DATA STRUCTURES AND ALGORITHMS

Dynamic data structures:

stack, queue, deque

Summary of the previous lecture

- Definition of list and examples of the list
- List features:
  - Head,
  - Tail,
  - Length
- Array implementation
  - Add, remove, insert node
- Linked list
  - Add, remove, insert node

Stack

The stack is a list-like structure in which elements may be inserted or removed from only one end i.e.

Stack is a data structure which works based on principle of last in first out (LIFO).

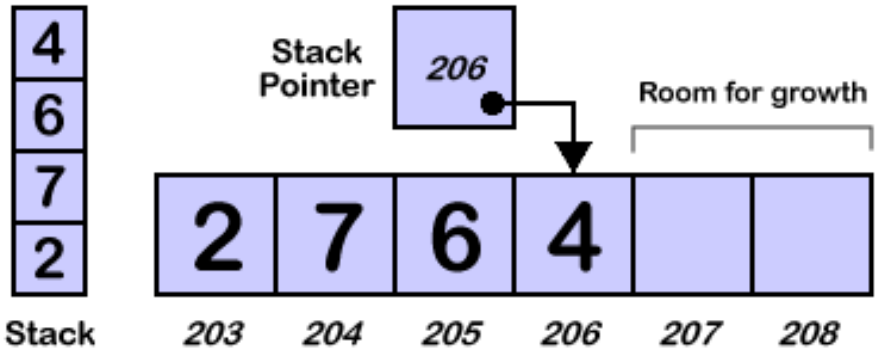
Stacks are used very frequently. Accountants used stacks long before the invention of the computer. They called the stack a "**LIFO**" list, which stands for "**Last-In, First-Out**."

Note that one implication of the LIFO policy is that stacks remove elements in reverse order of their arrival.

Stack

It is traditional to call the accessible element of the stack the **top** element.

Elements are not said to be inserted; instead they are **pushed** onto the stack. When removed, an element is said to be **popped** from the stack.



Stack

The **top item** is the item always the **last item** to enter the stack and it is always the first item to leave the stack since no other items can be removed until the top item is removed.

Push and pop functions:

Push (Stack, Item)

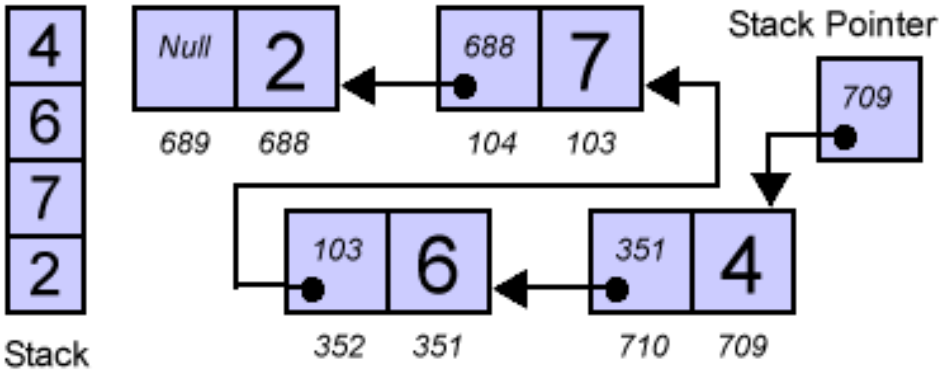
Item Pop (Stack)

The **Push** operation has two parameters which are,

- the stack
- an item to add.

The **Pop** operation only takes one parameter which is a stack.

Stack



In order to implement a stack using pointers, we need to link nodes together just like we did for the pointer implementation of the list.

Each node contains:

- the **stack item**
- the **pointer to the next node**.

Special pointer (**stack pointer**) is needed to keep track of the top of the stack.

Stack operations

push (stack, new-item)

Adds an item onto the stack.

item **top** ()

Returns the last item pushed onto the stack.

item **pop** ()

Removes the most-recently-pushed item from the stack.

bool **is-empty** ()

True iff no more items can be popped and there is no top item.

bool **is-full** ()

True iff no more items can be pushed.

int **get-size** ()

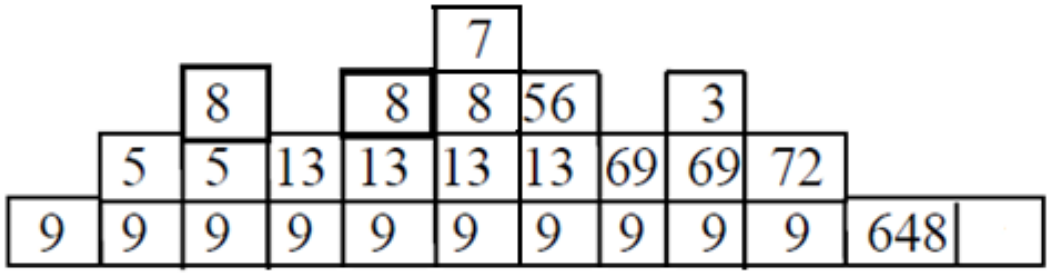
Returns the number of elements on the stack.

