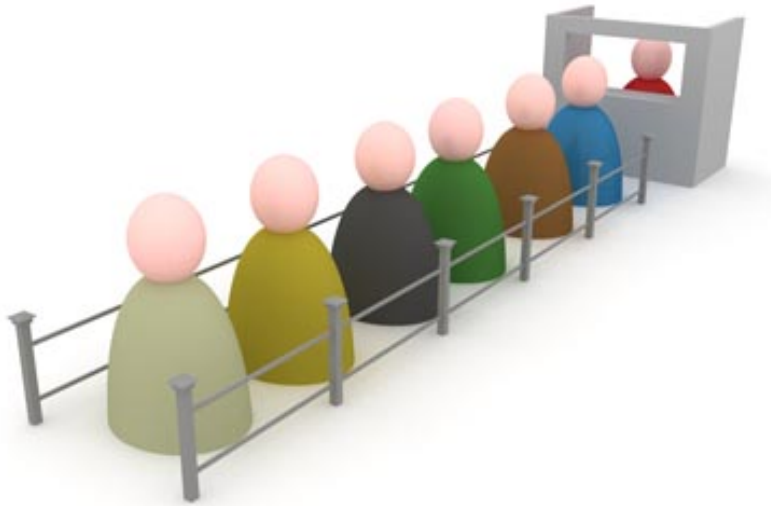
Lecture #5

- Stacks
- Queues



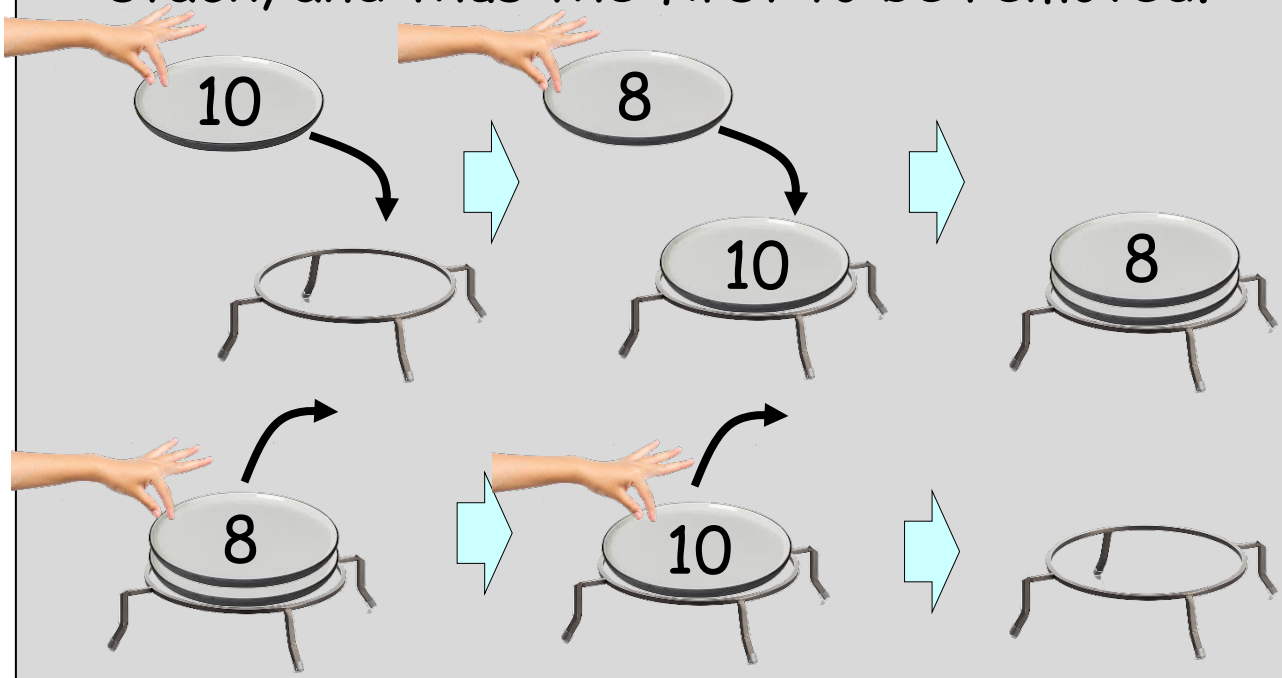

GOT A NEW
ERROR
PROGRESS
memegenerator.net

Stacks

What's the big picture?

A stack is a data structure that resembles a stack of plates at a buffet.

Just like a stack of plates, the last plate/value you add is at the top of the stack, and thus the first to be removed.



The last-added/first-removed aspect of stacks can help solve many hard problems!

Uses:

Use stacks to find a path through a maze, "undo" in a word processor, and evaluate math expressions.

The Stack: A Useful ADT

A stack is an ADT that holds a collection of items (like ints) where the elements are always added to one end.

Just like a stack of plates, the last item pushed onto the top of a stack is the first item to be removed.

Stack operations:

- put something on top of the stack (PUSH)
- remove the top item (POP)
- look at the top item, without removing it
- check to see if the stack is empty

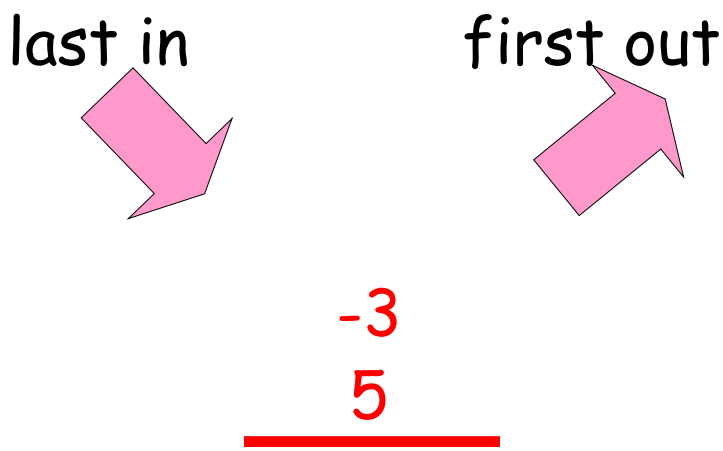
We can have a stack of any type of variable we like:
ints, Squares, floats, strings, etc.

The Stack

Note: The stack is called a Last-In-First-Out data structure.
Can you figure out why?

I can…

Push 5 on the stack.
Push -3 on the stack.
Push 9 on the stack.
Pop the top of the stack.
Look at the stack's top value.
Push 4 on the stack.
Pop the top of the stack
Pop the top of the stack
Look at the stack's top value.
Pop the top of the stack



Note: You can only access the top item of the stack, since the other items are covered.

Stacks

```
class Stack   // stack of ints
{
public:
   Stack();        // c'tor
   void push(int i);
   int pop();
   bool is_empty(void);
   int peek_top();
private:
   ...
};
```

```
int main(void)
{
    Stack is;

    is.push(10);
    is.push(20);

    ...
}
```

Question:
What type of data structure can we use to implement our stack?

Answer:
How about an array and a counter variable to track where the top of the stack is?

Implementing a Stack

```
const int SIZE  = 100;
class Stack
{
public:
    Stack()  {  m_top = 0;   }
    void push(int val)  {
        if (m_top >= SIZE) exit(-1); // overfl
        m_stack[m_top] = val;
        m_top += 1;
    }

    int pop()  {
        if (m_top == 0) exit(-1);
        m_top -= 1;
        return m_stack[m_top];
    }
    ...
private:
    int m_stack[SIZE];
    int m_top;
};
```

To initialize our stack, we'll specify that the first item should go in the 0th slot of the array.

Let's make sure we never over-fill (overflow) our stack! For this simple example, let's just exit if this happens.

Place our new value in the next open slot of the array… m_top specifies where that is!

Update the location where our next item should be placed in the array.

We can't pop an item from our stack if it's empty! This is called a stack "underflow!" Terminate our program!

Since m_top points to where our next item will be pushed… Let's decrement it to point it to where the current top item is!

Extract the value from the top of the stack and return it to the user.

Let's use an array to hold our stack items. This stack may hold a maximum of 100 items.

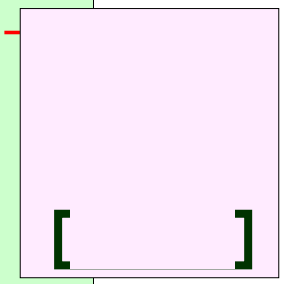
We'll use a simple int to keep track of where the next item should be added to the stack.

