CSC 212: Data Structures and Abstractions

11: Linked Lists

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

Practice

- Assuming that a character array starts at address `0x0100`
  - label the memory addresses of all elements

'h'	'e'	'l'	'l'	'o'	0				

- Assuming that an integer array starts at address `0x0100`
  - label the memory addresses of all elements

3	-12	2	4	10	20	22			

Practice

- Assume a dynamic array and efficient implementations
  - what is the cost of inserting 1 element at the end?
  - what is the cost of inserting 1 element at the front?
  - what is the cost of inserting 1 element at index `idx`?
  - what is the cost of performing deletions at those same locations?

Linked lists

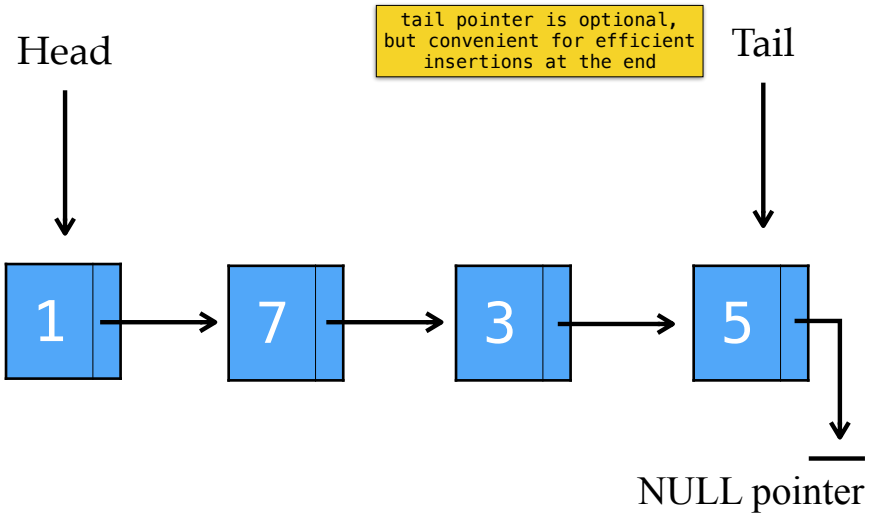
Linked list

- Definition
  - a linked list is a **linear data structure** that consists of a sequence of elements stored at **non-contiguous** locations in memory
- Typical operations
  - **insert**: add a new node to the list (rear, front, at index, by value)
  - **delete**: remove a node from the list (rear, front, at index, by value)
  - **search**: find a node with a specific value
  - **get**: get a value at an specific index
  - **traverse**: "visit" each node in the list

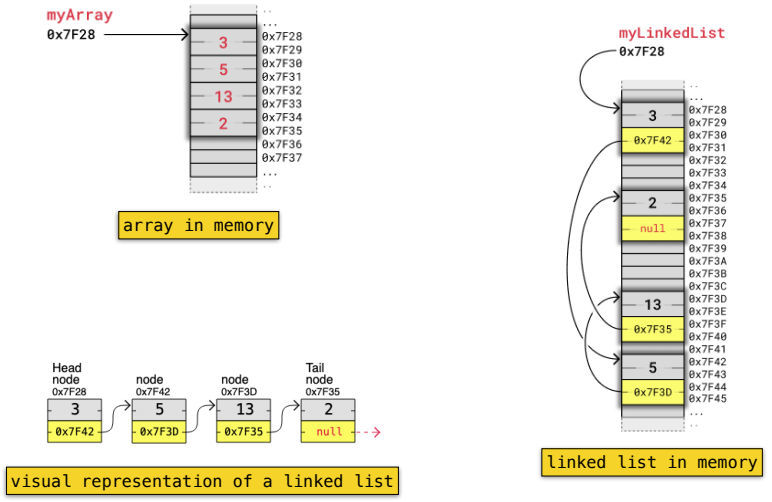
Linked lists

- Types of linked lists
  - **singly-linked list**: each node has a pointer to the next node
  - **doubly-linked list**: each node has a pointer to the next and previous nodes
  - **circular-linked list**: the last node has a reference to the first node
- Singly-linked list
  - each element is a **node** that contains a value and a pointer to a next node
  - the last node has a reference to **null**
  - the first node is called the **head**
  - the last node is called the **tail**
  - the length of the linked list is the number of nodes

