Lecture 10a: Linear Data Structures

CS 70: Data Structures and Program Development

Week of April 04, 2020

Motivation

ChunkyString (a CS70 Classic Linear Data Structure)

- HW 07 with current partner:
  - Plan and write pseudocode
  - Testing
    - 160 past student implementations
    - you try to break them
    - string-001 is a wrapper for `std::string`
- HW 08 with new partner:
  - Combine tests and plans
  - implement the core operations

Standard Linear Data Structures

Motivation

Every problem is different.

Why bother with "standardized" data structures? Arrays, stacks, vectors, trees, etc.

Motivation

Every problem is different.

Why bother with "standardized" data structures? Arrays, stacks, vectors, trees, etc.

We learn about standard data structures because doing so

- Gives us a common vocabulary
- Simplifies design
- Simplifies coding
- Simplifies documentation
- Simplifies maintenance
- Simplifies debugging
- Reusability
- Known performance characteristics

Stacks

Fundamental stack operations

Classically:

- `push(x)` – Add `x` onto top of stack
- `pop()` – Remove and return top of stack
- `isEmpty()` – Is the stack empty?

Fundamental stack operations

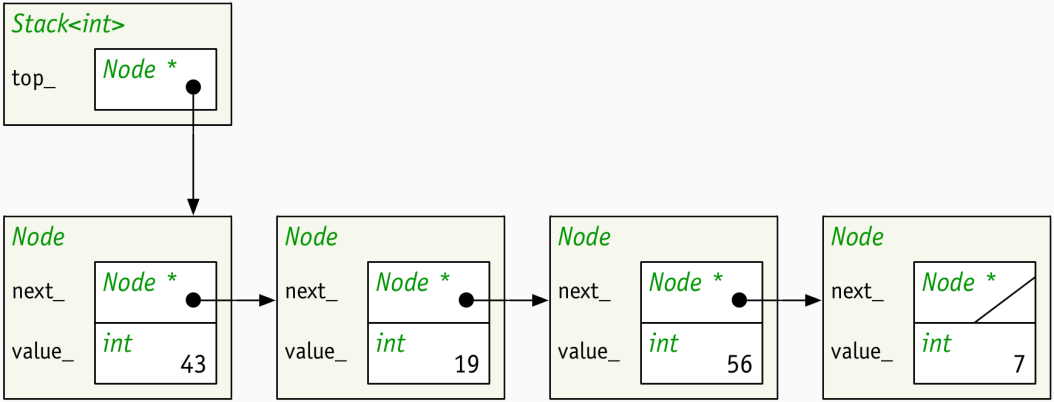
Classically:

- `push(x)` – Add `x` onto top of stack
- `pop()` – Remove and return top of stack
- `isEmpty()` – Is the stack empty?

For C++ `std::stack<T>` (`#include <stack>`)

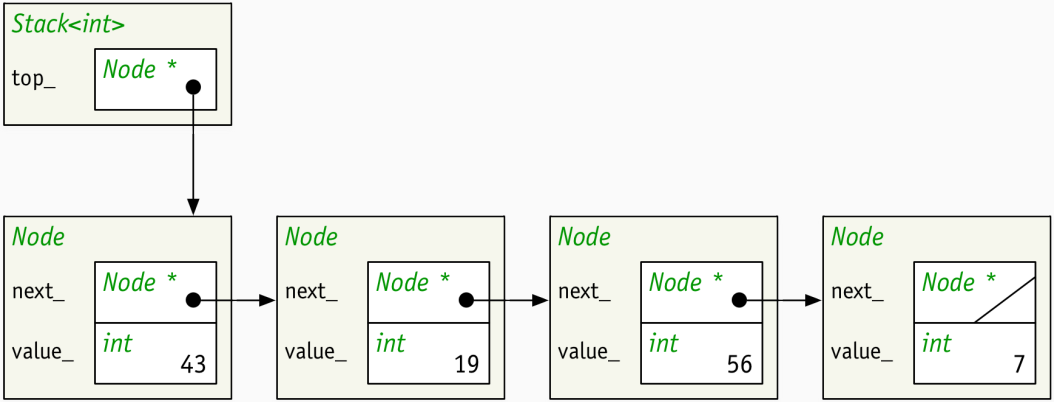
- `push(x)` – Add `x` onto top of stack
- `top()` – access the top element
- `pop()` – discards the top element
- `empty()` – Is the stack empty?

