
- Abstract Data Types (ADTs)

- The Queue ADT stores arbitrary objects
- A container of elements following the **First-In First-Out (FIFO)** principle
- Insertions at the rear, and removals from the front
- Main queue operations:
  - enqueue(e): insert element e at the rear
  - dequeue(): remove the front element
- Auxiliary queue operations:
  - front(): return the front element without removing it
  - size(): return the number of elements
  - empty(): check if the queue is empty
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an QueueEmpty

Queues
- Example of Queue Operation

Operation	Output	Queue	Operation	Output	Queue
enqueue(5)	–	(5)	enqueue(9)	–	(9)
enqueue(3)	–	(5,3)	enqueue(7)	–	(9,7)
dequeue()	–	(3)	enqueue(3)	–	(9,7,3)
enqueue(7)	–	(3,7)	enqueue(5)	–	(9,7,3,5)
dequeue()	–	(7)	size()	4	(9,7,3,5)
front()	7	(7)	dequeue()	–	(7,3,5)
dequeue()	–	()	enqueue(8)	–	(7,3,5,8)
dequeue()	“error”	()	dequeue()	–	(3,5,8)
front()	“error”	()	front()	3	(3,5,8)
empty()	true	()			

Queue



Queues
- Applications of Queues

- Direct applications
  - Waiting lists
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Queues
- STL Queue

- STL provides an implementation of a queue (std::queue)
  - Underlying implementation: based on the STL vector class
- Principal Member Functions
  - size(): return the number of elements
  - empty(): check if the queue is empty
  - push(e): enqueue e at the rear
  - pop(): dequeue the element at the front
  - front(): return a reference to the front element
  - back(): return a reference to the rear element
- Operations front, back, and pop on the empty queue are **undefined** – no exception is thrown, but may result in **program abortion**

Queues
- Queue Interface in C++

- Queue ADT Interface
  - Member functions “size”, “empty”, and “front” return values without altering the queue. Declared as **const**, ensuring contents remain unchanged

```
template <typename E>
class Queue {                                   // an interface for a queue
public:
  int size() const;                             // number of items in queue
  bool empty() const;                           // is the queue empty?
  const E& front() const throw(QueueEmpty);     // the front element
  void enqueue (const E& e);                    // enqueue element at rear
  void dequeue() throw(QueueEmpty);             // dequeue element at front
};
```

  - Calling front() or dequeue() on the empty queue throws the exception QueueEmpty

```
class QueueEmpty : public RuntimeException {
public:
  QueueEmpty(const string& err) : RuntimeException(err) { }
};
```

Queues
- Array-based Queue

- Array-based queue with capacity (N)
  - Need to track front and rear efficiently
- Inefficient Approach
  - Dequeue operation requires shifting elements forward : time complexity O(n)

- Efficient Approach : Circular Array
  - Achieve O(1) for enqueue and dequeue
  - Define three variables

    f index of the front element

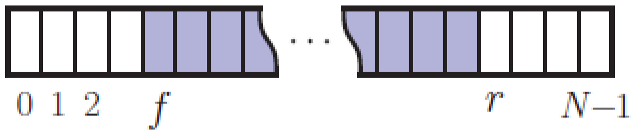
    r index after the rear element

    n number of elements in the queue
  - Operations

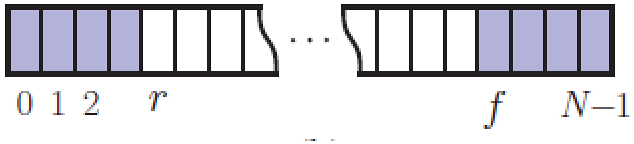
    enqueue increment ***r*** and ***n***

    dequeue increment ***f*** and decrement ***n***
  - Prevent out-of-bounds errors

normal configuration



wrapped-around configuration



Queues
- Queue Operations (Circular Array)

