CSC 212: Data Structures and Abstractions

11: Linked Lists (part 1)

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

Array insertions

0	1	2	3	4	5	6	7	8		n-2	n-1
23	-1	4	5	6	1	3			…		
							size				

`push_back(45)`

0	1	2	3	4	5	6	7	8		n-2	n-1
23	-1	4	5	6	1	3	45		…		
								size			

Computational cost?

Array insertions

0	1	2	3	4	5	6	7	8		n-2	n-1
23	-1	4	5	6	1	3			…		
							size				

`push_front(45)`

0	1	2	3	4	5	6	7	8		n-2	n-1
23	23	-1	4	5	6	1	3		…		
							size				

0	1	2	3	4	5	6	7	8		n-2	n-1
45	23	-1	4	5	6	1	3		…		
								size			

Computational cost?

Array insertions

0	1	2	3	4	5	6	7	8		n-2	n-1
23	-1	4	5	6	1	3			…		
							size				

`insert(4, 45)`

0	1	2	3	4	5	6	7	8		n-2	n-1
23	-1	4	5	6	6	1	3		…		
							size				

0	1	2	3	4	5	6	7	8		n-2	n-1
23	-1	4	5	45	6	1	3		…		
								size			

Computational cost?

(*) repeat the same analysis for remove (front, back, at index)

Memory representation of arrays

Address	Value
…	
0x0A08	
0x0A0C	
0x0A10	
0x0A14	
0x0A18	
0x0A1C	
0x0A20	
0x0A24	
0x0A28	
0x0A2C	
0x0A30	
0x0A34	
0x0A38	
0x0A3C	
0x0A40	
0x0A44	
0x0A48	
0x0A4C	
0x0A50	
0x0A54	
…	

…								
0x0A08								
0x0A28								
0x0A48								
0x0A68								
0x0A88								
0x0AA8								
0x0AC8								
…								

```
int var1;
int var2[7];
```

What happens to dynamic arrays?

Linked lists

Linked lists

- Definition
  - ✓ a linked list is a **linear data structure** in which elements (called **nodes**) are stored at **non-contiguous** memory locations
  - ✓ each node contains **data** and typically a **pointer** to the next node in the sequence
- Operations
  - ✓ **insert**: add a new node (at the front, rear, specific index, or by value)
  - ✓ **delete**: remove a node (from the front, rear, specific index, or by value)
  - ✓ **search**: find a node containing a specific value
  - ✓ **get**: retrieve the value at a specific position (requires traversal)
  - ✓ **traverse**: sequentially "visit" each node in the list

Memory representation of linked lists

…								
0x0A08		3		33	44	6		
0x0A28								
0x0A48				-5			18	
0x0A68		21						
0x0A88								
0x0AA8					32			
0x0AC8								
		…						

