Lecture 1: Stacks

EECG142- Data Structures

Textbook:
Data Structures via C++: Objects by Evolution
by A. Michael Berman

First year - EECE Department

Spring 2025

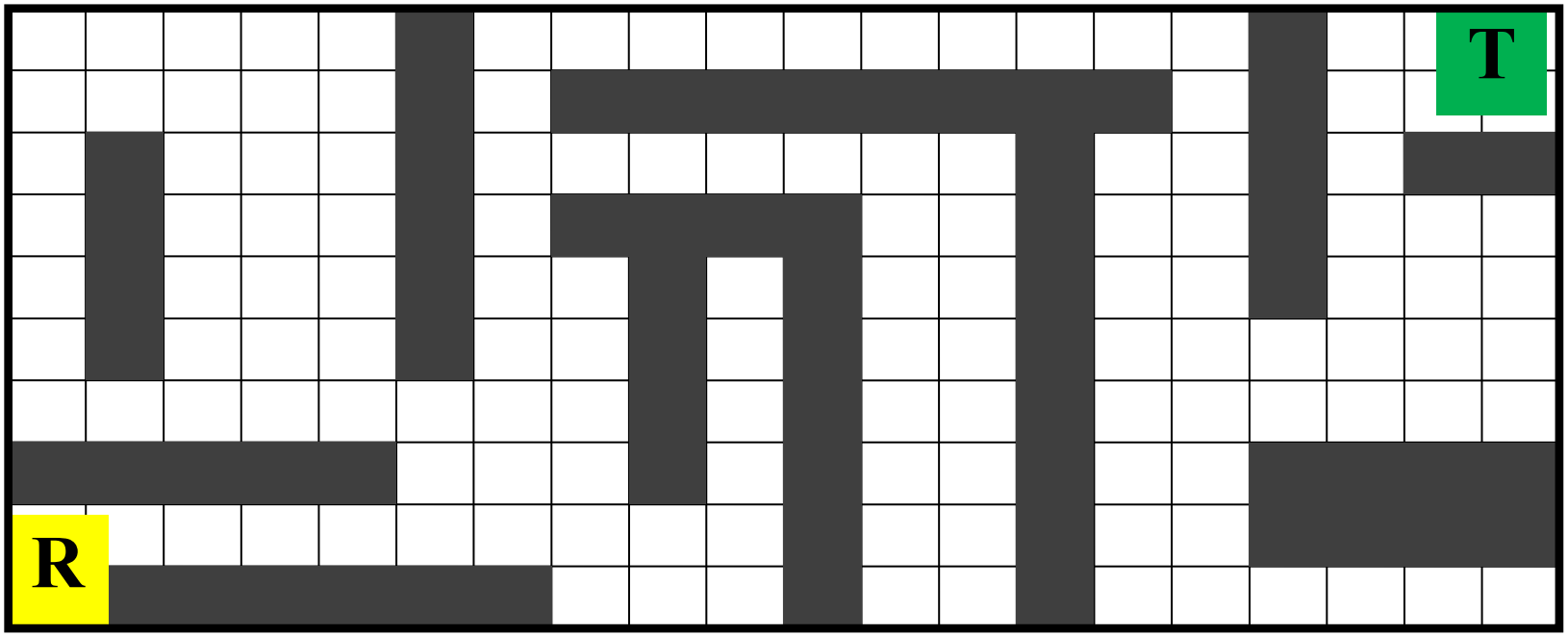
This lecture covers

1. Stack definition
2. Use-cases and Applications of stacks
3. Stack ADT creation using C++ Class templates
4. Stack implementation using Fixed-top and Floating-top arrays
5. Stack implementation using linked-lists
6. Comparison between array-based and linked-lists based implementations

