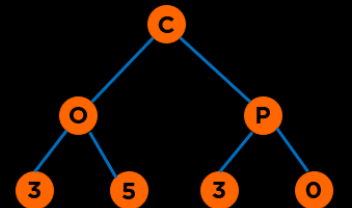


Lists, Stacks and Queues



Categories of Data Structures

Linear Ordered

Lists

Stacks

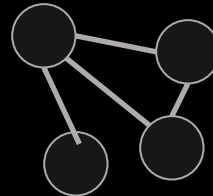
Queues



Non-linear Ordered

Trees

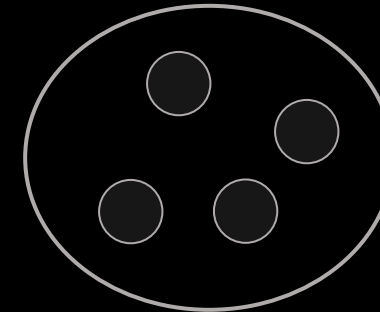
Graphs



Not Ordered

Sets

Tables/Maps



Agenda

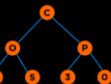
- **Data Structures**
- **Abstract Data Types**
- **Lists**
 - **Array Implementation**
 - **Linked List Implementation and its types**
 - **Lists in C++**
- **Stacks**
 - **Array Based and Linked List Based**
 - **Stacks in C++**
 - **Use Cases**
- **Queues**
 - **Array Based and Linked List Based**
 - **Queue in C++**
 - **Use cases**

Data Structures

Data Structures

A data structure is a way to store and organize data

- **Mathematical or Logical models (Abstract Data Types)**
 - **List: Store, Read, Modify**
- **Implementation (Concrete)**
 - **Arrays, Linked List, Vector, ArrayList, List**



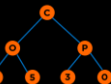
Abstract Data Types

Abstract Data Types

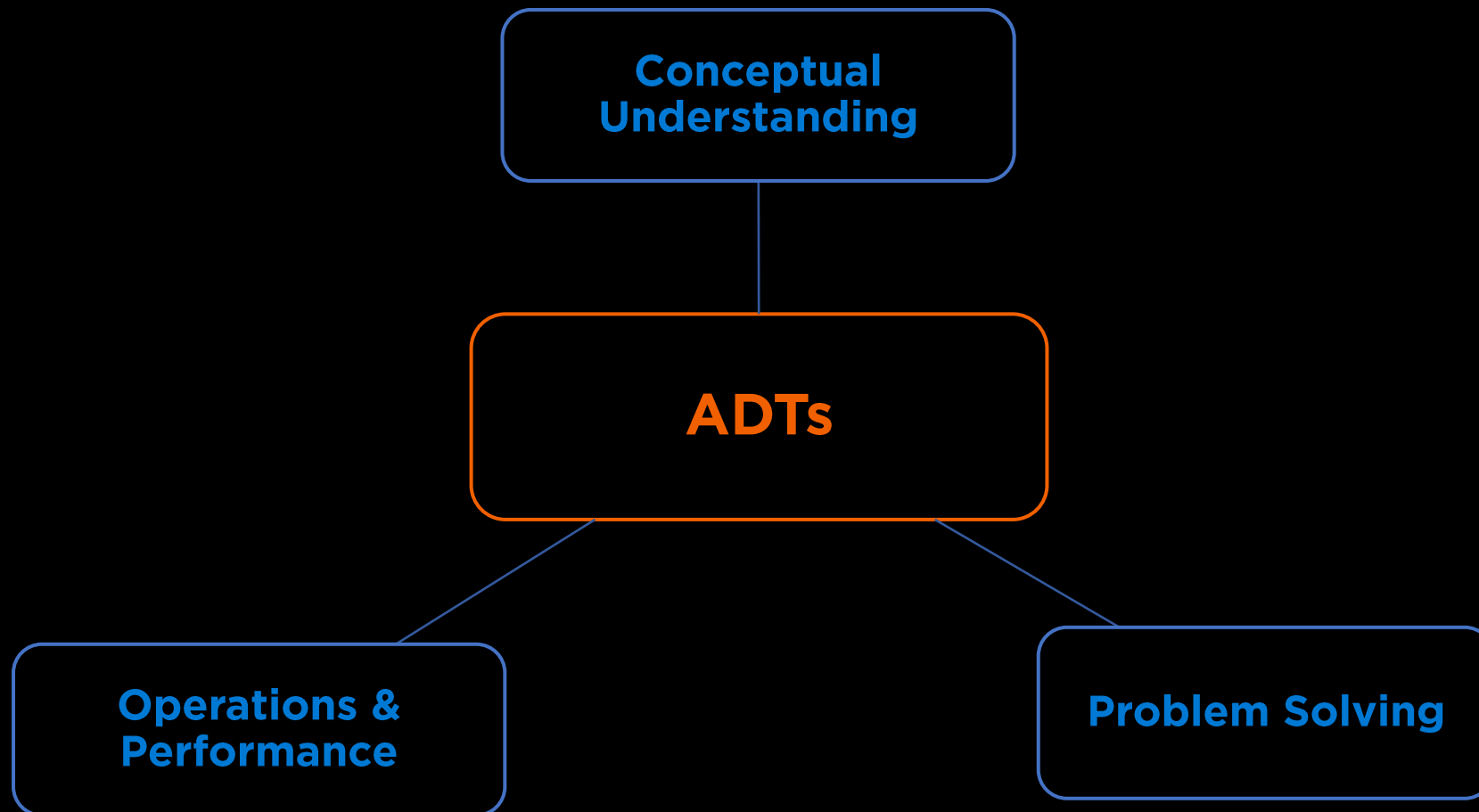
- Class of objects whose logical behavior is defined by a set of values and a set of operations
 - Define data (properties) and operations (behavior)
 - Don't care about implementation



https://en.wikipedia.org/wiki/Abstract_data_type



Abstract Data Types (ADTs)