- Use ***n*** to determine size and emptiness
- Increment indices ***f*** and ***r*** using :
  - $(\boldsymbol{f}+\mathbf{1})\ \boldsymbol{mod}\ \boldsymbol{N}$
  - $(\boldsymbol{r}+\mathbf{1})\ \boldsymbol{mod}\ \boldsymbol{N}$
- Exception Handling
  - QueueFull : Signals when queue capacity is exceeded
  - QueueEmpty : Signals when front or dequeue() is called on an empty queue
- Performance
  - Queue operations (enqueue, dequeue) in O(1) time
  - Space usage : O(N), N is array size at creation

Algorithm size():
  return n
Algorithm empty():
  return $(n = 0)$
Algorithm front():
  if empty() **then**
    throw QueueEmpty exception
  return $Q[f]$
Algorithm dequeue():
  if empty() **then**
    throw QueueEmpty exception
  $f \leftarrow (f+1) \bmod N$
  $n = n - 1$
Algorithm enqueue(e):
  if size() $= N$ **then**
    throw QueueFull exception
  $Q[r] \leftarrow e$
  $r \leftarrow (r+1) \bmod N$
  $n = n + 1$

Queues
- Circularly Linked List-based Queue

- Why Circularly Linked List?
  - Efficient access to **both front and rear** elements
  - Support dynamic expansion and contraction

```
typedef string Elem;                          // queue element type
class LinkedQueue {                           // queue as doubly linked list
public:
  LinkedQueue();                              // constructor
  int size() const;                           // number of items in the queue
  bool empty() const;                         // is the queue empty?
  const Elem& front() const throw(QueueEmpty); // the front element
  void enqueue(const Elem& e);                // enqueue element at rear
  void dequeue() throw(QueueEmpty);           // dequeue element at front
private:                                      // member data
  CircleList C;                               // circular list of elements
  int n;                                      // number of elements
};
```

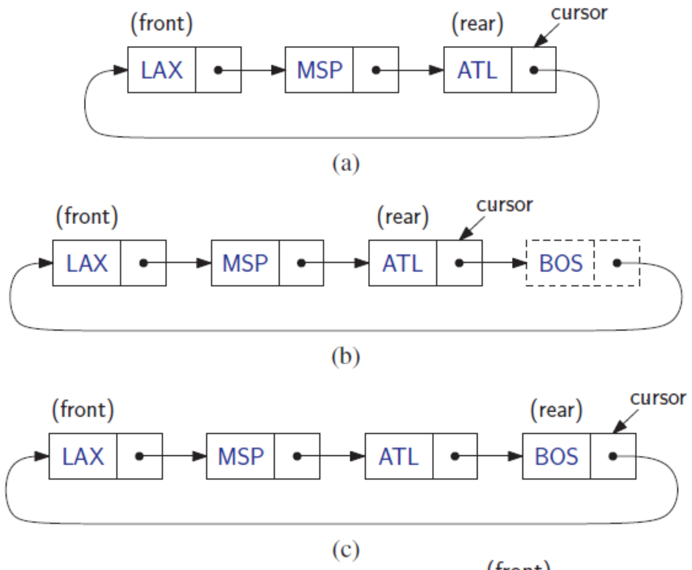
- CircleList Member Functions
  - back(): reference to the rear element
  - front(): reference to the front element
  - add(): insert a new node after the cursor
  - remove(): remove the node after the cursor
  - advance(): move the cursor to the next node

```
                                              // enqueue element at rear
void LinkedQueue::enqueue(const Elem& e) {
  C.add(e);                                   // insert after cursor
  C.advance();                                // ...and advance
  n++;
}
                                              // dequeue element at front
void LinkedQueue::dequeue() throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("dequeue of empty queue");
  C.remove();                                 // remove from list front
  n--;
}
```

