Stacks and Queues

Introduction

- Stacks and queues are higher level data structures
    - You can implement a stack as an array or as a linked-list
    - Not as fundamental
        - You can't implement a linked list as an array, or implement an array as a linked list

---

Stack

- Sequence of items: LIFO
    - The only place to insert an item is from one end
    - The only place to remove an item is from the same end

Stack operations

1. Create an empty stack `stack<classType> name`
2. Push an item onto the stack (add) `name.push(element)`
    - Possibilites of failure
        - If fixed-size: what if we run out of room -> could fail
        - If resizable: that's fine
3. Pop an item from the stack (remove) `name.pop()`
    - Possibilities of failure
        - What if the stack is empty?
4. Look at the top item on the stack (top of stack is the active end of stack)
    - `name.top()` returns the top element
    - Possibilities of failure
        - What if the stack is empty?
5. Is the stack empty? `name.empty()`

> Optional items:

6. Look at any item in the stack `name.size()`
7. How many items are in the stack

Stack in C++ Library

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

//Create an empty stack
stack<int> s; //in <>: the type of data you want in a stack

//Add items
s.push(10);
s.push(20);

//Visualization of stack
s: ===>
10 20

//Check top of stack
cout << s.top() << endl; //20

//Remove item
s.pop();

//Visualization
s: ===>
10

if (s.empty())
{
    cout << "Stack is empty!" << endl;
}
else
{
    cout << s.top() << endl; //writes 10
}

//How many items
cout << s.size() << endl;
```

- There is no member function to check any item of the stack in C++ library
- `s.pop()`: returns `void`, removes the top value
  - in other languages: pop may remove and return the value

    ```
    S.PUSH(30);
    int n = s.POP(); //n is 30, 30 was removed from the stack
    ```

  - in C++

```
s.push(30);
int n = s.top();
s.pop();
```

  - what if we call this when the stack is empty?
    - C++ throws an exception, undefined behavior
      - due to performance issues
- `s.push()` method
  - Would we run out of room?
    - In C++: the size can grow, but we can run out of memory

Scenario: Expression evaluation

- Prefix notation:
  - The operation comes before the operand: `f(x, y, z)`
    - The operation is `f`, the operands are `x`, `y`, `z`
  - Mathematical operations in prefix notation
    - `add(sub(8, div(6,2)), 1)`
    - `+ - 8 / 6 2 1`: there is a unique way to interpret this expression
      - Infix notation: 8-6/2+1
        - infix notation requires more rules of precedence,associativity rules, etc.
- Postfix notation:
  - The operation comes after the operand

    `8 6 2 / - 1 +`
  - Like prefix notation, its unambiguous
- Program that can evaluate infix notation for mathematical operation
  - Algorithm:
    - Go through infix string, convert sequence into postfix string
    - Given postfix string, visit each item, evaluate the answer
  - If infix to prefix, just reverse the string and do infix to postfix
  - Infix to postfix: Use **operator stack**

- Tricky: Associativity and precedence
- If the current item is an operand, append it to the result sequence
- If the current item is (, push it onto the stack:
    - the open parentheses is pushed onto the stack as a marker to tell the algorithm when to stop popping when it reaches a closed)
- If the current item is)
    - pop operators off the stack, appending them to the result sequence, until you pop an (, which you do not append to the sequence
- If the current item is an operator:
    - If the operator stack is empty, push the current operator onto the stack
    - If the stack is not empty:
        - If the top of the stack is `(`, push the operator onto stack
        - If the current operator has precedence greater than that of the operator at the top of the stack: push current operator onto the stack
        - Otherwise:
            - pop the top operator onto stack and append it to the result sequence
            - check again (loop)
- At the end of the input sequence, pop each operator off the stack and append it to the result sequence

```
//if operand, append to result
8 - 6 / 2 + 1
^
Operator stack: =====>

result: 8

//Operator, empty stack, push onto stack
8 - 6 / 2 + 1
  ^
Operator stack: =====>
-
result: 8

//operand, append to result
8 - 6 / 2 + 1
    ^
Operator stack: =====>
-
```

```
result: 8 6

//Operator precedence >= stack operator, push onto stack
8 - 6 / 2 + 1
      ^
Operator stack: =====>
- /
result: 8 6

//operand, append to result
8 - 6 / 2 + 1
        ^
Operator stack: =====>
- /
result: 8 6 2

//Operator precedence <= stack operator, pop operator off stack
8 - 6 / 2 + 1
          ^
Operator stack: =====>
-
result: 8 6 2 /

//Check again, precedence <= operator on top of stack, pop
8 - 6 / 2 + 1
          ^
Operator stack: =====>

result: 8 6 2 / -

//Check again, now stack empty, push operator onto stack
8 - 6 / 2 + 1
          ^
Operator stack: =====>
+
result: 8 6 2 / -

//operand, append to result
8 - 6 / 2 + 1
            ^
Operator stack: =====>
+
result: 8 6 2 / - 1

