CSCE 3110
Data Structures & Algorithms

- ADT, lists, stacks, queues
- Reading: Chap.3 Weiss

Contents

Elementary Data Structures

"Mankind's progress is measured by the number of things we can do without thinking."

Elementary data structures such as stacks, queues, lists, and heaps are the "off-the-shelf" components we build our algorithm from.

A data organization, management, and storage format that enables efficient access and modification.

There are two aspects to any data structure:

- The abstract operations which it supports (abstract data type, ADT).
- The implementation of these operations.

Data Abstraction

- That we can describe the behavior of our data structures in terms of abstract operations is why we can use them without thinking.
- *Push(x,s)* – Insert item x at the top of stack s.
- *Pop(s)* – Return (and remove) the top item of stack s.

- That there are different implementations of the same abstract operations enables us to optimize performance in difference circumstances.

Abstract Data Type

- An abstract data type is a set of objects with a set of operations.
- It is a mathematical abstraction.
- The operations are defined but not the implementation of the operations.
- Lists, sets, and graphs are examples of abstract data types.
- Classes implement ADTs, separating the interface from the implementation.

Fundamental Data Structures

Data structures can be neatly classified as either contiguous or linked depending upon whether they are based on arrays or pointers:

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.

- Linked data structures are composed of multiple distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

Pointers and Linked Structures

- Pointers represent the address of a location in memory. A cell-phone number can be thought of as a pointer to its owner as they move about the planet.
- In C, *p denotes the item pointed to by p, and &x denotes the address (i.e., pointer) of a particular variable x.
- A special NULL pointer value is used to denote structure terminating or unassigned pointers.

Array—Abstract Data Type

- The array is an abstract data type (ADT) that holds a collection of elements accessible by an index.
  - elements can be anything, primitives types such as integers to more complex types like instances of classes

Minimal Required Functionality

- set(i, v) -> Sets the value of index i to v
- get(i) -> Returns the value of index i in the array

Array – Data Structure

- An (simple) array is a structure of fixed-size data records such that each element can be efficiently located by its index or (equivalently) address.
- Advantages of contiguously-allocated arrays include:
  - Constant-time access given the index.
  - Arrays consist purely of data, so no space is wasted with links or other formatting information.
  - Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

Dynamic Arrays

- Unfortunately, we cannot adjust the size of simple arrays in the middle of a program's execution.
- Compensating by allocating extremely large arrays can waste a lot of space.
- With dynamic arrays we start with an array of size 1 and double its size from m to $2m$ each time we run out of space.
- How many times will we double for n elements?
- Only $\lceil \log_2 n \rceil$.

How Much Total Work?

- The apparent waste in this procedure involves the recopying of the old contents on each expansion.

Item No.:	1	2	3	4	5	6	7	8	9	10	…
Array size:	1	2	4	4	8	8	8	8	16	16	…
Cost:	1	2	3	1	5	1	1	1	9	1	…

$$\underbrace{(1+1+\cdots 1)}_{n \text{ items}} + \underbrace{(1+2+4+\cdots)}_{\lfloor \log_2(n-1) \rfloor + 1 \text{ items}}$$

Cost $\leq n + 2n = 3n = O(n)$

- Thus, each of the n elements move an average of only twice, and the total work of managing the dynamic array is the same $O(n)$ as a simple array.

Content

- Abstract data type
    - Array
- List, Stack, and Queue
    - ADT
    - implementations
    - algorithm analysis
    - applications
    - postfix conversion

List ADT

- $A_1, A_2, A_3, \ldots, A_N$ is a list of size N.
- A list can be implemented using an array.
- However, insertion and deletion are expensive.
- On average, half of the list must be moved.
- Worst case moves all elements.
- So, these operations are O(N).

List ADT

A list contains elements of same type arranged in sequential order

operations performed on the list:

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

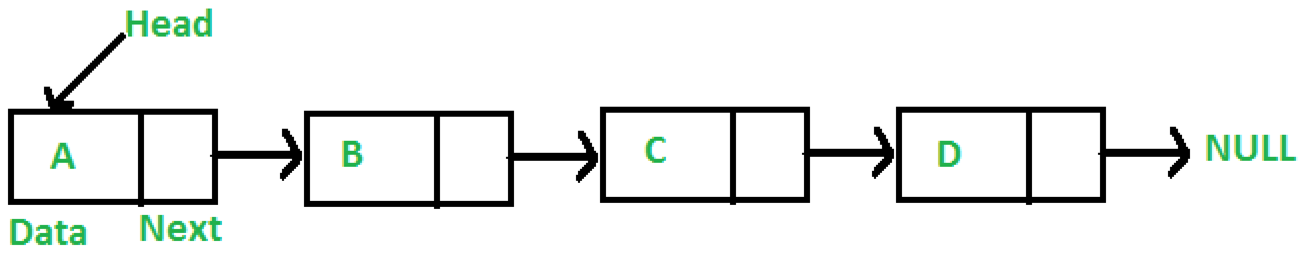
Array-based List

- On the other hand, accessing an item by its index can occur in O(1) time.
- Likewise, adding or removing from the end is O(1) time.

Linked List

- Linked lists use a series of nodes.
- Each node contains a pointer to the next node.
- This means the nodes need not be contiguous in memory.
- This allows a new node to be inserted or removed in O(1) time.

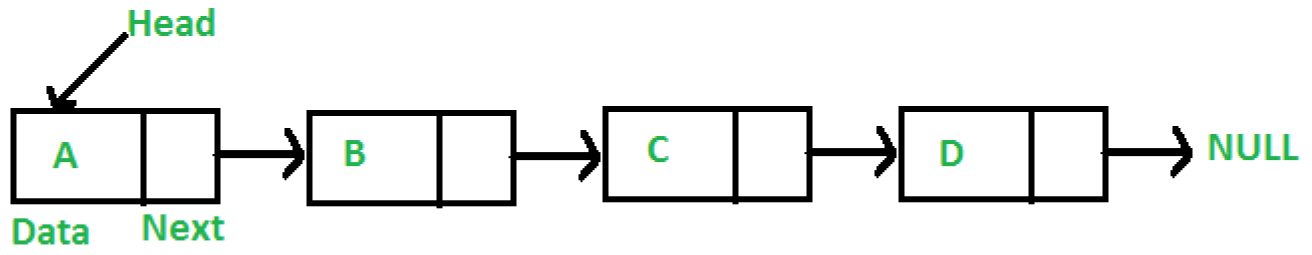
- On the other hand, going to the kth node is now O(N) time, because the list cannot be indexed the way an array can.
- The only way to find an item is to traverse the list.



Searching a List

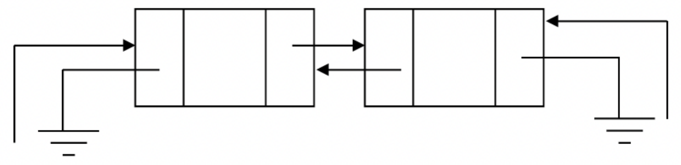
Searching in a linked list can be done iteratively or recursively.

```
list * search_list(list *l, item type x)
{
      if (l == NULL) return(NULL);
      if (l->item == x)
            return(l);
      else
            return(search_list(l->next, x) );
}
```

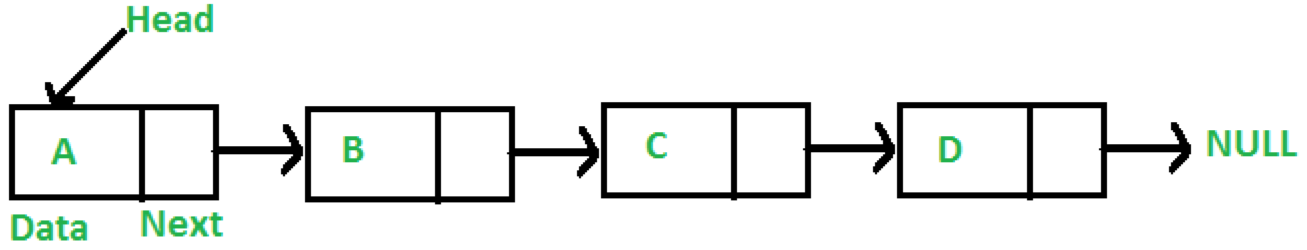


Linked List

```
// A linked list node
typedef struct list
{
   int data;
   struct list *next;
}list;
```



Empty list



Insertion into a List

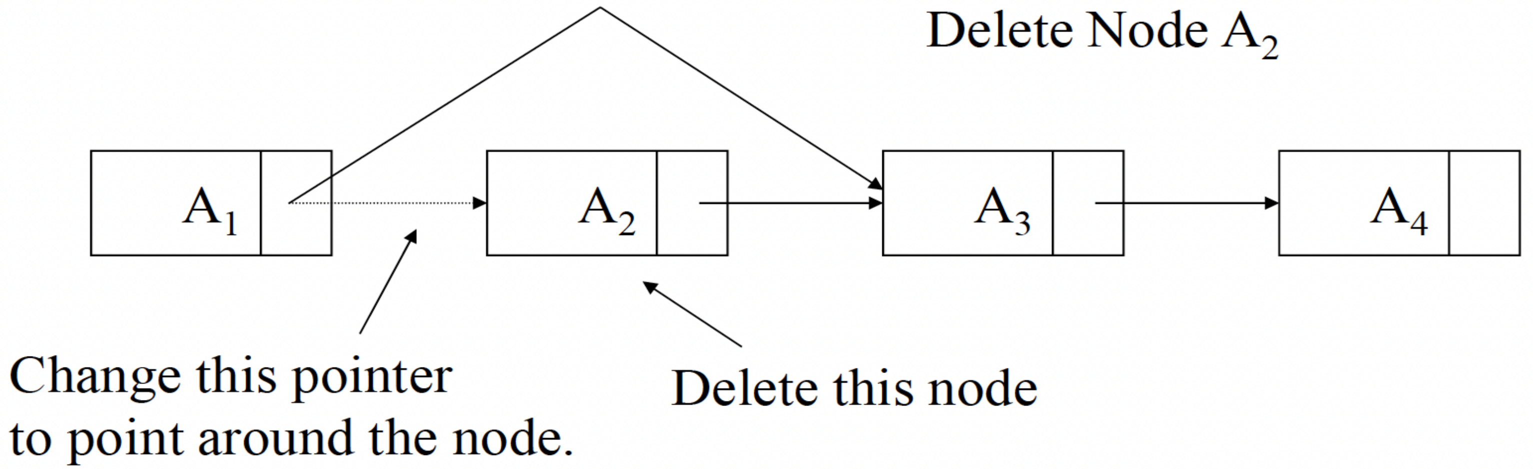
Since we have no need to maintain the list in any particular order, we might as well insert each new item at the head.

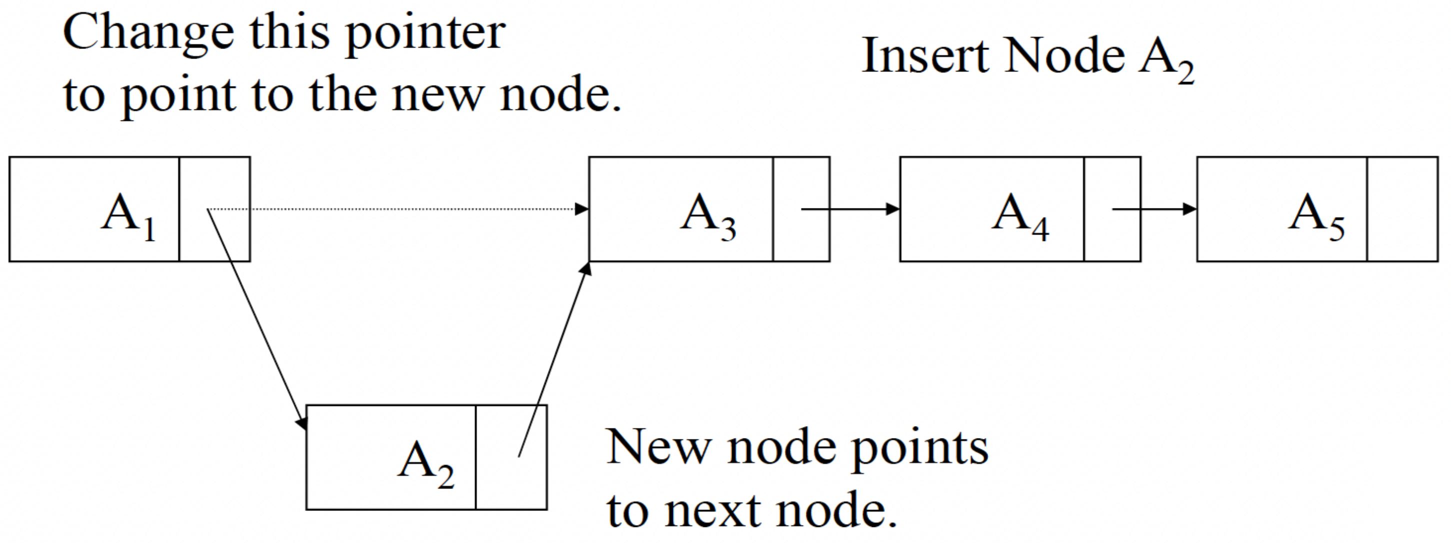
```
void insert_list(list **l, item type x)
{
      list *p;

      p = malloc( sizeof(list) );
      p->item = x;
      p->next = *l;
      *l = p;
}
```

Deleting from a List

```
delete_list(list **l, item type x)
{
        list *p; (* item pointer *)
        list *last = NULL; (* predecessor pointer *)
        p = *l;
        while (p != NULL && p->item != x) { (* find item to delete *)
                last = p;
                p = p->next;
        }
        if (p == NULL) (* item no found*)
                return;
        else
                last->next = p->next;
                free(p); (* return memory used by the node *)
}
```

Deleting from a List




Change this pointer to point to the new node.
Insert Node A_2
A_1
A_3
A_4
A_5
A_2
New node points to next node.

Advantages of Linked Lists

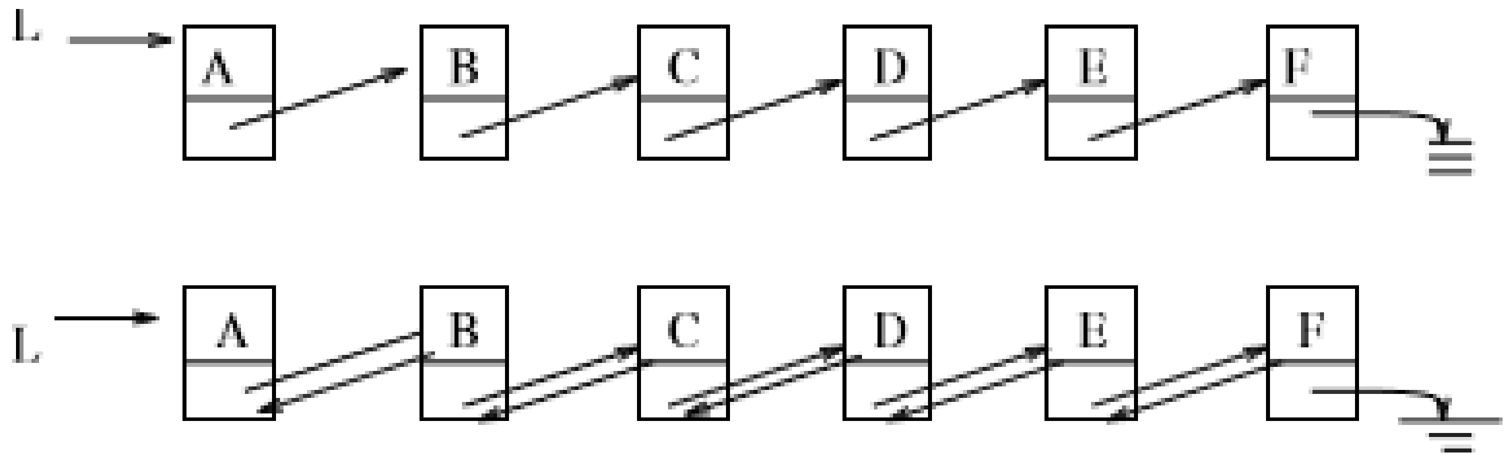
The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are simpler than for contiguous (array) lists.
3. With large records, moving pointers is easier and faster than moving the items themselves.

Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

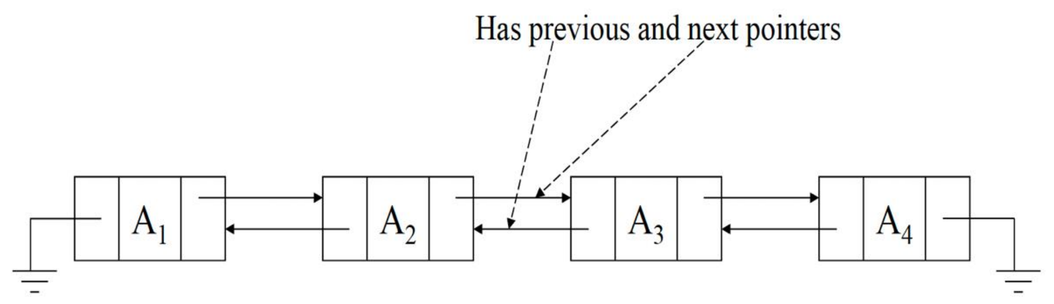
Singly or Doubly Linked Lists

- We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.



Doubly Linked Lists

- Doubly linked lists have a node pointing backwards to the previous node in addition to the one pointing forward to the next node.
- This costs more in terms of storage (for the extra pointer) and a little more in terms of time (to fix the extra pointer) on inserts and deletes.
- It saves time on deletes since a search is no longer required to find the previous node.

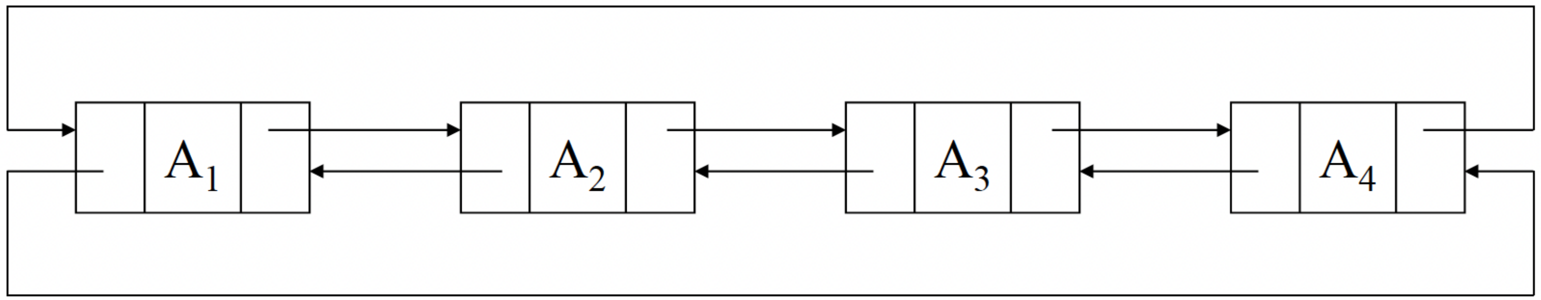


Circular Linked Lists

- A circular linked list links the last node back to the first node (or to the header node).
- The list itself can be singly or doubly linked.

Has pointers to other end, wraps around.

Can be singly or doubly linked.



ArrayList and LinkedList

ArrayList:

- It uses a **growable array** to store elements, which means it automatically expands when needed.

- Accessing elements (using get and set) happens in **constant time**, so it's fast when retrieving data.

- Modifying the structure of the list (like adding or removing elements in the middle) can be **costly** because the array may need to shift elements.

- It has a specific **capacity**, and you can provide an initial estimate to avoid frequent expansions.

- You can use trimToSize to reduce the list’s capacity and avoid wasting memory.

ArrayList and LinkedList

LinkedList:

•It uses a **linked list** structure, which stores elements in nodes connected by pointers.

•Modifying the list (adding or removing elements) is more efficient because it only involves updating pointers, so changes in the middle of the list are generally faster.

•However, accessing elements by an index (e.g., get) can be **expensive** because you have to traverse the list node by node, which takes more time.

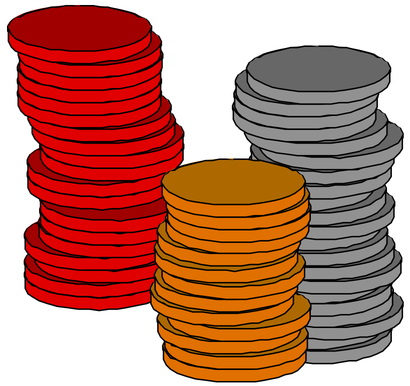
Both:
•For both lists, operations that involve **searching** through the list (like finding an element) take **linear time** (O(n)) because they have to go through each element in the worst case.

•**Use ArrayList** when you need **fast random access** to elements, but be cautious about modifications.

•**Use LinkedList** when you need **frequent modifications** (inserts/removals), but random access will be slower.

What is a stack?

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- = LIFO
- It means: the last element inserted is the first one to be removed
- Example



- Which is the first element to pick up?

Stack Applications

- **Real life**
    - Pile of books: When you stack books, the last book you place on the top is the first one you take off.
    - Plate trays: In a cafeteria, the plates are often stacked, and you take the top plate (the last one added) first.
- More applications related to computer science
    - Program execution stack
    - Evaluating expressions

Stack ADT

Objects: a finite ordered list with zero or more elements.

Methods:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

Implementing a Stack

- At least two different ways to implement a stack
  - Array:
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

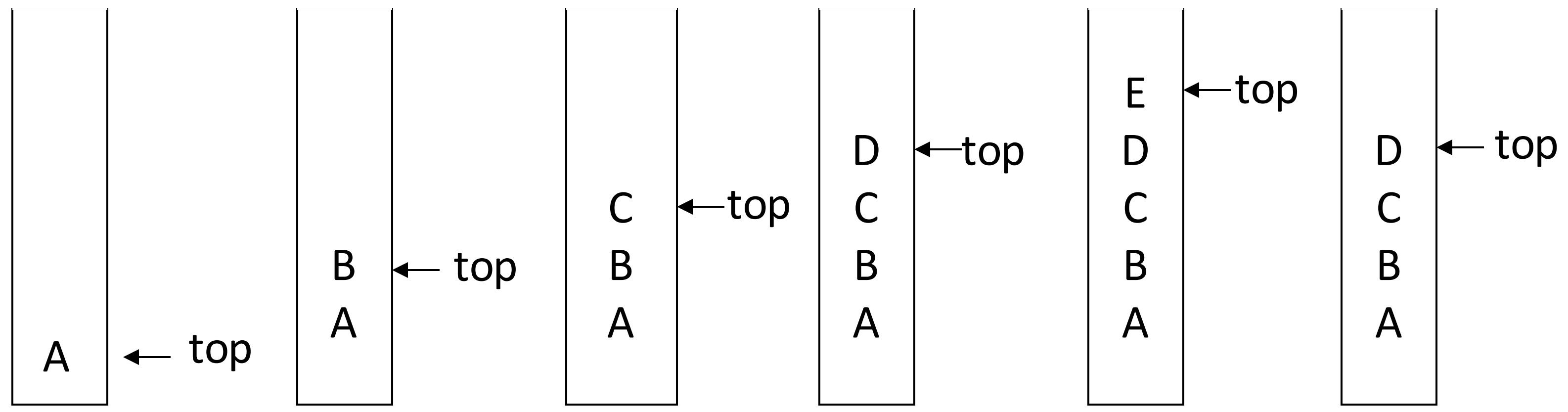
Implementing Stacks: Array

- A stack can be implemented using an array where elements are stored in contiguous memory locations.

- Advantages
  - best performance
- Disadvantage
  - fixed size

- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false, otherwise adds it into the correct spot
  - if array is empty, pop() returns null, otherwise removes the next item in the stack

Last In First Out



Array-based Stack Implementation

- Allocate an array of some size (pre-defined)
    - Maximum N elements in stack
- Associated with each stack is `top`
    - Bottom stack element stored at element 0
    - for an empty stack, set `top` to -1
    - last index in the array is the `top`
- Increment `top` when one element is pushed, decrement after pop
- Push
    - (1) Increment `top` by 1.
    - (2) Set Stack[`top`] = X
- Pop
    - (1) Set return value to Stack[`top`]
    - (2) Decrement `top` by 1

Stack class

```
class Stack {
public:
      Stack(int size = 10);                      // constructor
      ~Stack() { delete [] values; }             // destructor
      bool IsEmpty() { return top == -1; }
      bool IsFull() { return top == maxTop; }
      double Top();
      void Push(const double x);
      double Pop();
      void DisplayStack();
private:
      int maxTop;          // max stack size = size - 1
      int top;             // current top of stack
      double* values;      // element array
};
```

Create Stack

- The constructor of `Stack`
  - Allocate a stack array of `size`. By default, `size = 10`.
  - When the stack is full, `top` will have its maximum value, i.e. `size – 1`.
  - Initially `top` is set to -1. It means the stack is empty.

```
Stack::Stack(int size /*= 10*/) {
     maxTop        =     size - 1;
     values        =     new double[size];
     top           =     -1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed after the initialization.

Push Stack

- `void Push(const double x);`
  - Push an element onto the stack
  - If the stack is full, print the error information.
  - Note `top` always represents the index of the top element. After pushing an element, `top` increased by 1.

```
void Stack::Push(const double x) {
      if (IsFull())
            cout << "Error: the stack is full." << endl;
      else
            values[++top] = x;
}
```

Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don’t forgot to decrement `top`

```
double Stack::Pop() {
      if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      } else {
            return values[top--];
      }
}
```

Remaining Methods (array based)

```
double Stack::isEmpty() {
      return top == -1;
}
```

```
double Stack::isFull() {
      return top == maxTop;
}
```

Algorithm Analysis

- `push` $O(?)$
- `pop` $O(?)$
- `isEmpty` $O(?)$
- `isFull` $O(?)$

- What if `top` is stored at the beginning of the array?

Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is slower
  - can grow to an infinite size
- Basic implementation
  - `list` is initially empty
  - `push()` method adds a new item to the head of the list
  - `pop()` method removes the head of the list

List-based Stack Implementation: Push

```
void push(element item)
{
      /* add an element to the top of the stack */
      pnode temp = (pnode) malloc (sizeof (node));
      if (IS_FULL()) {
            fprintf(stderr, " The memory is full\n");
            exit(1);
      }
      temp->item = item;
      temp->next= top;
      top= temp;
}
```

Pop

```
element pop(pnode top) {
      /* delete an element from the stack */
      pnode temp = top;
      element item;
      if (IS_EMPTY(temp))  {
            fprintf(stderr,  “The stack is empty\n”);
            exit(1);
      }
      item = temp->item;
      top = temp->next;
      free(temp);
      return item;
}
```

Algorithm Analysis

- `push` $O(?)$
- `pop` $O(?)$
- `isEmpty` $O(?)$

Applications

- Balancing Symbols
- Evaluation of Postfix Expressions
- Infix to Postfix conversion

Balancing Symbols

- To check that every right brace, bracket, and parentheses must correspond to its left counterpart
  - e.g. [( )] is legal, but [( ]) is illegal

A stack is useful for checking symbol balance. When a closing symbol is found it must match the most recent opening symbol of the same type.

Applicable to checking `html` and `xml` tags!

Balancing Symbols

- Algorithm
  1. (1) Make an empty stack.
  2. (2) Read characters until end of file
     - i. If the character is an opening symbol, push it onto the stack
     - ii. If it is a closing symbol, then if the stack is empty, report an error
     - iii. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error
  3. (3) At end of file, if the stack is not empty, report an error

Mathematical Calculations

- What does 3 + 2 * 4 equal?
  2 * 4 + 3? (3 + 2) * 4?

- A mathematical expression cannot simply be evaluated left to right.
- The precedence of operators affects the order of operations.

- A challenge when evaluating a program.
- *Lexical analysis* is the process of interpreting a program.

Infix and Postfix Expressions

- The way we use to write expressions is known as infix notation
- Postfix notation is a notation that the operands appear before their operators.
- Postfix expression does not require any precedence rules
- 3 2 * 1 + is postfix of 3 * 2 + 1

Evaluation of Postfix Expressions

- Easy to do with a stack
- Given a proper postfix expression:
  - get the next token
  - if it is an operand (a number or variable) push it onto the stack
  - else if it is an operator （+ - * /）
    - Pop the stack twice to get two operands: the first pop is for the **right-hand operand**, the second for the **left-hand operand**.
      - pop the stack for the right-hand operand
      - pop the stack for the left-hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted, the result is the `top` (and only element) of the stack

Clicker Question

What does the following postfix expression evaluate to?

```
6 3 2 + *
```

A. 18

B. 36

C. 24

D. 11

E. 30

- Given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right-hand operand
    - pop the stack for the left-hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted, the result is the `top` (and only element) of the stack

Evaluation of Postfix Expressions

- The time to evaluate a postfix expression is $O(n)$
  - processing each element in the input consists of stack operations and thus takes constant time
- Evaluate the following postfix expressions and write out a corresponding infix expression: (Exponentiation has higher precedence than multiplication)