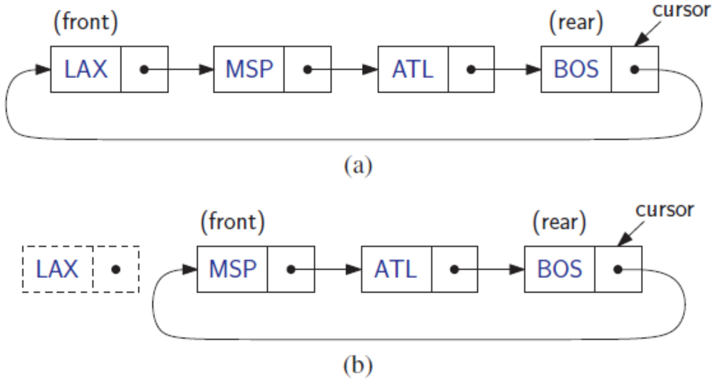
Queues
- Queue Operations (Circularly Linked List)

- Enqueue : O(1) time
  - Use add to insert the element after the cursor (rear of queue)
  - Use advance to make the new element the rear

- Dequeue : O(1) time
  - Check if queue is empty; throw QueueEmpty exception if true
  - Use remove to delete the element after the cursor (front of queue)
  - Update the size of the queue (*n*)



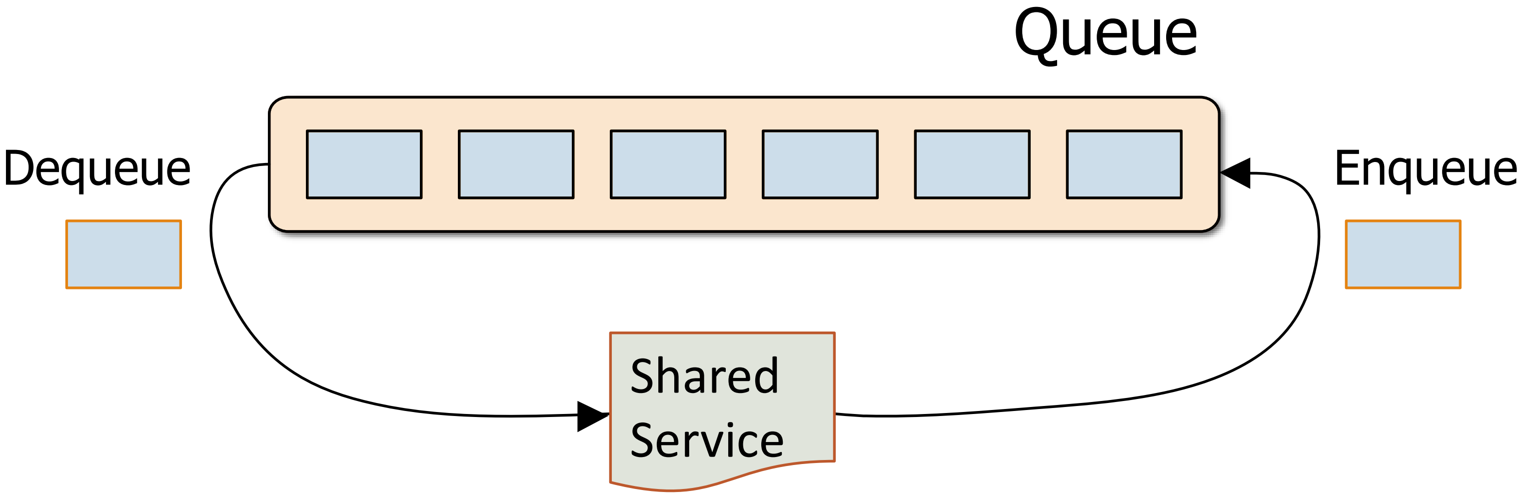
Enqueue



Dequeue

Queues
- Application : Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.front(); Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Deques
- Deque ADT(Abstract Data Type)

