CS 32 Week 4
Discussion 1B

TA: Yiyou Chen
LA: Ian Galvez

A couple things before we start...

- Midterm season can be really stressful!
- Just a reminder that your worth is not determined by your grades
- If you’re feeling like you’re struggling in this class, please don’t hesitate to reach out to the professor, TAs, LAs, or your peers.
- TAs/LAs hold lots of office hours during the week and I highly recommend coming to them!

A couple things before we start…

- If you’re not already doing so, find a partner or a group of people to work on the worksheet with! Ready… go!
- I’ll give y’all some time to break the ice a bit…
  - Rant about something
  - Rave about something
  - Argue whether pineapple on pizza is permissible

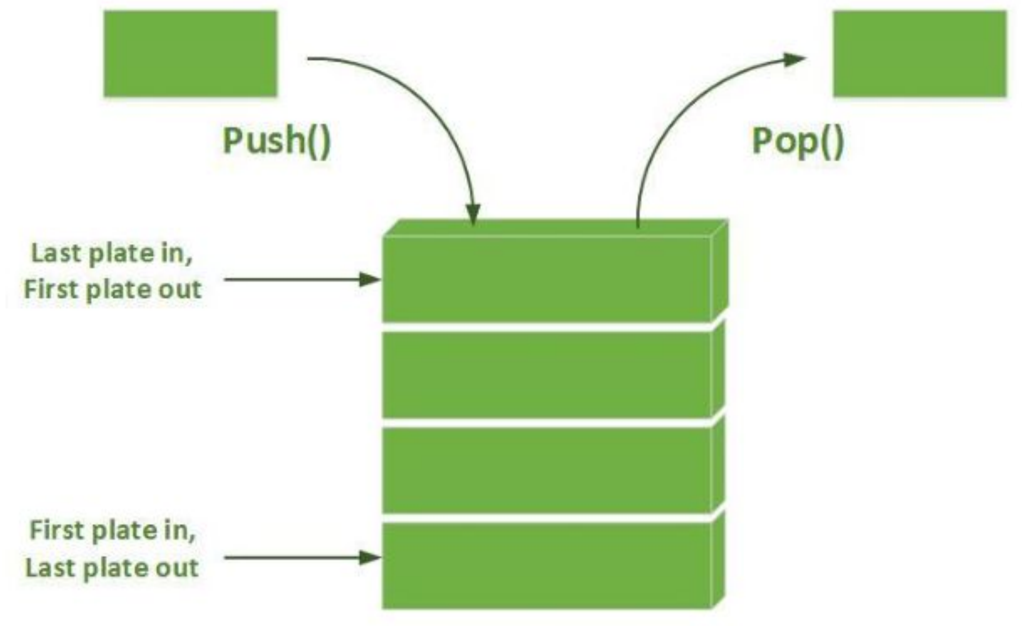
Topics

- **Stacks and Queues:**
  1. Definitions and Operations
  2. Infix->Postfix conversion and evaluation of postfix expressions using stacks
  3. Depth-first Search and Breadth-first Search

Stack

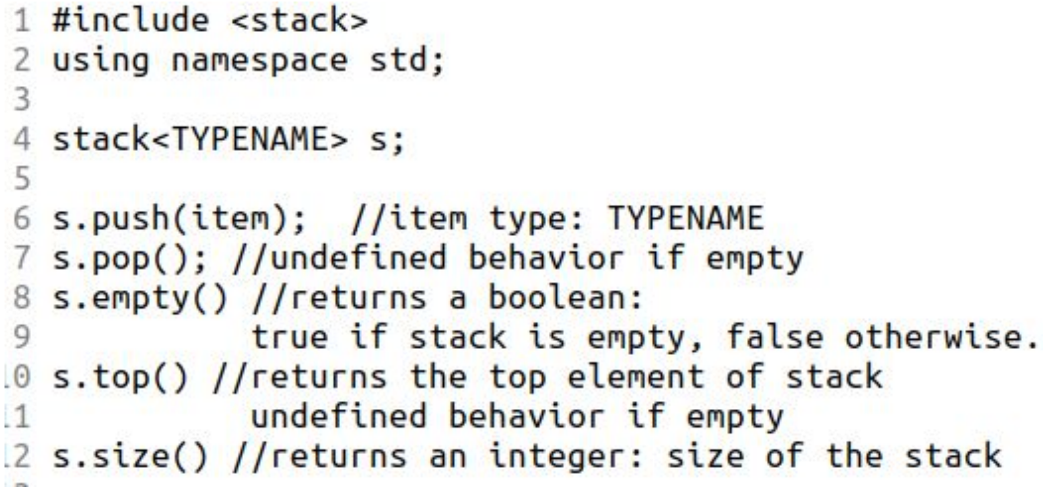
A sequential data structure.

