data_tructures(&algorithms, lecture02)

Doan Trung Tung, PhD – University of Greenwich (Vietnam)

Plan

Algorithm Analysis
Measuring algorithms by time, big-O analysis

01

Array popular algorithms
Searching, counting, accumulating, copying, …

02

Linked List
Definition, presentation, popular operations

03

Advanced Linked List
Doubly Linked List, Circular Linked List

04

Algorithms Analysis

Count execution, big-O, recursion analysis, ..

Algorithmic Performance

- There are two aspects of algorithmic performance:
  - Time
    - Instructions take time.
    - How fast does the algorithm perform?
    - What affects its runtime?
  - Space
    - Data structures take space
    - What kind of data structures can be used?
    - How does choice of data structure affect the runtime?

Analysis of Algorithms

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of specific implementations, computers, or data.
- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

Execution Time of Algorithms

- Each operation in an algorithm (or a program) has a cost.
  - Each operation takes a certain of time.

`count = count + 1;` ➔ take a certain amount of time, but it is constant

  - Sequence of operations

```
count = count + 1;
sum = sum + count;
```

Cost: c_1
Cost: c_2

➔ Total Cost = $c_1 + c_2$

Execution Time of Algorithms

- Simple If-Statement

Code	Cost	Times
`if (n < 0)`	c1	1
`absval = -n`	c2	1
`else`		
`absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

Execution Time of Algorithms

❖ Simple Loop

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

Execution Time of Algorithms

❖ Nested Loop

```
                                   Cost          Times
i=1;                               c1             1
sum = 0;                           c2             1
while (i <= n) {                   c3             n+1
    j=1;                           c4             n
    while (j <= n) {               c5             n*(n+1)
        sum = sum + i;             c6             n*n
        j = j + 1;                 c7             n*n
    }
    i = i +1;                      c8             n
}
```

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to n^2

General rules for estimation

- Loops: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- Nested Loops: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.
- Consecutive Statements: Just add the running times of those consecutive statements.
- If/Else: Never more than the running time of the test plus the larger of running times of S1 and S2.