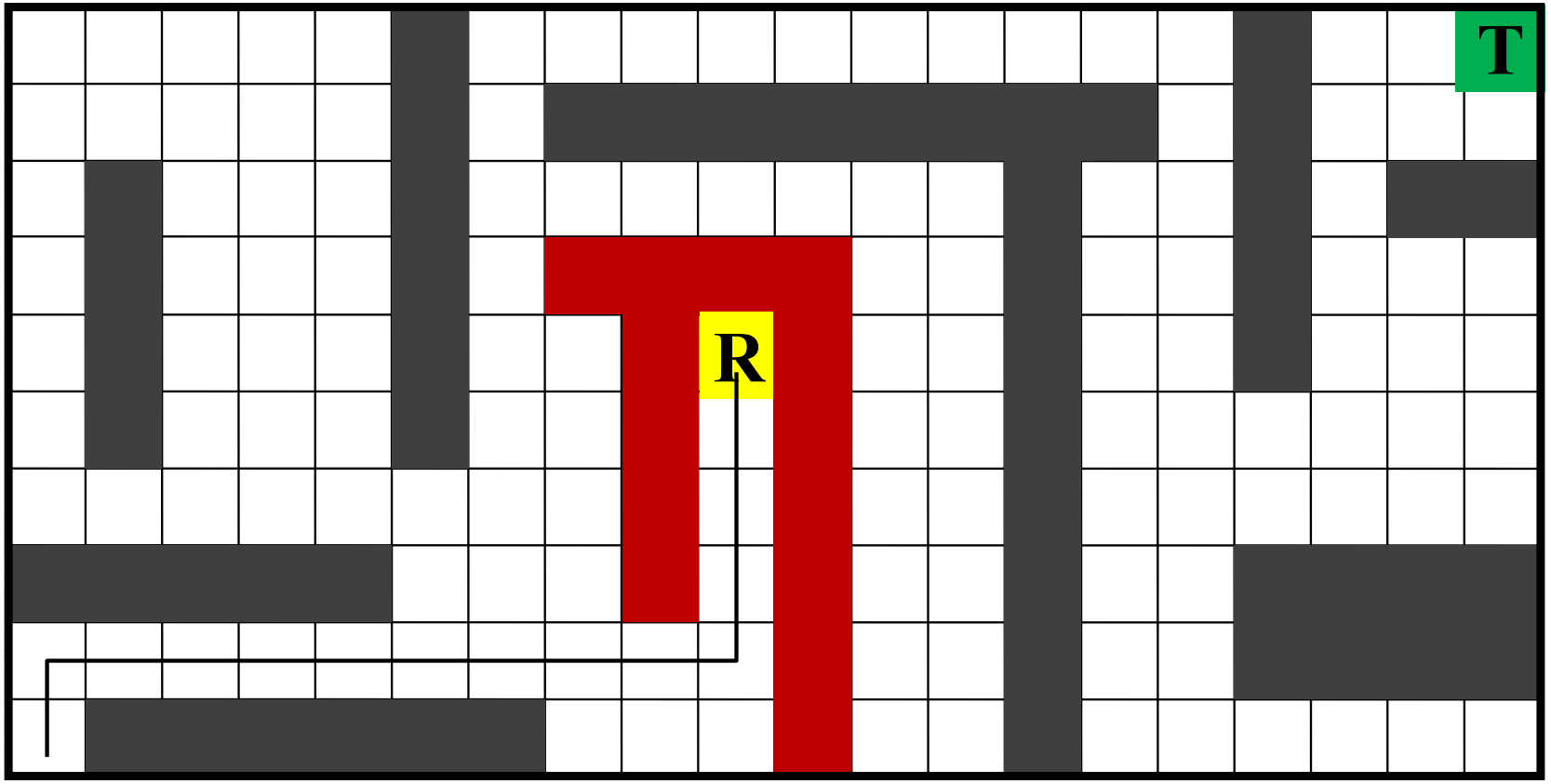
Robot Navigation Example

Suppose we have a robot that moves according to the following simple strategy:

Move in the direction of the target if possible. Else try any other direction.



What if the robot hits a dead end?