First in last out (last in first out).



https://appdividend.com

C++ Stack



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

stack<TYPENAME> s;

s.push(item);  //item type: TYPENAME
s.pop(); //undefined behavior if empty
s.empty() //returns a boolean:
          true if stack is empty, false otherwise.
s.top() //returns the top element of stack
          undefined behavior if empty
s.size() //returns an integer: size of the stack
```

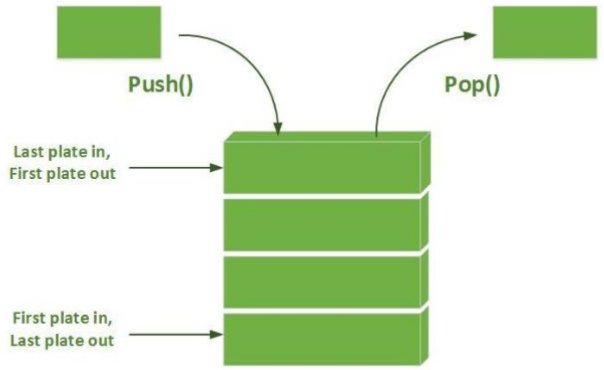


Image: https://appdividend.com

Queue

A sequential data structure.

First in First out (FIFO).

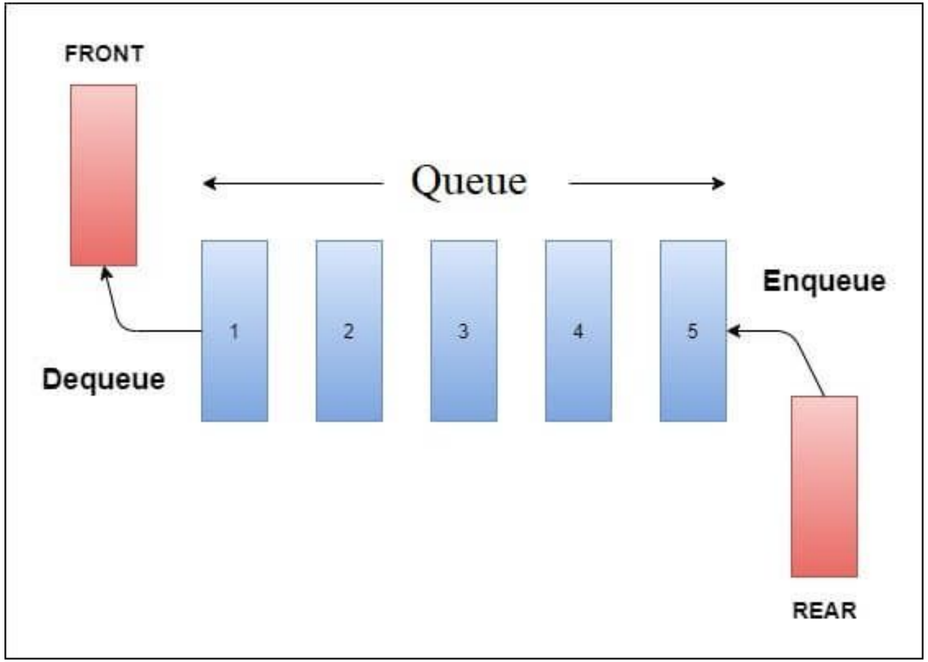


Image: https://www.tutorialandexample.com

C++ Queue

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

queue<TYPENAME> q;

q.push(item); //enqueue
q.pop(); //dequeue: undefined behavior if empty
q.empty() //returns a boolean:
          true if queue is empty, false otherwise.
q.front() //returns the front element of queue
          undefined behavior if empty
q.back() //returns the back element of queue
          undefined behavior if empty
q.size() //returns an integer: size of the queue
```