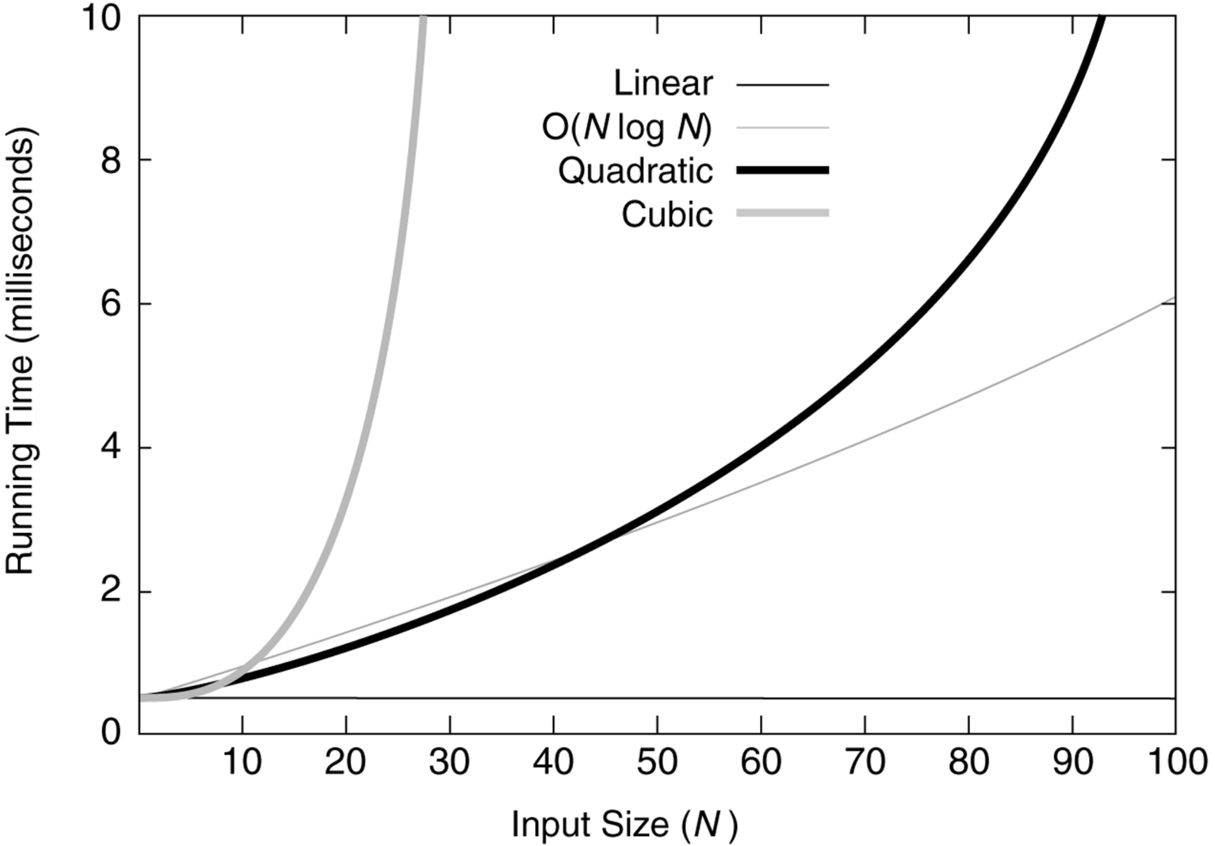
Algorithm Growth Rates

- We measure an algorithm's time requirement as a function of the *problem size n*.
  - Algorithm A requires $5*n^2$ time units to solve a problem of size n.
  - Algorithm B requires $7*n$ time units to solve a problem of size n.
- The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.
  - Algorithm A requires time proportional to n^2.
  - Algorithm B requires time proportional to n.
- An algorithm's proportional time requirement is known as ***growth rate***.