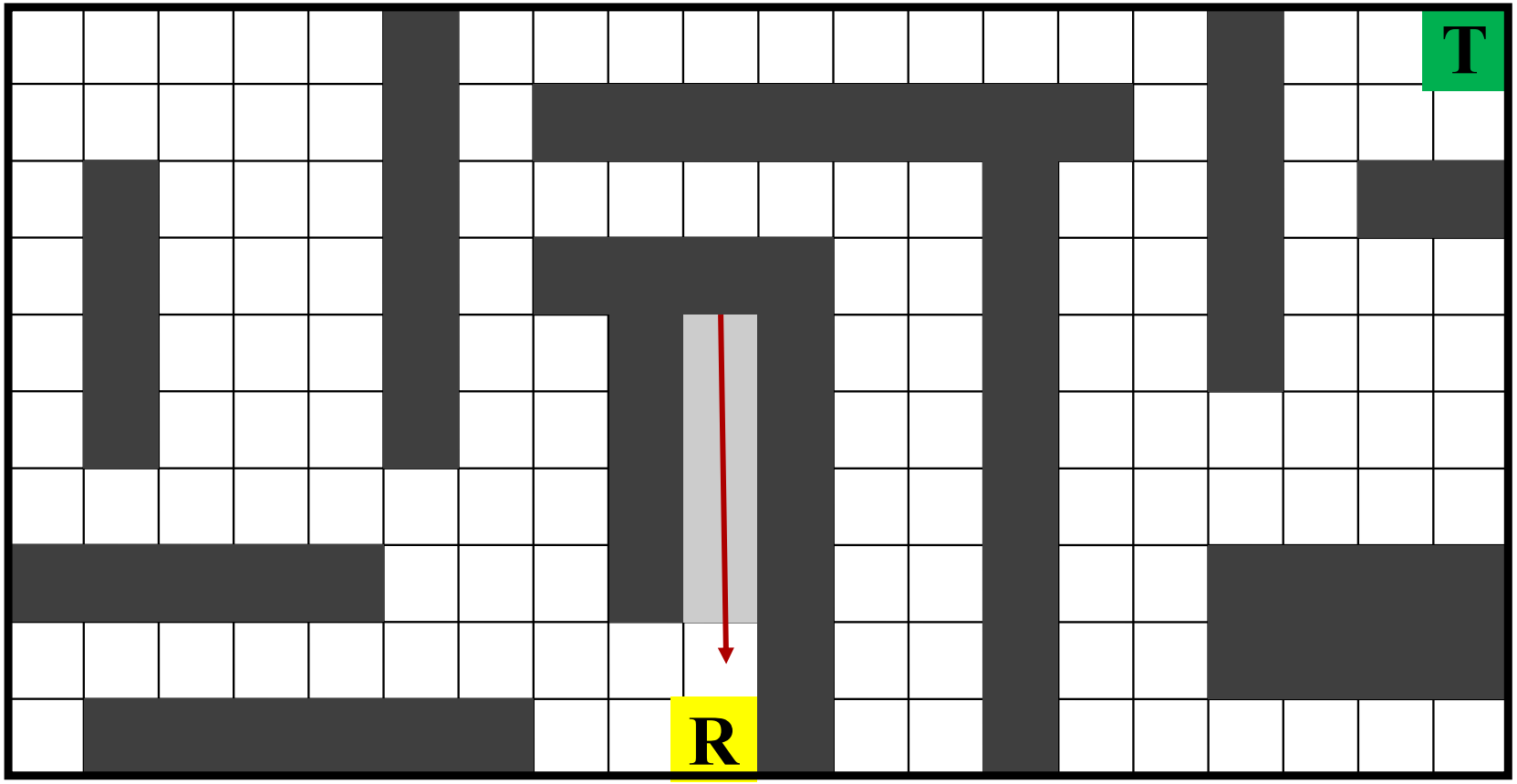


What if the robot hits a dead end?