Stack Implementation: Singly Linked List



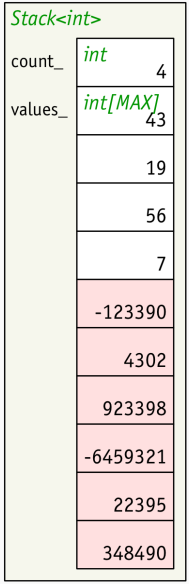
- How to return top element?
- How to pop (remove) top element?

Stack Implementation: Singly Linked List

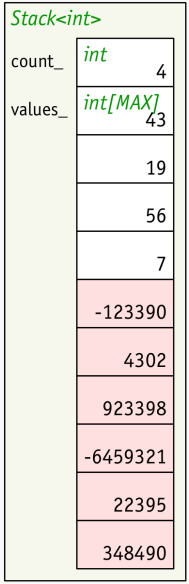


- How to push (insert) new top element?

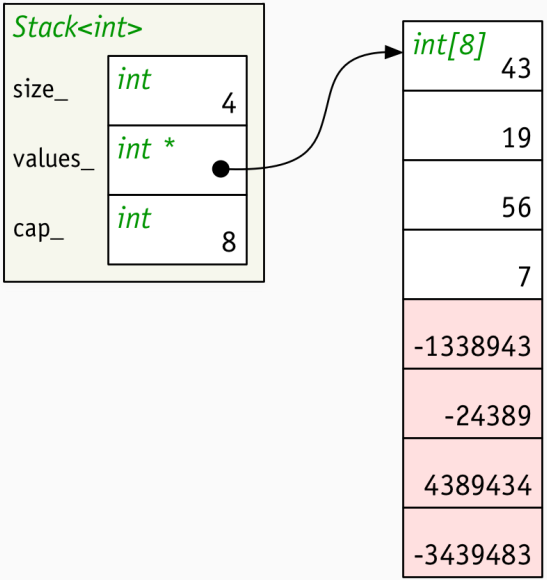
Stack Implementation: Static Array



Stack Implementation: Static Array



Stack Implementation: Dynamic Array



Tradeoffs: Arrays vs. Linked Lists

Arrays

- Time to access arbitrary element?

Arrays

- How much storage overhead?

Arrays

- How hard to do arbitrary insert?

Linked Lists

- Time to access arbitrary element?

Linked Lists

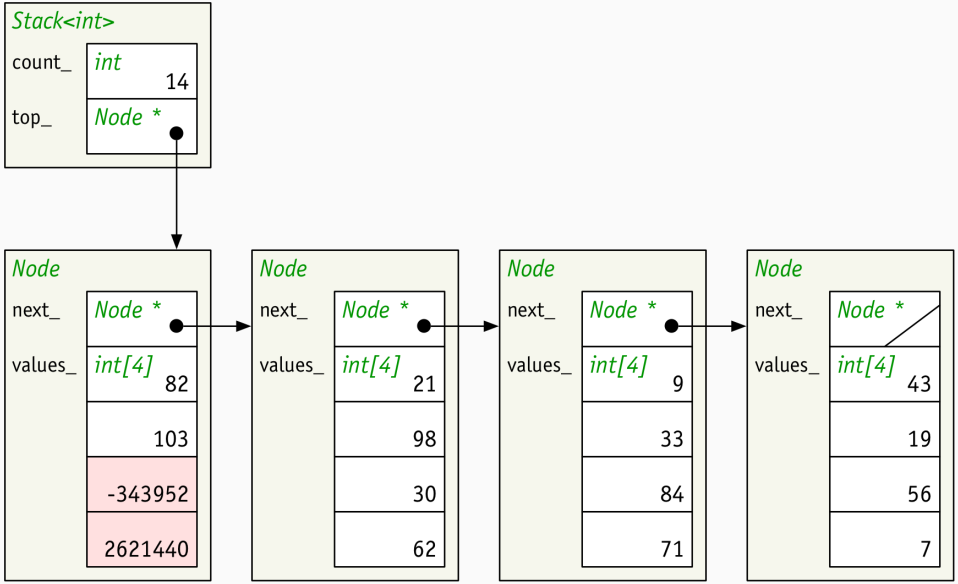
- How much storage overhead?

Linked Lists

- How hard to do arbitrary insert?

The Array / Linked List Spectrum

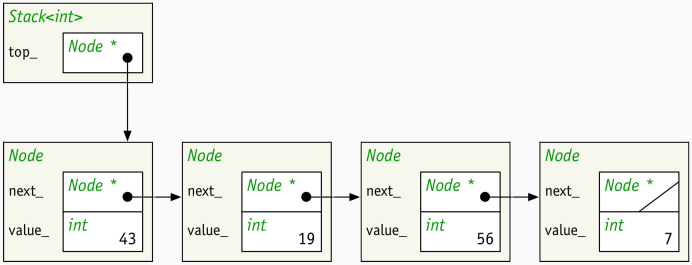
Stack Implementation: Chunky Lists (but why?)



