Implementering av Stackar

Nipun Thianwan

VT 2025

Introduktion

I denna uppgift har skribenten implementerat en räknare som använder sig av *Reverse Polish* notationen. Kort förklarat innebär detta att operanden i ett matematiskt uttryck placeras sist. Till exempel skrivs uttrycket $5 + 3$ som $5\ 3\ +$ i Reverse Polish-notation. Denna metod användes av flera äldre räknare, som till exempel HP-35 från 1970-talet. För att denna metod ska fungera krävs en datastruktur som kallas för *Stack*. En Stack kan liknas vid en hög och erbjuder två grundläggande operationer: *Push* och *Pop*.

Operationen *Push* lägger till ett element, exempelvis en operand eller siffra, på Stacken, där det förvaras. *Pop*, å andra sidan, tar bort det översta elementet på Stacken och returnerar det. En Stack bygger på principen *sist in, först ut* (LIFO, Last In, First Out). I denna rapport kommer både statisk och dynamisk Stack att behandlas.

En statisk stack

Målet med uppgiften var att implementera en statisk stack som följer LIFO-principen (Last In, First Out). Stacken skulle ha en fast storlek, och operationerna push (lägga till element) och pop (ta bort element) skulle implementeras. Den statiska stacken allokerar minne vid skapandet, och användaren måste hantera dess begränsade storlek. Uppgiften ger en grundläggande förståelse för stackens funktionalitet. Kodning och körning genomfördes med Visual Studio och Ubuntu-terminalen.

```
typedef struct stack {
    int top;
    int size;
    int *array;
} stack;

stack *new_stack(int size) {
    int *array = (int*)malloc(size * sizeof(int));
    if (!array) {
        printf("Memory allocation failed for stack array.\n");
        exit(EXIT_FAILURE);
    }
```

```
    stack *stk = (stack*)malloc(sizeof(stack));
    if (!stk) {
        printf("Memory allocation failed for stack structure.\n");
        exit(EXIT_FAILURE);
    }

    stk->top = -1;
    stk->size = size;
    stk->array = array;
    return stk;
}

void push(stack *stk, int val) {
    if (stk->top >= stk->size - 1) {
        printf("Stack overflow! Cannot push %d onto the stack.\n",
            val);
        return;
    }
    stk->array[++stk->top] = val;
}

int pop(stack *stk) {
    if (stk->top < 0) {
        printf("Stack underflow! Cannot pop from an empty stack.\n
           ");
        return -1;
    }
    return stk->array[stk->top--];
}
```

Efter implementationen testades koden med en stack av storlek 4, där flera `push`- och `pop`-operationer utfördes. Resultaten visade att stacken fungerade korrekt med tydlig felhantering för över- och underflöd. Minnesläckage undveks genom att frigöra allokerat minne.

Slutsatsen är att den statiska stacken är enkel och pålitlig för att hantera ett begränsat antal element. Dock är den mindre flexibel jämfört med dynamiska lösningar, men effektiv när maxstorleken är känd.

En dynamisk stack

En dynamisk stack skiljer sig från en statisk stack genom att den kan växa och krympa efter behov. Vid stacköverflöd fördubblas storleken genom att allokera en större array och kopiera över elementen. När antalet element understiger en fjärdedel av kapaciteten, krymps storleken för att spara minne. Uppgiften fokuserade på att implementera dessa mekanismer i C för att skapa en flexibel och effektiv stack som hanterar minnesallokering och elementkopiering på ett optimerat sätt. Kodning och körning genomfördes med Visual Studio och Ubuntu-terminalen. Kodmässigt ser de relevanta delarna ut som

följande:

```
void push(stack *stk, int val) {
    if (stk->top >= stk->size - 1) {
        int new_size = stk->size * 2;
        int *new_array = (int *)malloc(new_size * sizeof(int));
        for (int i = 0; i < stk->size; i++) {
            new_array[i] = stk->array[i];
        }
        free(stk->array);
        stk->array = new_array;
        stk->size = new_size;
        printf("Stack expanded to size: %d\n", new_size);
    }
    stk->array[++stk->top] = val;
}

int pop(stack *stk) {
    if (stk->top < 0) {
        printf("Stack underflow! Cannot pop from an empty stack.\n
           ");
        return -1;
    }
    int val = stk->array[stk->top--];
    if (stk->top < stk->size / 4 && stk->size > 4) {
        int new_size = stk->size / 2;
        int *new_array = (int *)malloc(new_size * sizeof(int));
        for (int i = 0; i <= stk->top; i++) {
            new_array[i] = stk->array[i];
        }
        free(stk->array);
        stk->array = new_array;
        stk->size = new_size;
        printf("Stack shrunk to size: %d\n", new_size);
    }
    return val;
}
```