Stack example

Lets calculate expression:

9*(((5+8)+(8*7))+3)

- *push(9);*
- *push(5);*
- *push(8);*
- *push(pop+pop);*
- *push(8);*
- *push(7);*
- *push(pop*pop);*
- *push(pop+pop);*
- *push(3);*
- *push(pop+pop);*
- *push(pop*pop);*
- *writeln(pop).*



Stack

The two approaches for the stack implementation can be used:

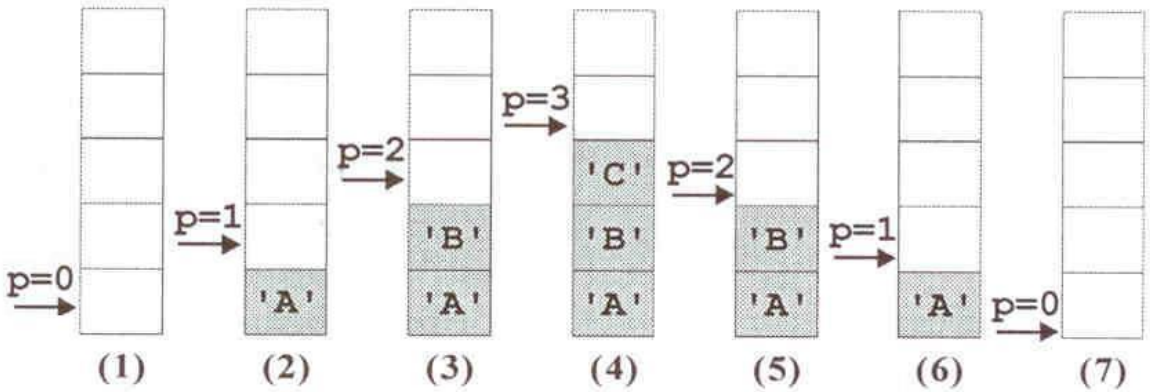
- **array-based stack**
- **linked stack**

Array-based stack uses "index" (integer) to manage top node. "Index" is somewhat like a current position value (because the "current" position is always at the top of the stack), as well as indicating the number of elements currently in the stack.

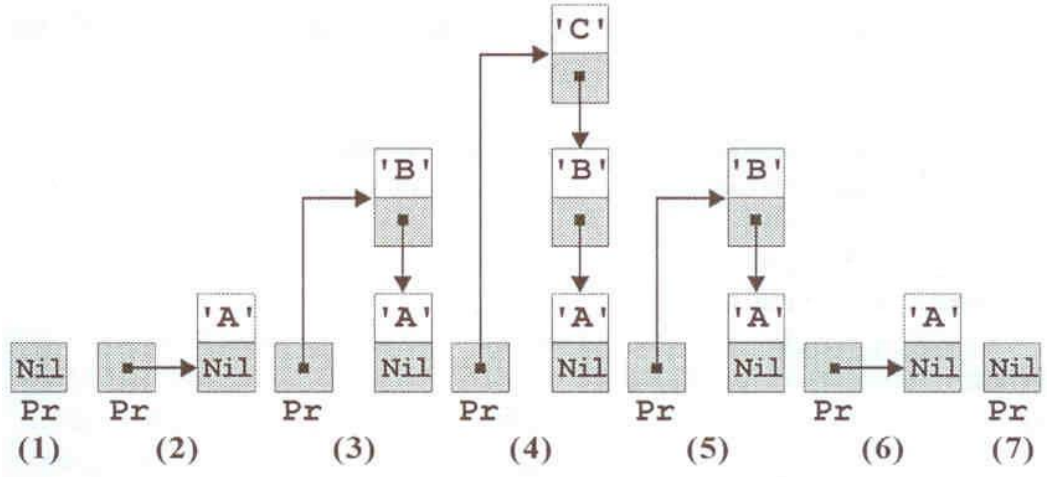
Linked stack uses pointer to top node to manage stack. The only data member is **top**, a pointer to the first (top) link node of the stack.

Stack implementation

Array based stack



List based stack



Array-based stack implementation

Stack.h

void **push** (int *s, int* top, int element);

int **pop** (int *s, int *top);

int **full** (int *top, const int size);

int **empty** (int *top);

void **init** (int *top);

Array-based stack implementation

```
// initialize stack pointer

void init (int *top)
{
   *top = 0;
}


// push an element into stack precondition: the stack is not full

void push (int *s, int* top, int element)
{
   s[ (*top)++ ] = element;
}
```

Array-based stack implementation

```
/* pop an element from stack precondition: stack is not empty */
int pop (int *s,int *top)
{
   return s[--(*top)];
}

/* report stack is full nor not return 1 if stack is full, otherwise return 0 */
int full (int *top, const int size)
{
   return *top == size ? 1 : 0;
}

/* report a stack is empty or not return 1 if the stack is empty, otherwise
   return 0  */
int empty (int *top)
{
   return *top == 0 ? 1 : 0;
}
```

Array-based stack implementation

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
#define size 3
void main()
{
   int top, element;
   int stack[size];
   // initialize stack
   init(&top);

   // push elements into stack
      while(!full(&top,size)) {
            element = rand();
            printf("push element %d into stack\n",element);
            push (stack, &top, element);
            getchar();
    }
```

```
printf ("stack is full\n");
// pop elements from stack

while(!empty(&top)) {
    element = pop(stack,&top);
    printf("pop element %d from
            stack\n",element);
    getchar();
}

printf("stack is empty\n");
getchar();
}

}
```

Linked stack

Example of the linked stack.

```
// Node element:
struct node{
        int data;
        struct node *next;
};
```

```
// Push an element :
void push (int skaicius) {
    node *v;
    v  =(struct node *)malloc(sizeof (struct
          node));
    v->data = skaicius;
    if ( listStart == NULL)          // First node
            v->next = NULL ;
    else
            v->next = listStart->next;
    listStart = v;
}
```

Linked stack

```
// Popped node:
int pop() {
node *v;
   if ( listStart == NULL)
            return 1 ;
    else {
             v = listStart;
             listStart = v->next;
             delete (v);
             }
return 0;
}
```

```
// Top node:
int ReadLast() {
node *v;
   if ( listStart != NULL)
       return  listStart ->data
   else
       return EXIT_FAILURE;
 }
```

Queue

Like the stack, the **queue is also a type of restricted list**.

Instead of restricting all the operations to only one end of the list as a stack does, the queue allows items to be added at the one end of the list and removed at the other end of the list.

This restrictions placed on a queue cause the structure to be a "First-In, First-Out" or **FIFO structure**.

.

Queue example

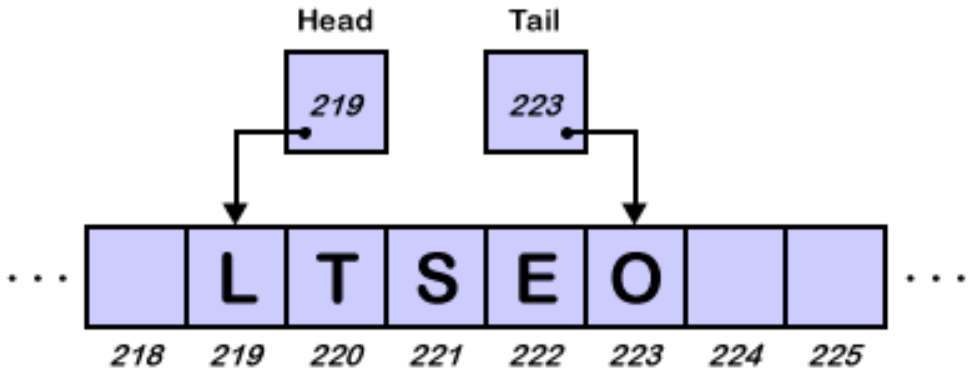
Queue operating principle is similar to customer lines at a in any bill payment store (eg. phone bill payment queue).

When **customer A** is ready to check out, he or she enters the tail (end) of the waiting line.

When the preceding customers have paid, then **customer A** pays and exits from the head of the line.

The bill-payment line is really a queue that enforces a "first come, first serve" policy.

Queue



Basic operations:

EnqueueItem (Queue, Item)

Item DequeueItem (Queue)

EnqueueItem() – "enter the queue item" - operation takes the Item parameter and adds it to the tail(end) of Queue.

DequeueItem() – "delete queue item" - operation removes the head item of Queue and returns this as Item.

Array-based queue

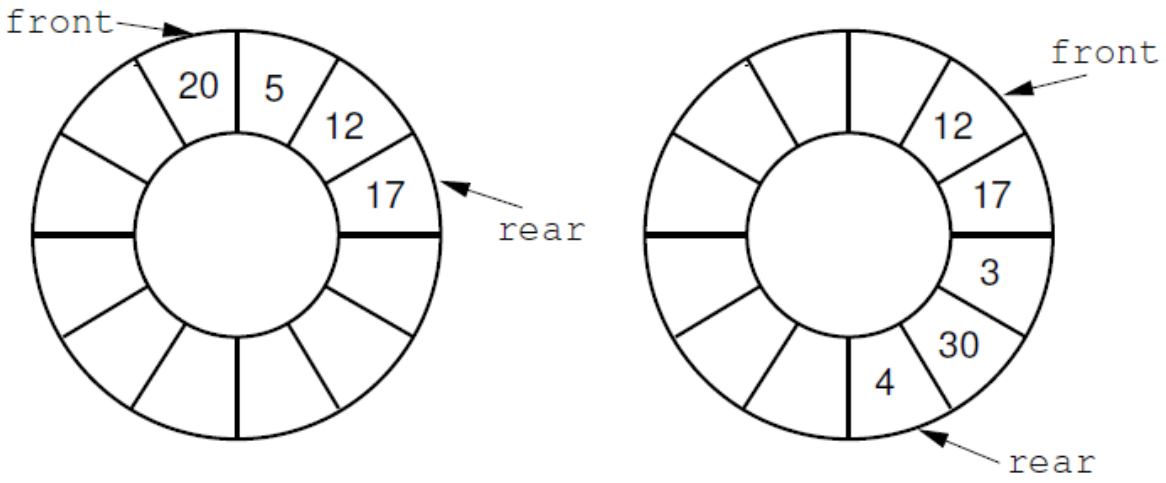
Assume that there are **n** elements in the queue. We could require that all elements of the queue be stored in the first **n** positions of the array.

Far more efficient implementation can be obtained by relaxing the requirement that all elements of the queue must be in the first **n** positions of the array.

We will still require that the queue be stored be in contiguous array positions, but the contents of the queue will be permitted to drift within the array.

This type of queue is called the circular queue.

Circular queue



Front – enqueue

Rear - dequeue

Array-based queue

queue.h