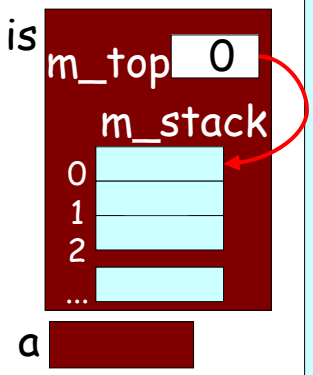
Stacks

```
const int SIZE  = 100;
class Stack
{
public:
    Stack() { m_top = 0; }
    void push(int val) {
        if (m_top >= SIZE) exit(-
        m_stack[m_top] = val;
        m_top += 1;
    }

    int pop() {
        if (m_top == 0) exit(-1);
        m_top -= 1;
        return m_stack[m_top];
    }

    ...
private:
     int m_stack[SIZE];
     int m_top;
};
```

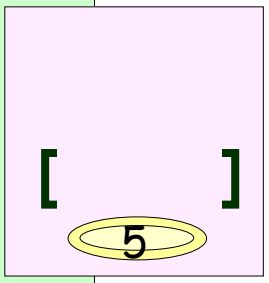
```
int main(void)
{
     Stack is;     // #0
     int a;

     is.push(5);   // #1
     is.push(10);  // #2
     a = is.pop();// #3
     cout << a;
     is.push(7);   // #4
}
```

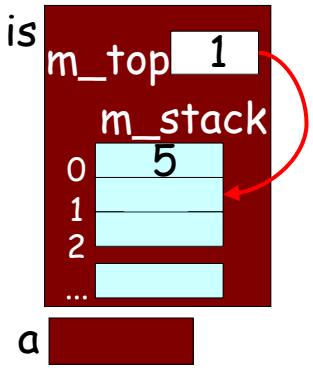
After line #0:



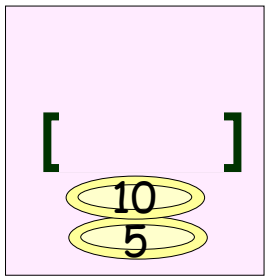


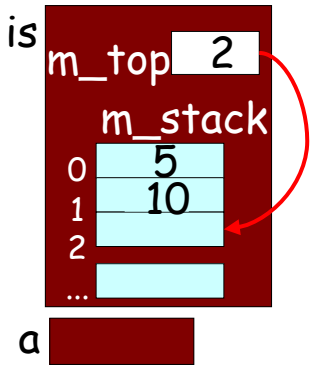
After line #1:



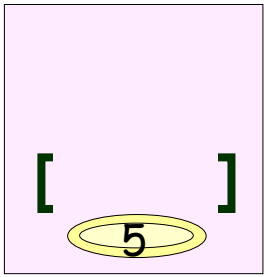


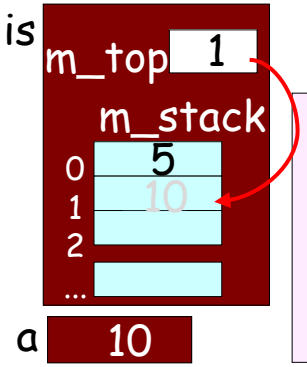
After line #2:



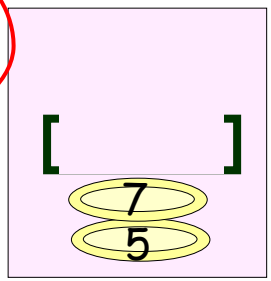


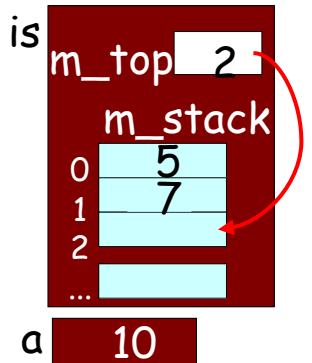
After line #3:





After line #4:





```
const int SIZE  = 100;
class Stack
{
public:
    Stack() { m_top = 0; }
    void push(int val) {
        if (m_top >= SIZE) exit(-1); // overflow
        m_stack[m_top] = val;
        m_top += 1;
    }

    int pop() {
        if (m_top == 0) exit(-1); // underflow
        m_top -= 1;
        return m_stack[m_top];
    }

    ...
private:
     int m_stack[SIZE];
     int m_top;
};
```

Always Remember:

When we push, we:

A. Store the new item in m_stack[m_top]
B. Post-increment our m_top variable
(post means we do the increment after storing)

Always Remember:

When we pop, we:

A. Pre-decrement our m_top variable
B. Return the item in m_stack[m_top]
(pre means we do the decrement before returning)

Stacks

Stacks are so popular that the C++ people actually wrote one for you. It's in the Standard Template Library (STL)!

```
#include <iostream>
#include <stack> // required!

int main()
{
  std::stack<int>  istack; // stack of ints

  istack.push(10);         // add item to top
  istack.push(20);

  cout << istack.top();  // get top value
  istack.pop();          // kill top value

  if (istack.empty() == false)
    cout << istack.size();
}
```

Here's the syntax to define a stack:

std::stack<type> variableName;
For example:
std::stack<string> stackOfStrings;
std::stack<double> stackOfDoubles;

And as with cin and cout, you can remove the std:: prefix if you add a using namespace std; command!

So to get the top item's value, before popping it, use the top() method!

Note: The STL pop() command simply throws away the top item from the stack… but it doesn't return it.

Stack Challenge

Show the resulting stack after the following program runs:

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

int main()
{
  stack<int>  istack;       // stack of ints

  istack.push(6);
  for (int i=0;i<2;i++)
  {
     int n = istack.top();
     istack.pop();
     istack.push(i);
     istack.push(n*2);
  }
}
```

Common Uses for Stacks

Stacks are one of the most USEFUL data structures in Computer Science.

They can be used for:

- Storing undo items for your word processor
  The last item you typed is the first to be undone!
- Evaluating mathematical expressions
  5 + 6 * 3 → 23
- Converting from infix expressions to postfix expressions
  A + B → A B +
- Solving mazes

In fact – they're so fundamental to CS that they're built into EVERY SINGLE CPU in existence!

A Stack… in your CPU!

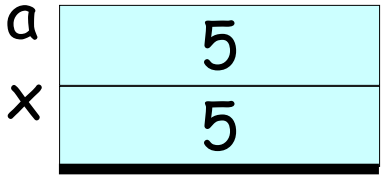
Did you know that every CPU has a built-in stack used to hold local variables and function parameters?


