lec11-linkedlist-stack-queue

April 18, 2018

1 Lecture 11: Linked List, Stack, Queue

Zewei Chu 4/18/2018

1.0.1 Announcement

Exam 1 is this Friday, 4/20, 2:30 in this classroom

1.1 Linked List

A linked list is a linear data structure where each element is a separate object. Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list.

- Usually we use malloc to dynamically create linked list.
- Free the linked list at the end of the program to prevent memory leak.
- Always free the node when you remove a node from the linked list

1.1.1 Linked List Operations

In [20]:
```
#include<stdio.h>
#include<stdlib.h>
typedef struct _node node;
struct _node{
    int value;
    node* next;
};

// print information of a node
void print_node(node* p){
    printf("node: %d \n", p->value);
}

// print the whole int list
void print_intlist(node* head){
    if (head == NULL){
        return;
```

```
    }
    print_node(head);
    print_intlist(head->next);
}

// free the whole int list
void free_intlist(node* head){
    node* tmp = head;
    printf("freeing linked list\n");
    while (head != NULL){
        tmp = head;
        head = head->next;
        free(tmp);
    }
}

// raise an error
void raise_error(char* msg, int code){
    printf("%s\n", msg);
    exit(code);
}

// check if the linked list is empty
int is_empty(node* head){
    return head == NULL;
}

// add v to the beginning of the linked list
// return the head node pointer
node* push_front(int v, node* head){
    node* newnode = (node*)malloc(sizeof(node));
    newnode->value = v;
    newnode->next = head;
    return newnode;
}

node* push_back(int v, node* head){
    node* tmp = head;
    node* newnode = (node*)malloc(sizeof(node));
    newnode->value = v;
    newnode->next = NULL;
    if (is_empty(head)){
        return newnode;
    }
    for (;tmp->next != NULL; tmp = tmp->next)
        ;
    tmp->next = newnode;
    return head;
```

```
}


// node_create  create a list containing length nodes.
// Return pointer to first node.
// Write iteratively using push_front
node* node_create(int length){
    if (length == 0){
        return NULL;
    }
    node* head = (node*)malloc(sizeof(node));
    head->value = 0;
    head->next = NULL;
    node* tmp = head;
    for (int i = 1; i < length; i++){
        tmp->next = (node*)malloc(sizeof(node));
        tmp->value = 0;
        tmp = tmp->next;
    }
    tmp->next = NULL;
    return head;
}



node* create_new_node(int v){
    node* newnode = (node*)malloc(sizeof(node));
    newnode->value = v;
    newnode->next = NULL;
    return newnode;
}

// a helper function on insert node after a given node
void insert_node_after(node* p, node* newnode){
    newnode->next = p->next;
    p->next = newnode;
}

// insert a node of value v at position pos
node* insert_node_at_pos(int v, node* head, int pos){
    node* newnode = create_new_node(v);
    if (is_empty(head)){
        return newnode;
    }
    if (pos == 0){
        newnode->next = head;
        return newnode;
    }
```

```
    node* curr = head;
    int i = 1;
    for (; curr->next!=NULL && i < pos; ++i)
        curr = curr->next;
    insert_node_after(curr, newnode);
    return head;
}

// remove a node at position pos,
// return the new head node pointer
node* remove_node_at_pos(node* head, int pos){
    if (is_empty(head)){
        return NULL;
    }
    node* curr = head;
    // if remove head node
    if (pos ==0){
        head = head->next;
        free(curr);
        return head;
    }
    // if not remove head node
    // find the node at the position right before the node to remove
    int i = 1;
    for (;curr != NULL && i < pos; i++)
        curr = curr->next;
    if (curr == NULL || curr->next == NULL)
        return head;

    node* tmp = curr->next;
    curr->next = tmp->next;
    free(tmp);
    return head;
}


int main(){
    int n = 10;
    node* head = node_create(n);
    int i = 0;
    node* curr = head;
    for (i = 0; i < n; ++i){
        curr->value = i;
        curr = curr->next;
    }
    // insert node
//     head = insert_node_at_pos(5, head, 0);
//     head = insert_node_at_pos(6, head, 1);
```

```
        //     head = insert_node_at_pos(7, head, 6);
        //     head = insert_node_at_pos(8, head, 8);
        //     head = insert_node_at_pos(9, head, 100);

            head = remove_node_at_pos(head, 8);

            print_intlist(head);

            printf("free linked list.....\n");
            free_intlist(head);
            return 0;
        }
```

```
node: 0
node: 1
node: 2
node: 3
node: 4
node: 5
node: 6
node: 7
node: 9
free linked list...
freeing linked list
```

Advantages of linked list

- Items can be added or removed from the middle of the list
- There is no need to define an initial size

Disadvantages

- There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
- Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
- Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.

