ECE 2620: C++, Data Structures & Algorithms
Stacks & Queues

Dr. Sarvesh Kulkarni

Department of Electrical & Computer Engineering
Villanova University, Villanova, PA 19085
Email: sarvesh.kulkarni@villanova.edu

Table of Contents

Creating New ADTs From Linked Lists

We will **reuse** the code that we wrote for linked lists to build even more abstract data types such as

- **Stacks**
- **Queues** (including **Priority Queues**)

Keep in mind that:

- We do not have to rely exclusively on linked lists to build these new ADTs (stacks, queues, priority queues)
- We can also build these ADTs from arrays or vectors
- Which approach you choose depends on whether you prefer small memory footprint or increased execution speed

  Therefore, your selection must be a conscious design choice, and not one of convenience!

Microprocessor-Level System Stacks

A **Stack** is a **Last-In First-Out (LIFO)** data structure
The data item inserted last, is the first to be extracted

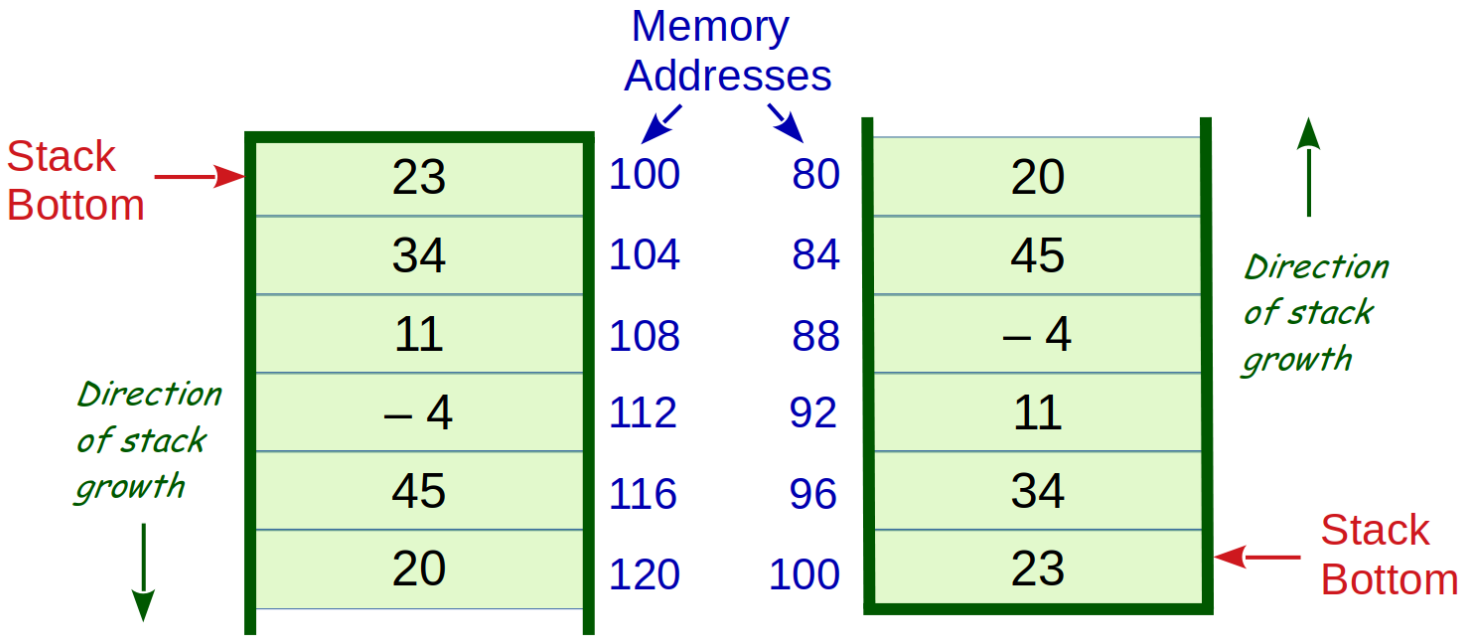


Figure: Two 32-bit microprocessor based stack implementations are shown; both stacks store the same data and have the same base memory address (100). When a new element is inserted, one stack grows downwards, whereas the other grows upwards

Stack Model

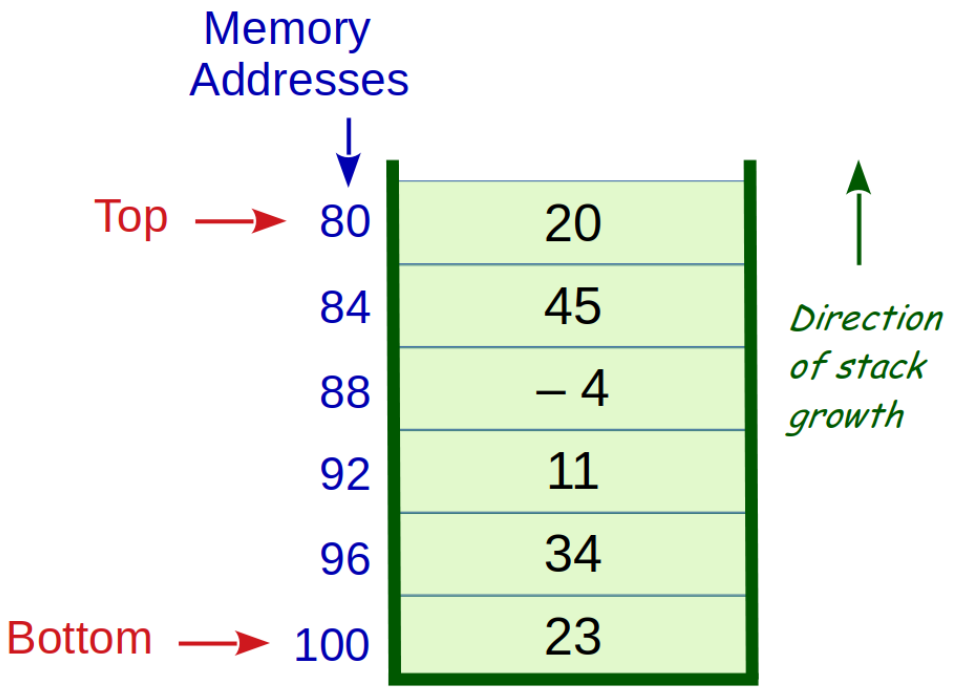


Figure: A Stack that grows upwards

Bottom marks the location of oldest element (at the bottom)
Top marks the location of the newest element (at the top)

Stack contents are tracked by Top & Bottom markers

Stack Operations

Stacks may be designed to hold basic data types (int, float, etc.) or user-defined data types (struct, class, etc.)

Five Stack Operations are defined & implemented as member fns:

1. **clear** clears stack
2. **isEmpty** checks if stack is empty
3. **push** pushes (stores) element on top of stack
4. **pop** pops (deletes) element from top of stack
5. **peek** reads element (non-destructively) on top of stack

The operation **peek** (sometimes called **topEl**), is non-essential
We can emulate it with the following member function calls:

```
el = mystack.pop();     // pops element, and stores it in 'el'
mystack.push(el);       // pushes element back to stack
```

Arithmetic Expressions: Infix & Postfix

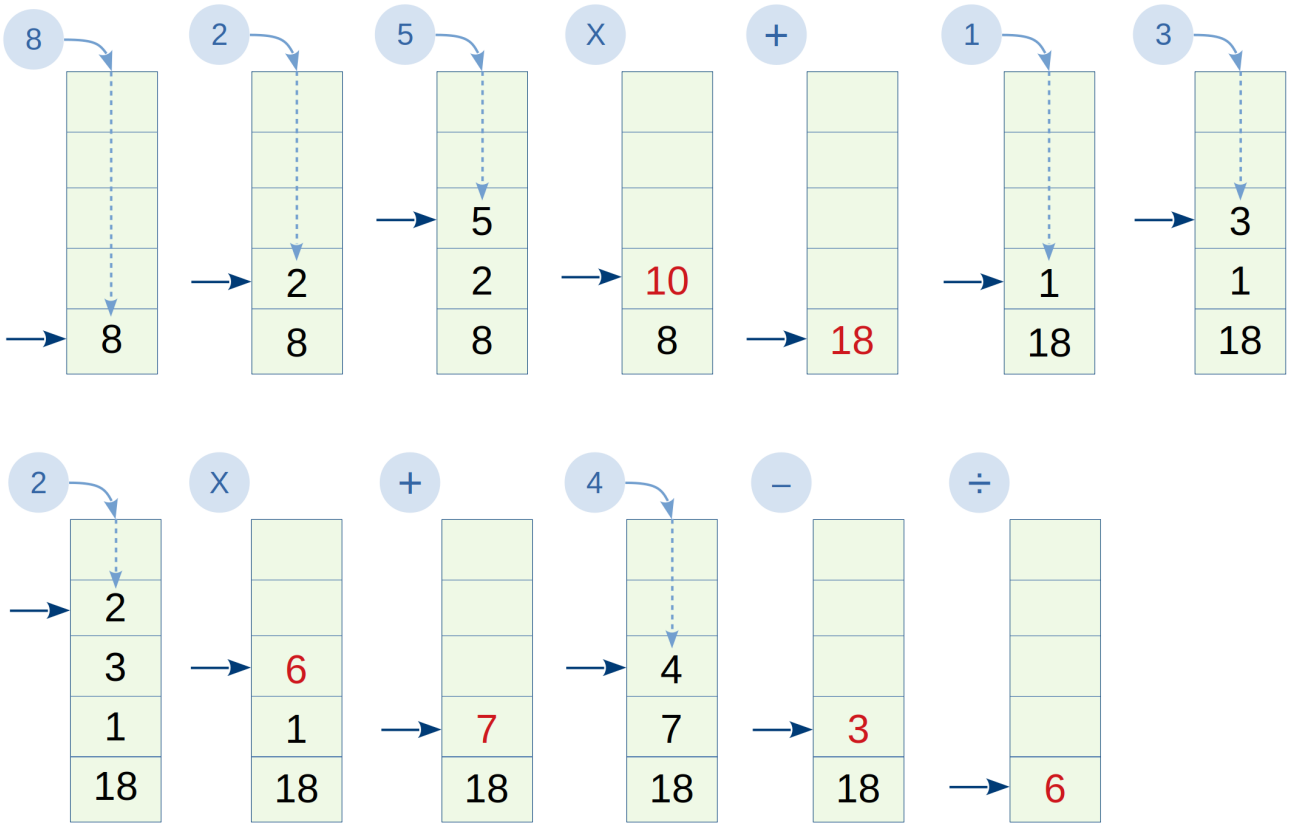
An arithmetic expression in **Infix** notation:
$(8 + 2 \times 5) \ / \ (1 + 3 \times 2 - 4)$

Same expression in **Postfix** (**Reverse Polish**) Notation:
$8 \quad 2 \quad 5 \quad \times \quad + \quad 1 \quad 3 \quad 2 \quad \times \quad + \quad 4 \quad - \quad /$

Infix $\rightarrow$ **Postfix** conversion is performed as follows: (take notes)

Evaluating a Postfix Expression With a Stack

In Postfix: 8 2 5 × + 1 3 2 × + 4 − /



Number is pushed, operator causes pop-evaluate-push

Arrow points to top of stack (toS)

When no postfix tokens remain, stack holds final answer

A Stack Constructed With a Linked List (LL)

Two possible singly linked list based stack implementations:

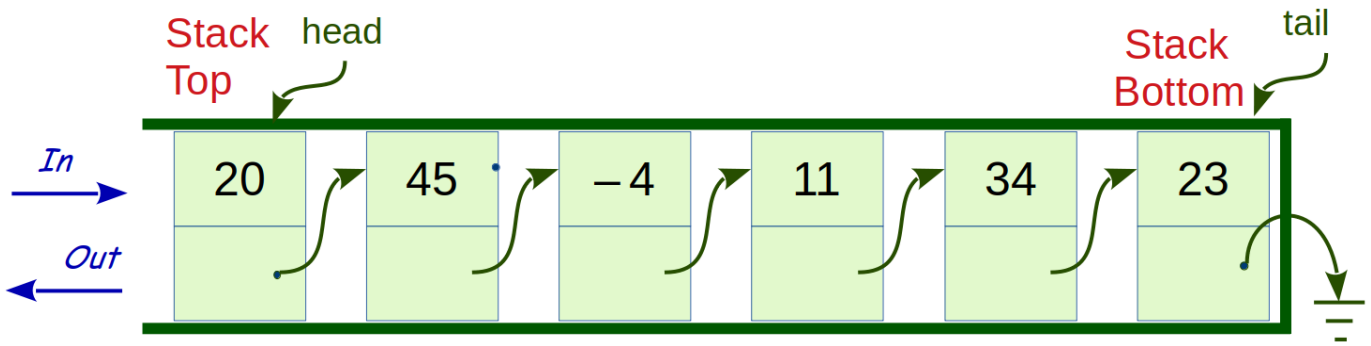


Figure: An SLL stack with stack-bottom at the tail

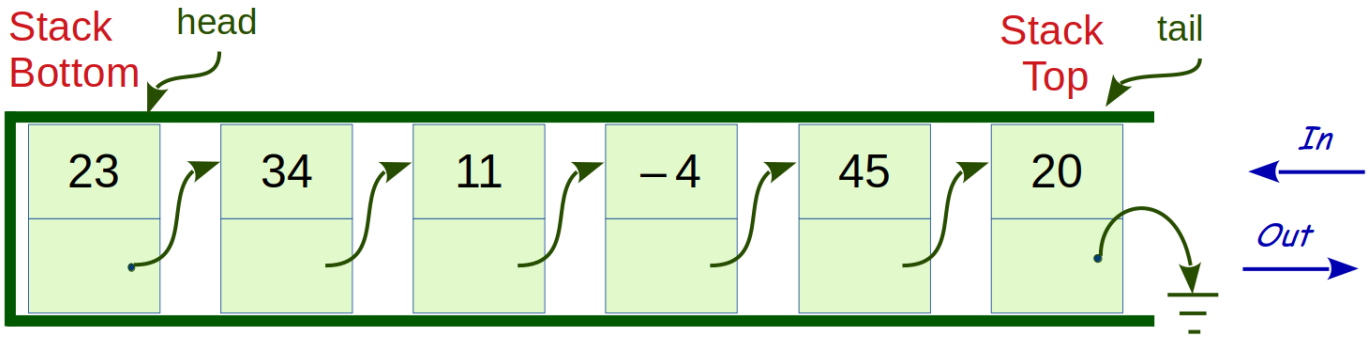


Figure: An SLL stack with stack-bottom at the head

Which one should we choose? **Think carefully!**

A Class for a Linked-List Stack (in header file llstack.h) (1)

```
#ifndef _LL_STACK
#define _LL_STACK

#include "intsllist.h"

class LLStack {
   public:
           // LLStack();              .. No constructor needed
           ~LLStack();                // destructor
           void  clear();             // clear out entire stack
           bool  isEmpty() const;     // check if empty stack
           void  push(const int& el); // push el onto stack
           int   pop();               // pop el from stack
           int   peek ();             // read el from stack

   private:
           intSLList  lst;  // Declare a list (to use as a stack)
};
#endif
```

Class **LLStack** - Member Fn Definitions (in llstack.cc) (2)

```
#include "llstack.h"

void  LLStack::clear() {
       while(!lst.isEmpty())
           lst.deleteFromHead();
}

LLStack::~LLStack() {
clear();
}

bool  LLStack::isEmpty() const {
       return  lst.isEmpty();
}
```

In member function **clear**, **deleteFromTail** would also have worked instead of **deleteFromHead** (inside the while loop)

So why shouldn't we?

Class **LLStack** - Member Fn Definitions (in llstack.cc) (3)

```
void  LLStack::push(const int& el) {
       lst.addToHead(el);
}

int  LLStack::pop() {
       return  lst.deleteFromHead();
}

int  LLStack::peek() {
       int top_el;

       top_el = lst.deleteFromHead();  // read & delete frm stack
       lst.addToHead(top_el);          // reinsert in stack

       return  top_el;
}
```

Asymptotic running time complexity (fill in):

Push: O(____) Pop: O(____) Clear: O(____)

Operating A Stack With Our **LLStack** Class Code

```
int  main() {
       LLStack  myStack;     // Declare stack object myStack
       int tmp;

       // First push 3 ints (in this order: 12, 4, 15) on stack
       myStack.push(12);
       myStack.push(4);
       myStack.push(15);
       // Now pop an element off this stack
       tmp = myStack.pop();
       cout << tmp << endl;  // Outputs 15 to the display
       // Push integer value -3 on stack
       myStack.push(-3);
       // Return top element from stack, but keep stack intact
       tmp = myStack.peek();
       cout << tmp << endl;  // Output will be -3

       return 0;
}
```

A Class for an Array-Stack (in header file astack.h) (1)

```
#ifndef _A_STACK
#define _A_STACK

class AStack {
   public:
           AStack();              // Constructor needs to init toS
           // ~AStack();          .. No destructor  needed
           // void  clear();      .. Deallocating array not possible
           bool  isEmpty() const;      // check if empty stack
           void  push(const int& el); // push el onto stack
           int   pop();                // pop el from stack
           int   peek();               // read el from stack-top

   private:
           int  x[MAXSIZE];     // Declare array (to use as stack)
           int  toS;            // array index of "top of stack"
};

#endif
```

Class **AStack** - Member Fn Definitions (in astack.cc)

```
#include "astack.h"
```

Queue Model

A **Queue** is a **First-In First-Out (FIFO)** data structure, sometimes also referred to as First-Come First-Served (FCFS)

The data item inserted first, is also the first to be extracted

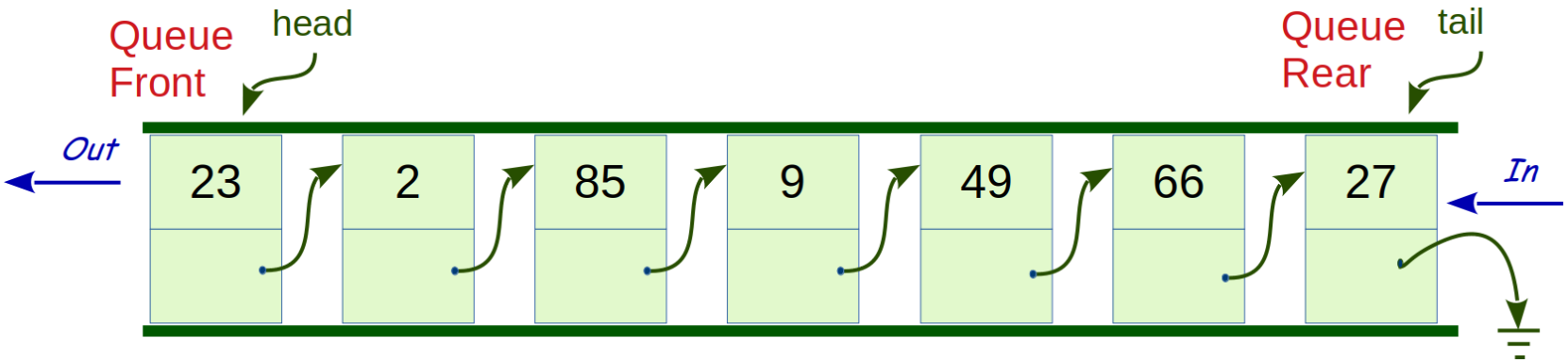


Figure: A queue implemented with a linked list

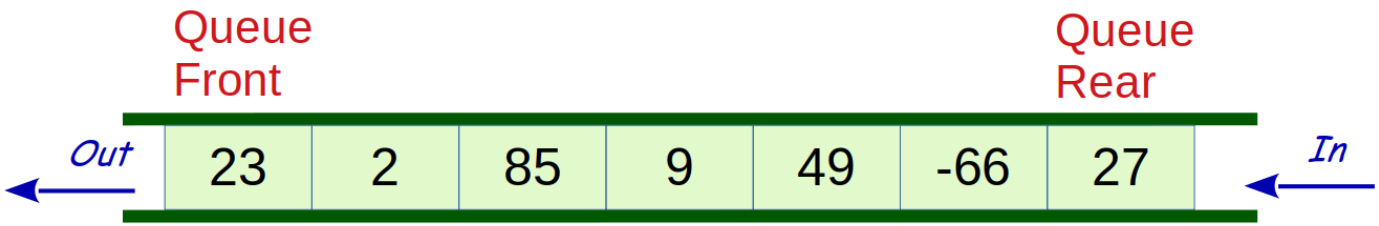


Figure: A queue implemented with an array

Queue Operations

Queues like Stacks, may be designed to hold basic data types (int, float, etc.) or user-defined data types (struct, class, etc.)