```
void enqueue (int *q, int *tail, int element);
int dequeue (int *q, int *head);
int full (int tail, const int size);
int empty (int head, int tail);
void init (int *head, int *tail);
```

Array-based queue

```
/*initialize queue pointer*/
void init(int *head, int *tail){
*head = *tail = 0;}

/* enqueue an element  precondition: the queue is not full*/
void enqueue(int *q,int *tail, int element){
 q[(*tail)++] = element;}

/* dequeue an element   precondition: queue is not empty*/
int dequeue(int *q,int *head){
return q[(*head)++];}

/* report queue is full nor not return 1 if queue is full, otherwise return 0*/
int full(int tail,const int size){
return tail == size ? 1 : 0;}

/* report queue is empty or not return 1 if the queue is empty, otherwise return 0*/
int empty(int head, int tail){  return head == tail ? 1 : 0; }
```

Array-based queue

```
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"
#define size 3
void main() {
int head,tail,element; int queue[size];
init(&head,&tail);                          // initialize queue

while( full(tail,size) != 1){               // enqueue elements
        element = rand();
        printf("enqueue element %d\n",element);
        enqueue(queue,&tail,element);
         printf("head=%d,tail=%d\n",head,tail);
       getchar(); }
printf("queue is full\n");
```

Array-based queue

```
// dequeue elements from
while( !empty(head,tail)) {
 element = dequeue(queue,&head);
     printf("dequeue element %d \n", element);
     getchar();
}
printf ("queue is empty\n");
getchar();
}
```

Deque

A **deque** is also a type of restricted list. It similar like a queue, except that new items may be added and existing can be removed from both the head and the tail.

It is convenient to make the deque using doubly-linked lists.

Main deque operations:

InsertFront (Item)

InsertRear (Item)

Item DeleteFront()

Item DeleteRear()

Example of Linked deque

Example of the linked deque.

Element:

```
struct node{
        int data;
        struct node *next;
        struct node *prev;
};

// head – head pointer
// tail  – tail pointer
```

```
void InsertFront (struct node *e,
struct node *head,
struct node *tail)
{
    if ( head == NULL)
        head = tail = e;
    else {
        e->next = head;
        head = e;
}
```

Example of Linked deque

```
void RemoveRear (struct node *head, struct node *tail)
{
 struct  node *p = tail;
 struct  node *d = head;

while (d->next->next != NULL )
        d= d->next;

  d->next = NULL;
  tail = d;
  free (p);
}
```

Exercises

No.1 Assume a array has the following configuration:

{ 2; 23; 15; 5; 9 }. Write a C code to delete the element with value 15, add new element 17, replace element 23 with new one 44.

No.2 Assume a linked list has the following configuration:

{ 2; 23; 15; 5; 9 }. Write a C code to delete the element with value 15 and to add new element 17, replace element 23 with new one 44.

No.3 Write a C code to implement array based stack. Make program Menu to realize user interaction.

Homework

No.1 Write program for array based queue.
All elements of the queue must be stored in the first **n** positions of the array. Make program Menu to realize user interaction.

No.2 Write program for array based **circular queue**. Make program Menu to realize user interaction.