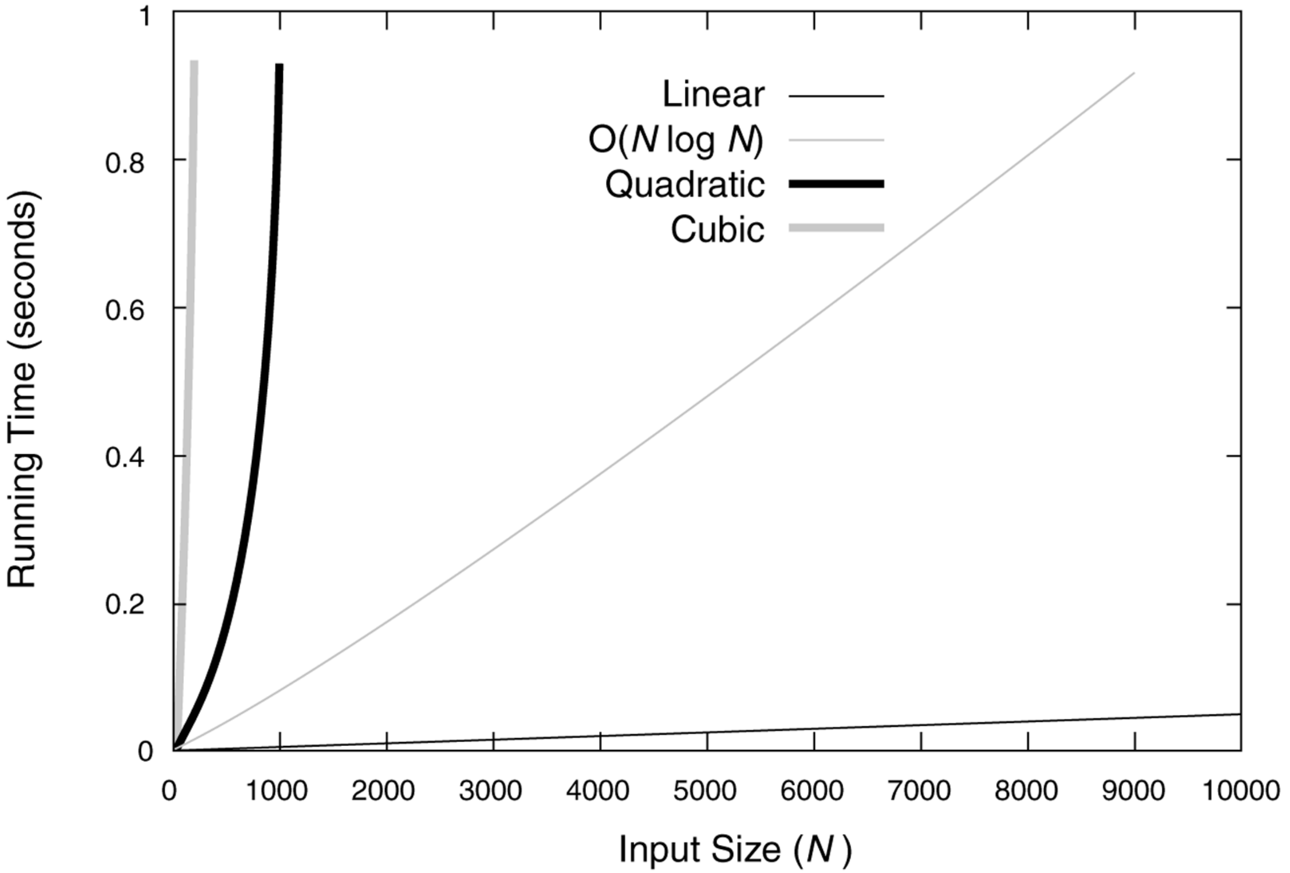
Common Growth Rates

Function	Growth Rate Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Common Growth Rates



Common Growth Rates



Big-O Notation

- If Algorithm A requires time proportional to f(n), Algorithm A is said to be order f(n), and it is denoted as O(f(n)).
- The function f(n) is called the algorithm's growth-rate function.
- If Algorithm A requires time proportional to n^2, it is O(n2).
- If Algorithm A requires time proportional to n, it is O(n).

Big-O Notation

Definition:

Algorithm A is order f(n) – denoted as O(f(n)) – if constants k and n_0 exist such that A requires no more than k*f(n) time units to solve a problem of size n $\geq n_0$.

The requirement of **n $\geq$ n_0** in the definition of O(f(n)) formalizes the notion of sufficiently large problems.

In general, many values of k and n can satisfy this definition.

Big-O Notation

If an algorithm requires $n^2 – 3*n+10$ seconds to solve a problem size n. If constants k and n_0 exist such that

$$k*n^2 > n^2 – 3*n+10 \quad \text{for all } n \geq n_0 .$$

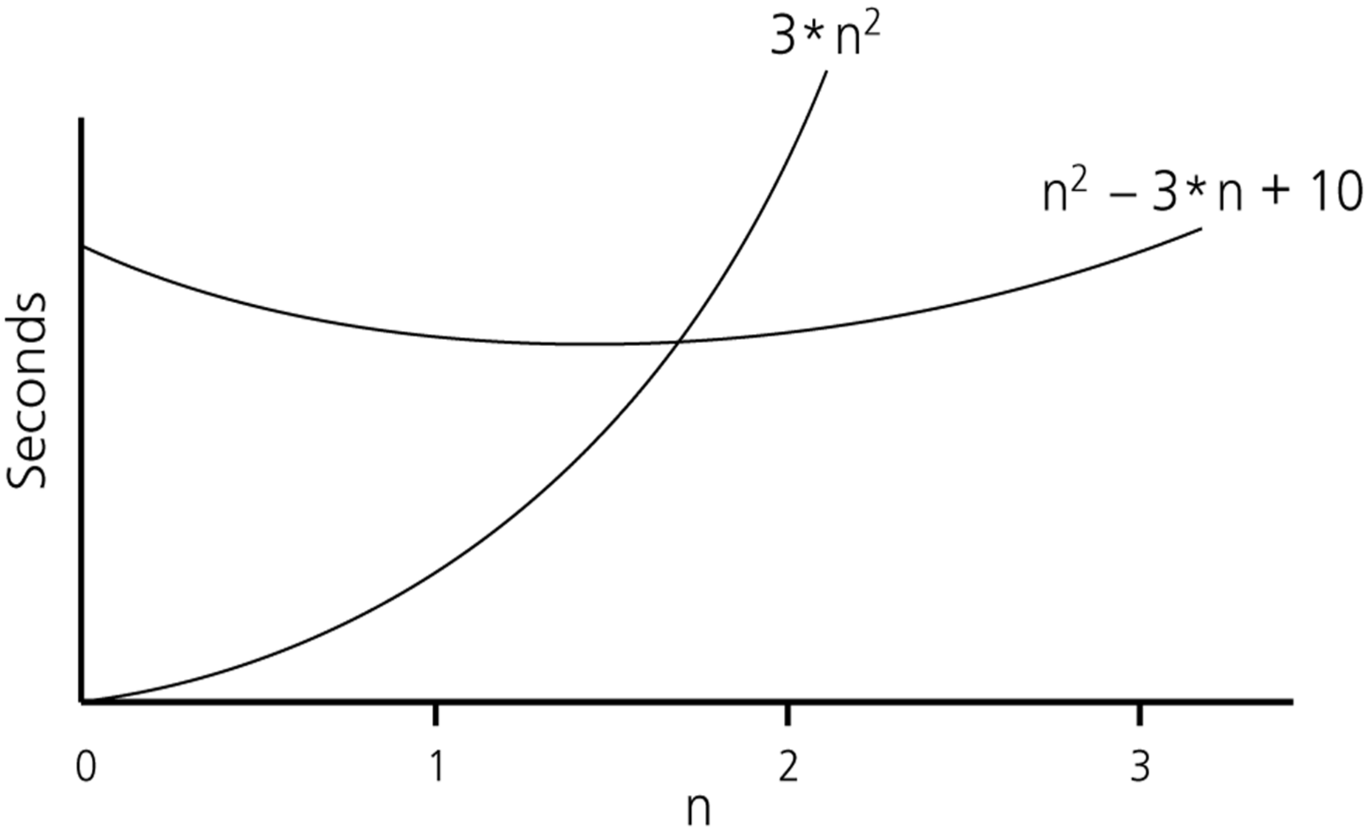
the algorithm is order n^2

(In fact, k is 3 and n_0 is 2, maybe more)

$$3*n^2 > n^2 – 3*n+10 \quad \text{for all } n \geq 2 .$$

Thus, the algorithm requires no more than $k*n^2$ time units for $n \geq n_0$ so it is **O(n^2)**

