

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 0×0100
 - label the memory addresses of all elements

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

- Assuming that an integer array starts at address 0×0100
 - label the memory addresses of all elements

3	-12	2	4	10	20	22			
---	-----	---	---	----	----	----	--	--	--

2

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?

3

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

5

Linked lists

Types of linked lists

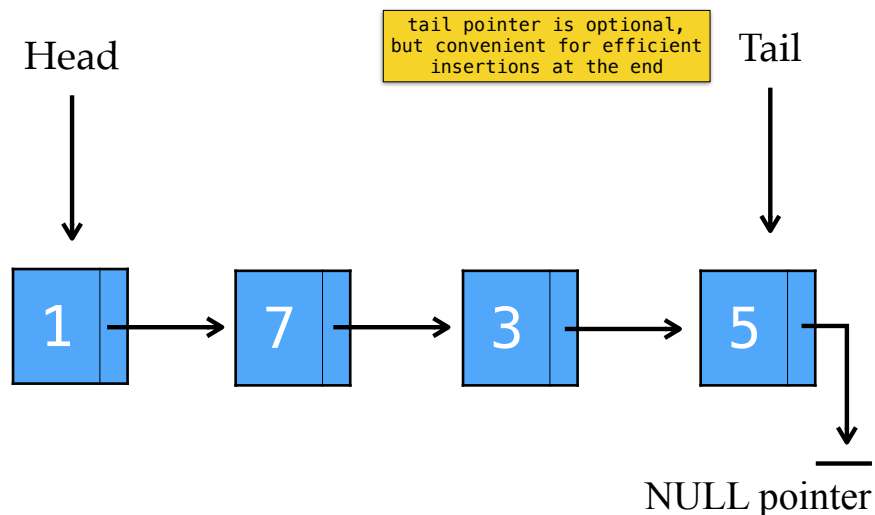
- ✓ **singly-linked list**: each node has a pointer to the next node
- ✓ **doubly-linked list**: each node has a pointer to the next and previous nodes
- ✓ **circular-linked list**: the last node has a reference to the first node

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

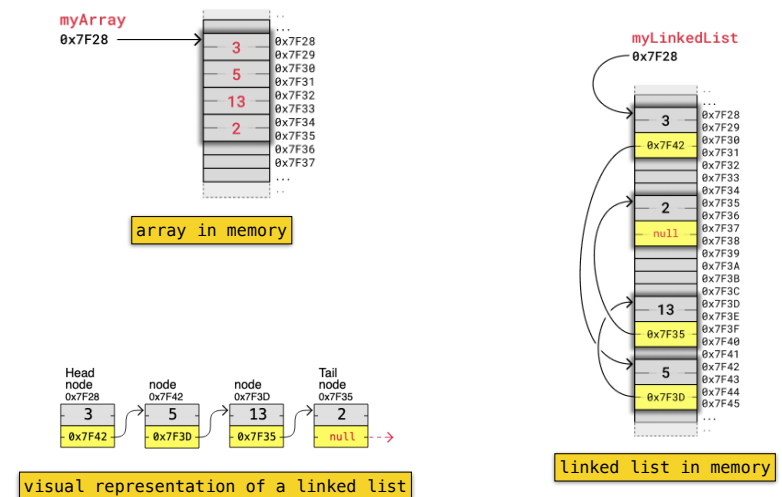
6

Singly-linked list



7

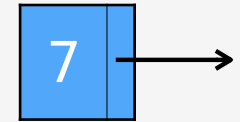
Singly-linked list and memory



8

Implementing a linked list

Representing a node



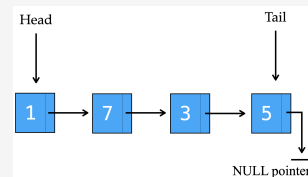
```
struct Node {  
    T data;  
    Node *next;  
    Node(const T& value) {  
        data = value; next = nullptr;  
    }  
};
```