//Reached the end of a string
8 - 6 / 2 + 1
             ^
Operator stack: =====>
+
result: 8 6 2 / - 1 +
```

- Evalute postfix expression: **Use operand stack**

`8 6 2 / - 1 +`

- Push operands onto stack
- When see operator, pop two items off the stack and apply operation
- Push result onto the stack

```
8 6 2 / - 1 +
^
Operand stack: =====>
8

8 6 2 / - 1 +
  ^
Operand stack: =====>
8 6

8 6 2 / - 1 +
    ^
Operand stack: =====>
8 6 2

//Operator '/' pop two items off
8 6 2 / - 1 +
      ^
Operand stack: =====>
8                                     6  /  2

//Push back result onto stack
8 6 2 / - 1 +
      ^
Operand stack: =====>
8  3

8 6 2 / - 1 +
        ^
Operand stack: =====>
                                      8  -  3

8 6 2 / - 1 +
          ^
Operand stack: =====>
5

8 6 2 / - 1 +
          ^
Operand stack: =====>
5 1

8 6 2 / - 1 +
            ^
Operand stack: =====>
```

```
                                     5   +  1

8 6 2 / - 1 +
            ^
Operand stack: =====>
6

//End of string: result = 6
```

- If the postfix string was properly formed, the result will always have one number on the stack
- Some complications:
    - Three operands: different operator symbol for it, just push three onto stack
    - One operator that means different things based on number of operands e.g. `-5` versus `8-5`
        - when converting from infix to postfix, change these into different symbols

- What about parentheses?

  `2 * (8 - (4 - 2) * 3) / 2`

  - Turn expression into equivalent postfix

```
//operand, append to result
2 * (8 - (4 - 2) * 3) / 2
^
operator stack:  ====>

result: 2

//operator, empty stack, push onto stack
2 * (8 - (4 - 2) * 3) / 2
  ^
operator stack:  ====>
*
result: 2

//Open (, push onto stack
2 * (8 - (4 - 2) * 3) / 2
    ^
operator stack:  ====>
* (
result: 2

//operand, append to result
2 * (8 - (4 - 2) * 3) / 2
     ^
operator stack:  ====>
```

```
* (
result: 2 8

//top operator on stack is (, push onto stack
2 * (8 - (4 - 2) * 3) / 2
       ^
operator stack:  ====>
* ( -
result: 2 8

//open parens, push onto stack
2 * (8 - (4 - 2) * 3) / 2
         ^
operator stack:  ====>
* ( - (
result: 2 8

//operand, append to result
2 * (8 - (4 - 2) * 3) / 2
          ^
operator stack:  ====>
* ( - (
result: 2 8 4

//top operator on stack is (, push onto stack
2 * (8 - (4 - 2) * 3) / 2
            ^
operator stack:  ====>
* ( - ( -
result: 2 8 4

//operand, append to stack
2 * (8 - (4 - 2) * 3) / 2
              ^
operator stack:  ====>
* ( - ( -
result: 2 8 4 2

//Closing ), keep popping operators and append to result
2 * (8 - (4 - 2) * 3) / 2
               ^
operator stack:  ====>
* ( - (
result: 2 8 4 2 -

//Pop open ( but not append to result
2 * (8 - (4 - 2) * 3) / 2
               ^
operator stack:  ====>
* ( -
result: 2 8 4 2 -

//* operator has greater precedence than -, push onto stack
2 * (8 - (4 - 2) * 3) / 2
```

```
                 ^
operator stack:  ====>
* ( - *
result: 2 8 4 2 -

//operand, append to result
2 * (8 - (4 - 2) * 3) / 2
                   ^
operator stack:  ====>
* ( - *
result: 2 8 4 2 - 3

//Closed ), keeping popping operators and append until open (, then pop (
2 * (8 - (4 - 2) * 3) / 2
                    ^
operator stack:  ====>
*
result: 2 8 4 2 - 3 * -

// Division operator does not have greater precedence than *, pop *
2 * (8 - (4 - 2) * 3) / 2
                      ^
operator stack:  ====>

result: 2 8 4 2 - 3 * - *

//Push Division operator onto stack now that its empty
2 * (8 - (4 - 2) * 3) / 2
                      ^
operator stack:  ====>
/
result: 2 8 4 2 - 3 * - *

//operand, append to result
2 * (8 - (4 - 2) * 3) / 2
                        ^
operator stack:  ====>
/
result: 2 8 4 2 - 3 * - * 2

//End of string
2 * (8 - (4 - 2) * 3) / 2
                        ^
operator stack:  ====>

result: 2 8 4 2 - 3 * - * 2 /
```

Queue

- Sequence of items: FIFO
- Stack v. Queue:
    - One active end v. Two active ends

Queue operations

1. Create an empty queue `queue<classType> name`
2. Enqueue an item (add a new item) `name.push(element)`
3. Dequeue an item (remove from other end) `name.pop()`
4. Look at the front item `name.front()`
    - We don't tend to need to look at the back of the queue, guy just waits
    - You can't look at the front of the queue if it has no items
    - returns a reference to the front most element
5. Is the queue empty `name.empty()`

> Optional operations:

6. Look at the back item in the queue `name.back()`
7. Look at any item in the queue
8. How many items are in the queue `name.size()`

Queue in C++