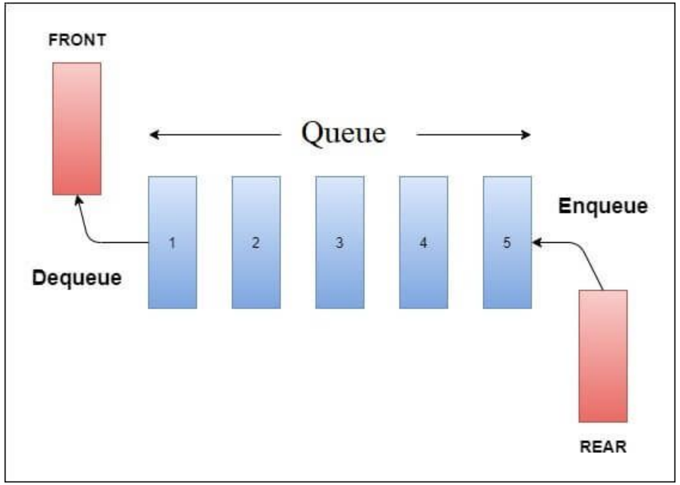


Image: https://www.tutorialandexample.com

Prefix, Infix, Postfix expressions

Pre-, in-, and post- tell us the location of **operators**.

E.g.

Infix: (3+2)*7-6/2

Postfix:

Prefix:

Prefix, Infix, Postfix expressions

Pre-, in-, and post- tell us the location of **operators**.

E.g.

Infix: (3+2)*7-6/2

Postfix: 3 2 + 7 * 6 2 / -

Prefix: - * + 3 2 7 / 6 2

Infix -> postfix using stack

Infix: 3+(2*7-6)/2

Postfix: 3 2 7 * 6 - 2 / +

Observations: Let O1, O2, O3, O4, … be the operators. Then we add On if O(n-1) < On >= O(n+1).

1. The order of operands doesn't change, so we just need to add the operators in correct order.
2. Use a stack to keep track of the operators of the highest precedence so far. No operators with equal or higher precedence should occur in the stack before the current (non '(' operator. If so, pop them out.
3. '(' has lowest precedence, but once paired with ')' they make the operators in between them highest precedence.

3 2	+ (
3 2 7	+ (* -
3 2 7 * 6	+ (-)
3 2 7 * 6 -	+ /
3 2 7 * 6 - 2 / +	

Exercise

Print the stack at each operand.

(5+6)/(3*((6+5)*7+8 - 2) + 9)

Exercise

Write out the stack at each operand.

(5+6)/(3*((6+5)*7+8 - 2) + 9)

5: (

5 6: (+

5 6 + 3: / (

5 6 + 3 6: / (* ((

5 6 + 3 6 5: / (* ((+

5 6 + 3 6 5 + 7: / (* (*

5 6 + 3 6 5 + 7 * 8: / (* (+

5 6 + 3 6 5 + 7 * 8 + 2 : / (* (-

5 6 + 3 6 5 + 7 * 8 + 2 - * 9: / (+

5 6 + 3 6 5 + 7 * 8 + 2 - * 9 + / :

Evaluate postfix using stack

Postfix: 3 2 7 * 6 - 2 / +

Each operator always acts on the last two operands before it.