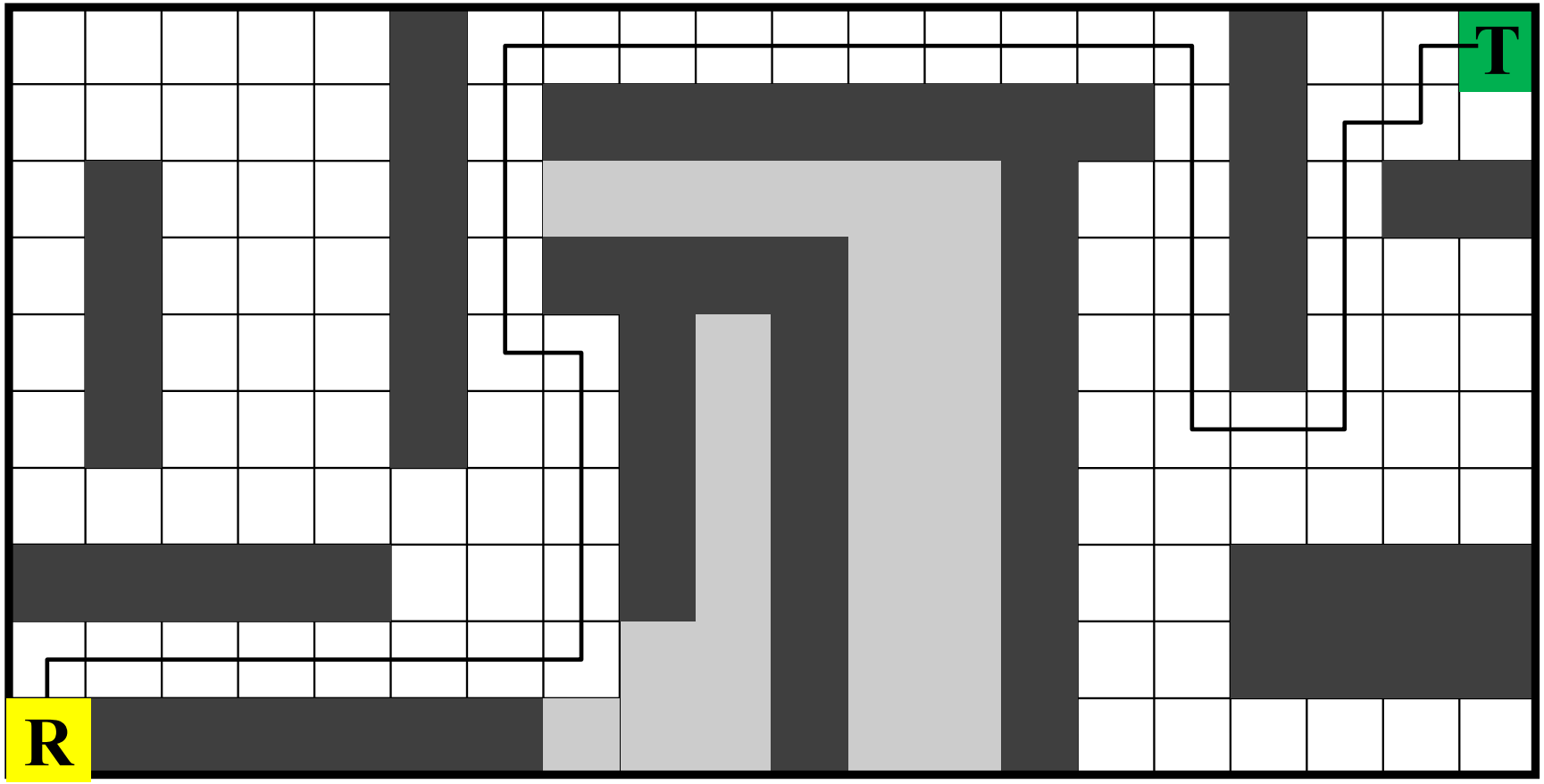
To get out of dead ends, the robot's moving strategy should be updated as follows:

- Mark the squares you visit while moving so as not to revisit them unless when you reach a dead end.
- If you reach a dead end, then backtrack (back up through your previous positions) until you finds an unexplored direction to try.
- In backtracking, the last visited step is the first you revisit.

Backtrack and try a new direction

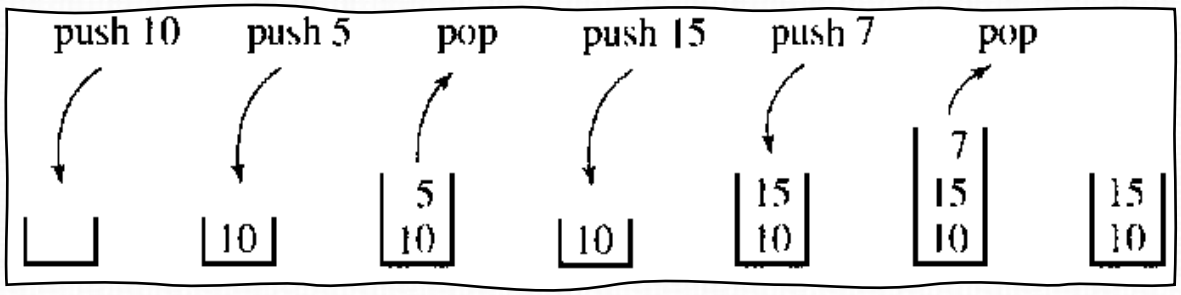


Robot's path to the target



Stack ADT

- A structure with “Last in First out” strategy is called a **stack**
- The last (**item**) you put (**push**), is the first to take out (**pop**)
- To handle the stack, we need to know where the top of the stack is stored and if the stack is empty (or full)



Matching Delimiter Example

examples of C++ statements that use delimiters properly:

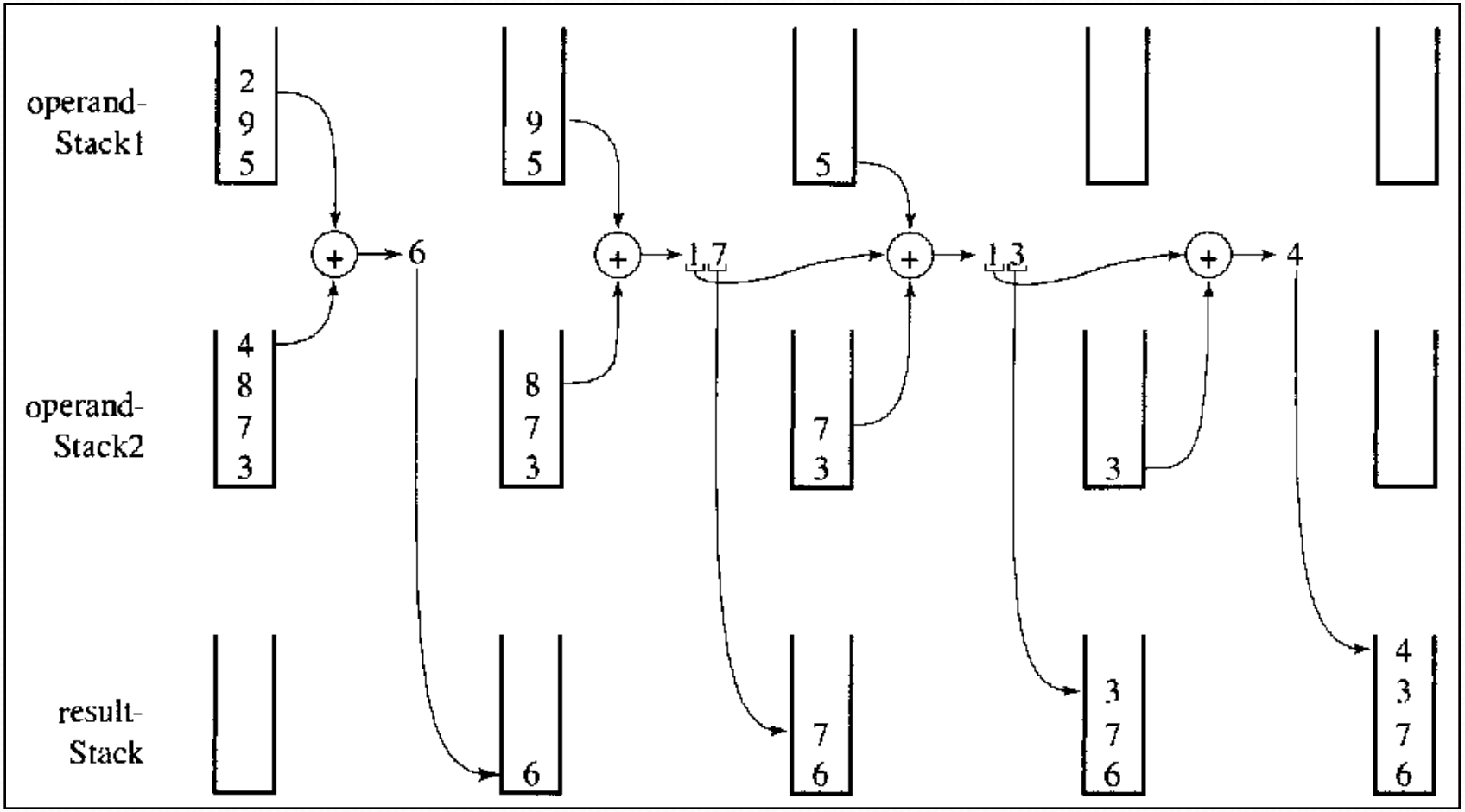
```
a = b + (c - d) * (e - f);
g[10] = h[i[9]] + (j + k) * l;
while (m < (n[8] + o)) { p = 7; /* initialize p */ r = 6; }
```

These examples are statements in which mismatching occurs:

```
a = b + (c - d) * (e - f));
g[10] = h[i[9]] + j + k) * l;
while (m < (n[8) + o]) { p = 7; /* initialize p */ r = 6; }
```

```
delimiterMatching(file)
  read character ch from file;
   while not end of file
       if ch is '(', '[', or '{'
             push(ch);
       else if ch is ')'
             read the next character;
             if this character is '*'
                   push(ch);
             else ch = the character read in;
                   continue; // go to the beginning of the loop;
       else if ch is ')', ']', or '}'
             if ch and popped off delimiter do not match
                 failure;
       else if ch is '*'
             read the next character;
             if this character is '/' and popped off delimiter is not '/'
                   failure;
             else ch = the character read in;
                   push back the popped off delimiter;
                   continue;
    // else ignore other characters;
        read next character ch from file;
   if stack is empty
        success;
   else failure;
```

Adding Large Numbers Example



Stack Operations

Push: Adds a new item to the top of the stack.

If you try to push an item to a full stack, it goes into an overflow state.

Pop: Removes an item from the top of the stack.

If you try to pop an item from an empty stack, it goes into underflow state.

Clear: Removes all stack elements.

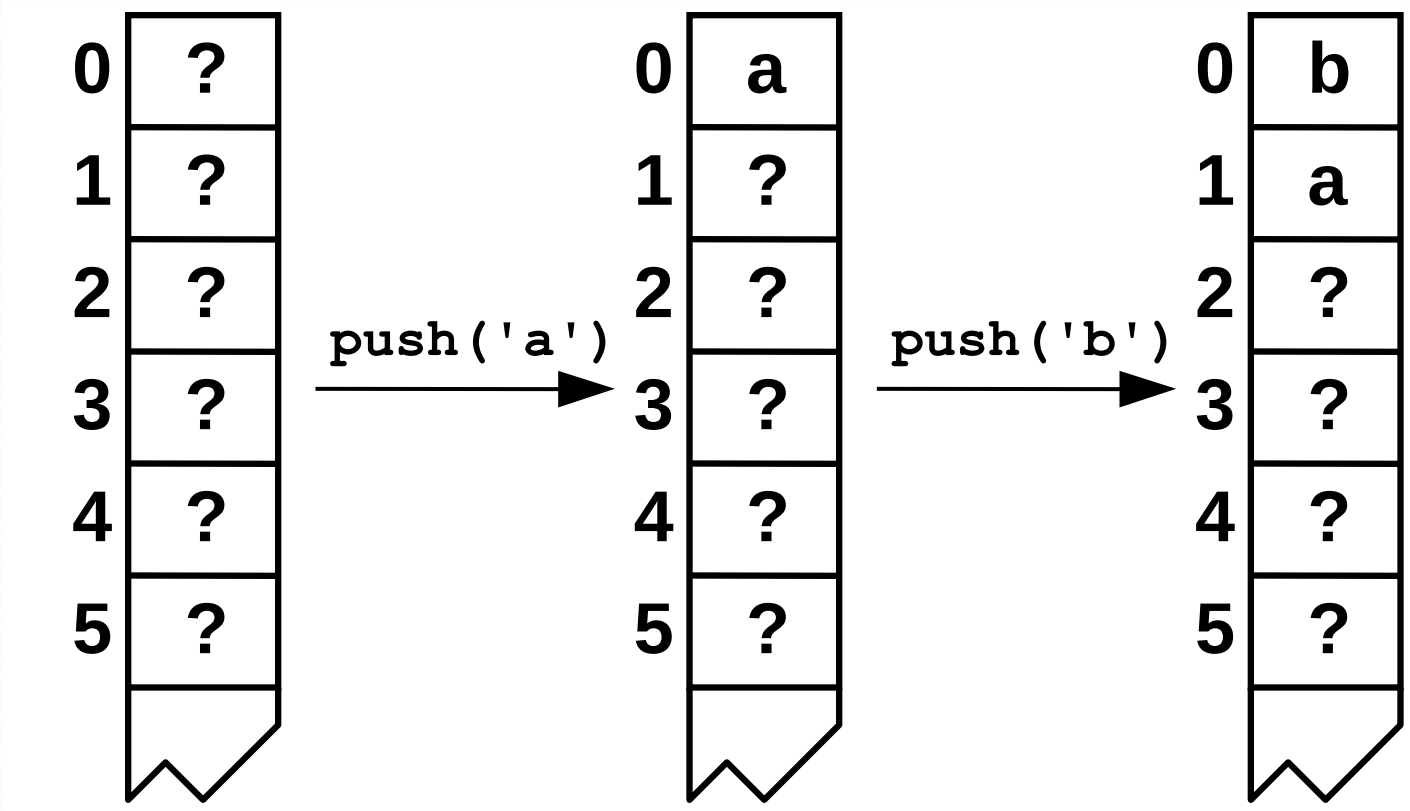
isEmpty: Checks if the stack is empty.

Top: Returns the value of the first element (without removing it).

Implement Stacks Using Arrays

The items of the stack are of the same data type, so it can be implemented using an array.

What is wrong with the following Fixed-Top implementation?



Implement Stacks Using Arrays

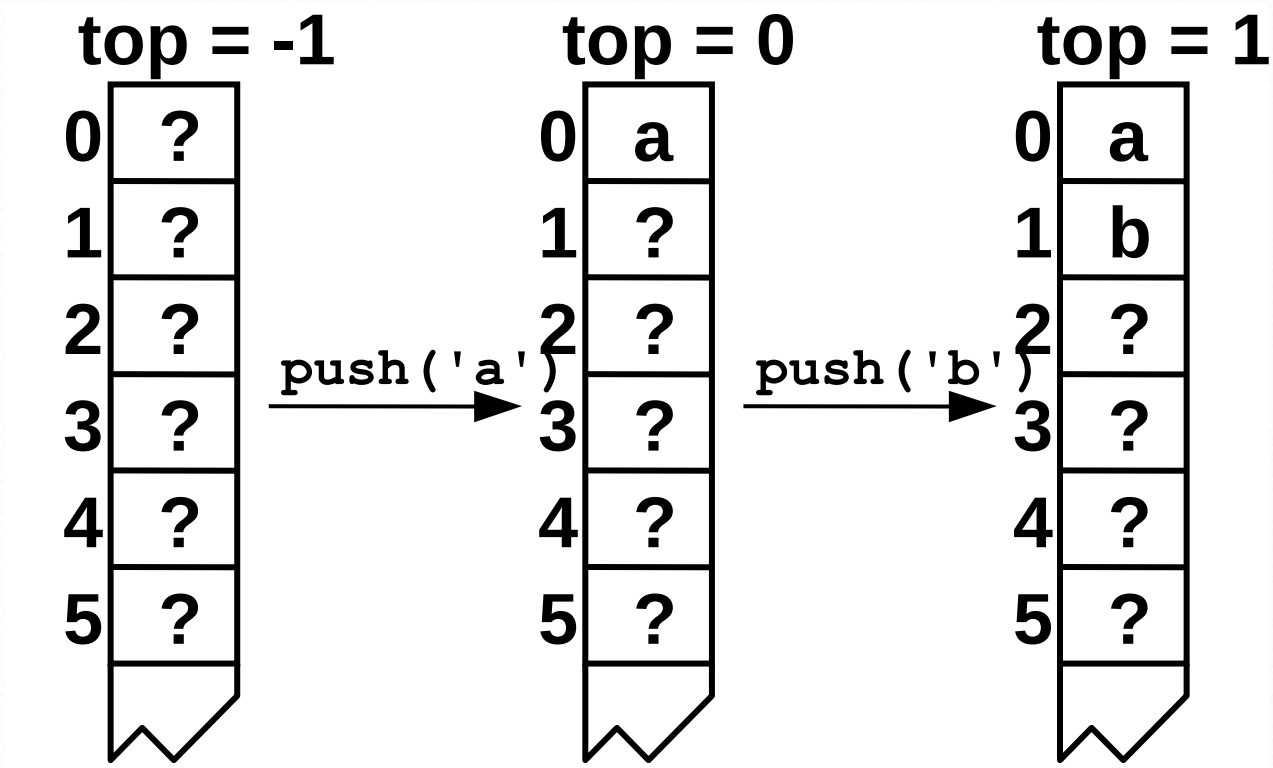
Fixed-Top array implementation have the following unnecessary overheads:

- All items need to be shifted down every time a new element is pushed.
- All items need to be shifted up every time a item is popped.

Implement Stacks Using Arrays

Floating-Top implementation fixes the data and shifts the top instead of fixing the top and shifting the data.

You need to know where the top is to do push and pop operations correctly.



Stack Header File

```
const int maxStackSize = 1000;
typedef char StackElementType;

class Stack {
public:
  Stack();
  void push(StackElementType item);
  StackElementType pop();
  StackElementType top();
  bool isEmpty();
private:
  StackElementType stackArray[maxStackSize];
  int topIndex;
};
```

Stack Implementation File

```
#include “stack_header.h“
Stack::Stack()
{
 topIndex = -1;
}
void Stack::push(StackElementType item)
{// ensure array bounds not exceeded
   assert(topIndex < maxStackSize-1);
   topIndex++;
   stackArray[topIndex] = item;
}
```

Stack Templates

- A stack template have a generic definition that allows us to later declare a stack of any needed data type.
- ‘template <class YourType>’ becomes part of the class definition and must also become the leading part of every member function.

Stack Template Header File

```
const int maxStackSize = 1000;
template < class StackElementType >
class Stack {
public:
  Stack();
  void push(StackElementType item);
  StackElementType pop();
  StackElementType top();
  bool isEmpty();
  bool isFull();
private:
  StackElementType stackArray[maxStackSize];
  int topIndex;
};
```

Stack Template Implementation

```
#include “stack_header.h”
template < class StackElementType >
Stack < StackElementType >::Stack()
{ topIndex = -1;
}

template < class StackElementType >
void  Stack < StackElementType >::push(StackElementType item)
{ assert(topIndex < maxStackSize-1);
   topIndex++;
   stackArray[topIndex] = item;
}
```

Stack Template Implementation

```
template < class StackElementType >
StackElementType Stack < StackElementType >::pop()
{ assert(topIndex >= 0);
  int returnIndex=topIndex;
  topIndex--;
  return stackArray[returnIndex];
}

template < class StackElementType >
StackElementType Stack < StackElementType >::top()
{ assert(topIndex >= 0);
  return stackArray[topIndex];
}
```

Stack Template Implementation

```
template < class StackElementType >
bool Stack < StackElementType >::isEmpty()
{
  return bool(topIndex == -1);
}

template < class StackElementType >
bool Stack < StackElementType >::isFull()
{
  return topIndex == maxStackSize - 1;
}
```

Main Program Example

The main program instantiates an object doubleStack declared as class ***Stack< double >***

The compiler associates type double with the <StackElementType> in the class template to produce the source code for a Stack class of type double.

```
#include “stack.h" // Stack class template definition
#include “stack.cpp" // Stack class template implementation
int main ( ) {
 Stack< double > doubleStack; // create a stack of doubles
double doubleValue = 1 . 1 ;
cout << "Pushing element s onto double Stack \n" ;
while ( !doubleStack.isFull () )   // push elements until stack is full
            {doubleStack.push (doubleValue);
            cout << doubleValue << ' ' ;
            doubleValue += 1 . 1 ;} // end while
cout << “\n Stack is full . Cannot push " << doubleValue ;
cout << "\n \n Popping elements from doubleStack \n" ;
while (!doubleStack.isEmpty () )    // pop elements from doubleStack
        { doubleValue = doubleStack . pop ();
        cout << doubleValue << ' ' ;}
cout << "\n Stack is empty . Cannot pop\n" ;
}
```

Linked-list implementation

- Array-based stacks are inefficient due to their static sizes
- Linked-List based stacks do not waste memory and do not run out of spaces (unless the heap is full)
- They need extra memory as they store the items and the pointer that point to their locations.

Stack Header File

```
template < class StackElementType >
class Stack {
public:
  Stack();
  void push(StackElementType e);
  StackElementType pop();
  StackElementType top();
  bool isEmpty();
private:
struct Node
typedef Node * Link;
struct Node {
    StackElementType data;
    Link next;
  };
Link head;
};
```

Stack Implementation

```
template < class StackElementType>
Stack < StackElementType >::Stack()
{
  head = NULL;
}
template < class StackElementType >
Void Stack < StackElementType >::push(StackElementType e)
{
  Link addedNode = new Node;
  assert(addedNode!=NULL);
  addedNode->data = e;
  addedNode->next = head;
  head = addedNode;
}
```

Stack Implementation

```
template <class StackElementType>
bool Stack<StackElementType>::isEmpty()
{    return head == null;}
template <class StackElementType>StackElementType
Stack<StackElementType>::pop()
{
if (isEmpty()) { throw std::underflow_error("Stack is empty");  }
StackElementType topData = head->data;  // Store top element
Link temp = head;
head = head->next;  // Move head to the next node
delete temp;  // Delete the old top node
return topData;  // Return the popped element's data}
```

Comparing Stack Implementations

- Linked-list based implementation does not waste space; however, it needs extra memory to store link pointers
- The implementation choice depends on the size of the stored data
- For example, consider storing 100 integer items in a stack

Array Implementation: 2Byte/item*100item + 4bytes(size and top index)= **204Bytes** (always reserved)

Linked-list Implementation: 6 bytes/ item (2 for value and four for next) + 4bytes for head pointer. It reaches 204Bytes after 33 elements only.

What would be the case if the stored data has a bigger size, e.g. a string of length 100Bytes?