- A deque supports insertion and deletion at both ends
  - Pronunciation : “Deck”
- Main deque operations:
  - insertFront(e): add e at the front
  - insertBack(e): add e at the rear
  - eraseFront(): remove the first element
  - eraseBack(): remove the last element
- Support deque operations:
  - front(): access the first element
  - back(): access the last element
  - size(): return the number of elements
  - empty(): check if the deque is empty

Deques
- STL Deque

- Based on **STL vector** class
- Dynamically resizes as elements are added
- Common operators

size():	Return the number of elements in the deque.
empty():	Return true if the deque is empty and false otherwise.
push_front(e):	Insert e at the beginning the deque.
push_back(e):	Insert e at the end of the deque.
pop_front():	Remove the first element of the deque.
pop_back():	Remove the last element of the deque.
front():	Return a reference to the deque's first element.
back():	Return a reference to the deque's last element.

- Include STL deque header : #include <deque>

Deques
- LinkedDeque Implementation

- Use a double linked list
  - Efficient access at both ends
  - Private Members
    - **D** – Stores deque elements
    - **n** – Tracks size
  - Front at the head, rear at the tail

```
typedef string Elem;                                  // deque element type
class LinkedDeque {                                   // deque as doubly linked list
public:
  LinkedDeque();                                      // constructor
  int size() const;                                   // number of items in the deque
  bool empty() const;                                 // is the deque empty?
  const Elem& front() const throw(DequeEmpty); // the first element
  const Elem& back() const throw(DequeEmpty); // the last element
  void insertFront(const Elem& e);                    // insert new first element
  void insertBack(const Elem& e);                     // insert new last element
  void removeFront() throw(DequeEmpty);               // remove first element
  void removeBack() throw(DequeEmpty);                // remove last element
private:                                              // member data
  DLinkedList D;                                      // linked list of elements
  int n;                                              // number of elements
};
```

Deques
- Adapter and Adapter Design Pattern

- Adapting existing data structures for special purposes
- Adapting “DLinkedList” to implement a deque
  - Mapping : each deque operation corresponds to a DLinkedList operation (e.g., insertFront → addFront)

- Adapter Design Pattern
  - An adapter (or wrapper) translates one interface to another
  - Like electric plug adapters for appliances in different countries
- Purpose
  - To reuse existing structures or classes by adapting their interfaces

Deques
- Adapter Examples

- Stack ADT with Deque
  - Stack ADT can be implemented using a deque
  - Each stack operation is translated to an equivalent deque operation
  - Operation mapping
    - push(*o*) → insertFront(*o*)
    - pop() → eraseFront()
    - top() → front()
  - Class DequeStack implements Stack ADT using LinkedDeque
- Queue ADT with Deque
  - Queue ADT can also be implemented using a deque
  - Operations translate seamlessly due to the deque’s symmetry
  - Operation mapping
    - enqueue(*e*) → insertBack(*e*)
    - dequeue() → eraseFront()
    - front() → front()

A Maze Problem
- Solving Maze Using Stack or Queue

- What is Maze Problem?
  - A Programming challenge to navigate a maze from start to finish
  - Uses constraints to avoid obstacles and dead ends
- Applications
  - Robotics, AI pathfinding, gaming, etc.

- Problem Statement
  - Input
    - A 2D grid representation the maze (0: obstacles, 1: open path)
    - Start point and destination coordinates
  - Output
    - A valid path from start to destination (if one exists)
  - Constraints
    - Paths can only one move up, down, left, or right

A Maze Problem
- Solving Maze Using Stack or Queue

- Using **Recursion**
  - Recursively explore potential paths
  - Mark cells as visited to avoid loops
  - Backtrack when a dead end is encountered
- Potential issues
  - Infinite loops without proper marking
  - Performance constraint for large mazes

- DFS (Depth First Search) with **Stack**
  - The stack keeps track of the current path being explored
  - Backtracking occurs naturally by popping elements off the stack when a dead-end is reached.
- BFS (Breadth First Search) with **Queue**
  - The queue ensures that all cells at the current distance are explored before moving to the next distance, leading to a breadth-wise search.