När fler element läggs till än den initiala storleken expanderar den dynamiska stacken från $4 \rightarrow 8 \rightarrow 16$, och krymper från $16 \rightarrow 8 \rightarrow 4$ när antalet element understiger en fjärdedel av kapaciteten. Stackens topp uppdateras korrekt vid varje `push` och `pop`, och elementen hanteras enligt LIFO-principen. Till skillnad från den statiska stacken i uppgift 1, som kunde orsaka stacköverflöd, anpassar den dynamiska stacken sin storlek efter behov, vilket gör den flexibel och minneseffektiv.

Koden testades genom att skapa en stack med initial storlek 4 och utföra flera `push`- och `pop`-operationer. Stacken expanderade och krympte korrekt, vilket visade att den hanterade minnet effektivt utan problem. Slutsatsen är att den dynamiska stacken är flexibel och passar väl för scenarier där antalet element varierar.

En tom stack

Uppgiften handlar om att hantera situationer där en operation utförs på en tom stack, som vid en pop-operation. Lösningen kan antingen returnera ett signalvärde, exempelvis -1, eller generera ett felmeddelande för att indikera att stacken är tom. Typmetoden för att lösa denna uppgift är densamma som i de två föregående uppgifterna, där felhantering implementeras för att säkerställa att programmet hanterar edge cases på ett säkert och förutsägbart sätt.

```
int main() {
stack *stk = stack(4);
int n = 10;
for(int i = 0; i < n; i++) {
push(stk, i+30);
}
for(int i = 0; i < stk->top; i++) {
printf("stack[%d] : %d\n", i, stk->array[i]);
}
int val = pop(stk);
while(val != 0) { // assuming 0 is returned when the stack is
   empty
printf("pop : %d\n", val);
val = pop(stk);
}
}
```

Resultatet visar att stacken expanderar dynamiskt från storlek 4 till 8 och sedan till 16. Element hanteras enligt *LIFO*-principen, och när stacken är tom returnerar `pop` -1 och ett felmeddelande (Stack underflow! Cannot pop from an empty stack). Funktionen fungerar som förväntat.

Kalkylatorn- The Calculator

För att implementera räknaren användes en tidigare byggd stack, som kompletterades med kod för att möjliggöra beräkningar. En skelettkod tillhandahölls som grund, och en dynamisk stack valdes för dess flexibilitet. Arbetet utfördes i Visual Studio Code, där endast heltal hanterades och funktionerna `push` och `pop` anpassades för stacken.

Felhantering lades till för att förhindra otillåten användning, som felplacerade operatorer, ogiltiga symboler och division med noll. Felmeddelanden visades vid dessa fall för att säkerställa korrekt funktionalitet. Delar av den slutliga koden ser ut som följer:

```
    while (run) {
        printf(" > ");
        fgets(buffer, n, stdin);

        if (buffer[0] == '\n' || strcmp(buffer, "exit\n") == 0) {
            run = 0;
        } else if (buffer[0] == '+') {
```

```
            int b = pop(stk);
            int a = pop(stk);
            if (a != -1 && b != -1) push(stk, a + b);
        } else if (buffer[0] == '-') {
            int b = pop(stk);
            int a = pop(stk);
            if (a != -1 && b != -1) push(stk, a - b);
        } else if (buffer[0] == '*') {
            int b = pop(stk);
            int a = pop(stk);
            if (a != -1 && b != -1) push(stk, a * b);
        } else if (buffer[0] == '/') {
            int b = pop(stk);
            int a = pop(stk);
            if (a != -1 && b != -1 && b != 0) push(stk, a / b);
            else if (b == 0) printf("Division by zero is not
               allowed.\n");
        } else {
            int val = atoi(buffer);
            if (val != 0 || buffer[0] == '0') {
                push(stk, val);
            } else {
                printf("Invalid input: %s", buffer);
            }
        }
    }
```

Med denna räknare kan vi ta redan på vad 4 2 3 * 4 + 4 * + 2 -. Vi vet att 4 2 3 * 4 + 4 * + 2 - kan skriva som (4 + (((2*3)+4)*4)) - 2 och efter att mata in alla data då ger det ut 42.