Big-O Notation

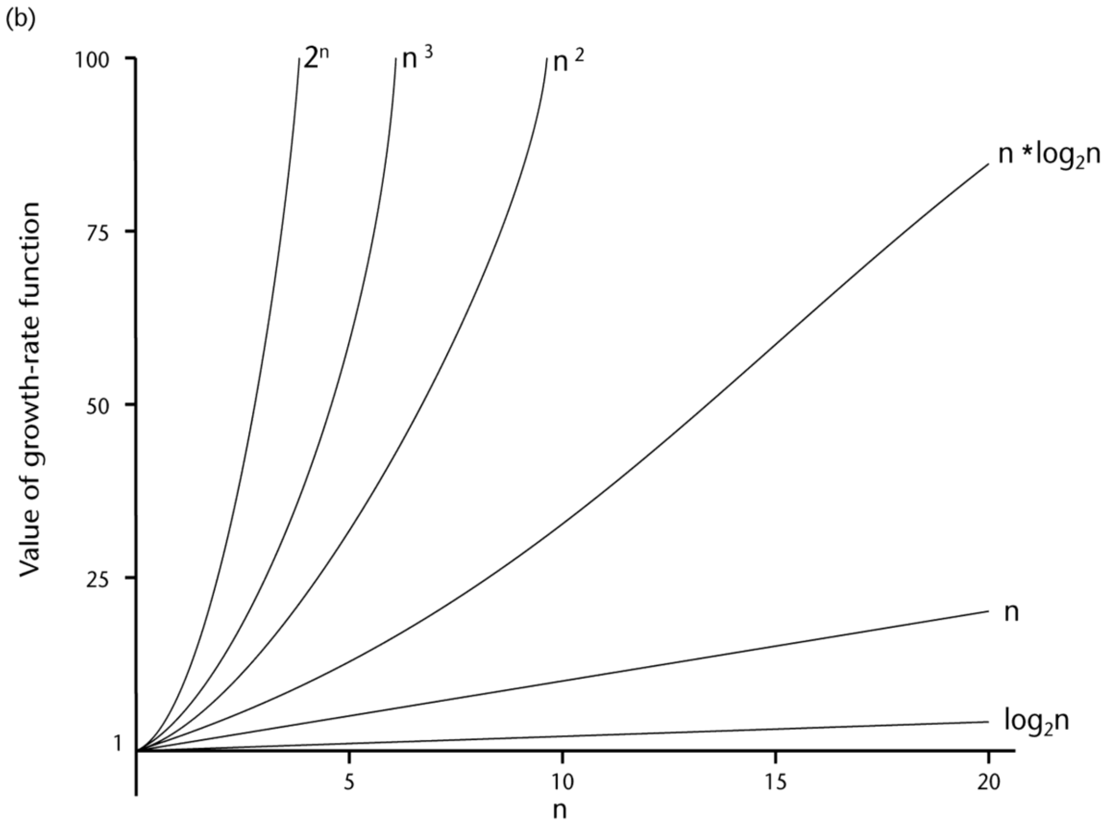


Popular Growth-Rate Functions

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Popular Growth-Rate Functions



Popular Growth-Rate Functions

$O(1)$	Time requirement is **constant**, and it is independent of the problem's size.
$O(\log_2 n)$	Time requirement for a **logarithmic** algorithm increases slowly as the problem size increases.
$O(n)$	Time requirement for a **linear** algorithm increases directly with the size of the problem.
$O(n*\log_2 n)$	Time requirement for a **$n*\log_2 n$** algorithm increases more rapidly than a linear algorithm.
$O(n^2)$	Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.
$O(n^3)$	Time requirement for a c**ubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.
$O(2^n)$	As the size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

Properties of Growth-Rate Functions

- We can ignore low-order terms in an algorithm's growth-rate function.
  - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
- We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.
  - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.
- $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
  - If an algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 + 4n^2) \rightarrow$ So, it is $O(n^3)$.

Counting & Growth-Rate Functions

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*c5
= (c3+c4+c5)*n + (c1+c2+c3)
= a*n + b

➔ So, the growth-rate function for this algorithm is **O(n)**

Counting & Growth-Rate Functions

Code	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`  i = i +1;`	c8	n
`}`		

$$
\begin{aligned}
T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8 \\
&= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3) \\
&= a*n^2 + b*n + c
\end{aligned}
$$

➔ So, the growth-rate function for this algorithm is **$O(n^2)$**

Counting recursive function

Algorithm BINARY-SEARCH (A, lo, hi, x)

if (lo > hi)

return FALSE ← c1

mid ← $\lfloor$(lo+hi)/2$\rfloor$

if x = A[mid]

return TRUE ← c2

if (x < A[mid])

BINARY-SEARCH (A, lo, mid-1, x) ← T(n/2)

if (x > A[mid])

BINARY-SEARCH (A, mid+1, hi, x) ← T(n/2)

=> T(n) = c + T(n/2)

$T(n) = c + T(n/2)$

$= c + c + T(n/4)$

$= c + c + c + T(n/8)$

Replace $n = 2^k$

$T(n) = c + c + \ldots + c + T(1)$

$= c\log n + T(1)$

So we have $T(n) = O(\log n)$

Analyze recursive function

```
Function factorial(n)
Begin
        if n = 0 then return 1
        else  return n*factorial(n-1);
End.
```

$T(0) = c$

$T(n) = b + T(n - 1)$

$= b + b + T(n - 2)$

$= b + b + b + T(n - 3)$

…

$= kb + T(n - k)$

Replace $k = n$, we have:

$T(n) = nb + T(n - n)$

$= bn + T(0)$

$= bn + c.$

So $T(n) = O(n)$.

Array algorithms

Popular & advanced array algorithms

Popular array algorithms

- Sum of all elements
- Search for an element
- Count number of appearances
- Delete one element
- Insert one element
- Find min/max element
- Reverse elements
- Sort elements ascending / descending

Advanced array algorithms

- Re-arrange elements based on condition
- Rotate an array left / right by k elements
- Find duplicate numbers
- Remove duplicate numbers
- Check if an array is a subset of another array

Linked List

Single Linked List, Double Linked List, Circular Linked List

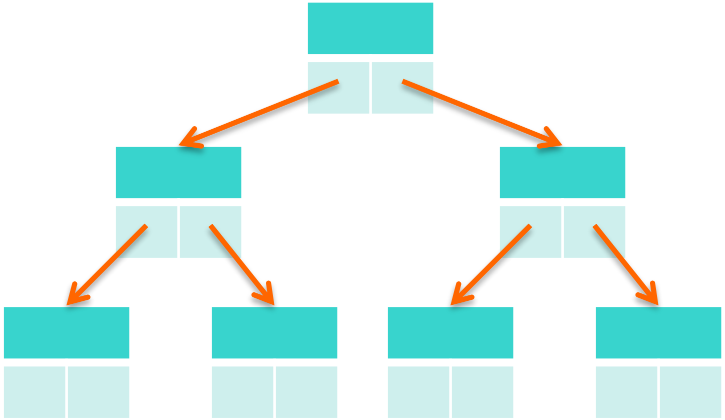
Drawbacks of Arrays

- Array is very useful data structure in many situations.
- However, it has some limitations
  - Fixed size
  - Need size information for creation
  - Inserting an element in the middle of an array leads to moving other elements around
  - Deleting an element from the middle of an array leads to moving other elements around
- Other data structures are more efficient for these cases

Self-referential structures

- Many dynamic data structures are implemented through the use of a self-referential structure
- A self-referential structure is an object, one of this object member is a reference to another object of its own type.
- With this arrangement, it's possible to create 'chains' of data of varying form

Linked List



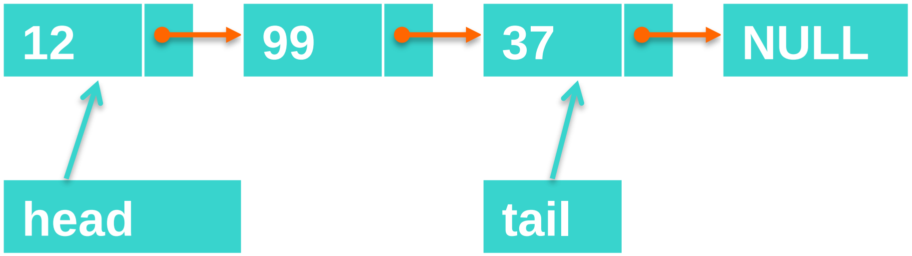
Tree

Linked List

- A collection of nodes storing data and links to other nodes
- A linear data structure composed of nodes
- Each node holds some info and reference to another node in the list
- Types of linked lists
  - Single linked list
  - Double linked list
  - Circular linked list

Single linked lists

- Its node contains two data fields: info and next.
  - Info stores information which is usable by user
  - Next links it to its successor in the sequence



- List operations: add (to head, to tail), remove (at head, at tail), find, insert, check empty, etc.

Declaration of list data structure

- Declare node structure

```c
#include <stdlib.h>
#include <stdio.h>

typedef struct str_node *link;
struct str_node
{
      int data;
      link next;
};

typedef struct str_node node;
```

Declaration of list data structure

- Declare list operations

```
node* create_node(const int data);
int is_empty(const node * const head);
void add_first(node **head, const int data);
void add_last(node **head, const int data);
node* get_last(node * const head);
node* find(node * const head, const int data);
void remove_first(node **head);
void remove_last(node **head);
void clear_list(node **head);
void print(node * const head);
```

Implementation of list operations

```c
node* create_node(const int data)
{
        node* n = (node*) malloc(sizeof(node));
        n->data = data;
        n->next = NULL;

        return n;
}
int is_empty(const node * const head)
{
        return head == NULL;
}
```