Singly-linked list



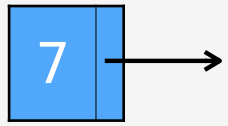
Singly-linked list and memory



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

Implementing a linked list

Representing a node




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

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

Representing a singly-linked list

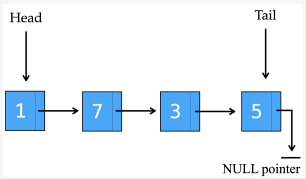
```
template <typename T>
class SLList {

    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
  - initialize head, tail, and size
  - constructor is (automatically) called once when the object is created
- destructor
  - call clear to delete all nodes
  - destructor is (automatically) called once when the object is destroyed
- clear
  - traverse the list deleting (freeing) all nodes
  - initialize head, tail, and size
  - clear can be called multiple times

Methods

- get_size
  - return the current number of nodes in the list
- empty
  - return true if the list is empty, false otherwise
- front
  - throw an exception if the list is empty
  - return the value of the first node
- back
  - throw an exception if the list is empty
  - return the value of the last node

Methods

- push_back
  - create a new node with the given value
  - add the node to the end of the list — adjust 'next' pointer
  - update the tail pointer
  - increment the size
- pop_back
  - throw an exception if the list is empty
  - delete (free) the last node from the list — adjust 'next' pointer
  - update the tail pointer — requires $O(n)$ traversal
  - decrement the size

Methods

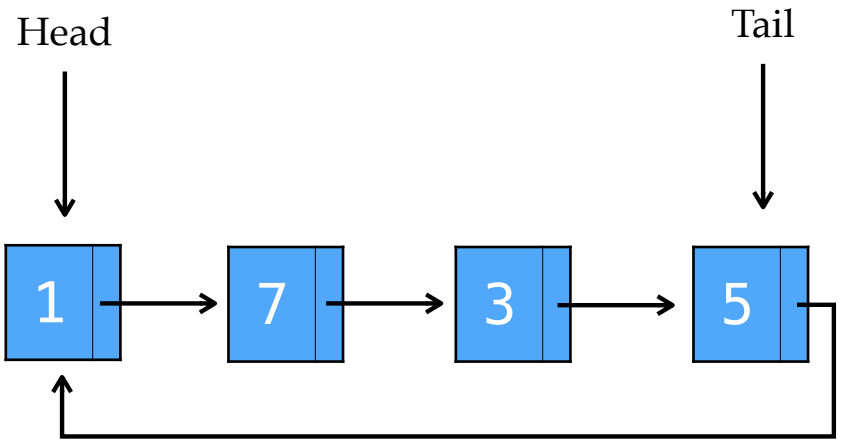
- push_front
  - create a new node with the given value
  - add the node to the beginning of the list -- adjust next pointer
  - update the head pointer
  - increment the size
- pop_front
  - throw an exception if the list is empty
  - remove the first node from the list
  - update the head pointer
  - decrement the size

Methods

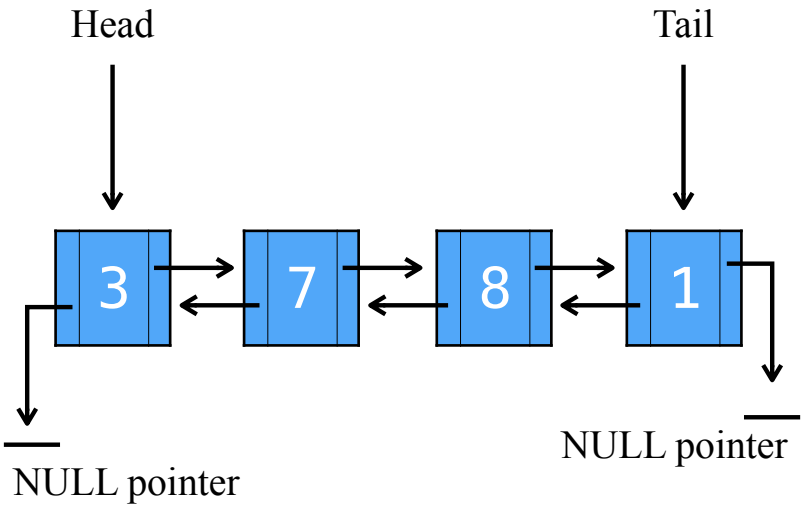
- print
  - use a temporary pointer to traverse the list starting from the head
  - print the value of each node
- search
  - use a temporary pointer to traverse the list starting from the head
