1. What is the time complexity of push and pop operations on a stack implemented using an array?
   1. **O(1)**
   2. O(n)
   3. O(log n)
   4. O(n^2)
2. How can you reverse a string using a stack in C?
   1. **Iteratively pushing characters onto the stack and then popping them**
   2. Recursively pushing characters onto the stack and then popping them
   3. Converting the string into an array and reversing the array
   4. None of the above
3. Which of the following is an advantage of using a stack to reverse a string?
   1. It uses less memory compared to other methods
   2. It is faster compared to other methods
   3. It is easier to implement compared to other methods
   4. **All of the above**
4. Which data structure is suitable for checking balanced brackets in an expression?
   1. **Stack**
   2. Queue
   3. Linked list
   4. Binary tree
5. How can a stack be used to check balanced brackets in an expression?
   1. **By pushing opening brackets onto the stack and popping them when a closing bracket is encountered**
   2. By pushing closing brackets onto the stack and popping them when an opening bracket is encountered
   3. By comparing opening and closing brackets using a counter variable
   4. None of the above
6. Which of the following expressions has balanced brackets?
   1. **"{[()]}"**
   2. "{{[()]}"
   3. "{[()]}}"
   4. "{([])}"
7. What is the time complexity of checking balanced brackets using a stack?
   1. O(1)
   2. **O(n)**
   3. O(log n)
   4. O(n^2)
8. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char str[] = "hello";

reverseString(str);

printf("%s", str);

return 0;

}

void reverseString(char\* str) {

// Code to reverse the string using a stack

}

* 1. olleh
  2. hello
  3. **Compilation error**
  4. Runtime error

1. Which of the following is a valid implementation of the reverseString() function using a stack?
   1. **Option A:**

**void reverseString(char\* str) {**

**// Code to reverse the string using a stack**

**}**

b) Option B:

char\* reverseString(char\* str) {

// Code to reverse the string using a stack

return str;

}

c) Option C:

void reverseString(char\* str) {

// Code to reverse the string using a stack

strcpy(str, reversedStr);

}

d) Option D:

void reverseString(char\* str) {

// Code to reverse the string using a stack

return reversedStr;

}

1. Which of the following is a disadvantage of using a stack to reverse a string?
   1. **It requires additional memory to store the stack**
   2. It is slower compared to other methods
   3. It can only reverse strings of limited length
   4. None of the above
2. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char str[] = "abcd";

reverseString(str);

printf("%s", str);

return 0;

}

void reverseString(char\* str) {

// Code to reverse the string using a stack

}

* 1. **dcba**
  2. abcd
  3. Compilation error
  4. Runtime error

1. Which of the following expressions has unbalanced brackets?
   1. "{[()]}"
   2. **"{{[()]}"**
   3. "{[()]}}"
   4. "{([])}"
2. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "{[()]}";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

* 1. **Balanced**
  2. Not Balanced
  3. Compilation error
  4. Runtime error

1. Which of the following is a valid implementation of the isBalanced() function to check balanced brackets using a stack?
   1. Option A:

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

b) Option B:

void isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

**c) Option C:**

**int isBalanced(char\* exp) {**

**// Code to check balanced brackets using a stack**

**return 1;**

**}**

d) Option D:

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

return 0;

}

1. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "{{[()]}}";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

* 1. **Balanced**
  2. Not Balanced
  3. Compilation error
  4. Runtime error

1. How can you handle the case when a closing bracket is encountered and the stack is empty while checking balanced brackets?
   1. Ignore the closing bracket
   2. **Report the expression as unbalanced**
   3. Push the closing bracket onto the stack
   4. None of the above
2. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "{[()]}}";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

* 1. Balanced
  2. **Not Balanced**
  3. Compilation error
  4. Runtime error

1. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "{[()]}}";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

* 1. Balanced
  2. **Not Balanced**
  3. Compilation error
  4. Runtime error

1. Which data structure is suitable for reversing a string without using a stack?
   1. Array
   2. **Linked list**
   3. Queue
   4. None of the above
2. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char str[] = "abcd";

reverseString(str);

printf("%s", str);

return 0;

}

void reverseString(char\* str) {

// Code to reverse the string without using a stack

}

* 1. **dcba**
  2. abcd
  3. Compilation error
  4. Runtime error

1. How can you reverse a string without using a stack in C?
   1. By iterating through the string from the last character to the first character
   2. **By using a temporary variable to swap characters in the string**
   3. By using the strrev() function from the string.h library
   4. None of the above
2. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main() {

char str[] = "abcd";

reverseString(str);

printf("%s", str);

return 0;

}

void reverseString(char\* str) {

int len = strlen(str);

for (int i = 0; i < len / 2; i++) {

char temp = str[i];

str[i] = str[len - i];

str[len - i] = temp;

}

}

* 1. dcba
  2. abcd
  3. **Compilation error**
  4. Runtime error

1. Which of the following expressions has unbalanced brackets?
   1. "{([])}"
   2. **"([]))"**
   3. "{(})"
   4. "{}()"
2. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "([]))";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

* 1. Balanced
  2. **Not Balanced**
  3. Compilation error
  4. Runtime error

1. How can you handle the case when an opening bracket is encountered and the stack is empty while checking balanced brackets?
   1. Ignore the opening bracket
   2. Report the expression as unbalanced
   3. **Push the opening bracket onto the stack**
   4. None of the above
2. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "([]))";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

// Code to check balanced brackets using a stack

}

* 1. Balanced
  2. **Not Balanced**
  3. Compilation error
  4. Runtime error

1. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char str[] = "hello";

reverseString(str);

printf("%s", str);

return 0;

}

void reverseString(char\* str) {

if (str == NULL)

return;

int len = 0;

while (str[len] != '\0')

len++;

for (int i = 0; i < len / 2; i++) {

char temp = str[i];

str[i] = str[len - i - 1];

str[len - i - 1] = temp;

}

}

* 1. **olleh**
  2. hello
  3. Compilation error
  4. Runtime error

1. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "{{[()]}}";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

if (exp == NULL)

return 0;

int len = 0;

while (exp[len] != '\0')

len++;

char stack[len];

int top = -1;

for (int i = 0; i < len; i++) {

if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')

stack[++top] = exp[i];

else if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}') {

if (top == -1)

return 0;

char opening = stack[top--];

if ((exp[i] == ')' && opening != '(') || (exp[i] == ']' && opening != '[') || (exp[i] == '}' && opening != '{'))

return 0;

}

}

if (top == -1)

return 1;

else

return 0;

}

* 1. **Balanced**
  2. Not Balanced
  3. Compilation error
  4. Runtime error

1. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "{{[()]}}";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

if (exp == NULL)

return 0;

int len = 0;

while (exp[len] != '\0')

len++;

char stack[len];

int top = -1;

for (int i = 0; i < len; i++) {

if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')

stack[++top] = exp[i];

else if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}') {

if (top == -1)

return 0;

char opening = stack[top--];

if ((exp[i] == ')' && opening != '(') || (exp[i] == ']' && opening != '[') || (exp[i] == '}' && opening != '{'))

return 0;

}

}

if (top == -1)

return 1;

else

return 0;

}

* 1. **Balanced**
  2. Not Balanced
  3. Compilation error
  4. Runtime error

1. What is the output of the following code?

#include <stdio.h>

#include <stdlib.h>

int main() {

char exp[] = "{{[()]}}";

if (isBalanced(exp))

printf("Balanced");

else

printf("Not Balanced");

return 0;

}

int isBalanced(char\* exp) {

if (exp == NULL)

return 0;

int len = 0;

while (exp[len] != '\0')

len++;

char stack[len];

int top = -1;

for (int i = 0; i < len; i++) {

if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')

stack[++top] = exp[i];

else if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}') {

if (top == -1)

return 0;

char opening = stack[top--];

if ((exp[i] == ')' && opening != '(') || (exp[i] == ']' && opening != '[') || (exp[i] == '}' && opening != '{'))

return 0;

}

}

if (top == -1)

return 1;

else

return 0;

}

* 1. **Balanced**
  2. Not Balanced
  3. Compilation error
  4. Runtime error