Five Queue Operations are defined & implemented as member fns:

1. **clear** clears queue
2. **isEmpty** checks if queue is empty
3. **enqueue** enqueues element at queue-rear
4. **dequeue** dequeues element from queue-front
5. **peek** reads element (non-destructively) from queue-front

The operation **peek** (sometimes called **frontEl**), is non-essential
But, we can easily implement it as well

A Class for a Linked-List Queue (in header file llqueue.h) (1)

```
#ifndef _LL_QUEUE
#define _LL_QUEUE

#include "intsllist.h"

class LLQueue {
   public:
           // LLQueue();                .. No constructor needed
           ~LLQueue();                  // destructor
           void  clear();               // clear out entire queue
           bool  isEmpty() const;       // check if empty queue
           void  enqueue(const int& el); // insert el into queue
           int   dequeue();             // remove el from queue
           int   peek();                // read el from queue-front

   private:
           intSLList  lst;  // Declare a list (to use as a queue)
};
#endif
```

Class **LLQueue** - Member Fn Definitions (in llqueue.cc) (2)

```
#include "llqueue.h"

void  LLQueue::clear() {
       while(!lst.isEmpty())
           lst.deleteFromHead();
}

LLQueue::~LLQueue() {
clear();
}

bool  LLQueue::isEmpty() const {
       return  lst.isEmpty();
}
```

In member function **clear**, **deleteFromTail** would also have worked instead of **deleteFromHead** (inside the while loop)

So why shouldn't we?

Class **LLQueue** - Member Fn Definitions (in llqueue.cc) (3)

```
void  LLQueue::enqueue(const int& el) {
       lst.addToTail(el);
}

int  LLQueue::dequeue() {
       return  lst.deleteFromHead();
}

int  LLQueue::peek() {
       int front_el;
       front_el = lst.deleteFromHead();  // read & delete from qu
       lst.addToHead(front_el);          // reinsert into queue

       return  front_el;
}
```

Why not enqueue at the head and dequeue from the tail?

Asymptotic running time complexity (fill in):

Enqueue: O(____) Dequeue: O(____) Clear: O(____)

Operating A Queue With Our **LLQueue** Class Code

```
int  main() {
       LLQueue  myQueue;     // Declare stack object myQueue
       int tmp;

       // First enqueue 3 ints (in this order: 12, 4, 15)
       myQueue.enqueue(12);
       myQueue.enqueue(4);
       myQueue.enqueue(15);
       // Now dequeue an element from this queue
       tmp = myQueue.dequeue();
       cout << tmp << endl;  // Outputs 12 to the display
       // Enqueue integer value -3
       myQueue.enqueue(-3);
       // Return front element from queue without deleting
       tmp = myQueue.peek();
       cout << tmp << endl;  // Output will be 4

       return 0;
}
```

A Class for an Array-Queue (in header file aqueue.h) (1)

```
#ifndef _A_QUEUE
#define _A_QUEUE

class AQueue {
   public:
           AQueue();              // constructor inits curr marker
           // ~AQueue();         .. No destructor  needed
           // void  clear();     .. Deallocating array not possible
           bool  isEmpty() const;     // check if empty queue
           void  enqueue(const int& el); // insert el into queue
           int   dequeue();           // remove el from queue
           int   peek();              // read el from queue-front

   private:
           int  x[MAXSIZE];    // Declare array (to use as queue)
};

#endif
```

Class **AQueue** - Member Fn Definitions (in aqueue.cc)

```
#include "aqueue.h"
```

Priority Queues (take notes in class)