```
void bar(int b)
{
  cout << b << endl; // #3
}

void foo(int a)
{
  cout << a << endl; // #2
  bar(a*2);          // #4
}

int main(void)
{
  int x = 5;  // #1
  foo( x );
}
```

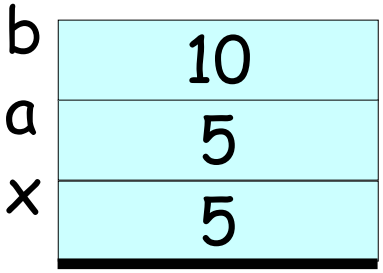
- Every time you *declare a local variable*, your program pushes it on the PC's stack in memory automatically!
- When you pass a value to a function, the CPU pushes that value onto a *stack* in the computer's memory.
- … when your function returns, the values are popped off the stack and go away.

After line #1:

After line #2:



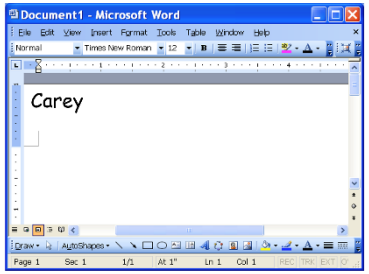
After line #3:



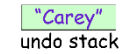
After line #4:

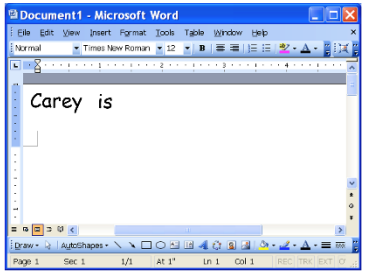
Undo!

- So how does the UNDO feature of your favorite word processor work? It uses a stack, of course!
- Every time you type a new word, it's added to the stack!
- Every time you cut-and-paste an image into your doc, it's added to the stack!
- And even when you delete text or pictures, this is tracked on a stack!
- In this way, the word processor can track the last X things that you did and properly undo them!
- When the user hits the undo button… The word processor pops the top item off the stack and removes it from the document!

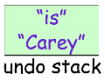


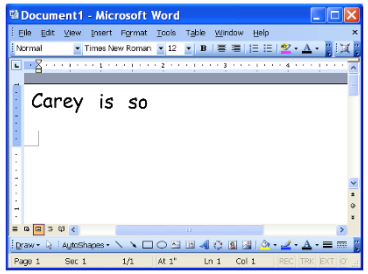
The user has typed "Carey" into the word processor. It's added to the stack.



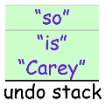


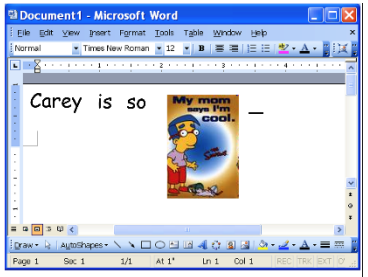
The user has typed "is" into the word processor. It's added to the stack.



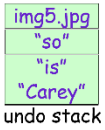


The user has typed "so" into the word processor. It's added to the stack.

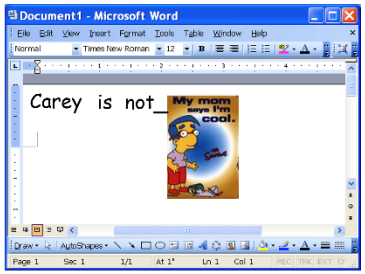


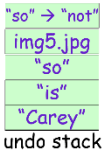


The user has pasted a photo into the word processor. It's noted on the stack.

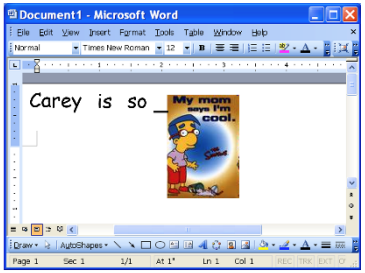


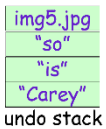
The user has replaced "so" with "not" into the word processor. The change is added to the stack.



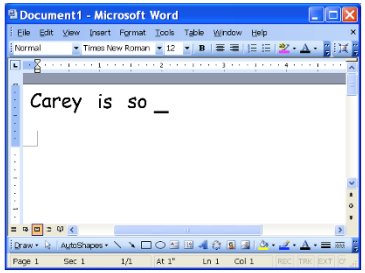


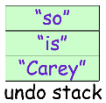
The user has hit the undo button and the "so" -> "not" is popped from the stack and reversed in the doc.



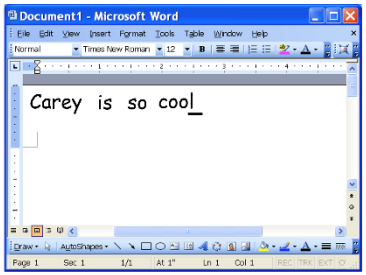


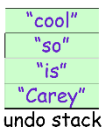
The user has hit the undo button and the image is popped from the stack and removed from the doc.





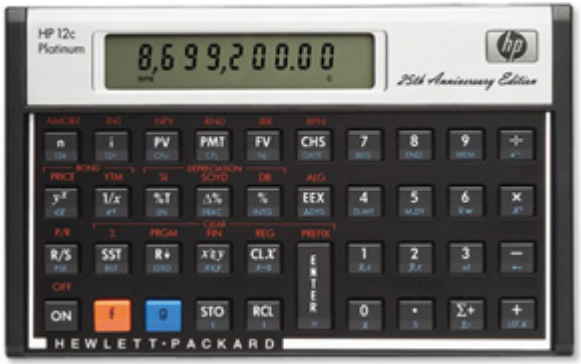
The user has typed "cool" into the word processor. It's added to the stack.





Postfix Expression Evaluation

- Most people are used to infix notation, where the operator is in-between the two operands, e.g.: A + B
- Postfix notation is another way to write algebraic expressions – here the operator follows the operands: A B +
- Postfix expressions are easier for a computer to compute than infix expressions, because they're *unambiguous*.
  - Ambiguous infix expression example: 5 + 10 * 3
  - Is that (5+10) * 3 or 5 + (10 * 3)
- To understand infix expressions, the computer has to be equipped with precedence rules (which is complicated).
- As we'll see, postfix expressions have no such ambiguity!
- If you've ever used an HP calculator, you've used postfix notation! (Ask you parents!)



Here are some infix expressions and their postfix equivalents:

Infix	Postfix
15 + 6	15 6 +
9 – 4	9 4 -
(15 + 6) * 5	15 6 + 5 *
7 * 6 + 5	7 6 * 5 +
3 + (4 * 5)	3 4 5 * +

Postfix Evaluation Algorithm

- Here's the algorithm to evaluate a postfix expression. It's SUPER simple!
- In the next few handout slides, we'll show the algorithm in action computing the result for a simple postfix expression: 7 6 * 5 +

Inputs: postfix expression string
Output: number representing answer
Private data: a stack

1. Start with the left-most token.
2. If the token is a number:
   a. Push it onto the stack
3. Else if the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to v1 and v2 (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!

Postfix Evaluation Algorithm

- Here we've already run through lines 1 and 2.
- We started with the left-most token (7) and determined it was a number.
- So we pushed it onto the stack in 2a
- After this, we'll run line 4, which checks to see if there are more tokens (there are, we still have 6 * 5 + left to process), so we:
  - Advance to the second token (6) and,
  - Go back to step #2

Inputs: postfix expression string
Output: number representing answer
Private data: a stack

7 6 * 5 +

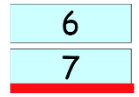
1. Start with the left-most token.
2. If the token is a number:
   → a. Push it onto the stack
3. Else if the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to v1 and v2 (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!

- We can see here we've advanced to the second token (6)
- Line #2 has run and determined that this second token (6) is also a number.
- So we pushed it onto the stack in 2a as well.
- Now our stack has two values.
- After this we'll run line 4, which checks to see if there are more tokens (there are, we still have * 5 + left to process) so we:
  - Advance to the third token (*) and,
  - Go back to step #2

Inputs: postfix expression string
Output: number representing answer
Private data: a stack

7 6 * 5 +

1. Start with the left-most token.
2. If the token is a number:
   → a. Push it onto the stack
3. Else if the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to v1 and v2 (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!



Postfix Evaluation Algorithm

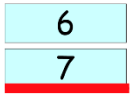
- We can see here we've advanced to the third token (*)
- Line #2 has run and determined that this second token is NOT a number.
- Line #3 has also run and determined that the token is an operator. * is an operator.
- So in step 3a, we are about to pop the top two values off of the stack into two temporary variables so we can apply the operator to them.
- We're about to pop the top value on the stack (6) into a variable called v2; this belongs in v2 because it comes LATER in the expression
- Then we'll pop the second to last value (7) into a variable called v1; this belongs in v1 because it became earlier in the expression

- Here we have already run steps 3a, 3b and 3c.
- We popped 6 into v2
- We popped 7 into v1
- We applied the current operator (*) to v1 * v2, which is 7 * 6 and got the result of 42.
- We then pushed the result of 42 back onto the stack
- NOTE: The details of which value goes into v1 and which goes into v2 are critical! What if instead of a *, the operator was division, for example: 7 / 6 is far different than 6 / 7
- So by the end of line 3c, our stack contains only one value, the result of 7 * 6.
- We advance to line 4, which sees that we have more tokens (e.g., 5 +) left, so we:
  - Advance to the fourth token (*) and,
  - Go back to step #2

Inputs: postfix expression string
Output: number representing answer
Private data: a stack

7 6 * 5 +

1. Start with the left-most token.
2. If the token is a number:
   a. Push it onto the stack
3. Else if the token is an operator:
   → a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to v1 and v2 (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!

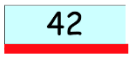


Inputs: postfix expression string
Output: number representing answer
Private data: a stack

7 6 * 5 +

1. Start with the left-most token.
2. If the token is a number:
   a. Push it onto the stack
3. Else if the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to v1 and v2 (e.g., v1 / v2)
   → c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!

v1 = 7 v2 = 6
temp = 7*6 = 42



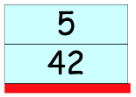
Postfix Evaluation Algorithm

- We run into another number, so we push it too onto the stack
- We'll stop the example here, but you can see what will happen next.
- We'll end up reaching the + operator, and then we'll pop 5 into v2, and 42 into v1
- Then we'll add v1 + v2 (42 + 5) and get a result of 47
- This will be pushed onto the stack as the only item
- Finally, we'll reach the end of our expression, and line #5 will run.
- It will see that we have one item on the stack, which is the answer!!!
- We return that value.
- If we had more than one value on the stack when we reached the end of the expression, it means there was an error.
- For example, 1 2 3 4 +
- Would result in a stack with:
  7
  2
  1
- This is an incomplete expression, and should result in an error!

Inputs: postfix expression string
Output: number representing answer
Private data: a stack

7 6 * 5 +

1. Start with the left-most token.
2. If the token is a number:
   → a. Push it onto the stack
3. Else if the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to v1 and v2 (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!



Class Challenge

Given the following postfix expression: 6 8 2 / 3 * -

Show the contents of the stack *after* the 3 has been processed by our postfix evaluation algorithm.

Reminder:

1. Start with the left-most token.
2. If the token is a number:
   a. Push it onto the stack
3. If the token is an operator:
   a. Pop the top value into a variable called v2, and the second-to-top value into v1.
   b. Apply operator to the two #s (e.g., v1 / v2)
   c. Push the result of the operation on the stack
4. If there are more tokens, advance to the next token and go back to step #2
5. After all tokens have been processed, the top # on the stack is the answer!

Infix to Postfix Conversion

- Stacks can also be used to convert infix expressions to postfix expressions:
- Since people are more used to infix notation…
- You can let the user type in an infix expression…
- And then convert it into a postfix expression.
- Finally, you can use the postfix evaluation alg (that we just learned) to compute the value of the expression.

For example,

From: (3 + 5) * (4 + 3 / 2) - 5
To: 3 5 + 4 3 2 / + * 5 -

Or

From: 3 + 6 * 7 * 8 - 3
To: 3 6 7 * 8 * + 3 -

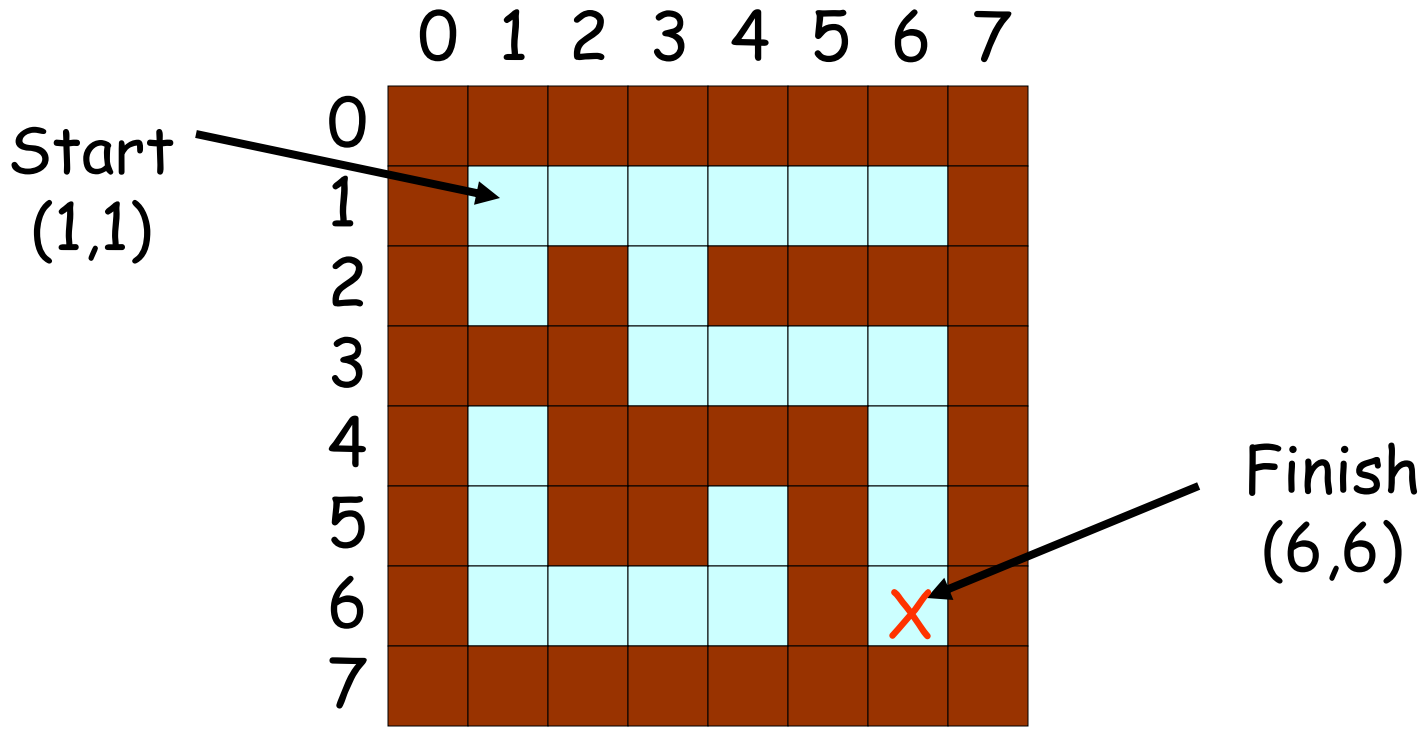
Infix to Postfix Conversion

Inputs: Infix string
Output: postfix string (initially empty)
Private data: a stack

1. Begin at left-most Infix token.
2. If it's a #, append it to end of postfix string followed by a space
3. If its a "(", push it onto the stack.
4. If it's an operator *and the stack is empty*:
   a. Push the operator on the stack.
5. If it's an operator and the stack is NOT empty:
   a. Pop all operators with greater or equal precedence off the stack and append them on the postfix string.
   b. Stop when you reach an operator with lower precedence or a (.
   c. Push the new operator on the stack.
6. If you encounter a ")", pop operators off the stack and append them onto the postfix string until you pop a matching "(".
7. Advance to next token and GOTO #2
8. When all infix tokens are gone, pop each operator and append it } to the postfix string.

Solving a Maze with a Stack!

We can also use a stack to determine if a maze is solvable:



Solving a Maze with a Stack!

Inputs: 10x10 Maze in a 2D array,
Starting point (sx,sy)
Ending point (ex,ey)
Output: TRUE if the maze can be solved, FALSE otherwise
Private data: a stack of *points*

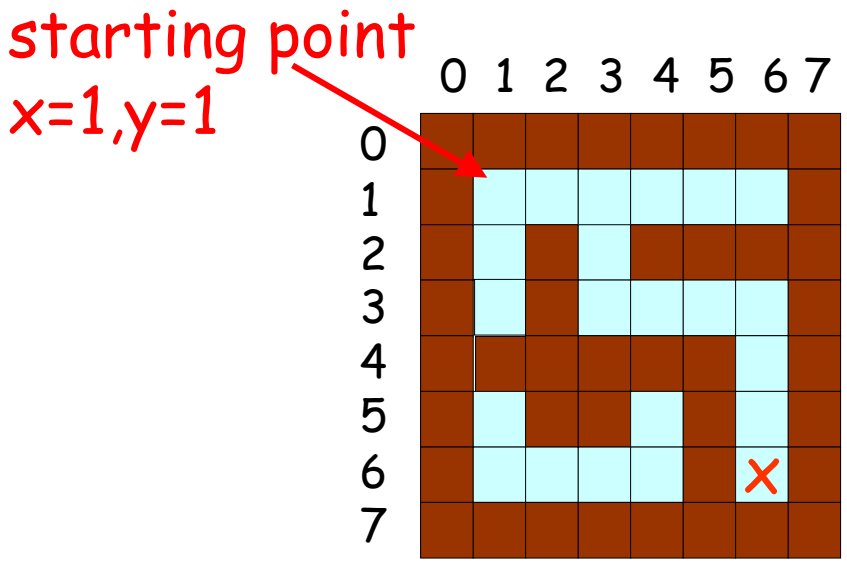
- Let's define a new Point class to help represent the pair of values (x,y) at the same time.
- We'll then create a stack of these Points.
- So each row of the stack will hold and x,y pair.

```
class Point
{
public:
  Point(int x, int y);
  int getx() const;
  int gety() const;
private:
  int m_x, m_y;
};
```

```
class Stack
{
public:
   Stack();     // c'tor
   void push(Point &p);
   Point pop();
   ...
private:
   ...
};
```

Solving a Maze with a Stack!

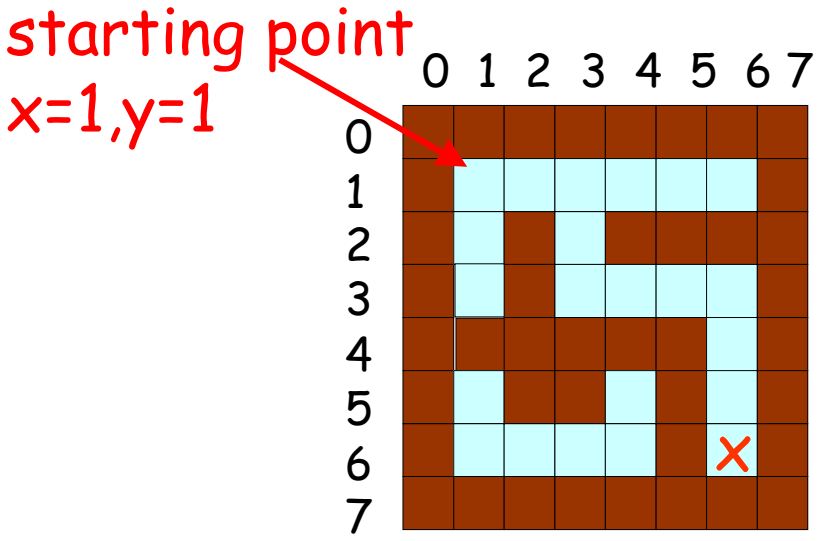
1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. While the stack is not empty:
   A. POP the top point off the stack into a variable.
   B. If we're at the endpoint, DONE! Otherwise…
   C. If slot to the WEST is open & is undiscovered
      Mark (curx-1,cury) as "discovered"
      PUSH (curx-1,cury) on stack.
   D. If slot to the EAST is open & is undiscovered
      Mark (curx+1,cury) as "discovered"
      PUSH (curx+1,cury) on stack.
   E. If slot to the NORTH is open & is undiscovered
      Mark (curx,cury-1) as "discovered"
      PUSH (curx,cury-1) on stack.
   F. If slot to the SOUTH is open & is undiscovered
      Mark (curx,cury+1) as "discovered"
      PUSH (curx,cury+1) on stack.
4. If the stack is empty and we haven't reached our goal position, then the maze is unsolvable.



- This is the algorithm for searching a maze using a stack.
- It follows a common pattern used by most stack (or queue) based algorithms:
  - Push the starting item (e.g., point 1,1) onto the stack
  - While the stack is not empty:
    - Extract the top item (e.g., x,y point) from the stack
    - Process the item (e.g., see if you've reached the goal position in the maze, etc.)
    - Identify related items to the item you just extracted (e.g., neighboring x,y points in the maze) that **haven't been processed yet**
    - Add them to the stack
  - If the stack is empty, you're done.

Solving a Maze with a Stack!

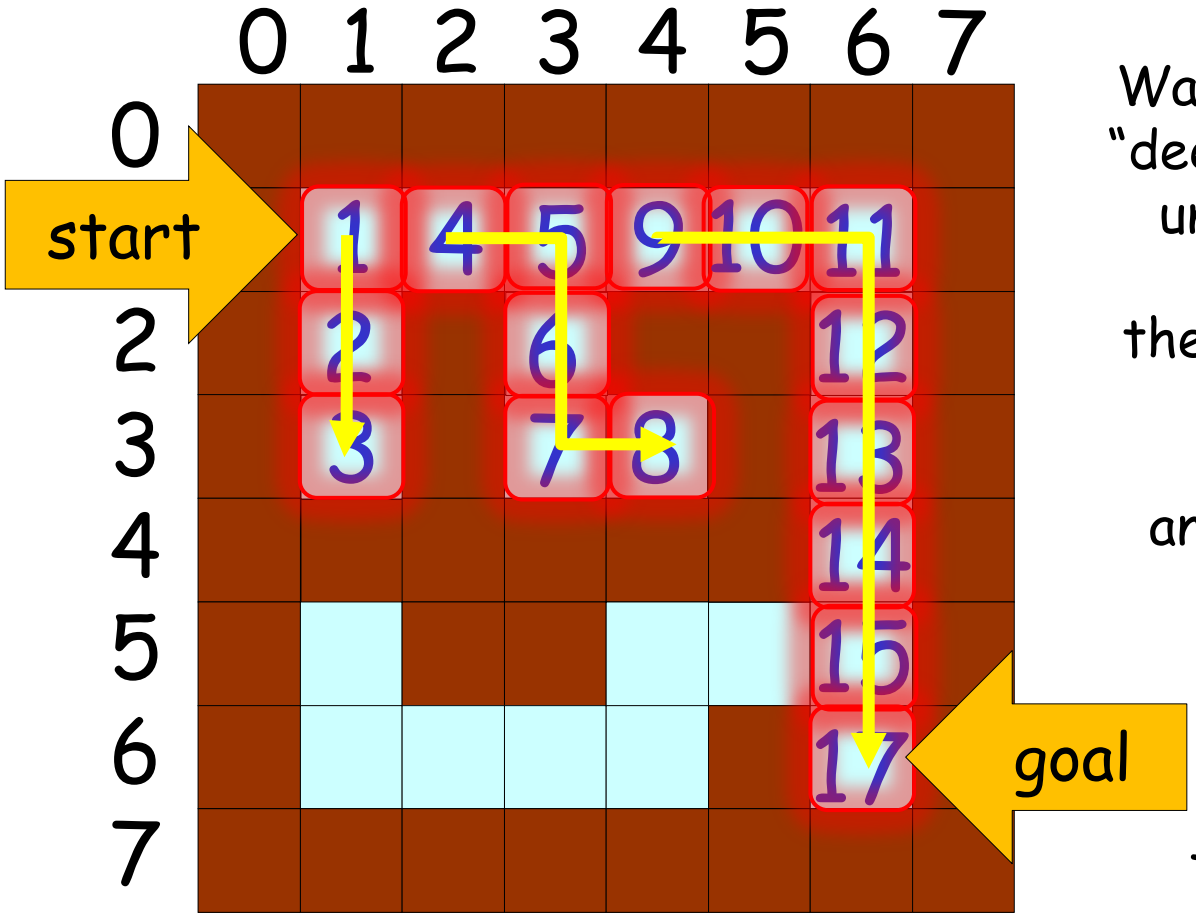
1. PUSH starting point onto the stack.
2. Mark the starting point as "discovered."
3. While the stack is not empty:
   A. POP the top point off the stack into a variable.
   B. If we're at the endpoint, DONE! Otherwise…
   C. If slot to the WEST is open & is undiscovered
      Mark (curx-1,cury) as "discovered"
      PUSH (curx-1,cury) on stack.
   D. If slot to the EAST is open & is undiscovered
      Mark (curx+1,cury) as "discovered"
      PUSH (curx+1,cury) on stack.
   E. If slot to the NORTH is open & is undiscovered
      Mark (curx,cury-1) as "discovered"
      PUSH (curx,cury-1) on stack.
   F. If slot to the SOUTH is open & is undiscovered
      Mark (curx,cury+1) as "discovered"
      PUSH (curx,cury+1) on stack.
4. If the stack is empty and we haven't reached our goal position, then the maze is unsolvable.



The stack-based maze-solving algorithm is called a "**depth-first search**" because the use of a stack causes it to explore "deep" down passages 'til it hits a dead end, then unravel back to the last junction where it again explores deep to another dead end, etc. **It's very similar to the recursive maze searching algorithm.**

Solving a Maze with a Stack

Depth-first Search Visualization



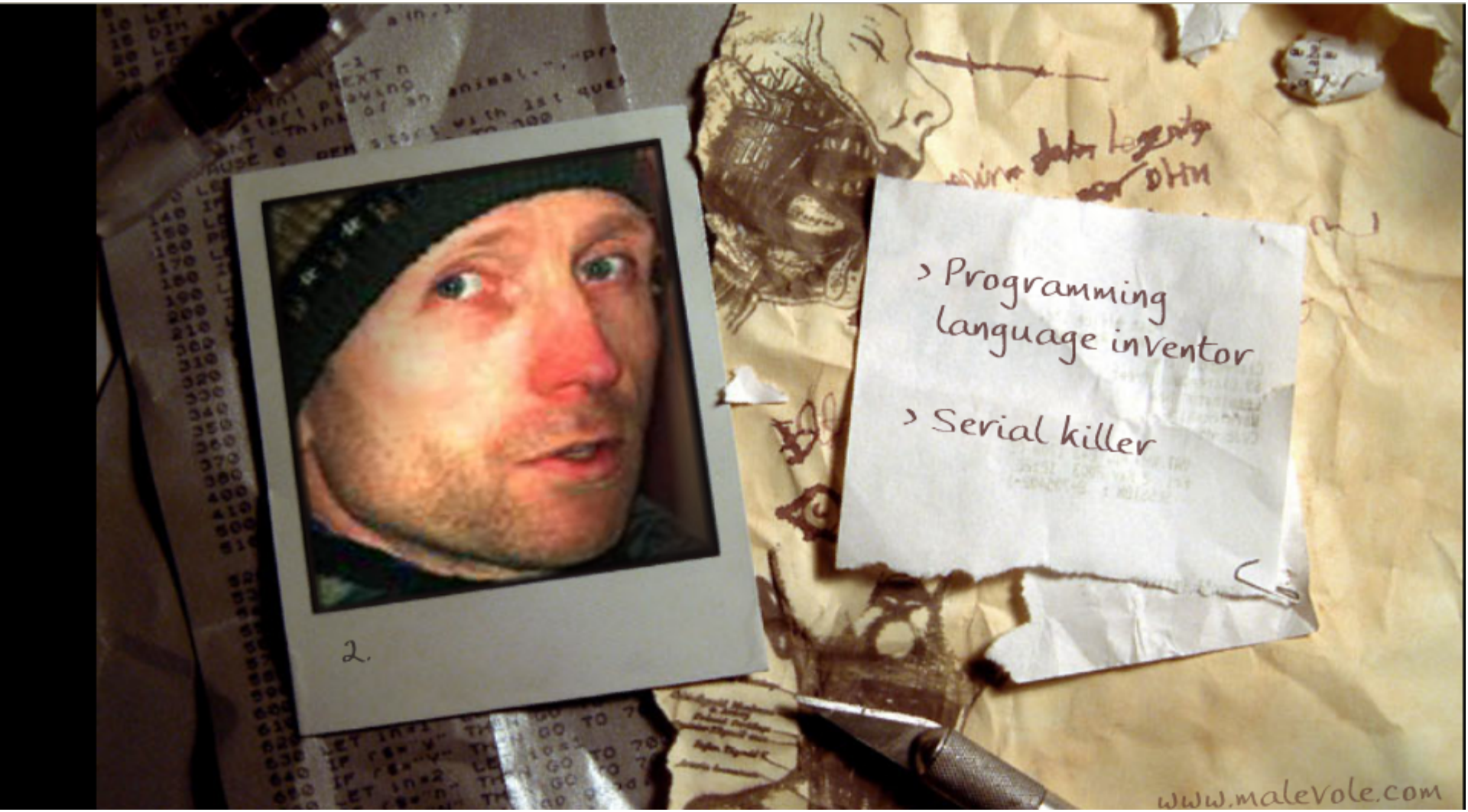
Watch how our search goes "deep" in the same direction until it hits a dead end…

then it "backtracks" to the last junction…

and goes deep in another direction.

This is why it's called a "depth-first" search.

Your favorite game!



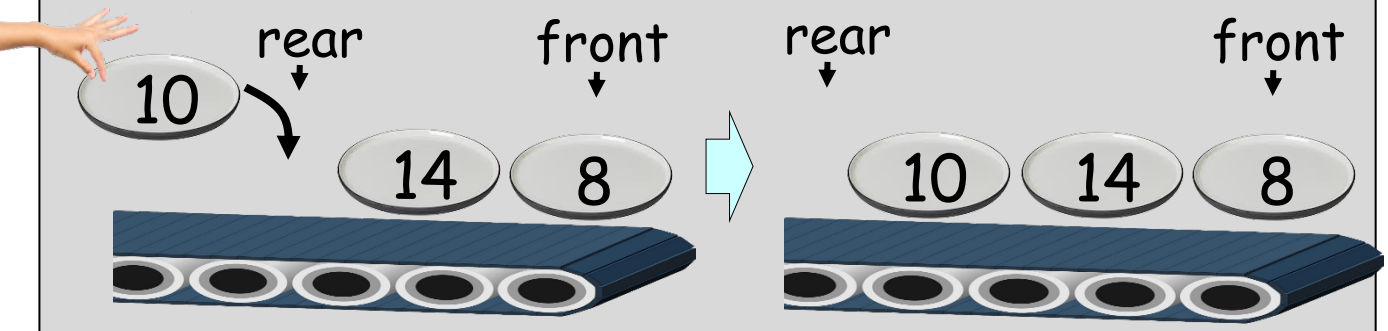

stack.top()
stack.push(“2nd”);
stack.push(“1st”);

Queues

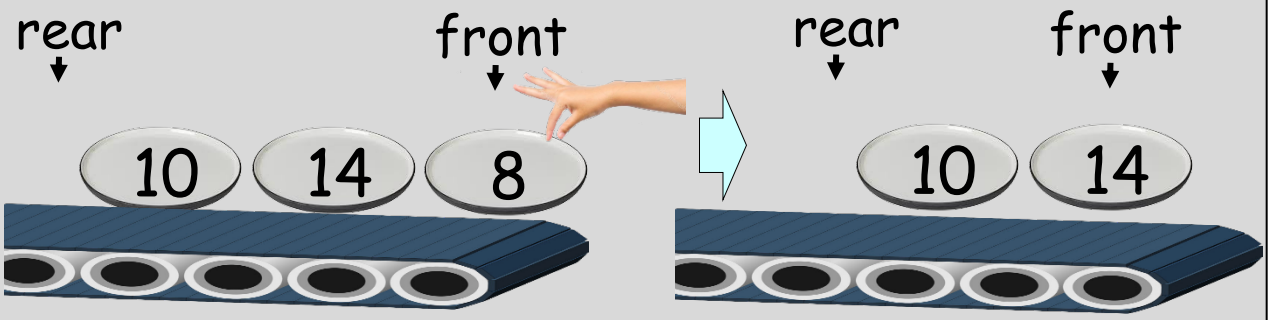
What's the big picture?

A queue is a super-useful* data structure that resembles a conveyer belt of dishes.

Dishes are added to the rear of the queue...



and dishes are extracted from its front...



The first-in/first-out property of queues is opposite of the last-in-first-out property of stacks.

Uses:

Use queues for vehicle navigation, implementing twitter feeds, queueing songs on Spotify, etc.

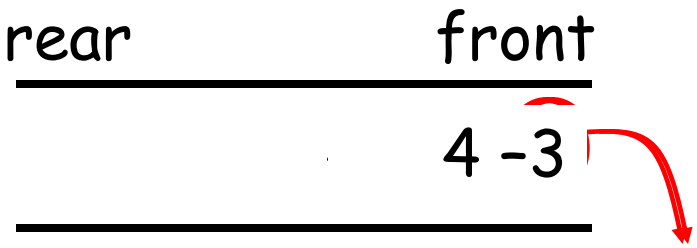
Another ADT: The Queue

The queue is another ADT that is just a like a line at the store or at the bank.

The first person in line is the first person out of line and served.

This is called a FIFO data structure:
FIRST IN, FIRST OUT.

Every queue has a *front* and a *rear*. You enqueue items at the *rear* and dequeue from the front.



What data structures could you use to implement a queue?

The Queue Interface

void enqueue(int a):
Inserts an item on the rear of the queue

int dequeue():
Removes and returns the top item from the front of the queue

bool isEmpty():
Determines if the queue is empty

int size():
Determines the # of items in the queue

int getFront():
Gives the value of the top item on the without removing it like dequeue

Like a Stack, we can have queues of any type of data! Queues of strings, Points, Nerds, ints, etc!

Common Uses for Queues

Often, data flows from the Internet faster than the computer can use it. We use a queue to hold the data until the browser is ready to display it...

Every time your computer receives a character, it enqueues it:

```
internetQueue.enqueue(c);
```

Every time your Internet browser is ready to get and display new data, it looks in the queue:

```
while (internetQueue.isEmpty() == false)
{
    char ch = internetQueue.dequeue();

    cout << ch;  // display web page...
}
```

Common Uses for Queues

You can also use queues to search through mazes!

If you use a queue instead of a stack in our searching algorithm, it will search the maze in a different order…

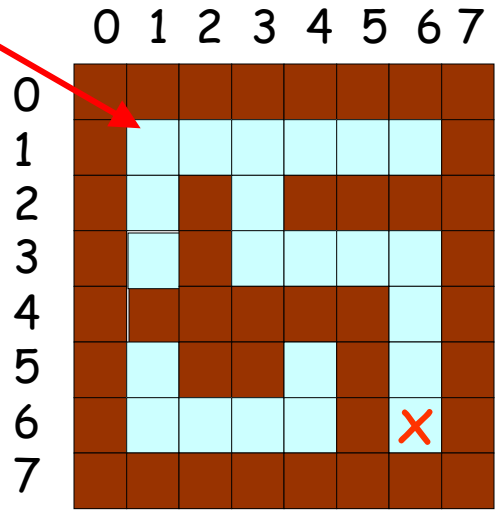
Instead of always exploring the last x,y location pushed on top of the stack first…

The new algorithm explores the oldest x,y location inserted into the queue first.

Solving a Maze with a Queue!

(AKA Breadth-first Search)

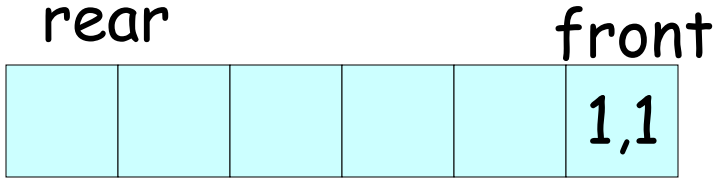
sx,sy = 1,1



1. Insert starting point onto the queue.
2. Mark the starting point as "discovered."
3. While the queue is not empty:
   A. Remove the front point from the queue.
   B. If we're at the endpoint, DONE! Otherwise…
   C. If slot to the WEST is open & is undiscovered
      Mark (curx-1,cury) as "discovered"
      INSERT (curx-1,cury) on queue.
   D. If slot to the EAST is open & is undiscovered
      Mark (curx+1,cury) as "discovered"
      INSERT (curx+1,cury) on queue.
   E. If slot to the NORTH is open & is undiscovered
      Mark (curx,cury-1) as "discovered"
      INSERT (curx,cury-1) on queue.
   F. If slot to the SOUTH is open & is undiscovered
      Mark (curx,cury+1) as "discovered"
      INSERT (curx,cury+1) on queue.
4. If the queue is empty and we haven't reached our goal position, then the maze is unsolvable.

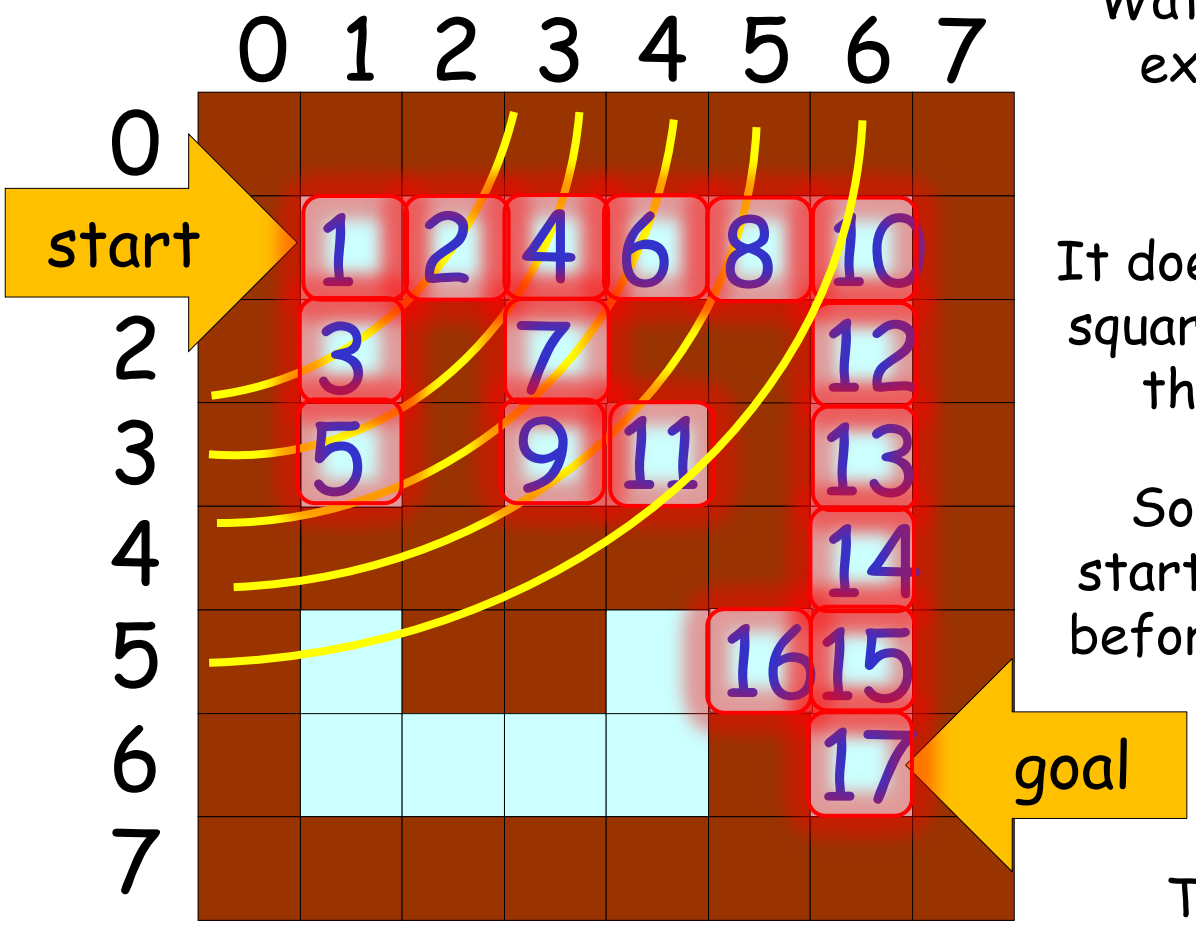
And so on…

curx,cury=



Solving a Maze with a Stack

Breadth-first Search Visualization



Watch how the squares we explore expand out like ripples in a pond…

It does this because each new square to explore is added to the END of the queue…

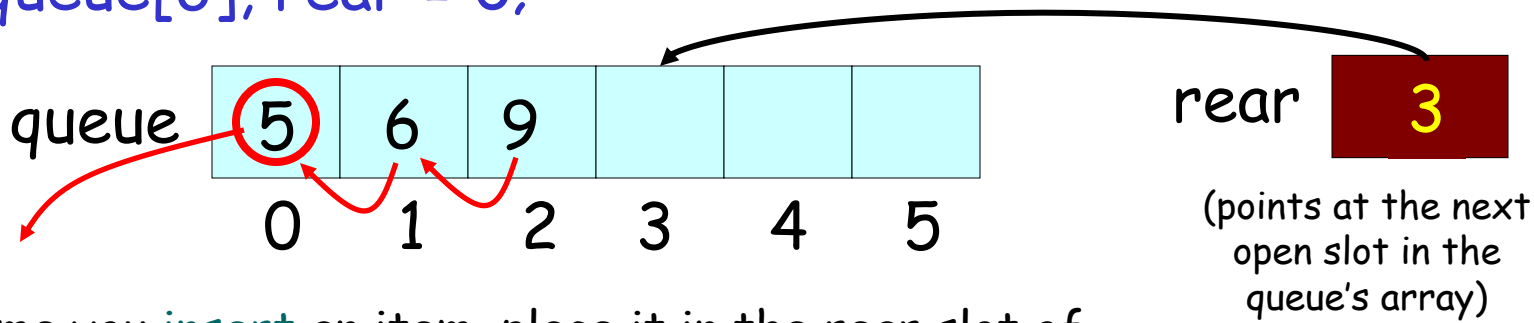
So squares closer to the starting square are explored before squares further away.

This is why it's called a "breadth-first" search.

Queue Implementations

We can use an array and an integer to represent a queue:

```
int queue[6], rear = 0;
```



- Every time you insert an item, place it in the rear slot of the array and increment the rear count
- Every time you dequeue an item, move all of the items forward in the array and decrement the rear count.

What's the problem with the array-based implementation?

If we have N items in the queue, what is the cost of:
(1) inserting a new item, (2) dequeuing an item

Right! If there are N items in our queue, to remove the first item requires us to shift the other N-1 items over. If N is large, that's SLOW!

Queue Implementations

We can also use a linked list to represent a queue:

- Every time you insert an item, add a new node to the end of the linked list.
- Every time you dequeue an item, take it from the head of the linked list and then delete the head node.

Of course, you'll want to make sure you have both head and tail pointers…

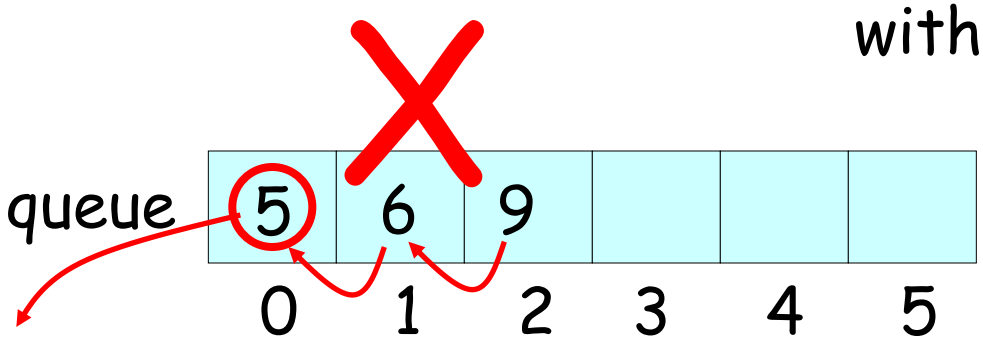
or your linked-list based queue will be really inefficient!

If we do this, we can extract the first item in the queue in just 1 step. No shifting of N items required.

The Circular Queue

The circular queue is a clever type of array-based queue.

Unlike our previous array-based queue, we never need to shift items with the circular queue!

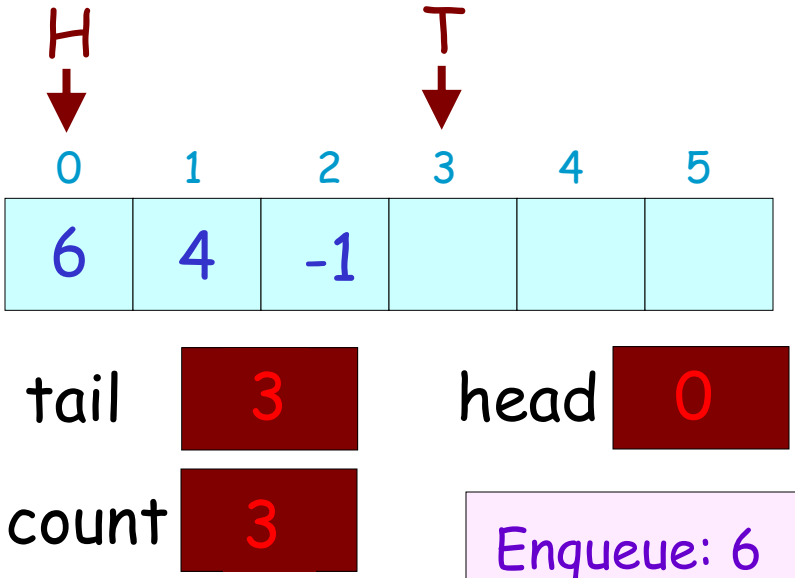


Let's see how it works!

The Circular Queue

Private data:

- arr: Array that holds the items in our queue
- head: Int that specifies the slot in the array that represents the current "front" of the queue.
- tail: Int that specifies the slot in the array that represents the rear of the queue (where the next new item will be added at the end).
- count: The count of the number of items in the queue at the current time. If it's zero, there are no items in the queue.



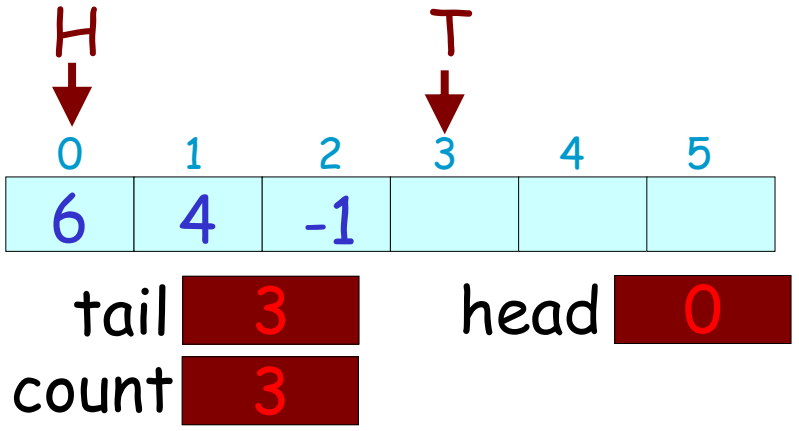
- To initialize your queue, set:
  count = head = tail = 0
- To insert a new item, place it in arr[tail] and then increment the tail & count values
- To dequeue the head item, fetch arr[head] and increment head and decrement count
- If the head or tail go past the end of the array, set it back to 0.

Enqueue: 6
Enqueue: 4
Enqueue: -1

The Circular Queue

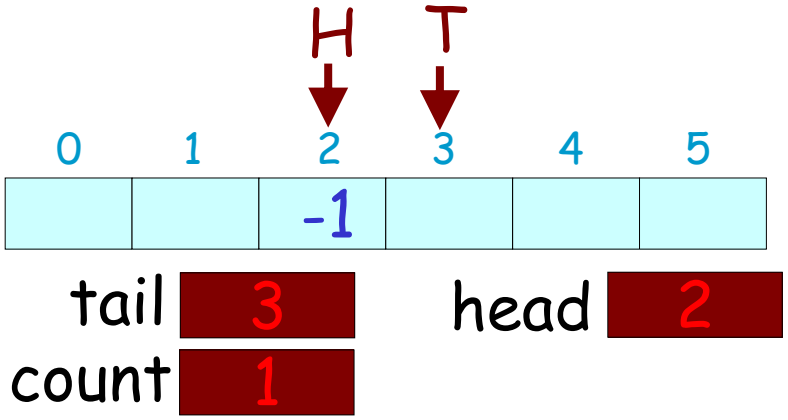
Here's the queue after adding 3 items: 6, 4, -1:

- Notice how our head points at slot #0. This is the first item in the queue and the next to be extracted when we "dequeue" an item.
- Notice how our tail points at slot #3. This is the empty slot where our next item will be added.
- The tail does not point at the last item in the queue, but rather at the next open slot AFTER the last item.



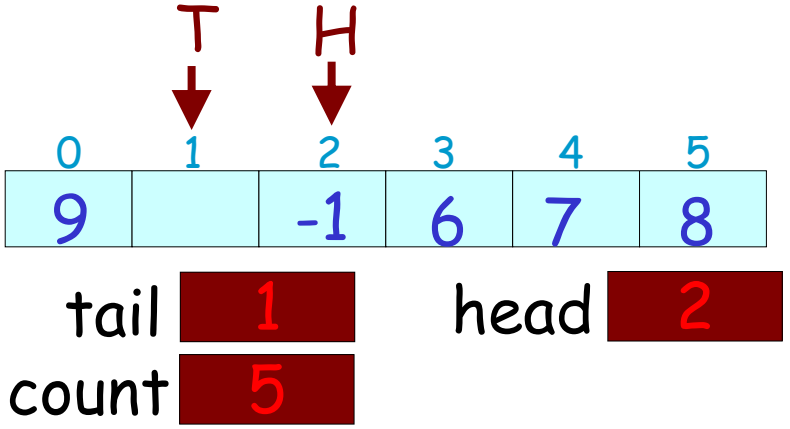
Here's the queue after extracting the first two items:

- Since we removed two items, our head shifts over by two spots. It now points at the -1 in slot #2, which is the new first item in the queue.
- Our tail position stays the same, since we'll still add the next item to slot #3.



Here's the queue after adding 4 items: 6, 7, 8, and 9:

- We added 6, 7 and 8 in slots 3, 4 and 5.
- Then we hit the end of the array, so we wrapped around our tail integer to slot 0. That's where we added 9.
- This "wrapping around" lets us re-use previously used slots.



A Queue in the STL!

The people who wrote the Standard Template Library also built a queue class for you:

```
#include <iostream>
#include <queue>

int main()
{
        std::queue<int>  iqueue;   // queue of ints

        iqueue.push(10);           // add item to rear
        iqueue.push(20);
        cout << iqueue.front();    // read the front item
        iqueue.pop();              // discard front item
        if (iqueue.empty() == false)
          cout << iqueue.size();
}
```

Class Challenge

Given a circular queue of 6 elements, show the queue's contents, and the Head and Tail pointers after the following operations are complete:

enqueue(5)
enqueue(10)
enqueue(12)
dequeue()
enqueue(7)
dequeue()
enqueue(9)
enqueue(12)
enqueue(13)
dequeue()