```
#include <queue>

using namespace std;

//Create
queue<int> q;

//Enqueue
q.push(10);
q.push(20);

//Visualization
q: x --> x x x x x --> x
        20    10

//Look at front of queue
cout << q.front() << endl; //10

//Dequeue
q.pop();
```

```
//Visualization
q: x --> x x x x x --> x
            20

//Check if queue empty
if (q.empty())
{
    cout << "Queue is empty!" << endl;
}
else
{
    cout << q.front() << endl; //writes 20
}

//Check size
cout << q.size() << endl; //writes 1

//Can look at the back of the queue
cout << q.size() << endl; //writes 20
```

- You can't look at any item in the queue
- `q.pop()` returns void (just as in stack)
- If you try to ask about or remove front/back of empty queue -> undefined behavior

---

Implementation of Stack and Queue

Implement a stack as an array

- If I have an array with elements
- To push an item, just insert at the end
- To pop an item, just remove at the end
- Everything is efficient as long as we know where the top of the stack it
    - We need an integer subscript or pointer to keep track
    - `top` indicate *just past the last item* instead of last item itself
        - Scenario:
            - Underlying integer array of one item in the stack
            - `top` would take integer subscript `0` (since item at index 0) and we only have 1 item
            - When we pop the top --> `top` would be `-1`
            - HOWEVER: if use pointer representation

            - `top` points to index 0
            - if we pop, we subtract 1 from the pointer -> undefined behavior to have a pointer to element negative 1
        - `top` is also going to be the size of the stack
    - To push an item:
        - `top` tells me where to store the new item
        - increment `top`
        - trying to push to a full capacity
            - if `top` is index just past array -> fail
            - resizable array: `top` just past array tells you to allocate a bigger array and delete the old array
    - To pop an item:
        - subtract 1 from `top`
        - we really don't do anything to the popped item, since we don't care, the limits have changed
        - popping from an empty stack
            - `top` is `0` tells you the stack is empty
            - if try to look at top of the stack (`top` index - 1) that is undefined behavior
            - if pop from stack, `top` will become `-1`
                - then if we try checking we will be looking into `-2`
                - then if we try pushing we will be pushing to `-1`, `top` will end up at 0 again

Implement a stack using a linked list

- Head pointer points to the front, item points to the next
- This is BAD!
    - Everytime I want to push I have to traverse the list
    - What if we have tail pointer? --> now we can push easily
    - But what if we want to pop? --> to go backwards we have to do a doubly-linked list
- Rethink this: the 'end' of the stack be the end the head pointer points to
    - Push:
        - Allocate a new node

    - Copy the head in there, make head point to first node
    - Push one more: allocate one node, set value, copy pointer to next pointer, make head point to new node

  -Typically call the head point `top`

  - Pop:
    - Save a copy of the top pointer
    - Change top to point to second item
    - If try to pop from empty stack --> following nullptr --> crash

Implement a Queue Using An Array

- Enqueue
  - 10 at index 0, 20 at index 1, etc.
- Dequeue
  - Get rid of something
- Need to keep track of where the first and last item is: two pointers
  - Point to head: front
    - Dequeue, remove item, advance head pointer
  - Tail pointer point to end, for enqueue
- If fixed sized array
  - Enqueue and dequeue process will lead to an array like this:

    ```
    [] [] [] [] [] []... [23] [87]
                          ^    ^
                          h    t
    ```

  - When we reach the end, do we want to allocate more memory? NO
    - Queue has only two items, but array capacity is a 100, 98 empty spots
      - If `t` is pointing to very last item, then copy all items to the beginning of the array
  - Solution 1:
    - Copy items back to the beginning

      ```
      [23][87]...[][][]
      ```

  - Allocate more memory only when truly full
  - **Disadvantages**: costly
    - 98 elements in array, reached the end -> our algorithm tells us to move copy 98 elements to beginning to give 2 more empty spots
    - Expensive copy
    - A very short time later, we hit the end AGAIN -> have to shift again
- Solution 2: Wrap around back to the beginning
  - Pretend that the next item after index 99 is 0

    ```
    [87][][][][][][][21]
    ^                  ^
    t                  h
    ```

  - If at last item, then reset to beginning
  - Allocate more memory if truly full
  - Treat linear memory as if it were stored in a circle
    - Ring buffer, 'circular array'
  - We have two pointers, tail pointer points just *past* the last

  ```
  [8][3][][][][][]
  ^     ^
  h     t
  ```

  - When we dequeue:
    - when queue empty -> head pointer equal to the tail pointer
    - but NOT the other way around
      - When the queue has 99 items in it, add one more tail pointer advances to the head pointer -> the queue is FULL

        ```
        [6][7][][8][9]
               ^ ^
               t h
        ```

    - hence have to keep track of the size `size` variable

        - Growing the array:
            - Copy, start at head, copy into index 0 of new array, go until 99, etc.
                - Copy the element the `head` pointer is pointing to into index 0, NOT actually index 0

C++ Library Implementation of Stack and Queue

- As long as popping and pushing are cheap -> use that