`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`

10

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();  
};
```



11

Methods

- Constructor
- Destructor
- clear

12

Methods

- `get_size`
- `empty`
- `front`
- `back`

13

Methods

- `push_back`
- `pop_back`

14

Methods

- `push_front`
- `pop_front`

15

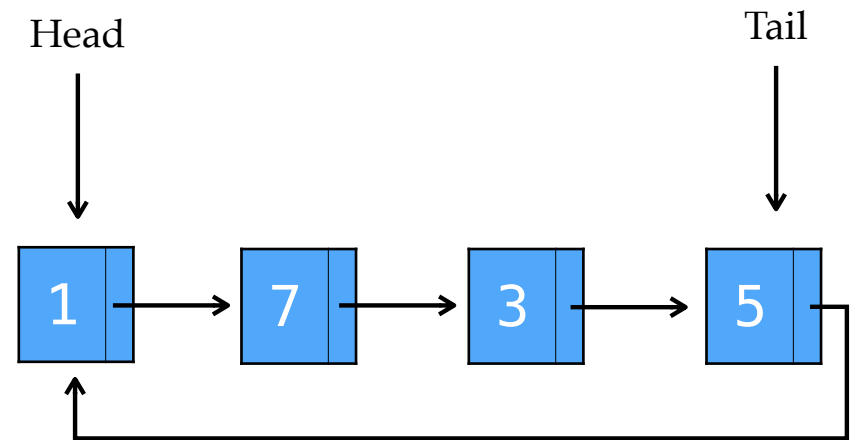
Methods

- `print`
- `search`

16

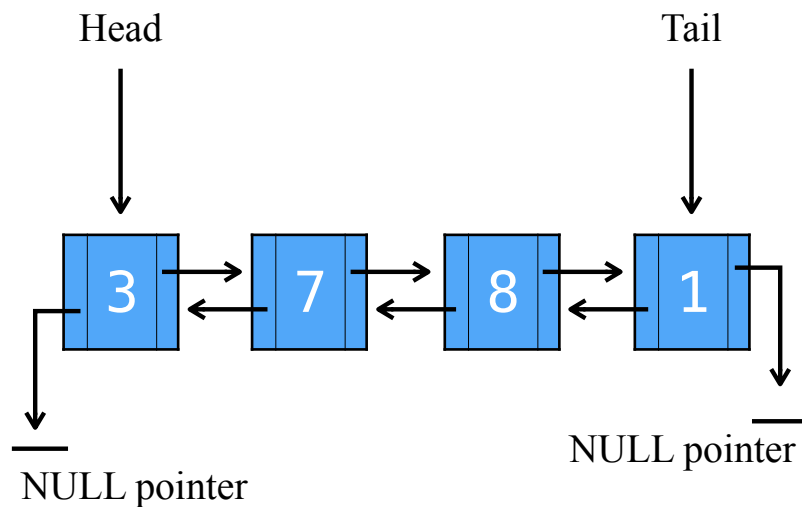
Other types of linked lists

Circular singly linked list



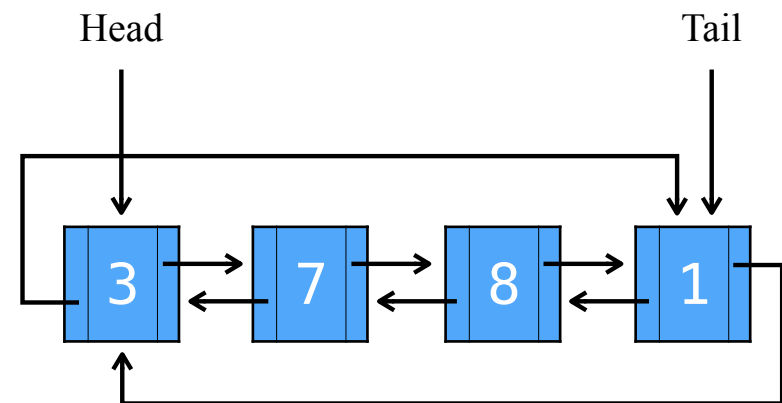
18

Doubly linked list



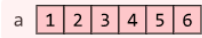

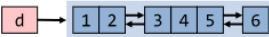
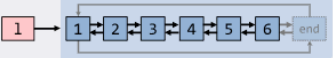
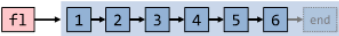
19

Circular doubly linked list



20

Linked lists in the STL

<code>array<T, size></code>	
	fixed-size contiguous array
<code>vector<T></code>	
	dynamic contiguous array; amortized $O(1)$ growth strategy; C++'s "default" container
<code>deque<T></code>	
	double-ended queue; fast insert/erase at both ends
<code>list<T></code>	
	doubly-linked list; $O(1)$ insert, erase & splicing; in practice often slower than vector
<code>forward_list<T></code>	
	singly-linked list; $O(1)$ insert, erase & splicing; needs less memory than <code>list</code> ; in practice often slower than vector

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

22

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

Dequeues

- ✓ insert and remove from both ends
- ✓ constant time complexity for all insert/remove operations

24

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)
```

25

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>