```
2 5 ^ 1 −              (2 ^ 5) - 1
2 3 2 4 * + *          2 * (3 + (2 * 4))
1 2 3 4 ^ * +          1 + (2 * (3 ^ 4))
1 2 − 3 2 ^ 3 * 6 / +
```

(1 - 2) + ((3 * (3 ^ 2)) / 6)

- Given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right-hand operand
    - pop the stack for the left-hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted, the result is the `top` (and only element) of the stack

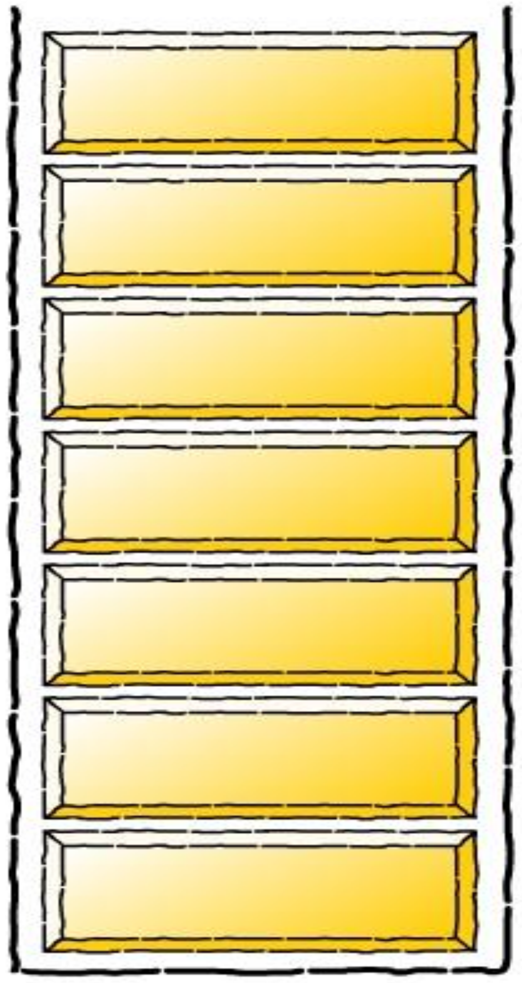
Infix to Postfix Conversion

Requires operator precedence parsing algorithm

- parse: To determine the syntactic structure of a sentence or other utterance

- Operands: add to the output expression
- **Open parenthesis**: push onto stack
- Close parenthesis: pop stack symbols until an open parenthesis appears
- Operators:
  - compare on stack and off stack precedence, except open parenthesis
  - Pop all stack symbols until a symbol of lower precedence appears. Then push the operator
- End of input: Pop all remaining stack symbols and add to the output expression

stack

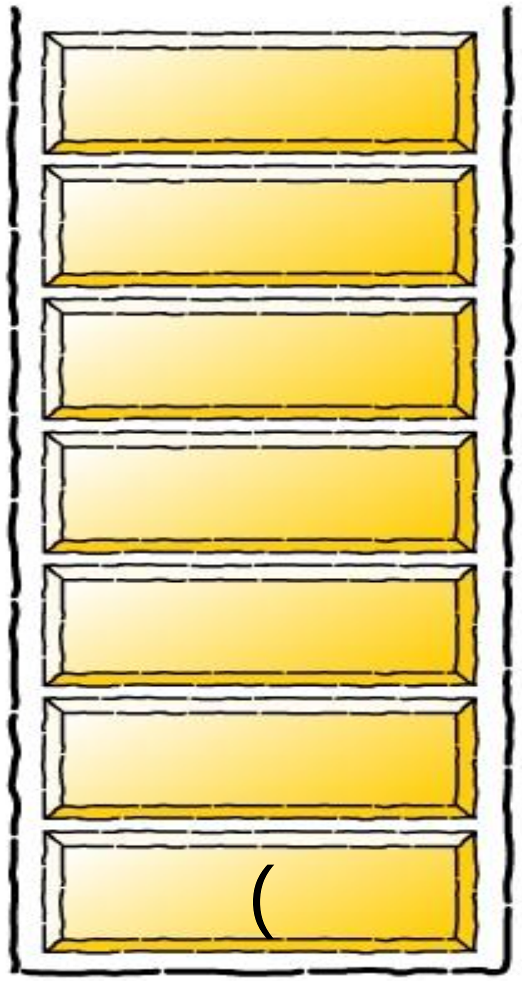


infix expression

(a + b - c) * d – (e + f)

postfix expression

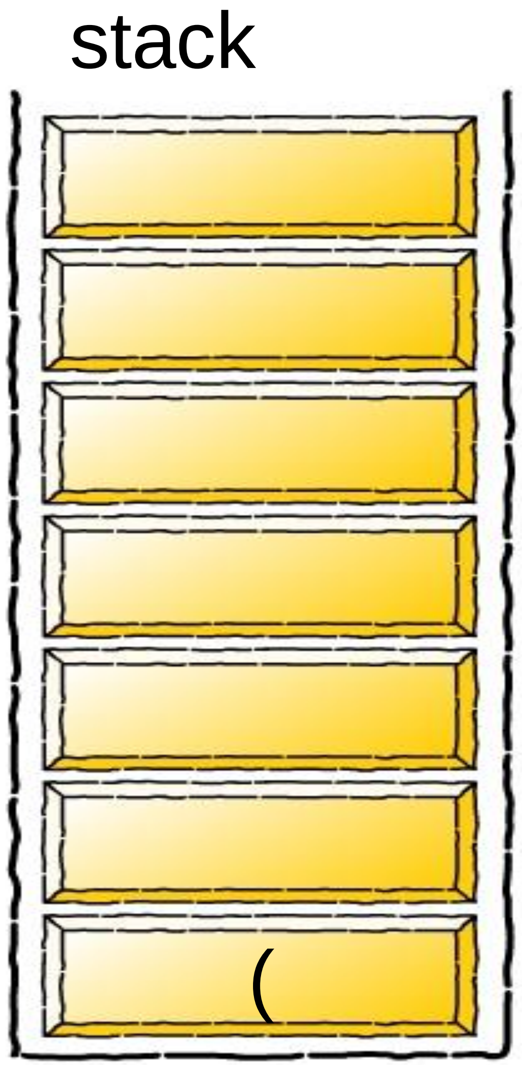
stack



infix expression

a + b - c) * d – (e + f)

postfix expression

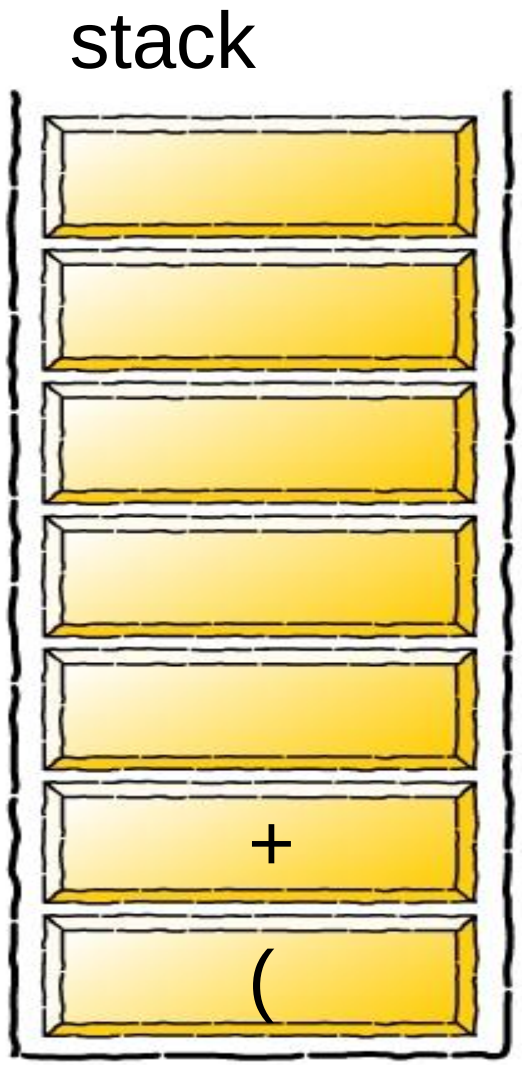


infix expression

+ b - c) * d – (e + f)

postfix expression

a

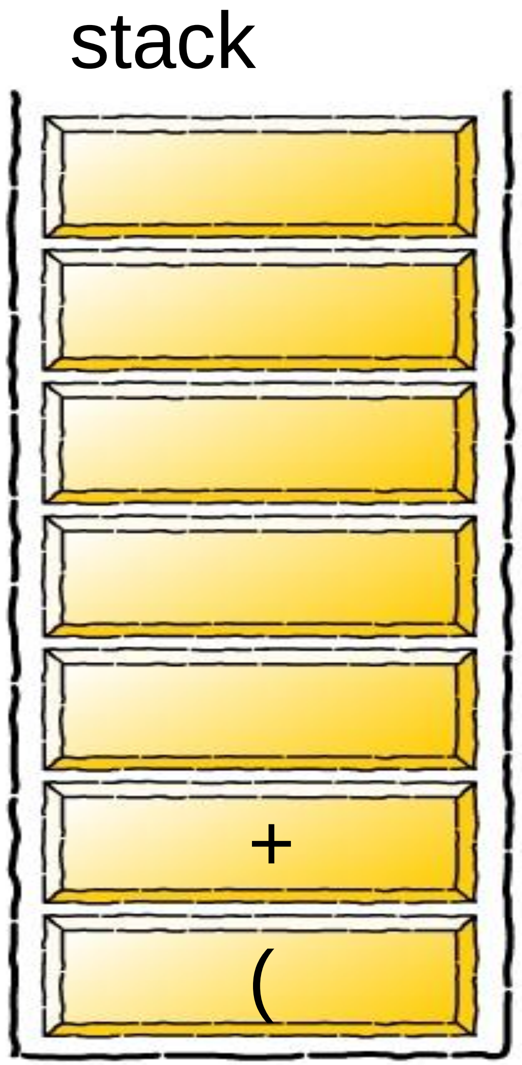


infix expression

b - c) * d – (e + f)

postfix expression

a

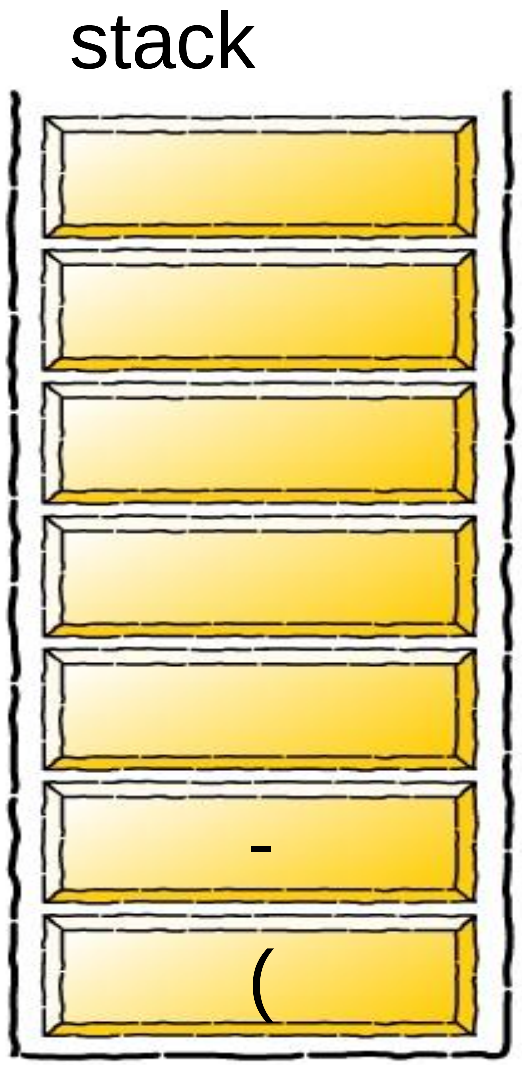


infix expression

- c) * d – (e + f)

postfix expression

a b

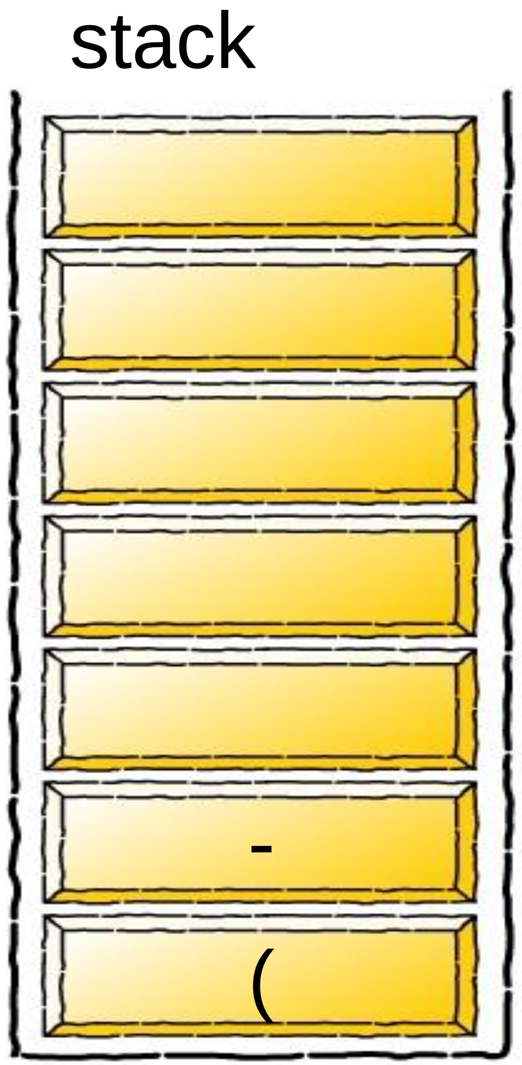


infix expression

c) * d – (e + f)

postfix expression

a b +



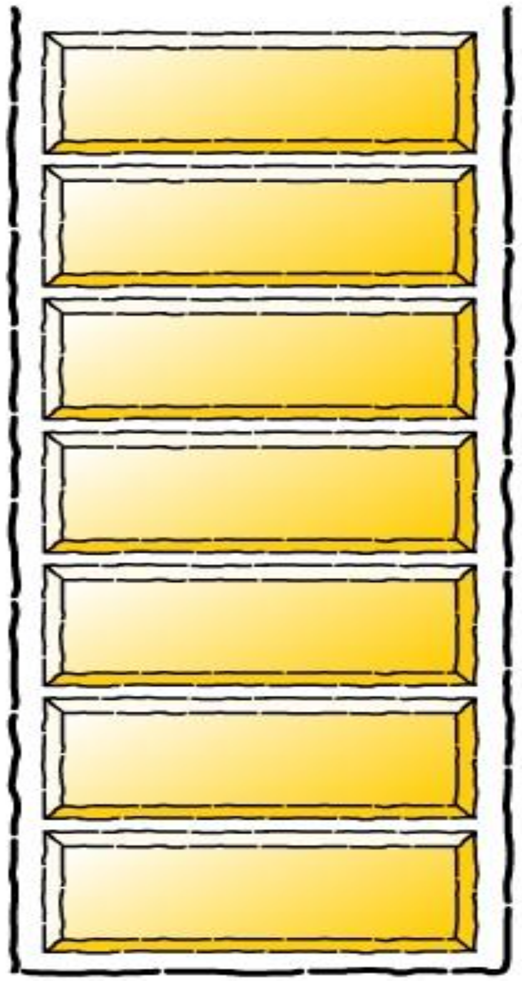
infix expression

) * d – (e + f)

postfix expression

a b + c

stack

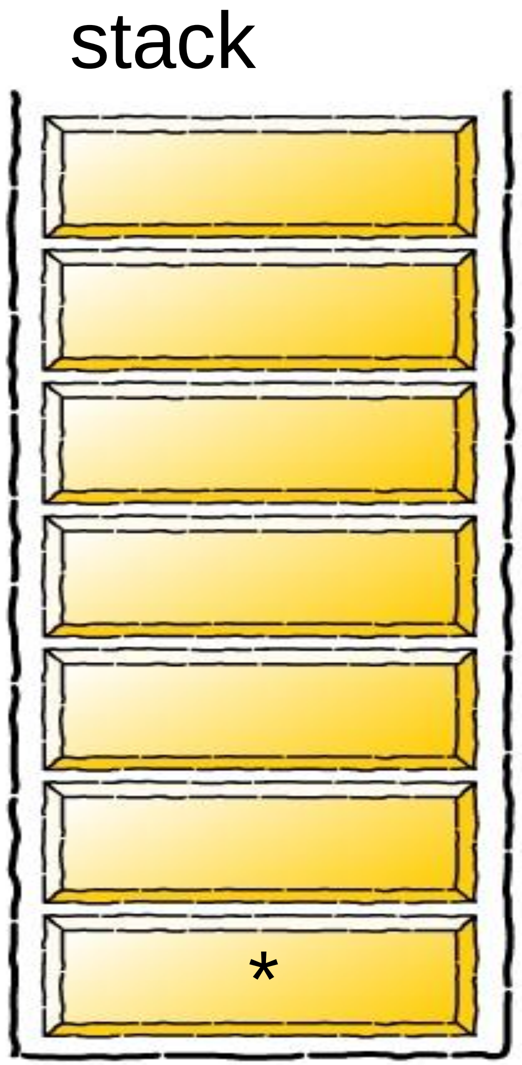


infix expression

* d – (e + f)

postfix expression

a b + c –

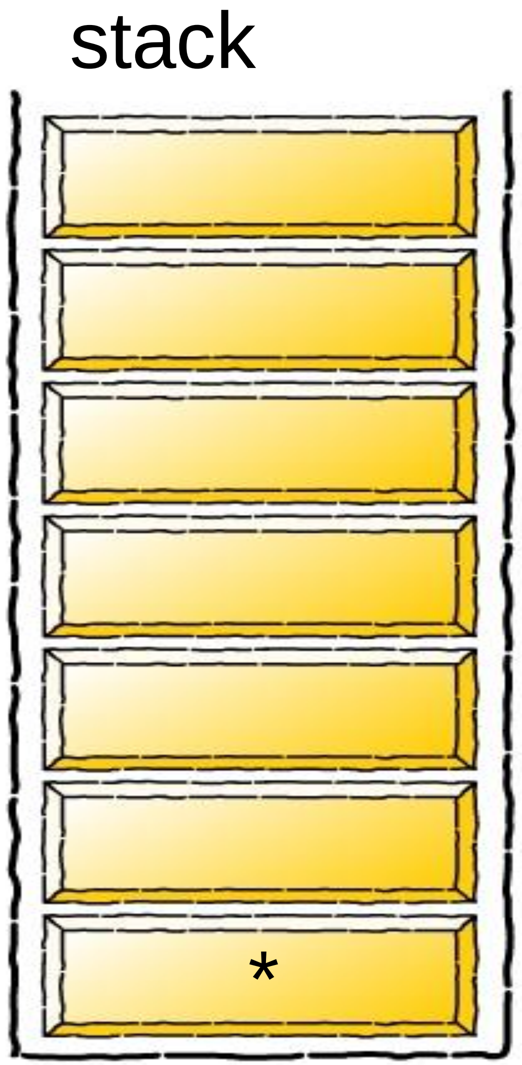


infix expression

d – (e + f)

postfix expression

a b + c –



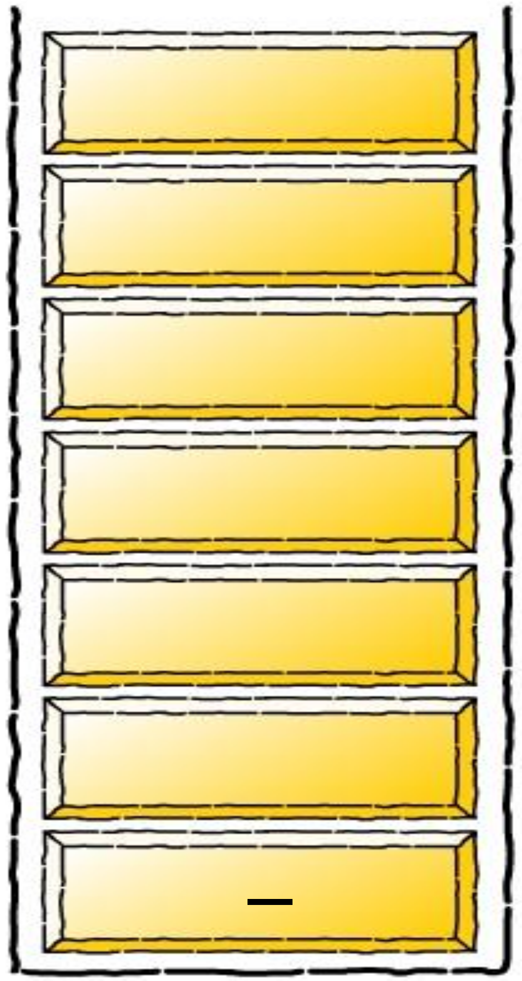
infix expression

– (e + f)

postfix expression

a b + c – d

stack

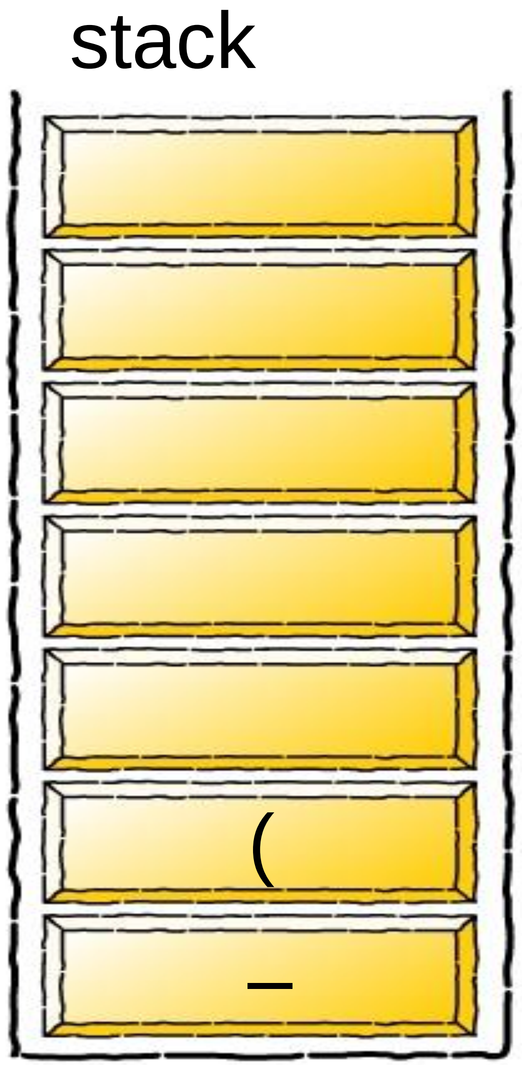


infix expression

(e + f)

postfix expression