Stack:

3 2 7 *

3 14 6 -

3 8 2 /

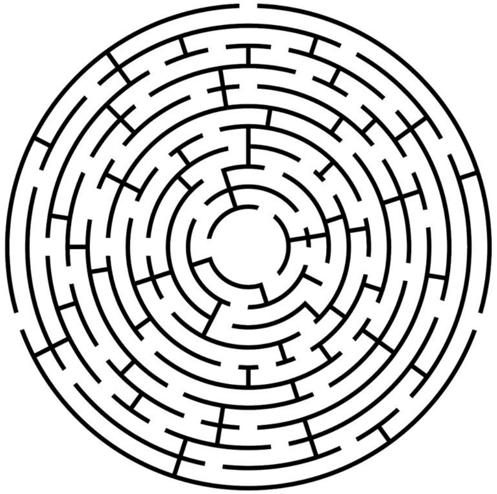
3 4 +

7

Searching Algorithms

Depth-first search (DFS): I’ll stick to this path till I reach a dead end!

Breadth-first search (BFS): I’ll explore all paths at the same time!

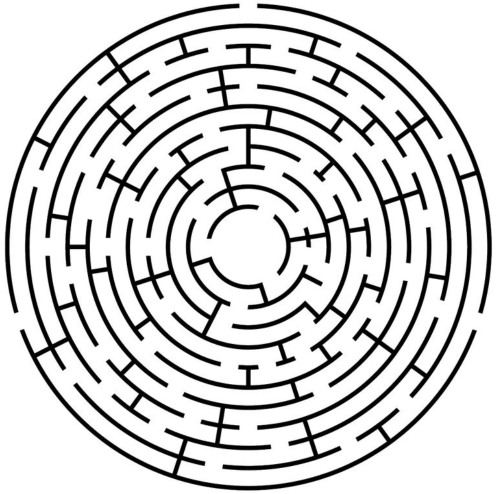


https://www.popsci.com

Searching Algorithms

Depth-first search (DFS): I’ll stick to this path till I reach a dead end!

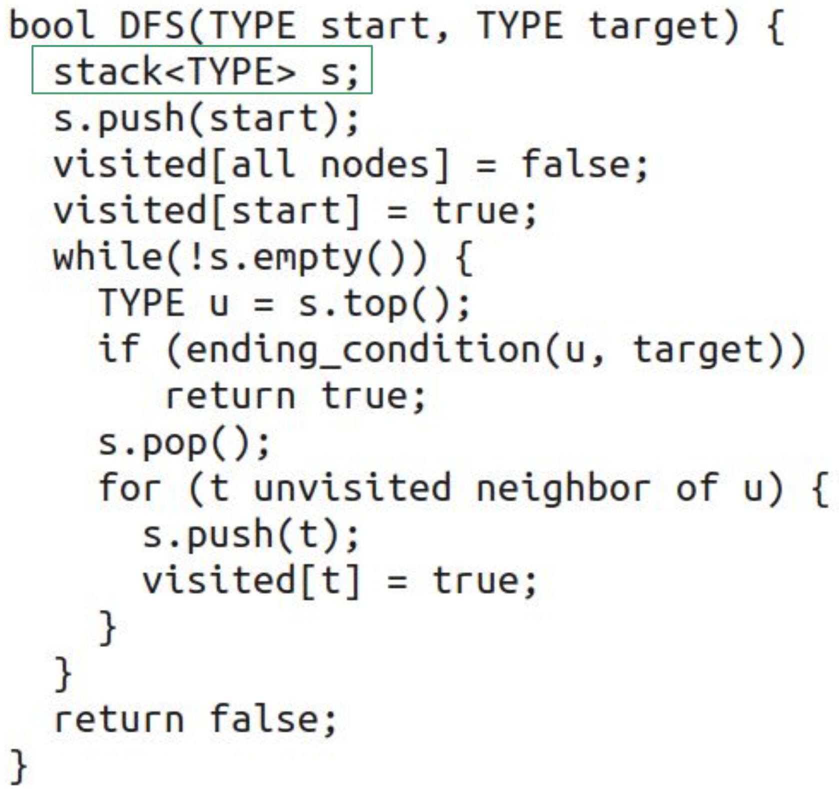
Breadth-first search (BFS): I’ll explore all paths at the same time!