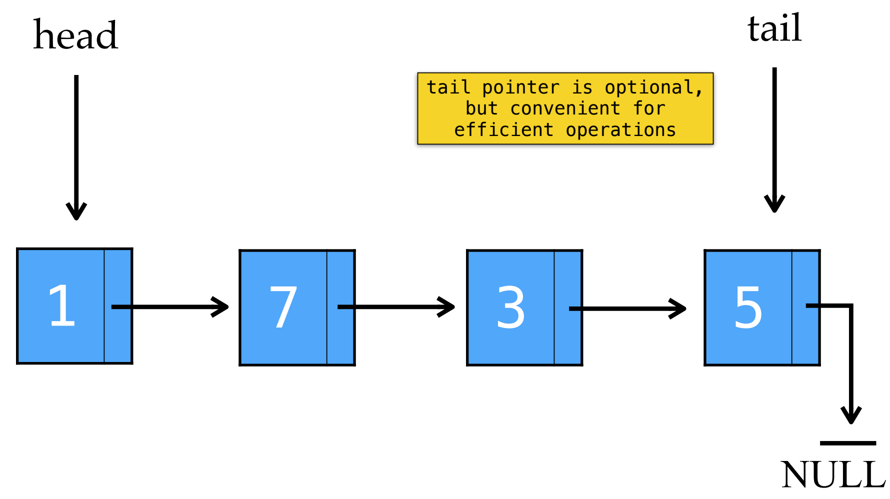
```
{3, -5, 6, 18, 21, 33, 44, 32}
```

Types of linked lists

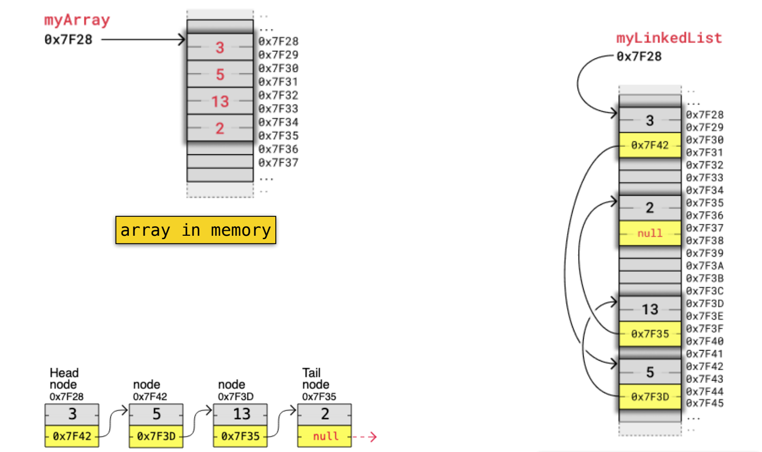
- Singly-linked list
  - each **node** contains a value and a pointer to the next node
  - the first node is called the **head**, the last node is the **tail**
  - the tail node points to **null**
  - the **length** of the linked list is the number of nodes
  - enables traversal only from the head towards the tail

- Doubly-linked list
  - each **node** contains a value, a pointer to the next node, a pointer to the previous node
  - the first node is called the **head**, the last node is the **tail**
  - the head node's previous pointer and the tail node's next pointer are **null**
  - the **length** of the linked list is the number of nodes
  - enables traversal in both directions

Singly-linked list



Singly-linked list and memory



array in memory

visual representation of a linked list

linked list in memory

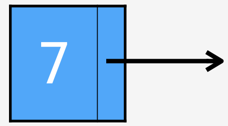
https://www.w3schools.com/dsa/dsa_theory_linkedlists_memory.php

Implementing a linked list

Representing a node

```
template <typename T>
struct Node {
    T data;
    Node<T> *next;

    Node(const T& value) {
        data = value;
        next = nullptr;
    }
};
```



`struct` representing a node in a linked list using templates. It contains a <u>value</u> of type `T`, a <u>pointer</u> to the next node, and a <u>constructor</u> that initializes the value and sets the next pointer to `nullptr`

Representing a singly-linked list

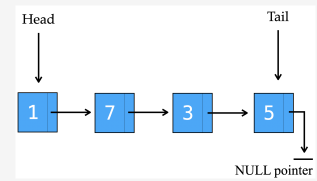
```
template <typename T>
class SLList {

    private:
        struct Node {
            T data;
            Node *next;
            Node(const T& value) { data = value; next = nullptr; }
        };

        Node *head;
        Node *tail;
        size_t size;

    public:
        SLList() { head = tail = nullptr; size = 0; }
        ~SLList() { clear(); }

        size_t get_size() { return size; }
        bool empty() { return size == 0; }
        void clear();
        T& front();
        T& back();
        void push_front(const T& value);
        void pop_front();
        void push_back(const T& value);
        void pop_back();
        void print();
};
```



Methods

- constructor
  - ✓ invoked automatically
  - ✓ initializes an empty list
  - ✓ sets head and tail to nullptr and size to zero
- destructor
  - ✓ invoked automatically
  - ✓ calls clear() to delete all dynamically allocated nodes
- clear()
  - ✓ traverse the list and deletes each node
  - ✓ resets head and tail to nullptr and size to zero