a b + c – d *

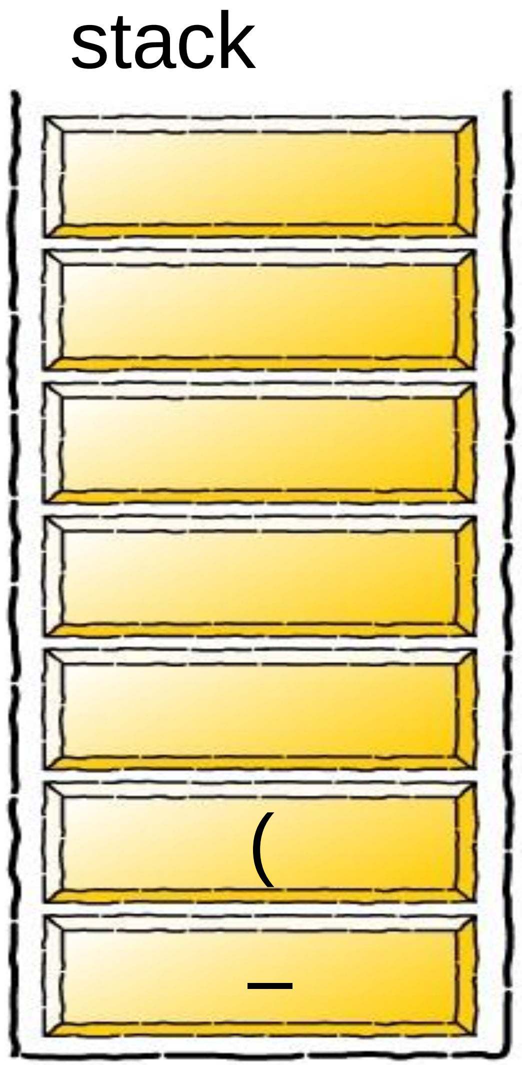


infix expression

e + f)

postfix expression

a b + c – d *

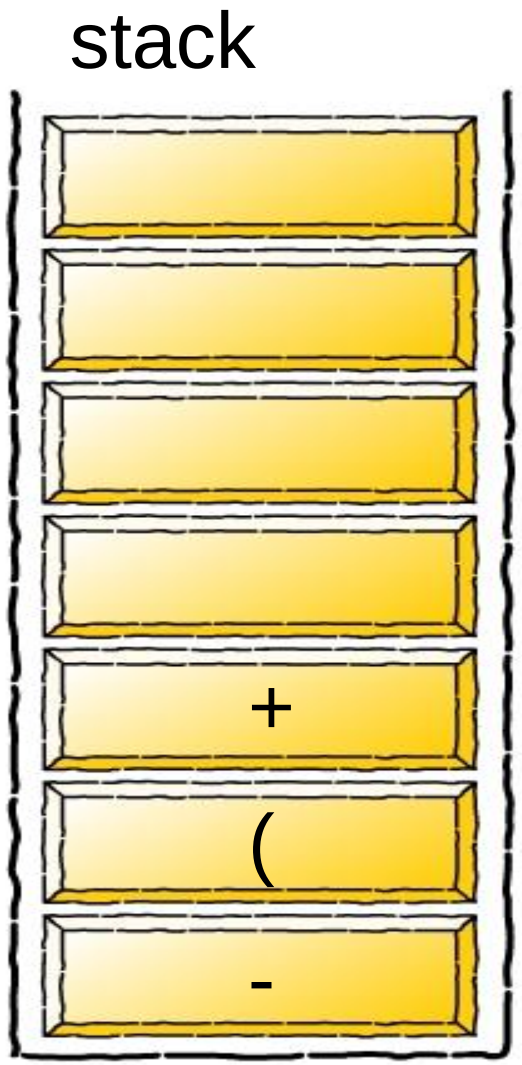


infix expression

+ f)

postfix expression

a b + c – d * e

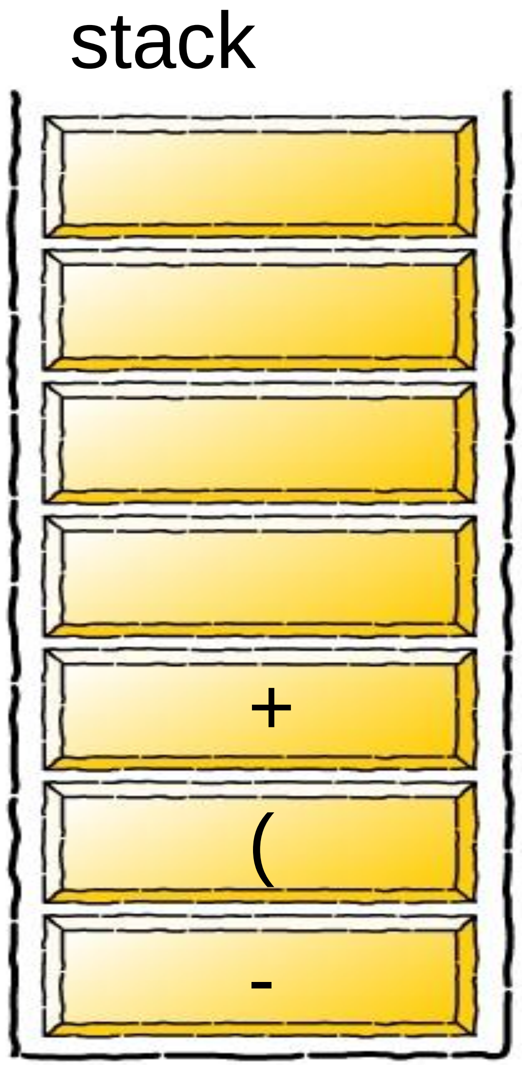


infix expression

f)

postfix expression

a b + c – d * e



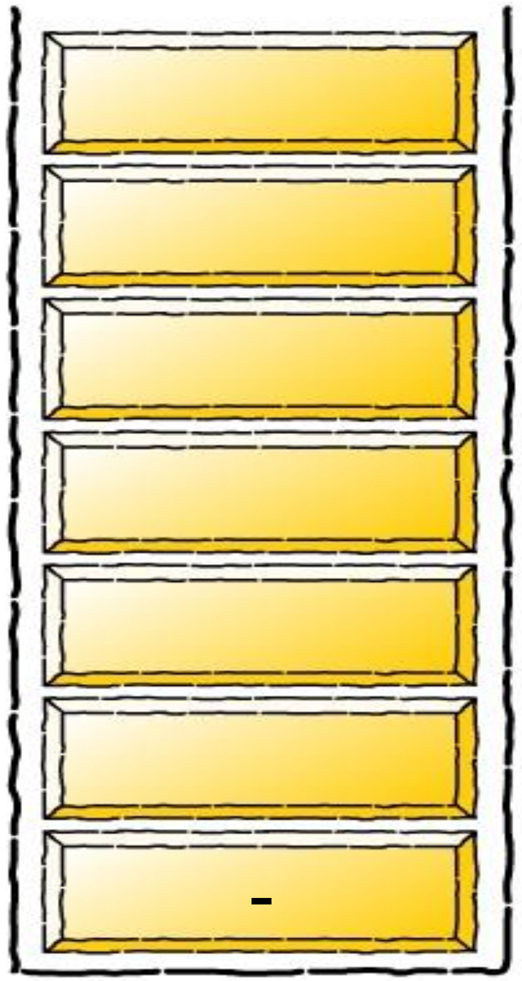
infix expression

)

postfix expression

a b + c – d * e f

stack

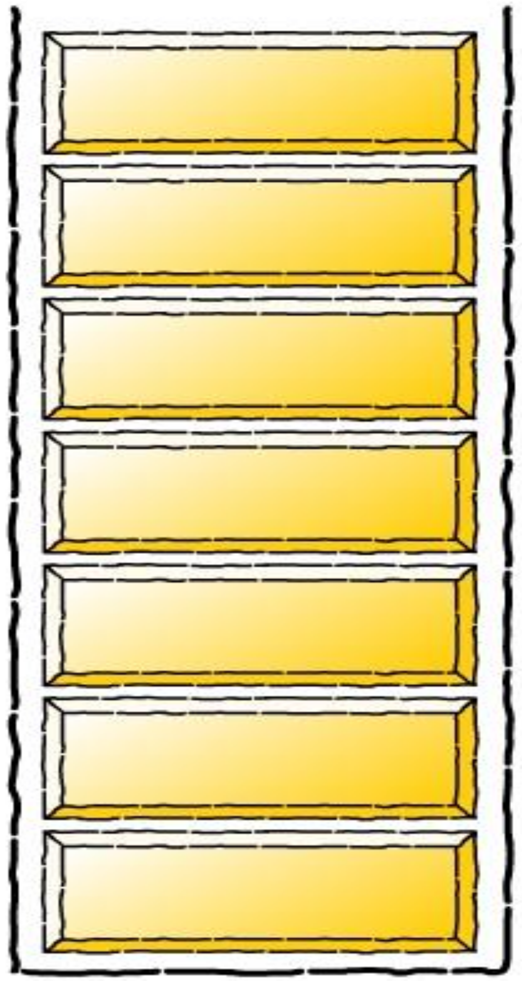


infix expression

postfix expression

a b + c – d * e f +

stack



infix expression

postfix expression

a b + c – d * e f + –

Infix to Postfix

- Convert the following equations from infix to postfix:

  (2 ^ 3) ^ 3 + 5 * 1 — 2 3 ^ 3 ^ 5 1 * +

  1 + 2 - 1 * 3 / 3 + 2 ^ 2 / 3 — 1 2 + 1 3 * 3 / - 2 2 ^ 3 / +

- Operands: add to the output expression
- Open parenthesis: push onto stack
- Close parenthesis: pop stack symbols until an open parenthesis appears
- Operators:
  - compare on stack and off stack precedence, except open parenthesis
  - Pop all stack symbols until a symbol of lower precedence appears. Then push the operator
- End of input: Pop all remaining stack symbols and add to the output expression

Infix & Postfix

Infix Notation:

- **Definition**: The operator is placed **between** the operands.
- **Example**:

  A + B or 3 * (4 + 5)
- **Common in**: Standard arithmetic expressions we use in daily life.
- **Precedence and parentheses**: Parentheses are used to enforce precedence and grouping of operations.

Postfix Notation :

•Definition: The operator is placed after the operands.

- Example:

  A B + or 3 4 5 + *

•No need for parentheses: Since the position of the operator defines precedence, parentheses are not needed.

•Usage: Commonly used in stack-based evaluations, such as in certain calculators or programming languages.

Queue Overview

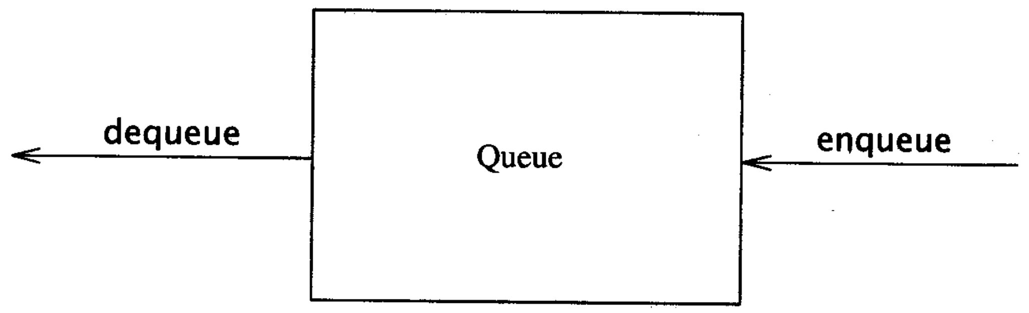
- Queue ADT （ focuses on how it behaves rather than how it's implemented）

- Basic operations of queue
    - Enqueuing, dequeuing etc.

- Implementation of queue
    - Array
    - Linked list

The Queue ADT

- Another form of restricted list
  - Insertion is done at one end, whereas deletion is performed at the other end
- Basic operations:
  - enqueue: insert an element at the rear of the list
  - dequeue: delete the element at the front of the list



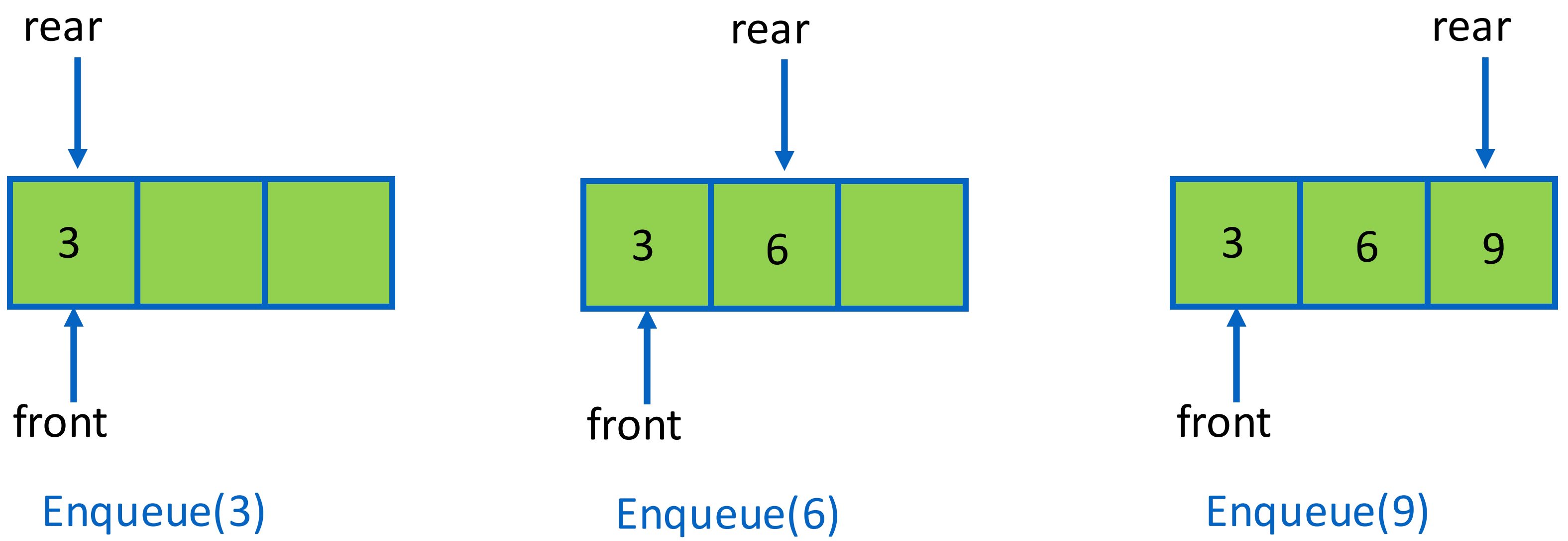
- First-in First-out (FIFO) list

Queue Applications

- **Real life examples**
  - *Waiting in line:* at a bank, a movie theater, or a grocery store, people stand in line, and the first person to arrive is the first to be served. This is a classic queue in action.
  - *Waiting on hold for tech support:* When you call customer support, your call is placed in a queue. The system handles calls in the order they arrive, so the first person to call is the first to be assisted.
- Applications related to Computer Science
  - Threads: In multithreading, queues help manage multiple threads that are waiting for execution. Threads are placed in a queue and processed one by one based on their priority or arrival time.
  - Job scheduling (e.g., Round-Robin algorithm for CPU allocation): This is a common method used in operating systems for allocating CPU time. In the Round-Robin scheduling algorithm, processes are placed in a queue and are assigned a time slice (or quantum) to execute. Once a process finishes its slice, it is moved to the back of the queue, ensuring each process gets a fair share of CPU time. This approach ensures a balanced distribution of resources among multiple tasks.

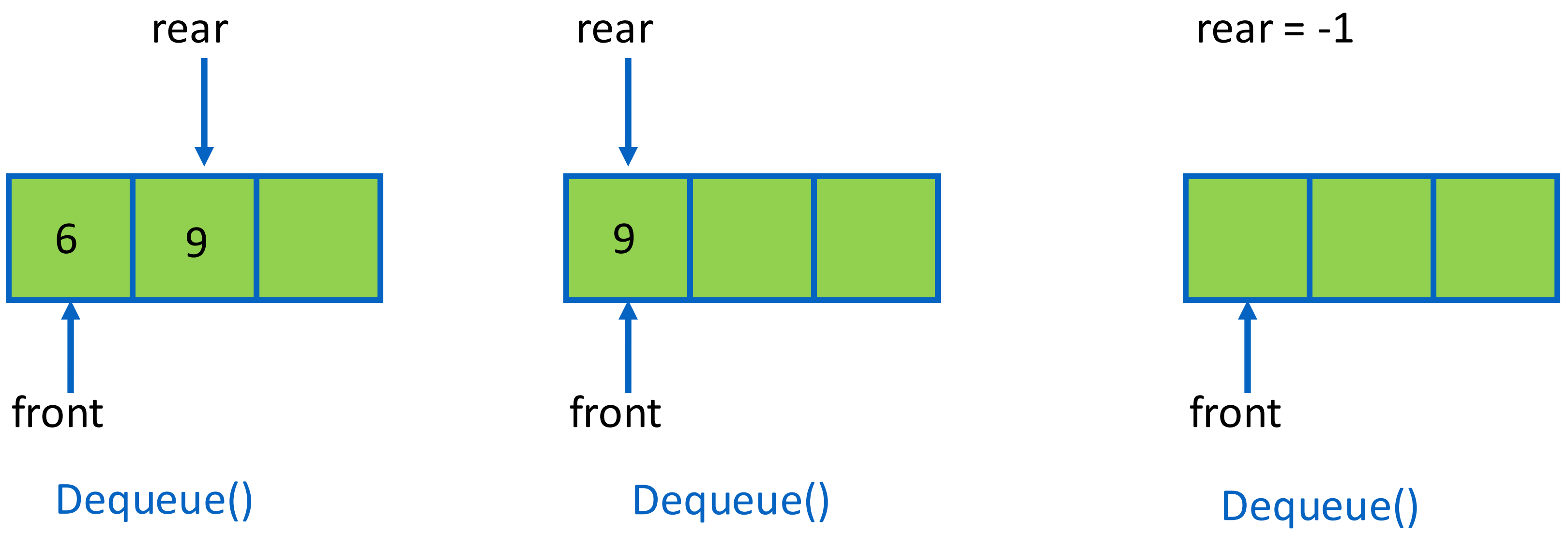
Queue Implementation of Array

- There are several different algorithms to implement Enqueue and Dequeue
- Naïve way
    - When enqueuing, the front index is always fixed, and the rear index moves forward in the array.



Queue Implementation of Array

- Naïve way
  - When enqueuing, the front index is always fixed, and the rear index moves forward in the array.
  - When dequeuing, the element at the front the queue is removed. Move all the elements after it by one position. (Inefficient!!!)



Queue Implementation of Array

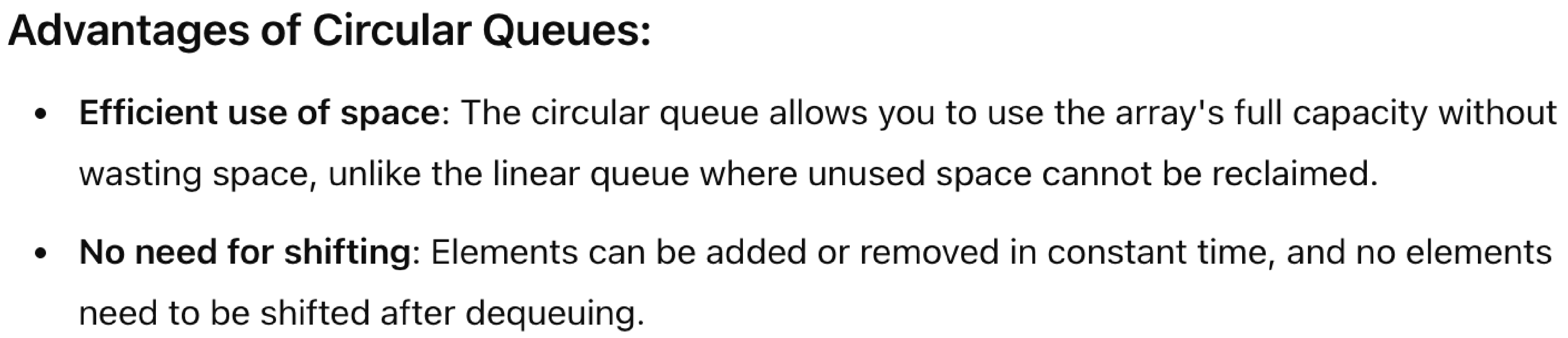
- Better way
  - When an item is enqueued, make the <u>rear index</u> move forward.
  - When an item is dequeued, the <u>front index</u> moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed).

```
  ────→
EEEEOOOOO
OEEEEOOOO  (after 1 dequeue, and 1 enqueue)
OOEEEEEOO  (after another dequeue, and 2 enqueues)
OOOOEEEEE  (after 2 more dequeues, and 2 enqueues)
```

The problem here is that the rear index cannot move beyond the last element in the array.

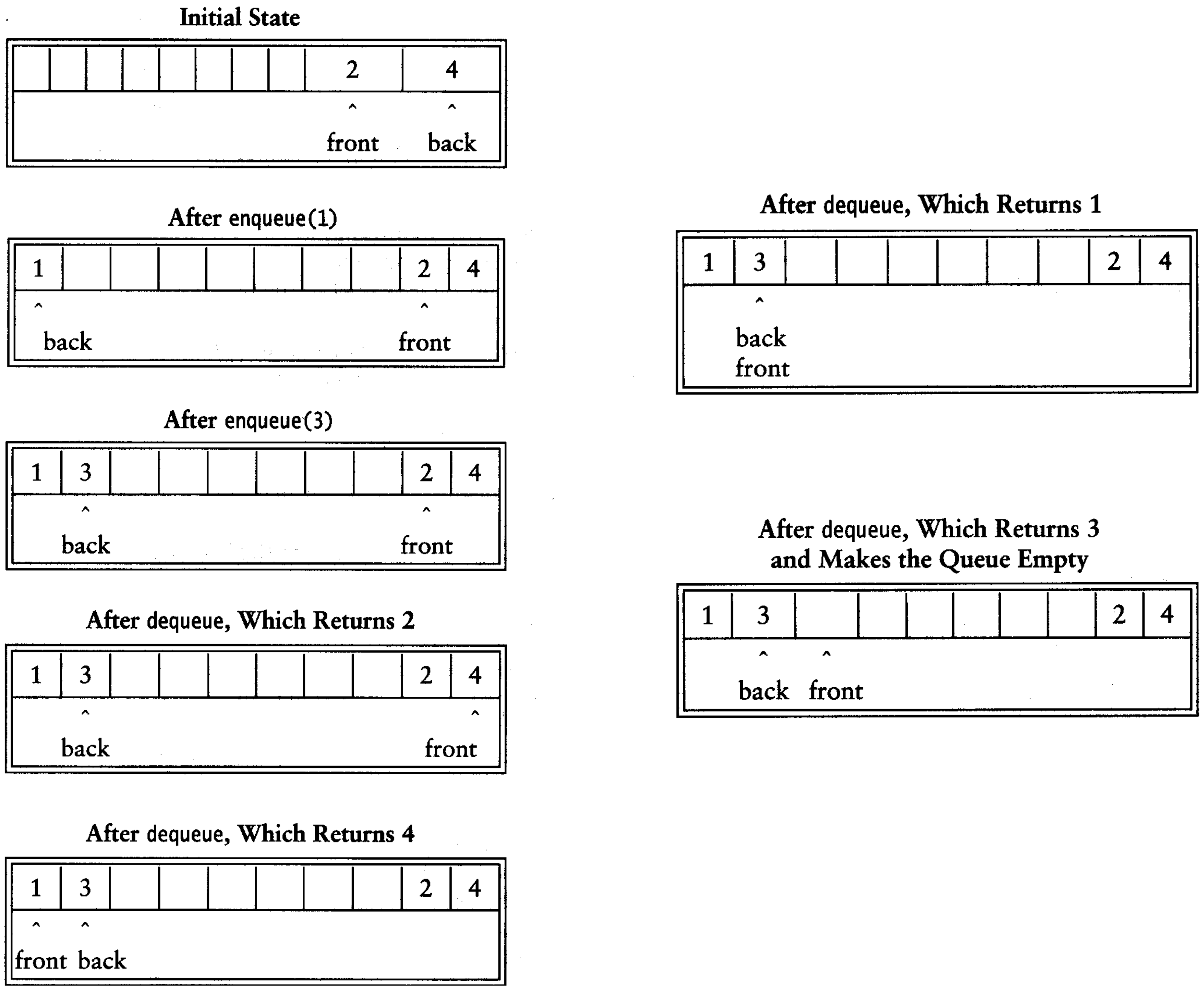
Implementation using Circular Array

- Using a circular array
- When an element moves past the end of a circular array, it wraps around to the beginning, e.g.
    - OOOOO7963 → 4OOOO7963 (after Enqueue(4))
    - After Enqueue(4), the rear index moves to 0.



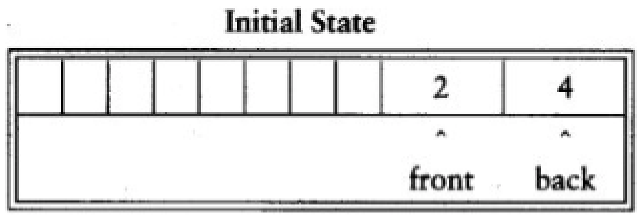
Advantages of Circular Queues:

- **Efficient use of space**: The circular queue allows you to use the array's full capacity without wasting space, unlike the linear queue where unused space cannot be reclaimed.
- **No need for shifting**: Elements can be added or removed in constant time, and no elements need to be shifted after dequeuing.


Initial State
2
4
front
back
After enqueue(1)
1
2
4
back
front
After enqueue(3)
1
3
2
4
back
front
After dequeue, Which Returns 2
1
3
2
4
back
front
After dequeue, Which Returns 4
1
3
2
4
front
back
After dequeue, Which Returns 1
1
3
2
4
back
front
After dequeue, Which Returns 3
and Makes the Queue Empty
1
3
2
4
back
front

Empty or Full?

- Empty queue
  - rear = front - 1
- Full queue?
  - the same!
  - Reason: n values to represent $n + 1$ states
- Solutions
  - Use a boolean variable to say explicitly whether the queue is empty or not
  - Make the array of size $n + 1$ and only allow n elements to be stored
  - Use a counter of the number of elements in the queue



Queue Implementation of Circular Array

```
class Queue {
public:
        Queue(int size = 10);                   // constructor
        ~Queue() { delete [] values; }   // destructor
        bool IsEmpty(void);
        bool IsFull(void);
        bool Enqueue(double x);
        bool Dequeue(double & x);
        void DisplayQueue(void);
private:
        int front;          // front index
        int rear;           // rear index
        int counter;  // number of elements
        int maxSize;  // size of array queue
        double* values;     // element array
};
```

Create Queue

- Allocate a queue array of `size`. By default, `size = 10`.
- `front` is set to `0`, pointing to the first element of the array
- `rear` is set to `−1`. The queue is empty initially.

```
Queue::Queue(int size /* = 10 */) {
       values       =      new double[size];
       maxSize             =      size;
       front               =      0;
       rear                =      -1;
       counter             =      0;
}
```

Max = 9

IsEmpty & IsFull

- Since we keep track of the number of elements that are actually in the queue: `counter`, it is easy to check if the queue is empty or full.

```
bool Queue::IsEmpty() {
      if (counter)  return false;
      else                return true;
}
bool Queue::IsFull() {
      if (counter < maxSize)      return false;
      else                        return true;
}
```

Enqueue

```
bool Queue::Enqueue(double x) {
	if (IsFull()) {
		cout << "Error: the queue is full." << endl;
		return false;
	}
	else {
		// calculate the new rear position (circular)
		rear = (rear + 1) % maxSize;
		// insert new item
		values[rear]  = x;
		// update counter
		counter++;
		return true;
	}
}
```

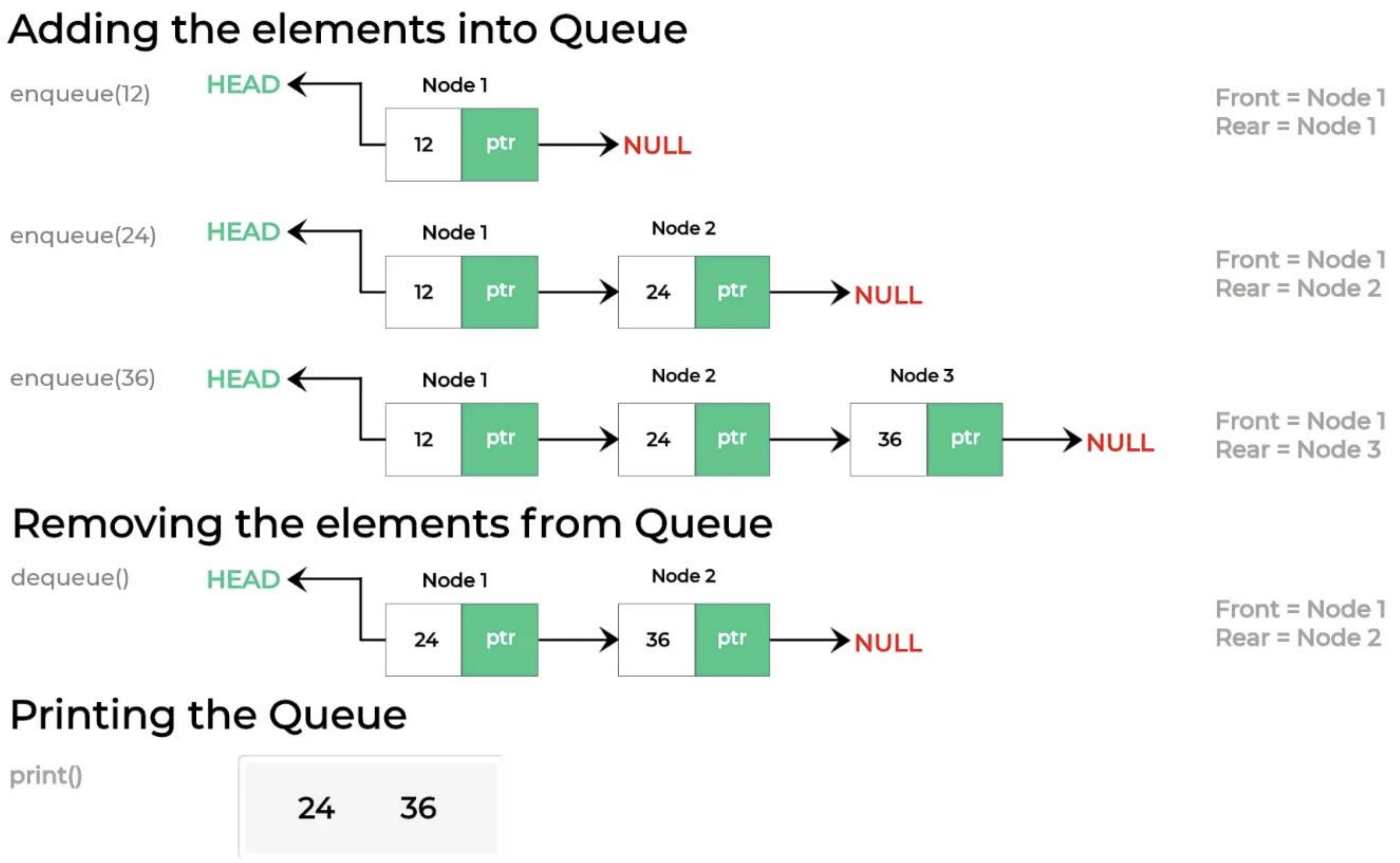
Dequeue

```
bool Queue::Dequeue(double & x) {
        if (IsEmpty()) {
                cout << "Error: the queue is empty." << endl;
                return false;
        }
        else {
                // retrieve the front item
                x = values[front];
                // move front
                front  = (front + 1) % maxSize;
                // update counter
                counter--;
                return true;
        }
}
```

Algorithm Analysis

enqueue	O(?)
dequeue	O(?)
size	O(?)
isEmpty	O(?)
isFull	O(?)

List-based Queue Implementation



List-based Queue Implementation: Enqueue

```
void enqueue(element item)
{  /* add an element to the rear of the queue */
       pnode temp = (pnode) malloc(sizeof (queue));
       if (IS_FULL(temp)) {
              fprintf(stderr, " The memory is full\n");
              exit(1);
       }
       temp->item = item;
       temp->next = NULL;
       if (front)  { (rear) -> next= temp;}
       else front = temp;
       rear = temp;
}
```

Dequeue

```
element dequeue(pnode &front)
{/* delete an element from the queue */
      pnode temp = front;
      element item;
      if (IS_EMPTY(front))  {
            fprintf(stderr,  "The queue is empty\n");
            exit(1);
      }
      item = temp->item;
      front = temp->next;
      free(temp);
      return item;
}
```

Algorithm Analysis

• enqueue	O(?)
• dequeue	O(?)
• size	O(?)
• isEmpty	O(?)
• isFull	O(?)