Q: Is there any case when DFS will be inefficient?

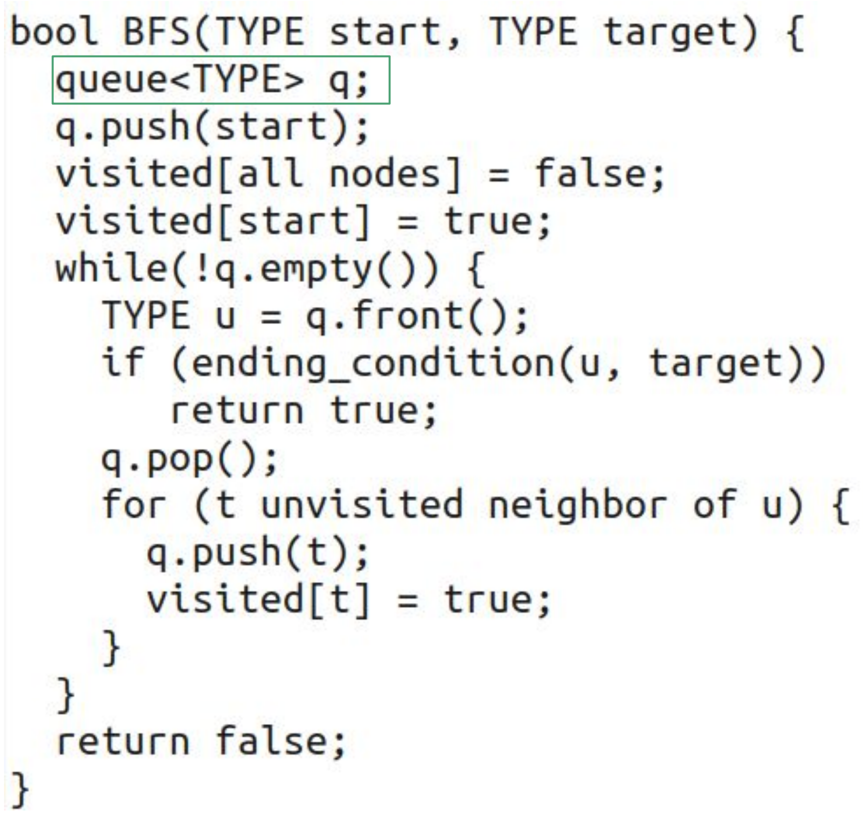
https://www.popsci.com

Depth-First Search (DFS) using a stack



```
bool DFS(TYPE start, TYPE target) {
  stack<TYPE> s;
  s.push(start);
  visited[all nodes] = false;
  visited[start] = true;
  while(!s.empty()) {
    TYPE u = s.top();
    if (ending_condition(u, target))
       return true;
    s.pop();
    for (t unvisited neighbor of u) {
      s.push(t);
      visited[t] = true;
    }
  }
  return false;
}
```

Breadth-First Search (BFS) using a queue



```
bool BFS(TYPE start, TYPE target) {
  queue<TYPE> q;
  q.push(start);
  visited[all nodes] = false;
  visited[start] = true;
  while(!q.empty()) {
    TYPE u = q.front();
    if (ending_condition(u, target))
       return true;
    q.pop();
    for (t unvisited neighbor of u) {
      q.push(t);
      visited[t] = true;
    }
  }
  return false;
}
```

Worksheet Time!

How should we approach the problems?

- Check your intuition! Ask yourself the following questions:
    - What is our input? Is it an int? An array? A stack? A pointer?
    - What is our output? Do we write anything to cout?
    - Before you write any code, think about how you'd intuitively solve the problem step-by-step. Translate that into pseudocode!
    - Try to think through other test cases - if you gave your function this input, what would the expected output be?
    - Using your pseudocode, turn that into valid C++ code
    - Check test cases to make sure your code is sound!

Problem 1

- Given a string of `'('`, `')'`, `'['`, and `']'`,
  write a function to check if the input string is valid.
- Validity is determined by each `'('` having a corresponding `')'`,
  and each `'['` having a corresponding `']'`,
  with parentheses being properly nested and brackets being properly nested

Examples:

```
“[()([])[[([][])]]]” → Valid
“((([(])))“ → Invalid
“(()))” → Invalid
“()[]” → Valid
```

Problem 1 – Intuition

- Every open-paren needs exactly one close-paren (and vice versa)
- Our stack maintains the sequence of opening parentheses and brackets, and removes an opening symbol upon seeing the matching closing one.
- How can we immediately tell if our input string is invalid?
    - If we have a closing symbol, the top of the stack MUST be the matching opening symbol!
    - If the stack is empty, this is invalid (e.g. we have the string “`)`” – where’s the open-paren??)
    - We also can’t have something like this: “`([)]`” - no interleaving allowed!
- Hint, hint: We’ve been talking an awful lot about stacks…

Problem 1 – **Solution**

```
bool isValid(string symbols) {
      stack<char> openers;
      for (int k = 0; k != symbols.size(); k++) {
            char c = symbols[k];
            switch (c) {
                  case '(':
                  case '[':
                        openers.push(c);
                        break;
                  case ')':
                        if ( openers.empty() || openers.top()!='(' )
                              return false;
                        openers.pop();
                        break;
                  case ']':
                        if (openers.empty() || openers.top()!='[')
                              return false;
                        openers.pop();
                        break;
            }
      }
      return openers.empty();
}
```

Problem 2

Give an algorithm for reversing a queue of integers Q.
Only the following standard operations are allowed on queue:

`Q.push(x)`	Add an item x to the back of the queue.
`Q.pop()`	Remove an item from the front of the queue.
`Q.front()`	Return the item at the front of the queue
`Q.empty()`	Check if the queue is empty or not.

Hint: You may use an additional data structure if you wish.

Example:
Input: `Q = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]`
Output: `Q = [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]`

Problem 2 – ***Solution***

```
void reverseQueue(queue<int>& Q){
    // use an auxiliary stack
    stack<int> S;
    while (!Q.empty()) {
        S.push(Q.front());
        Q.pop();
    }
    while (!S.empty()) {
        Q.push(S.top());
        S.pop();
    }
}
```

Problem 3

- Write a function `findNextInts` that takes in two integer arrays of size n: `sequence` and `results`.
  - This function assumes that `sequence` already contains a sequence of positive integers.
  - For each position i (from 0 to n-1) of sequence, this function should find the smallest index j such that j > i and sequence[j] > sequence[i], and put sequence[j] in results[i].
  - If there is no such j, put -1 in sequence[i].
  - Try to do this without nested for loops both iterating over the array! (Hint: #include <stack>).

```
void findNextInts(const int sequence[], int results[], int n);
```

```
int seq[] = {2, 6, 3, 1, 9, 4, 7 }; // Only positive integers!
int res[7];
findNextInts(seq, res, 7);
for (int i = 0; i < 7; i++) { // Should print: 6 9 9 9 -1 7 -1
     cout << res[i] << " ";
cout << endl;
```

Notice that the last value in results will always be set to -1,
since there are no integers in sequence after the last one!

Problem 4

Evaluate the following postfix expression. Show your work!

```
9 5 * 8 - 6 7 * 5 3 - / *
```

Problem 5

- Implement a **`Stack`** class using only queues as data structures.
- This class should implement the `empty`, `size`, `top`, `push`, and pop member functions, as specified by the standard library's implementation of stack.
- (The implementation will not be very efficient.)

Problem 6

- Implement a **`Queue`** class using only stacks as data structures.
- This class should implement the `empty`, `size`, `front`, `back`, `push`, and pop member functions, as specified by the standard library's implementation of queue.
- (The implementation will not be very efficient, you may use local variables for storage within your functions.)
- (Another side note: It's quite a bit more challenging to implement queues using stacks than the other way around!)

Solutions for Problems 3-6

to be posted on CS 32 Website