1.2 Queue

Queue is a linear data structure which follows First In First Out (FIFO) operations. A queue has the same idea of consumers for products where the consumer that came first gets served first.

A queue supports four basic operations:

- Equeue: add an item to the queue

- Dequeue: remove and item from the queue
- Front: get the front item from the queue
- Rear: get the last item from the queue

reference:

- https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/
- https://www.geeksforgeeks.org/queue-set-2-linked-list-implementation/

In [15]:
```c
#include<stdio.h>
#include<stdlib.h>

typedef struct _QNode QNode;
struct _QNode{
    int value;
    struct _QNode *next;
};

typedef struct _Queue Queue;
struct _Queue{
    QNode *front, *rear;
};

QNode* newNode(int v){
    QNode* p = (QNode*)malloc(sizeof(QNode));
    p->value = v;
    p->next = NULL;
    return p;
}

Queue* createQueue(){
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->front = q->rear = NULL;
    return q;
}

void printQNode(QNode* p){
    printf("QNode: %d\n", p->value);
}

void printQueue(Queue* q){
    QNode* p = q->front;
    printf("Queue: ");
    while (p != NULL){
        printf("%d ", p->value);
        p = p->next;
    }
    printf("\n");
```

```
}

void enQueue(Queue* q, int v){
    QNode* p = newNode(v);
    if (q->rear == NULL){
        q->front = q->rear = p;
        return;
    }
    q->rear->next = p;
    q->rear = p;
}

QNode* deQueue(Queue* q){
    if (q->front == NULL)
        return NULL;

    QNode* p = q->front;
    q->front = q->front->next;

    if (q->front == NULL)
        q->rear = NULL;
    return p;
}

void freeQueue(Queue* q){
    QNode* p = q->front;
    printf("Freeing queue...");
    while (p != NULL){
        free(p);
        p = p->next;
    }
}

int main(){
    Queue* q = createQueue();
    QNode* p = NULL;
    enQueue(q, 1);
    enQueue(q, 2);
    enQueue(q, 3);
    enQueue(q, 4);
    printQueue(q);
    p = deQueue(q);
    printQNode(p);
    free(p);
    p = deQueue(q);
    printQNode(p);
    free(p);
```

```
    printQueue(q);
    freeQueue(q);
    return 0;
}
```

```
Queue: 1 2 3 4
QNode: 1
QNode: 2
Queue: 3 4
Freeing queue...
```

1.3 Stack

A stack is a First In Last Out (FILO) data structure.
It should have three basic operations

- push: add an item in the stack
- pop: remove an element from the stack
- isEmpty: returns 1 if stack is empty, else 0

In [25]:
```
#include <stdio.h>
#include <stdlib.h>

typedef struct _StackNode StackNode;
struct _StackNode{
    int value;
    StackNode* next;
};

typedef struct _Stack Stack;
struct _Stack{
    StackNode* head;
};


// raise an error
void raise_error(char* msg, int code){
    printf("%s\n", msg);
    exit(code);
}

// create a new node with value v
StackNode* newNode(int v){
    StackNode* newnode = (StackNode*) malloc(sizeof(StackNode));
    newnode->value = v;
    newnode->next = NULL;
    return newnode;
}
```

```c
// create a new empty stack
Stack* newStack(){
    Stack* s = (Stack*)malloc(sizeof(Stack));
    s->head = NULL;
    return s;
}

int isEmpty(Stack* s){
    if (s->head == NULL)
        return 1;
    else
        return 0;
}

// push a value into stack
void push(Stack* s, int v){
    StackNode* newnode = newNode(v);
    newnode->next = s->head;
    s->head = newnode;
}

// pop value out of stack
int pop(Stack* s){
    if (isEmpty(s))
        raise_error("stack is empty", -1);
    StackNode* tmp = s->head;
    s->head = tmp->next;
    int v = tmp->value;
    free(tmp);
    return v;
}

void freeStack(Stack* s){
    StackNode* p = s->head;
    printf("Freeing stack...");
    while (p != NULL){
        free(p);
        p = p->next;
    }
}

int main(){
    Stack* s = newStack();
    int v = 0;
    push(s, 0);
    push(s, 1);
    push(s, 2);
    push(s, 3);
```

```
    v = pop(s);
    printf("poped value: %d\n", v);
    v = pop(s);
    printf("poped value: %d\n", v);
    v = pop(s);
    printf("poped value: %d\n", v);

    freeStack(s);
    return 0;
}
```

```
poped value: 3
poped value: 2
poped value: 1
Freeing stack...
```