Implementation of list operations

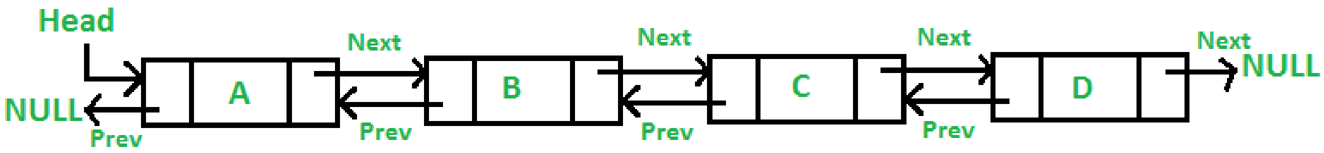
```c
void add_first(node **head, const int data)
{
	node *n = create_node(data);
	if (is_empty(*head))
	{
		*head = n;
	}
	else
	{
		n->next = *head;
		*head = n;
	}
}
```

Implementation of list operations

```c
void remove_first(node **head)
{
    if (is_empty(*head)) return;
    node *p = *head;
    if (p->next == NULL)
    {
        free(p);
        *head = NULL;
    }
    else
    {
        *head = p->next;
        p->next = NULL;
        free(p);
    }
}
```

Double Linked List

- In doubly linked list, each node has two reference fields
  - one to the successor and
  - one to the predecessor



Declaration of Double Linked List

- Declare struct node

```
typedef struct str_dnode *dlink;
struct str_dnode
{
     int data;
     dlink next;
     dlink prev;
};

typedef struct str_dnode dnode;
```

Declaration of Double Linked List

- Declare list operators

```c
dnode* create_dl_node(const int data);
int is_dl_empty(const dnode * const head);
void add_first_dl(dnode **head, const int data);
void add_last_dl(dnode **head, const int data);
dnode* get_last_dl(dnode * const head);
dnode* find_dl(dnode * const head, const int data);
void remove_first_dl(dnode **head);
void remove_last_dl(dnode **head);
void clear_list_dl(dnode **head);
void print_dl(dnode * const head);
```

Implementation of Double Linked List

```
dnode* create_dl_node(const int data)
{
    dnode* n = (dnode*) malloc(sizeof(dnode));
    n->data = data;
    n->next = n->prev = NULL;

    return n;
}
```

Implementation of Double Linked List

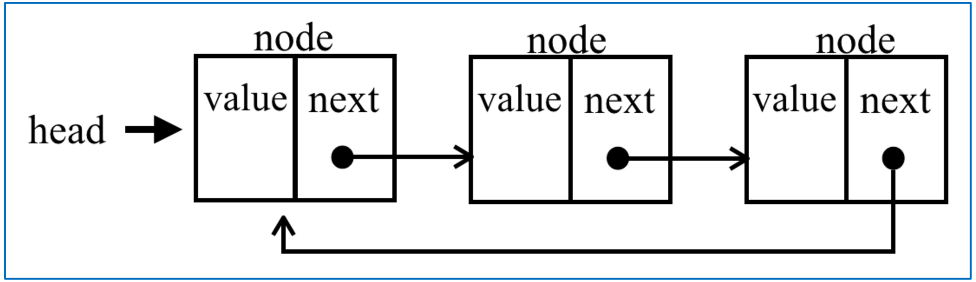
```
void add_first_dl(dnode **head, const int data)
{
    dnode *n = create_dl_node(data);

    if (is_dl_empty(*head))
    {
        *head = n;
    }
    else
    {
        n->next = *head;
        (*head)->prev = n;
        *head = n;
    }
}
```

Implementation of Double Linked List

```
void remove_first_dl(dnode **head)
{
    if (is_dl_empty(*head)) return;
    if ((*head)->next == NULL)
    {
        free(*head);
        *head = NULL;
    }
    else
    {
        dnode* n = *head;
        *head = n->next;
        (*head)->prev = NULL;
        free(n);
    }
}
```

Circular Linked List

- Circular Single Linked List



- Circular Double Linked List