Methods

- get_size()
  - ✓ returns the current number of nodes in the list
- empty()
  - ✓ returns true if the list is empty, false otherwise
- front()
  - ✓ throws an exception if the list is empty
  - ✓ otherwise, returns the value stored in the <u>first</u> node
- back()
  - ✓ throws an exception if the list is empty
  - ✓ otherwise, returns the value stored in the <u>last</u> node

Methods

- push_back(value)
  - ✓ creates a new node containing the given value
  - ✓ adds the node to the end of the list
  - ✓ updates all necessary pointers, including the tail
  - ✓ increments the size counter
  - ✓ if the list was empty, the new node becomes both head and tail
- pop_back()
  - ✓ throws an exception if the list is empty
  - ✓ traverses the list to find the second-to-last node
  - ✓ removes the last node from the list and frees its memory
  - ✓ updates all necessary pointers, with tail now pointing to the second-to-last node
  - ✓ decrements the size counter
  - ✓ if the list becomes empty, sets head and tail to nullptr

Methods

- push_front(value)
  - ✓ creates a new node containing the given value
  - ✓ adds the node to the beginning of the list
  - ✓ updates all necessary pointers, including the head
  - ✓ increments the size counter
  - ✓ if the list was empty, the new node becomes both head and tail
- pop_front()
  - ✓ throws an exception if the list is empty
  - ✓ removes the first node from the list and frees its memory
  - ✓ updates all necessary pointers, with head now pointing to the second node
  - ✓ decrements the size counter
  - ✓ if the list becomes empty, sets head and tail to nullptr

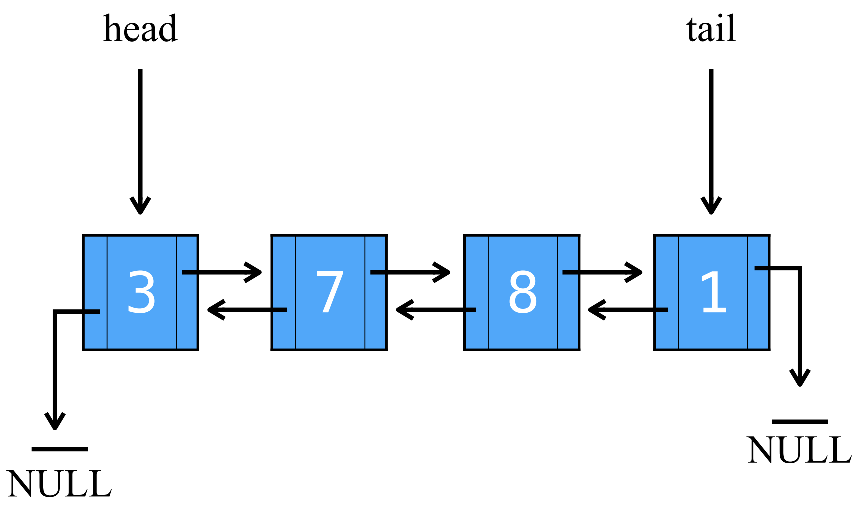
Methods

- print
  - ✓ uses a temporary pointer to traverse the list starting from the head
  - ✓ prints the value stored in each node during traversal
- search
  - ✓ uses a temporary pointer to traverse the list starting from the head
  - ✓ compares each node’s value with the target value
  - ✓ returns true if the value is found; otherwise, returns false

Practice

- Provide the computational cost for each of the operations listed in the previous slides
- Design, implement and provide the computational cost for the following methods
  - ✓ insert_at(index, value)
  - ✓ remove_at(index)

Doubly linked list



Practice

- Implement a class DoublyLinkedList
  - ✓ include all the methods seen for the singly linked list
- Complete the following table with rates of growth
  - ✓ assume all linked lists use a “tail” pointer

Operation	Dynamic Array	Singly-linked list	Doubly-linked list
Append 1 element			
Remove 1 element from the end			
Insert 1 element at index idx			
Remove 1 element from index idx			
Read element from index idx			
Write (update) element at index idx			