Class Exercise

Suppose we'd like to have iterators for our stacks.

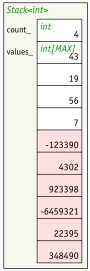
- What data must the iterator contain?
- What would ++ do?
- What would `begin()` and `end()` be?
- What order do we get stack items?
- If we push or pop, do iterators become "invalid"?



Class Exercise

Suppose we'd like to have iterators for our stacks.

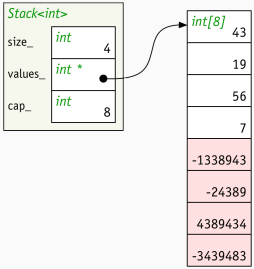
- What data must the iterator contain?
- What would ++ do?
- What would `begin()` and `end()` be?
- What order do we get stack items?
- If we push or pop, do iterators become "invalid"?



Class Exercise

Suppose we'd like to have iterators for our stacks.

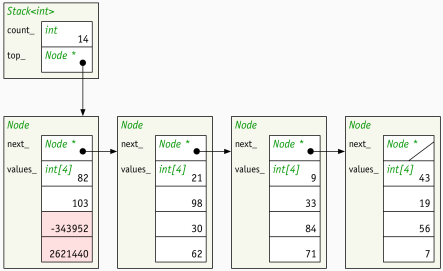
- What data must the iterator contain?
- What would ++ do?
- What would `begin()` and `end()` be?
- What order do we get stack items?
- If we push or pop, do iterators become "invalid"?



Class Exercise

Suppose we'd like to have iterators for our stacks.

- What data must the iterator contain?
- What would ++ do?
- What would `begin()` and `end()` be?
- What order do we get stack items?
- If we push or pop, do iterators become "invalid"?



Queues

Fundamental queue operations

Classically:

- `enqueue(x)` – Add `x` onto back of queue
- `dequeue()` – Remove and return front of queue
- `isEmpty()` – Is the queue empty?

Fundamental queue operations

Classically:

- `enqueue(x)` – Add `x` onto back of queue
- `dequeue()` – Remove and return front of queue
- `isEmpty()` – Is the queue empty?

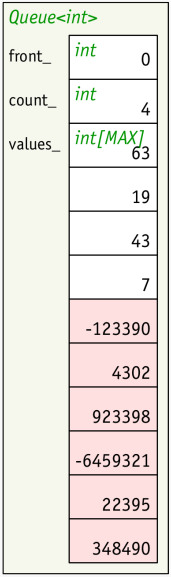
For C++ `std::queue<T>` (`#include <queue>`)

- `push(x)` – adds `x` to the back of the queue
- `front()` – access the front element
- `back()` – access the back element
- `pop()` – discards the front element
- `empty()` – Is the queue empty?

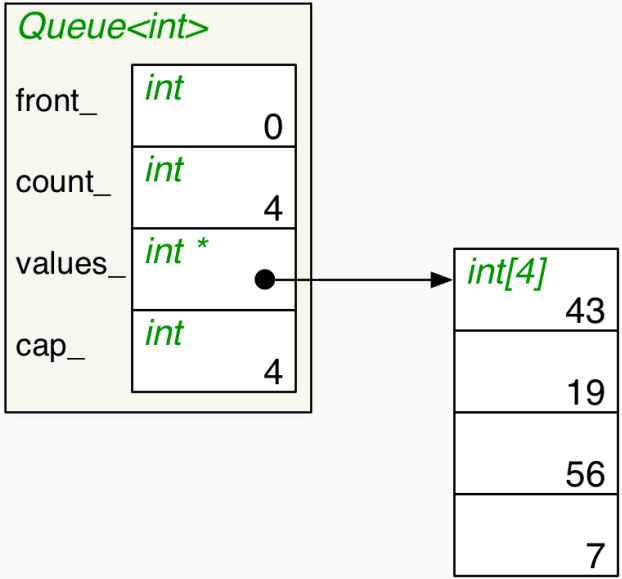
Stack or Queue?

a. FIFO
b. LIFO
c. FILO
d. LILO

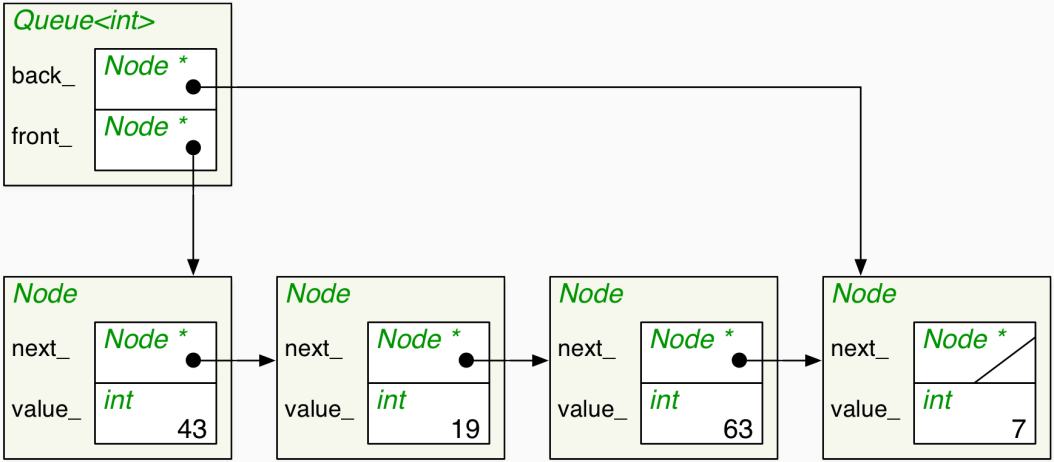
Queue Implementation: Static Array



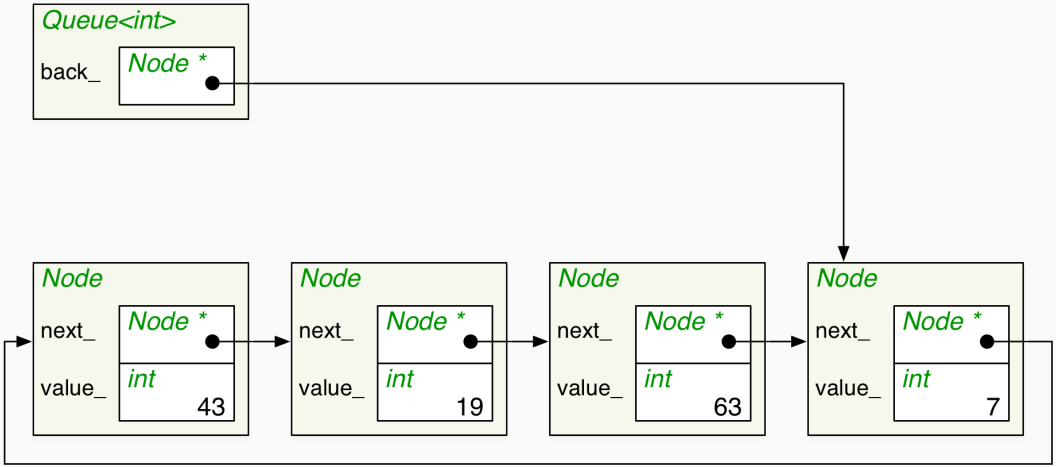
Queue Implementation: Dynamic Array



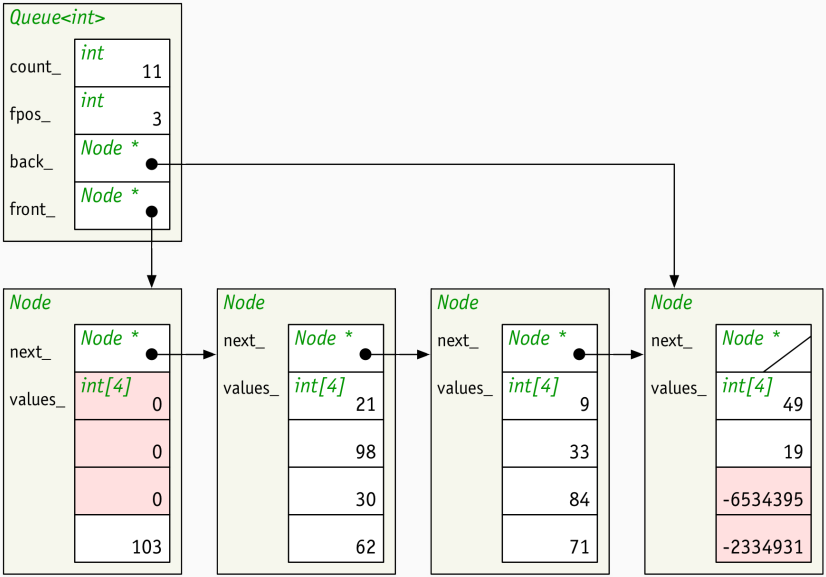
Queue Implementation: Linked List



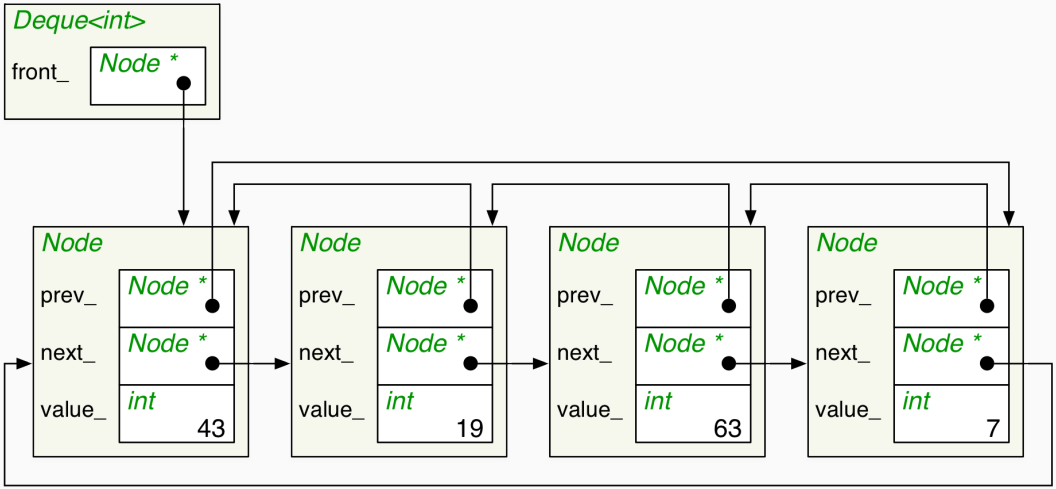
Queue Implementation: Circular Linked List



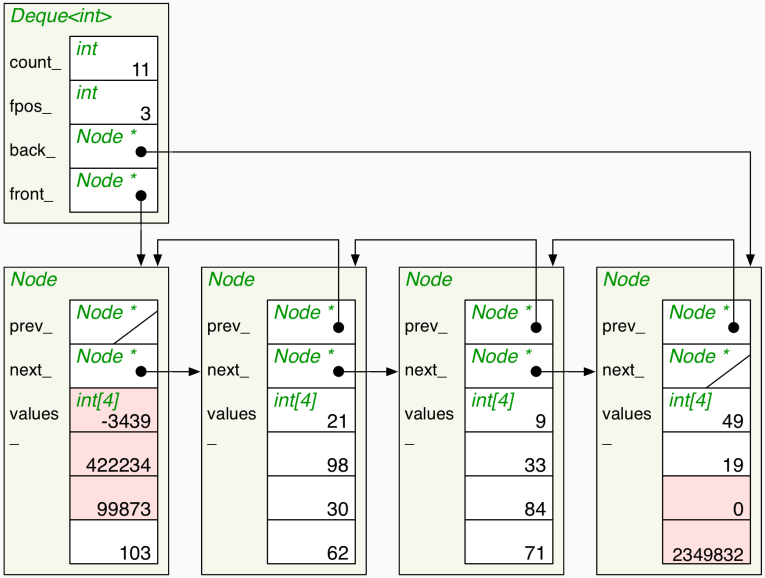
Queue Implementation: Chunky List



Deque (Double Ended Queue): via Circular Linked List



Chunky List


Deque<int>
count_
int
11
fpos_
int
3
back_
Node *
front_
Node *
Node
prev_
Node *
next_
Node *
values
int[4]
-3439
422234
99873
103
Node
prev_
Node *
next_
Node *
values
int[4]
21
98
30
62
Node
prev_
Node *
next_
Node *
values
int[4]
9
33
84
71
Node
prev_
Node *
next_
Node *
values
int[4]
49
19
0
2349832

Lists

A `List` class is more general than any of the previous data structures.

- Usually allows insertion and deletion at any point in the list
- May allow lists to be merged/spliced in constant time

Several varieties:

- Singly linked
- Circular singly linked
- Doubly linked
- Circular doubly linked

Learning Targets

For each of stacks, queues, chunky versions:

1. I can explain the public interface
2. I can describe several different implementation strategies
3. I can identify advantages & disadvantages of different implementations
4. I can design iterators for each implementation.