  - compare the value of each node with the target value
  - return true if the value is found, false otherwise

Other types of linked lists

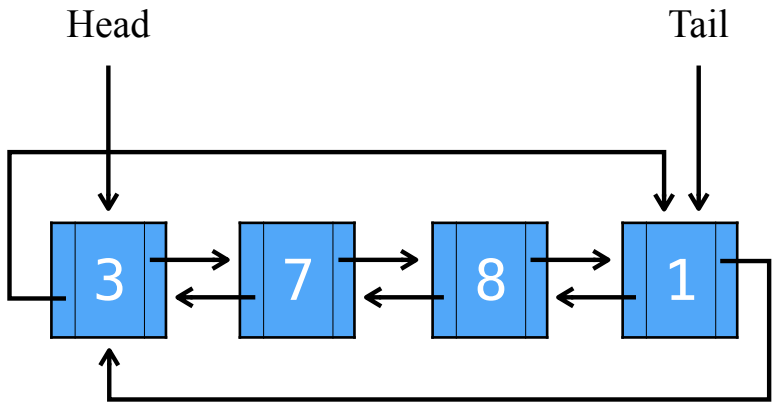
Circular singly linked list



Doubly linked list



Circular doubly linked list

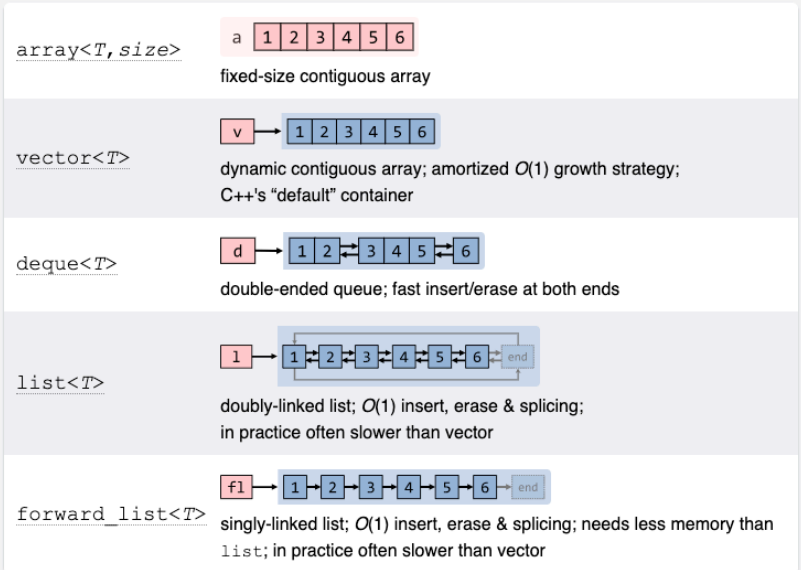


Practice

- Complete the following table with rates of growth
  - assume 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			

Linked lists in the STL



https://hackingcpp.com/cpp/std/sequence_containers.html

Collection	Description	Implementation	Random Access	Insertion/Deletion	Memory Overhead
`std::string`	character sequence	contiguous memory	O(1)	O(n) at arbitrary positions; O(1) amortized at end	low
`std::array`	fixed-size array	contiguous memory	O(1)	Not designed for insertion/deletion	none
`std::vector`	dynamic array	contiguous memory	O(1)	O(n) at arbitrary positions; O(1) amortized at end	low
`std::deque`	double-ended queue	segmented array	O(1)	O(1) at both ends; O(n) in middle	medium
`std::list`	doubly-linked list	non-contiguous nodes	O(n)	O(1) at any position with iterator	high
`std::forward_list`	singly-linked list	non-contiguous nodes	O(n)	O(1) at front; O(n) elsewhere	medium

Others

Linked lists and other data structures

- Stacks
  - insert and remove from the same end
  - constant time complexity for both operations
- Queues
  - insert at one end and remove from the other end
  - constant time complexity for both operations
- Deques
  - insert and remove from both ends
  - constant time complexity for all insert/remove operations

Run this code

```
import time

n = 100000

start = time.time()
array = []
for i in range(n):
    array.append('s')
print(time.time() - start)

start = time.time()
array = []
for i in range(n):
    array = array + ['s']
print(time.time() - start)
```


Practice

- Why the difference in time?

How are lists implemented in CPython?

CPython's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list `a[i]` an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

CPython is the reference implementation of the Python programming language (primarily written in C)

https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython