Stack, Queue, and Heap Program with Explanations

September 23, 2024

1 Introduction

This document provides the full C code for implementing a stack, queue, and heap, along with detailed explanations for each part of the code. The program switches between these functionalities based on the user's input and performs various operations such as pushing, popping, enqueueing, dequeueing, heapify, extracting the minimum, and more.

2 Code with Explanations

We will go through each part of the code step by step, explaining what each function does.

2.1 Header Inclusions and Global Variables

We start by including necessary headers and defining global variables.

```
#include<stdio.h>       // Standard input/output library
#include<stdlib.h>      // Standard library for exit function

#define M 1000          // Maximum size for stack, queue, and heap
int s[M], t=-1;         // Stack array 's' and top index 't'
int q[M], f=0, r=0, c=0; // Queue array 'q', front 'f', rear 'r', and count 'c'
int h[M], hs=0;         // Heap array 'h' and heap size 'hs'
```

- We define the maximum size for all data structures as 1000.
- `s[M]`: An array used to implement the stack.
- `t`: A variable to keep track of the top of the stack.
- `q[M]`: An array used to implement the queue.
- `f`, `r`, `c`: These are pointers for the front, rear, and the count of the elements in the queue.
- `h[M]`: An array used to implement the heap.
- `hs`: Heap size.

2.2 Stack Functions

The following functions handle stack operations:

```
// Push an element to the stack
void psh(int v, int m) {
    if (t == m-1) {      // If the stack is full
        printf("-1\n");
        exit(0);
    }
    s[++t] = v;          // Push the element and increment the top
}
```

```
// Pop an element from the stack
int pp() {
    if (t == -1) {        // If the stack is empty
        printf("-1\n");
        exit(0);
    }
    return s[t--];        // Return the top element and decrement the top
}

// Print the stack elements
void ps() {
    for (int i = 0; i <= t; i++)
        printf("%d ", s[i]);
    printf("\n");
}
```

- **psh**: This function pushes an element onto the stack. If the stack is full, it prints `-1` and exits.
- **pp**: This function pops the top element from the stack. If the stack is empty, it prints `-1` and exits.
- **ps**: This function prints all elements of the stack.

2.3 Queue Functions

The following functions handle queue operations:

```
// Enqueue an element to the queue
void enq(int v, int m) {
    if (c == m-1) {     // If the queue is full
        printf("-1\n");
        exit(0);
    }
    q[r] = v;           // Insert at rear
    r = (r + 1) % m;    // Update rear with circular increment
    c++;                // Increase count
}

// Dequeue an element from the queue
int deq(int m) {
    if (c == 0) {       // If the queue is empty
        printf("-1\n");
        exit(0);
    }
    int v = q[f];       // Get the front element
    f = (f + 1) % m;    // Update front with circular increment
    c--;                // Decrease count
    return v;           // Return dequeued element
}

// Print the queue elements
void pq(int m) {
    if (c == 0) {       // If the queue is empty
        printf("\n");
        return;
    }
    int i = f;
    for (int j = 0; j < c; j++) {
        printf("%d ", q[i]);
```

```
        i = (i + 1) % m;
    }
    printf("\n");
}
```

- **enq**: This function enqueues an element into the queue. It handles overflow by printing `-1` and exiting.
- **deq**: This function dequeues an element from the front of the queue. If the queue is empty, it prints `-1` and exits.
- **pq**: This function prints all elements in the queue, starting from the front to the rear.

2.4 Heap Functions

The following functions handle heap operations:

```
// Swap two elements
void swp(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Heapify a subtree rooted at index 'i'
void hf(int i) {
    int sm = i, l = 2*i + 1, r = 2*i + 2;
    if (l < hs && h[l] < h[sm])
        sm = l;
    if (r < hs && h[r] < h[sm])
        sm = r;
    if (sm != i) {
        swp(&h[i], &h[sm]);
        hf(sm);
    }
}

// Build a heap from an array 'v' of size 'n'
void bh(int v[], int n, int m) {
    if (n > m) {        // Check if the array size exceeds the limit
        printf("-1\n");
        exit(0);
    }
    hs = n;
    for (int i = 0; i < n; i++)
        h[i] = v[i];
    for (int i = n/2 - 1; i >= 0; i--)
        hf(i);          // Build the heap by heapifying from the bottom up
}

// Print the heap elements
void ph() {
    for (int i = 0; i < hs; i++)
        printf("%d ", h[i]);
    printf("\n");
}

// Extract the minimum element from the heap
int em() {
```

```
    if (hs == 0) {        // If the heap is empty
        printf("-1\n");
        exit(0);
    }
    int r = h[0];         // Root element is the minimum
    h[0] = h[--hs];       // Replace root with last element and reduce heap size
    hf(0);                // Heapify the new root
    return r;
}

// Decrease the value of element at index 'i' to 'nv'
void dk(int i, int nv) {
    if (i < 0 || i >= hs || h[i] < nv)
        return;
    h[i] = nv;
    while (i != 0 && h[(i-1)/2] > h[i]) {
        swp(&h[i], &h[(i-1)/2]);
        i = (i-1)/2;
    }
}
```

- **swp**: This function swaps two integers, used in heap operations.
- **hf**: This function heapifies a subtree rooted at index `i`.
- **bh**: This function builds a heap from an array `v` of size `n`. If the array size exceeds the maximum allowed size, it prints `-1` and exits.
- **ph**: This function prints all elements in the heap.
- **em**: This function extracts the minimum element from the heap.
- **dk**: This function decreases the value of the element at index `i` in the heap to a new value `nv`.

2.5 Main Program

The main function determines which mode (stack, queue, or heap) to use, based on user input.

```
int main() {
    int o, m;
    scanf("%d%d", &o, &m);      // Read the option 'o' and maximum size 'm'

    if (o == 0) {    // Stack mode
        int a, b;
        while (scanf("%d", &a) != EOF) {
            if (a == 0) ps();  // Print stack
            else if (a == 1) { scanf("%d", &b); psh(b, m); } // Push
            else if (a == 2) printf("%d\n", pp());  // Pop
            else if (a == 3) break;  // Exit
        }
    } else if (o == 1) {  // Queue mode
        int a, b;
        while (scanf("%d", &a) != EOF) {
            if (a == 0) pq(m);  // Print queue
            else if (a == 1) { scanf("%d", &b); enq(b, m); } // Enqueue
            else if (a == 2) printf("%d\n", deq(m));  // Dequeue
            else if (a == 3) break;  // Exit
        }
    } else if (o == 2) {  // Heap mode
        int a1, a2, a3;
```

```
        while (scanf("%d", &a1) != EOF) {
            if (a1 == 0) ph();  // Print heap
            else if (a1 == 1) { scanf("%d", &a2); int v[M]; for (int i = 0; i < a2; i++)
// Build heap
            else if (a1 == 2) { scanf("%d%d", &a2, &a3); dk(a2-1, a3); }  // Decrease ke
            else if (a1 == 3) printf("%d\n", em());  // Extract minimum
            else if (a1 == 4) break;  // Exit
        }
    }
    return 0;
}
```

- The program first reads two integers: `o` (the option for mode: stack, queue, or heap) and `m` (the maximum size).
- Based on `o`, it enters the appropriate mode and processes the respective commands until the user exits the loop.

3 Conclusion

This program demonstrates the implementation of three fundamental data structures: stack, queue, and heap. Each functionality can be used separately based on user input, and all standard operations are supported.