SCT211-0504/2021

ADTs and Algorithms

CAT 1

GATMACH YUOL NYUON

1.

A singly linked list is a dynamic data structure that allows efficient insertion and deletion of nodes, where each node points to the next node in a sequence. A node in a singly linked list contains the following two components:

Data Field:

- This field holds the value or information that the node is meant to store. For example, if the node is meant to store integers, this field could hold values like 1, 27, 39, etc.

Pointer (Next):

- This is a pointer to the next node in the linked list.
- It stores the memory address of the next node or NULL if it is the last node in the list.

[Data | Next] -> [Data | Next] -> NULL

The key characteristics of a singly linked list are:

One-directional: The nodes are connected in one direction only, from the first node (head) to the last node (tail).

Dynamic Size: It can grow or shrink dynamically by adding or removing nodes, making it more memory-efficient than arrays.

Tail End: The last node has its **pointer (Next)** set to **NULL**, indicating the end of the list.

No Indexing: Unlike arrays, a singly linked list does not provide direct indexing; traversal is required to access elements.

Snippet code:

```
struct Node {
    int data;        // Data stored in the node
    struct Node* next; // Pointer to the next node
};
```

2.

To create a queue using two stacks, we utilize two stacks: **Stack1** for **enqueue operations** and **Stack2** for **dequeue operations**. This approach ensures that the **First-In-First-Out (FIFO)** property of a queue is preserved, even though stacks inherently follow a **Last-In-First-Out (LIFO)** order.

Steps to Implement Queue Using Two Stacks

Enqueue Operation:

- We push the element onto Stack1.
- This ensures that the elements are added in the same order they arrive.

Dequeue Operation:

- If Stack2 is empty:
  - Transfer all elements from Stack1 to Stack2 by popping each element from Stack1 and pushing it onto Stack2.
  - This reverses the order, making the first element added to Stack1 the top element of Stack2.
- Finally, pop the top of Stack2, which corresponds to the front element of the queue.

Pseudocode

Enqueue(x):

1. Push x onto Stack1.

Dequeue():

1. If Stack2 is empty:
    - While Stack1 is not empty:
        - Pop the top of Stack1 and push it onto Stack2.
2. Pop the top of Stack2 (this will be the front of the queue).

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct Stack {

   int arr[MAX];

   int top;

} Stack;

// Initialize stack

void initStack(Stack* s) {

   s->top = -1;
```

```
}

// Push operation

void push(Stack* s, int data) {

    if (s->top == MAX - 1) {

        printf("Stack overflow!\n");

        return;

    }

s->arr[++(s->top)] = data;

}

// Pop operation

int pop(Stack* s) {

    if (s->top == -1) {

        printf("Stack underflow!\n");

        return -1;

    }

    return s->arr[(s->top)--];

}
```

```
// Check if stack is empty
int isEmpty(Stack* s) {
    return s->top == -1;
}
// Queue structure using two stacks
typedef struct Queue {
    Stack stack1;
    Stack stack2;
} Queue;
// Initialize queue
void initQueue(Queue* q) {
    initStack(&(q->stack1));
    initStack(&(q->stack2));
}
// Enqueue operation
void enqueue(Queue* q, int data) {
    push(&(q->stack1), data);
```

```
    printf("Enqueued: %d\n", data);
}
// Dequeue operation
int dequeue(Queue* q) {
    if (isEmpty(&(q->stack2))) {
        while (!isEmpty(&(q->stack1))) {
            push(&(q->stack2), pop(&(q->stack1)));
        }
    }
    if (isEmpty(&(q->stack2))) {
        printf("Queue is empty!\n");
        return -1;
    }
    int front = pop(&(q->stack2));
    printf("Dequeued: %d\n", front);
    return front;
}
```

```
// Main function to demonstrate
int main() {
    Queue q;
    initQueue(&q);
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    dequeue(&q);
    dequeue(&q);
    enqueue(&q, 40);
    dequeue(&q);
    dequeue(&q);
    return 0;
}
```

Example Workflow

Enqueue Operations:

- **Enqueue 5, 20, 35:**

- Stack1: [5, 20, 35]
- **Stack2**: []

Dequeue Operation:

- **Dequeue:**
  - Since Stack2 is empty, transfer elements from **Stack1** to **Stack2**.
  - **Stack1**: []
  - **Stack2**: [35, 20, 5]
  - Pop the top of **Stack2.**
  - **Output:** 5.

Enqueue Operation:

- **Enqueue 40:**
  - Push 40 onto **Stack1.**
  - Stack1: [40]
  - Stack2: [35, 20]

Dequeue Operation:

- **Dequeue:**
  - Pop from Stack2.
  - **Output:** 20.

3.

i) Traversing in reverse order: **Stack**

A stack is suitable because it uses the **Last-In-First-Out (LIFO)** principle, allowing reverse traversal.

ii) Ensuring first-in-first-out processing: **Queue**

A queue is ideal as it uses **the First-In-First-Out (FIFO)** principle, ensuring items are processed in the order they arrive.

iii) Tracking function calls in recursion: **Stack**

Recursion uses the system call stack to store function calls, making the stack the natural data structure for this purpose.

4.

Adjacency matrix representation of the graph:

	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	1	0	0	1
D	0	1	0	0

Explanation:

Row **i**, column **j** has 1 if there is a directed edge from node **i** to node **j**, and **0** otherwise.

5.

To delete a node after a given node in a linked list, the following steps should be followed:

- Identify the given node (let's call it **current**).

- Locate the node to be deleted, which is the one immediately after **current**. Let's call this node **toDelete.**
- Adjust the next pointer of the current node to point to the node after **toDelete.**
- Free the memory occupied by the **toDelete** node.

Example:

Let's say we have the following linked list:

5 -> 15 -> 30 -> 45

If we want to delete the node after **15** (i.e., **30**):

1. **Identify the current node:** The node containing 15.
2. **Locate the node to delete:** The node immediately after **current** is 30.
3. **Adjust the next pointer:**
   - Change the next pointer of the node containing 15 to point to the node after **30** (i.e., **45**).
4. **Free memory:** Deallocate the memory for the node containing 30.

The linked list will now look like this:

5 -> 15 -> 45