Lists

List

- **Ordered Collection of Data (Ordered = Position)**
- **Elements have some position**
- **Linear Structure**
- **Can have some size or grow/shrink**
- **No limit on nature of elements**

List - Characteristics

- Data
- Operations

List - Characteristics

- **Data**

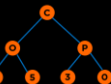
- **Items**
- **Number of Items (Size)**
- **Capacity**

- **Operations**

- **Read/Write an element**
- **Add or remove an element**
- **Find an element**
- **Count**
- **Traverse the list (Printing)**

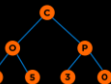
List – Example: Array

Characteristics			
Operations			
Performance			
Benefits			
Drawbacks			



List – Example: Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Contiguous Indices▪ Elements are stored contiguously in memory▪ Allows Random Access		
Operations			
Performance			
Benefits			
Drawbacks			



List – Example: Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Contiguous Indices▪ Elements are stored contiguously in memory▪ Allows Random Access		
Operations	<ul style="list-style-type: none">▪ Adding – Beginning, Middle, End▪ Removal – Beginning, Middle, End		
Performance		Add	Remove
	Beginning		
	End		
	Middle		
Benefits			
Drawbacks			

List – Example: Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Contiguous Indices▪ Elements are stored contiguously in memory▪ Allows Random Access		
Operations	<ul style="list-style-type: none">▪ Adding – Beginning, Middle, End▪ Removal – Beginning, Middle, End		
Performance		Add	Remove
	Beginning	$O(n)$	$O(n)$
	End	$O(1)$	$O(1)$
	Middle	$O(n)$	$O(n)$
Benefits			
Drawbacks			

List – Example: Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Contiguous Indices▪ Elements are stored contiguously in memory▪ Allows Random Access		
Operations	<ul style="list-style-type: none">▪ Adding – Beginning, Middle, End▪ Removal – Beginning, Middle, End		
Performance		Add	Remove
	Beginning	$O(n)$	$O(n)$
	End	$O(1)$	$O(1)$
	Middle	$O(n)$	$O(n)$
Benefits	<ul style="list-style-type: none">▪ Constant Access Time, $arr[i] = arr + (i * \text{sizeof}(\text{type}))$		
Drawbacks			

List – Example: Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Contiguous Indices▪ Elements are stored contiguously in memory▪ Allows Random Access		
Operations	<ul style="list-style-type: none">▪ Adding – Beginning, Middle, End▪ Removal – Beginning, Middle, End		
Performance		Add	Remove
	Beginning	$O(n)$	$O(n)$
	End	$O(1)$	$O(1)$
	Middle	$O(n)$	$O(n)$
Benefits	<ul style="list-style-type: none">▪ Constant Access Time, $\text{arr}[i] = \text{arr} + (i * \text{sizeof}(\text{type}))$		
Drawbacks	<ul style="list-style-type: none">▪ Expensive for adding/removing elements from front		

List – Example: Single Linked List

List – Example: Single Linked List

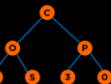
```
class Node
{
    public:
        Node *next;
        int data;
};
```

```
int main ()
{
    Node* obj = new Node;
    obj -> data = 10;
    obj -> next = nullptr;

    obj -> next = new Node;
    obj -> next -> data = 20;
    obj -> next -> next = nullptr;

    obj -> next -> next = new Node;
    obj -> next -> next -> data = 30;
    obj -> next -> next -> next = nullptr;

    return 0;
}
```



List – Example: Single Linked List

```
class Node
{
    public:
        Node *next;
        int data;
};
```

```
int main ()
{
    Node* obj = new Node;
    obj -> data = 10;
    obj -> next = nullptr;

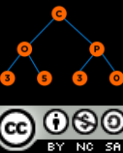
    obj -> next = new Node;
    obj -> next -> data = 20;
    obj -> next -> next = nullptr;

    obj -> next -> next = new Node;
    obj -> next -> next -> data = 30;
    obj -> next -> next -> next = nullptr;

    return 0;
}
```

Problem:

Messy and too hard to read, Error prone



List – Example: Single Linked List

```
class Node
{
    public:
        Node *next;
        int data;
        Node(int d, Node* n);
    private:
        int size();
};

Node::Node(int d, Node* n)
{
    this -> data = d;
    this -> next = n;
}

int Node::size()
{
    if(this == nullptr)
        return 0;
    else
        return 1 + (this->next)->size();
}

int main ()
{
    Node* obj = new Node(10, nullptr);
    obj = new Node(20, obj);
    obj = new Node(30, obj);
    cout << "Size = " << obj -> size();
    return 0;
}
```

List – Example: Single Linked List

```
class Node
{
    public:
        Node *next;
        int data;
        Node(int d, Node* n);
    private:
        int size();
};
```

```
Node::Node(int d, Node* n)
{
    this -> data = d;
    this -> next = n;
}
```

```
int Node::size()
{
    if(this == nullptr)
        return 0;
    else
        return 1 + (this->next)->size();
}
```

```
int main ()
{
    Node* obj = new Node(10, nullptr);
    obj = new Node(20, obj);
    obj = new Node(30, obj);
    cout << "Size = " << obj -> size();
    return 0;
}
```

Problem:

Too much details

List – Example: Single Linked List

Encapsulate Node: Better Design

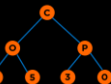
```
struct Node
{
    int data;
    Node *next;
};
```

or

```
class Node
{
public:
    Node *next;
    int data;
};
```

```
class List
{
public:
    int size;
    Node* head;
    Node* tail;

public:
    List();
    void push_front(int);
};
```



List – Example: Single Linked List

Encapsulate Node: Better Design

```
struct Node
{
    int data;
    Node *next;
};
```

or

```
class Node
{
public:
    Node *next;
    int data;
};
```

```
class List
{
public:
    int size;
    Node* head;
    Node* tail;

public:
    List();
    void push_front(int);
};

List::List()
{
    size = 0;
    head = new Node;
    tail = new Node;
    head -> next = tail;
}
```

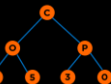
```
void List::push_front(int x)
{
    Node* temp = new Node;
    temp -> data = x;
    temp -> next = head->next;
    head -> next = temp;
    size++;
}
```

```
int main()
{
    List obj;
    obj.push_front(15);
    obj.push_front(20);
    obj.push_front(15);
    return 0;
}
```

List – Example: Single Linked List

Do not expose internal details to the client

```
int main()
{
    List myList;
    myList.push_front(15);
    myList.push_front(20);
    myList.push_front(15);
    return 0;
}
```



List – Example: Single Linked List

Characteristics			
Operations			
Performance			
Benefits			
Drawbacks			

List – Example: Single Linked List

Characteristics	<ul style="list-style-type: none">▪ Consists of Nodes<ul style="list-style-type: none">○ Data○ Pointer to Next Node▪ Stores Similar Elements▪ Elements are linked in memory but stored non-contiguously▪ Does not allow Random Access		
Operations			
Performance			
Benefits			
Drawbacks			

List – Example: Single Linked List

Characteristics	<ul style="list-style-type: none"> Consists of Nodes <ul style="list-style-type: none"> Data Pointer to Next Node Stores Similar Elements Elements are linked in memory but stored non-contiguously Does not allow Random Access 		
Operations	<ul style="list-style-type: none"> Adding (Add) – PushFront(Key), PushBack(Key) Removal (Remove) – PopFront, PopBack Access (Get) – TopFront, TopBack Find(Key), Erase(Key), Empty() AddBefore(Node, Key), AddAfter(Node, Key) 		
Performance			
Benefits			
Drawbacks			

List – Example: Single Linked List

Characteristics	<ul style="list-style-type: none"> Consists of Nodes <ul style="list-style-type: none"> Data Pointer to Next Node Stores Similar Elements Elements are linked in memory but stored non-contiguously Does not allow Random Access 		
Operations	<ul style="list-style-type: none"> Adding (Add) – PushFront(Key), PushBack(Key) Removal (Remove) – PopFront, PopBack Access (Get) – TopFront, TopBack Find(Key), Erase(Key), Empty() AddBefore(Node, Key), AddAfter(Node, Key) 		
Performance	PushFront	PushBack	AddBefore
	PopFront	PopBack	AddAfter
	TopFront	TopBack	
	Find	Erase	Empty
Benefits			
Drawbacks			

List – Example: Single Linked List

Characteristics	<ul style="list-style-type: none"> Consists of Nodes <ul style="list-style-type: none"> Data Pointer to Next Node Stores Similar Elements Elements are linked in memory but stored non-contiguously Does not allow Random Access 		
Operations	<ul style="list-style-type: none"> Adding (Add) – PushFront(Key), PushBack(Key) Removal (Remove) – PopFront, PopBack Access (Get) – TopFront, TopBack Find(Key), Erase(Key), Empty() AddBefore(Node, Key), AddAfter(Node, Key) 		
Performance	PushFront – $O(1)$	PushBack – $O(n)$	AddBefore – $O(n)$
	PopFront – $O(1)$	PopBack – $O(n)$	AddAfter – $O(1)$
	TopFront – $O(1)$	TopBack – $O(n)$	
	Find – $O(n)$	Erase – $O(n)$	Empty – $O(1)$
Benefits			
Drawbacks			

List – Example: Single Linked List

Characteristics	<ul style="list-style-type: none"> Consists of Nodes <ul style="list-style-type: none"> Data Pointer to Next Node Stores Similar Elements Elements are linked in memory but stored non-contiguously Does not allow Random Access 		
Operations	<ul style="list-style-type: none"> Adding (Add) – PushFront(Key), PushBack(Key) Removal (Remove) – PopFront, PopBack Access (Get) – TopFront, TopBack Find(Key), Erase(Key), Empty() AddBefore(Node, Key), AddAfter(Node, Key) 		
Performance	PushFront – $O(1)$	PushBack – $O(n)$	AddBefore – $O(n)$
	PopFront – $O(1)$	PopBack – $O(n)$	AddAfter – $O(1)$
	TopFront – $O(1)$	TopBack – $O(n)$	
	Find – $O(n)$	Erase – $O(n)$	Empty – $O(1)$
Benefits	<ul style="list-style-type: none"> Adding/Removing in front is faster, $O(1)$ 		
Drawbacks			

List – Example: Single Linked List

Characteristics	<ul style="list-style-type: none"> Consists of Nodes <ul style="list-style-type: none"> Data Pointer to Next Node Stores Similar Elements Elements are linked in memory but stored non-contiguously Does not allow Random Access 		
Operations	<ul style="list-style-type: none"> Adding (Add) – PushFront(Key), PushBack(Key) Removal (Remove) – PopFront, PopBack Access (Get) – TopFront, TopBack Find(Key), Erase(Key), Empty() AddBefore(Node, Key), AddAfter(Node, Key) 		
Performance	PushFront – $O(1)$	PushBack – $O(n)$	AddBefore – $O(n)$
	PopFront – $O(1)$	PopBack – $O(n)$	AddAfter – $O(1)$
	TopFront – $O(1)$	TopBack – $O(n)$	
	Find – $O(n)$	Erase – $O(n)$	Empty – $O(1)$
Benefits	<ul style="list-style-type: none"> Adding/Removing in front is faster, $O(1)$ 		
Drawbacks	<ul style="list-style-type: none"> Expensive for random access, $O(n)$ TopBack, PushBack, PopBack and AddBefore are expensive 		

List : Single Linked List with Tail

Characteristics	<ul style="list-style-type: none"> Consists of Nodes and has a Tail <ul style="list-style-type: none"> Data Pointer to Next Node Stores Similar Elements Elements are linked in memory but stored non-contiguously Does not allow Random Access 		
Operations	<ul style="list-style-type: none"> Adding – PushFront(Key), PushBack(Key) Removal – PopFront, PopBack Access – TopFront, TopBack Find(Key), Erase(Key), Empty() AddBefore(Node, Key), AddAfter(Node, Key) 		
Performance	PushFront – $O(1)$	PushBack – $O(1)$	AddBefore – $O(n)$
	PopFront – $O(1)$	PopBack – $O(n)$	AddAfter – $O(1)$
	TopFront – $O(1)$	TopBack – $O(1)$	
	Find – $O(n)$	Erase – $O(n)$	Empty – $O(1)$
Benefits	<ul style="list-style-type: none"> Solves the issue of pushing back and getting last element 		
Drawbacks	<ul style="list-style-type: none"> Expensive for PopBack and AddBefore 		

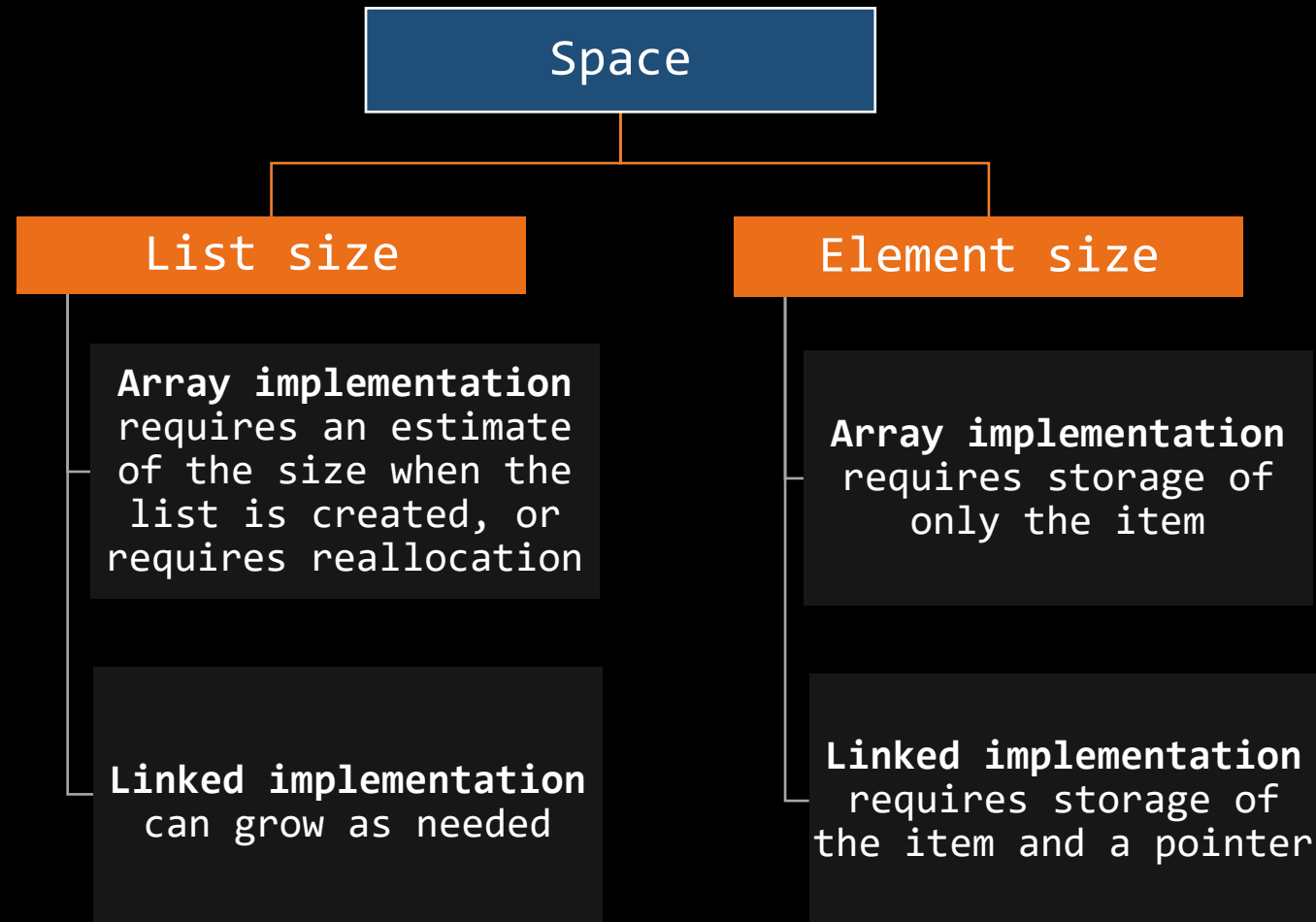
List : Doubly Linked List with Tail

Characteristics	<ul style="list-style-type: none"> Consists of Nodes <ul style="list-style-type: none"> Data Pointer to Next and Previous Node Stores Similar Elements Elements are linked in memory but stored non-contiguously Does not allow Random Access 		
Operations	<ul style="list-style-type: none"> Adding – PushFront(Key), PushBack(Key) Removal – PopFront, PopBack Access – TopFront, TopBack Find(Key), Erase(Key), Empty() AddBefore(Node, Key), AddAfter(Node, Key) 		
Performance	PushFront – $O(1)$	PushBack – $O(1)$	AddBefore – $O(1)$
	PopFront – $O(1)$	PopBack – $O(1)$	AddAfter – $O(1)$
	TopFront – $O(1)$	TopBack – $O(1)$	
	Find – $O(n)$	Erase – $O(n)$	Empty – $O(1)$
Benefits	<ul style="list-style-type: none"> Solves the issue of PopBack and AddBefore 		
Drawbacks	<ul style="list-style-type: none"> Extra memory 		

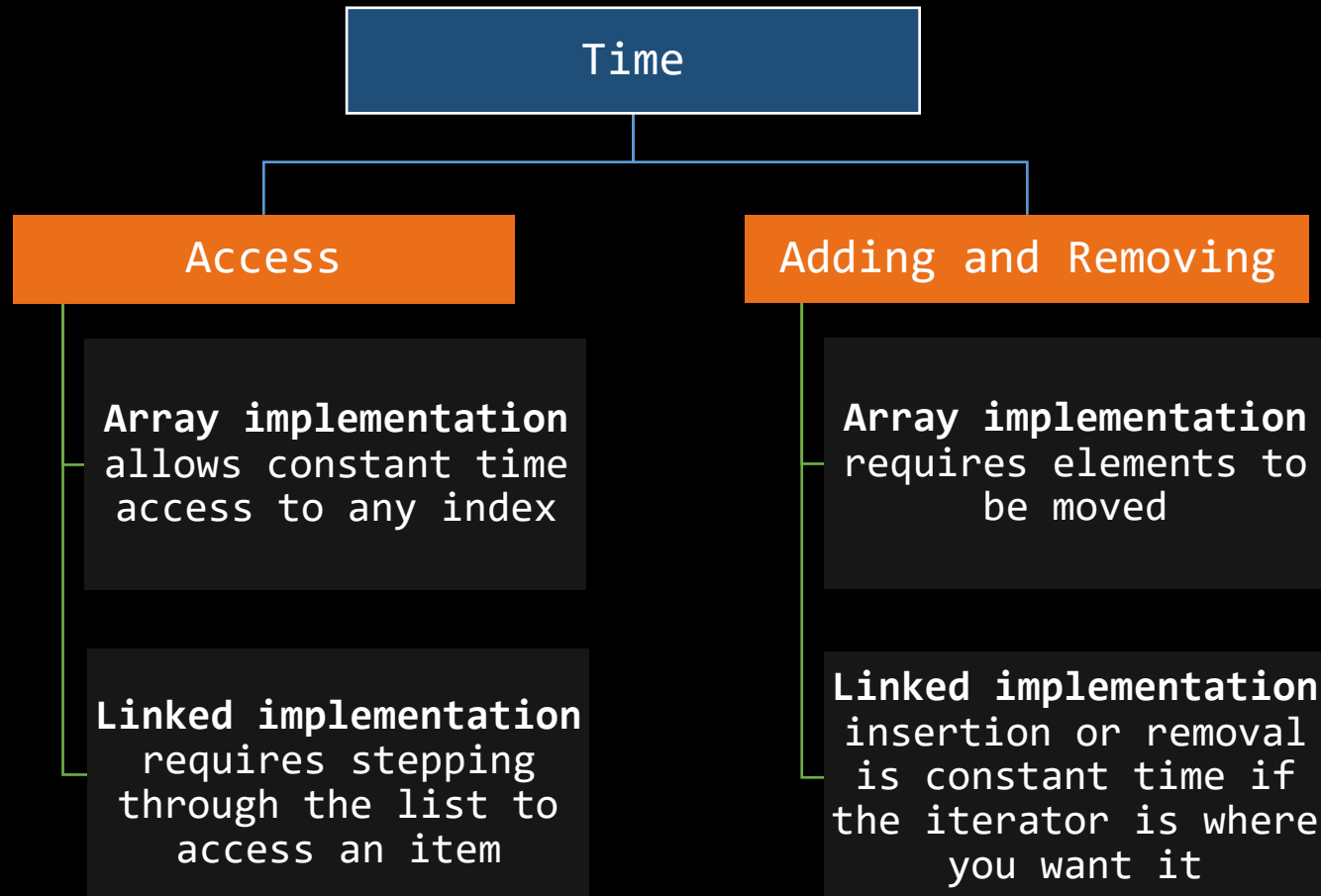
List – Example: Circular Linked List

- **Single Circular**
- **Doubly Circular**

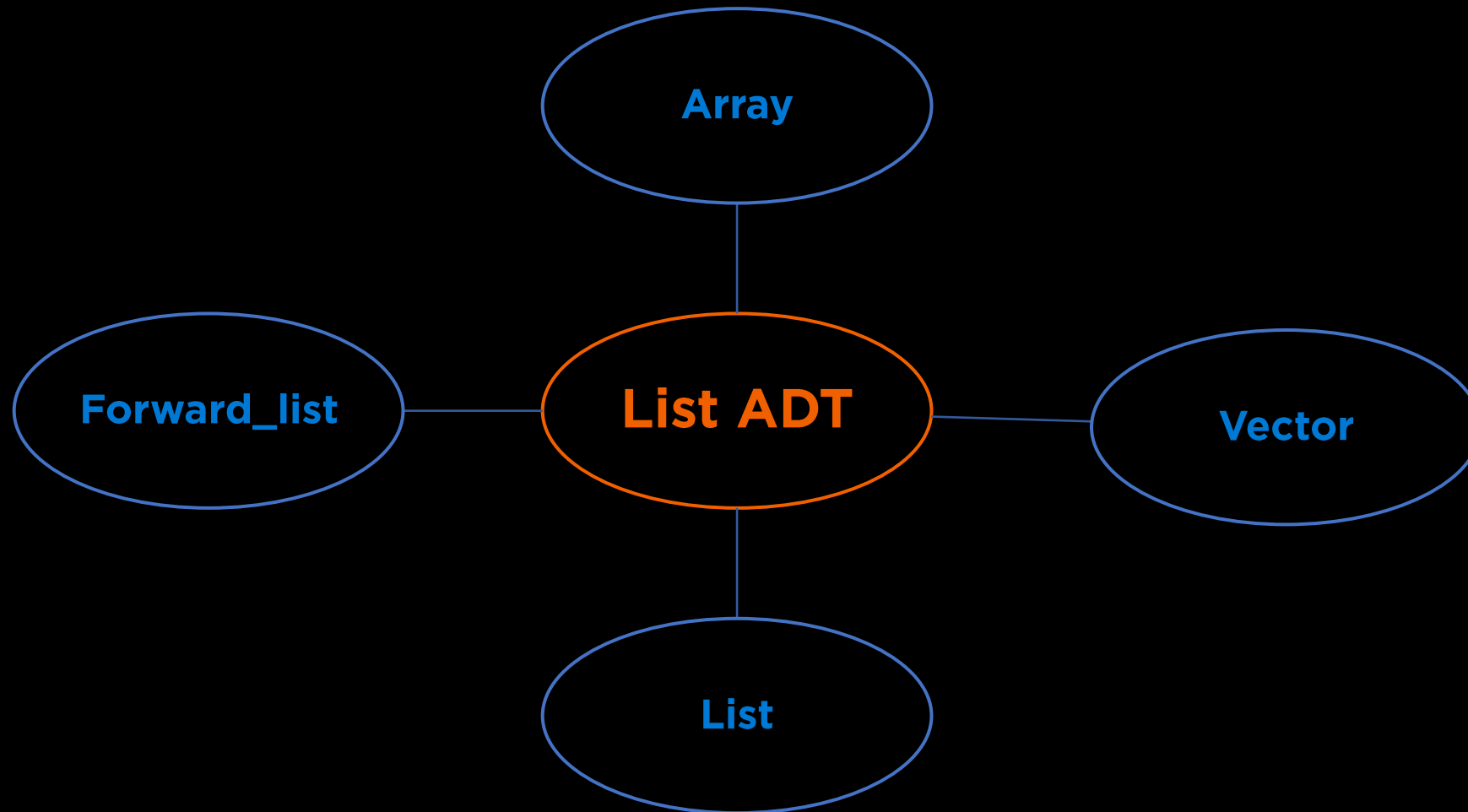
Array vs Linked List: Space



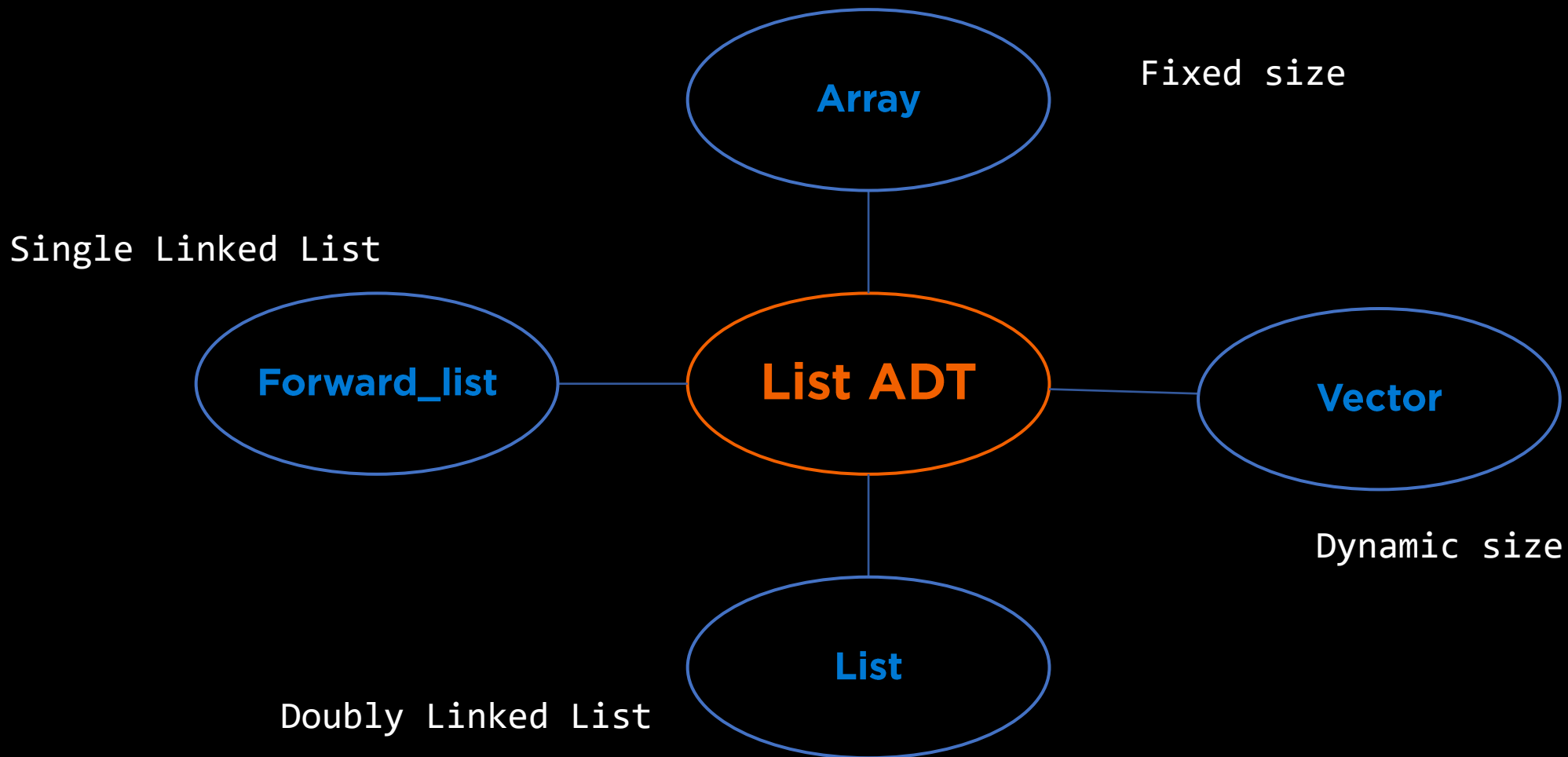
Array vs Linked List: Time



Lists in C++ Standard Template Library



Lists in C++ Standard Template Library



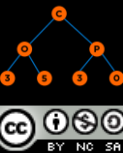
Lists in C++ STL: List

List containers are implemented as doubly-linked lists

Function	Operation
Constructor(n, val)	Constructs a container with n elements. Each element is a copy of val (if provided).
void push_front (val)	Inserts a new element at the beginning of the list, right before its current first element.

```
std::list<int> mylist (4,100);    [100, 100, 100, 100]
mylist.push_front (200);          [200, 100, 100, 100, 100]
```

<http://www.cplusplus.com/reference/list/list/>



Lists in C++ STL: List

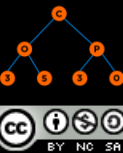
Printing a List

```
01 list<int> mylist (4,100);    // [100, 100, 100, 100]
02 mylist.push_front (200);    // [200, 100, 100, 100, 100]
03 for(list<int>::iterator it = mylist.begin(); it != mylist.end(); ++it)
04     cout << ' ' << *it;
```

Prints: 200 100 100 100 100

All containers have Iterators!

<http://www.cplusplus.com/reference/list/list/>



Iterators

- **Variables to keep track of where we are in a data set**
- **Iterator Class**
 - Operators to advance to next data (++)
 - Dereference Operator (*) to access data
 - Operators to compare two iterators (!=)
 - Assignment operator (=)
- **Container or the main data structure you are implementing Class**
 - begin() methods
 - end() methods

Templated Singly Linked List: https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/lists/list_declaration.html

Iterators

category				properties	valid expressions
all categories				<i>copy-constructible, copy-assignable and destructible</i>	X b(a); b = a;
				Can be incremented	++a a++
Random Access	Bidirectional	Input		Supports equality/inequality comparisons	a == b a != b
				Can be dereferenced as an <i>rvalue</i>	*a a->m
		Output		Can be dereferenced as an <i>lvalue</i> (only for <i>mutable iterator types</i>)	*a = t *a++ = t
				<i>default-constructible</i>	X a; X()
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	{ b=a; *a++; *b; }
				Can be decremented	--a a-- *a--
				Supports arithmetic operators + and -	a + n n + a a - n a - b
				Supports inequality comparisons (<, >, <= and >=) between iterators	a < b a > b a <= b a >= b
				Supports compound assignment operations += and -=	a += n a -= n
				Supports offset dereference operator ([])	a[n]

<http://www.cplusplus.com/reference/iterator/>

Iterators: Forward

```
1. // Forward Lists - support forward iterators: https://www.cplusplus.com/reference/forward_list/forward_list/
2.
3. std::forward_list<int> flist_container {1, 2, 3, 4, 5};
4.
5. for(auto it = flist_container.begin(); it != flist_container.end(); it++)
6. {
7.     std::cout << *it << std::endl;
8.     std::cout << *(--it); //This line will throw an error as iterator does not support going backward
9.     std::cout << *(it + 2); //This line will throw an error as iterator does not support random access
10.    break;
11. }
```

Iterators: Bidirectional

```
1. // Lists - support bidirectional iterators: https://www.cplusplus.com/reference/list/list/
2. std::list<int> list_container {1, 2, 3, 4, 5};
3.
4. for(auto it = list_container.begin(); it != list_container.end(); it++)
5. {
6.     std::cout << *it << " ";
7.     std::cout << *--it << std::endl; //This line is undefined behavior
8.     //std::cout << *(it + 2); //This line will throw an error as iterator does not support random access
9.     break;
10. }
```

Iterators: Random Access

```
1. // Vectors - support random access iterators: https://cplusplus.com/reference/vector/vector/
2.
3. std::vector<int> vector_container {1, 2, 3, 4, 5};
4.
5. for(auto it = vector_container.begin(); it != vector_container.end(); it++)
6. {
7.     std::cout << *it << " ";
8.     std::cout << *--it << " "; //This line is undefined behavior
9.     std::cout << *(it + 2);
10.    break;
11. }
12.
```

Merge Two Sorted Linked Lists of Integers

Merge Two Sorted Linked Lists of Integers

Pseudocode:

```
given ListA, ListB both sorted
create mergedList
```

```
place iterA at the head of list A, iterB at the head of list B
itemA is item iterA is pointing to and itemB is item iterB is pointing to
```

```
while iterA is not at end of ListA and iterB is not at end of ListB
    if itemA == itemB
        make a copy of itemA and add it to mergedList
        move iterA and iterB forward
    else if itemA < itemB
        make a copy of itemA and add it to mergedList
        move iterA forward
    else
        make a copy of itemB and add it to mergedList
        move iterB forward
```

Merge Two Sorted Linked Lists of Integers

Pseudocode:

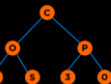
```
given ListA, ListB both sorted
create mergedList
```

```
place iterA at the head of list A, iterB at the head of list B
itemA is item iterA is pointing to and itemB is item iterB is pointing to
```

```
while iterA is not at end of ListA and iterB is not at end of ListB
    if itemA == itemB
        make a copy of itemA and add it to mergedList
        move iterA and iterB forward
    else if itemA < itemB
        make a copy of itemA and add it to mergedList
        move iterA forward
    else
        make a copy of itemB and add it to mergedList
        move iterB forward
```

```
while iterA is not at end of list
    make a copy of itemA and add it to mergedList
    move iterA forward
```

```
while iterB is not at end of list
    make a copy of itemB and add it to mergedList
    move iterB forward
```



Merge Two Sorted Linked Lists of Integers (Union)

```
01 int arr1[] = {16,22,77,129};
02 int arr2[] = {1,2,7,29,77,155,166};
03
04 //Creating Lists from Arrays
05 list<int> list1 (arr1, (arr1 + (sizeof(arr1) / sizeof(int)))) ;
06 list<int> list2 (arr2, (arr2 + (sizeof(arr2) / sizeof(int)))) ;
07
08 //Final Merged Sorted List
09 list<int> list3;
10
11 list<int>::iterator l1 = list1.begin();
12 list<int>::iterator l2 = list2.begin();
13
14 while(l1 != list1.end() && l2 != list2.end())
15 {
16     if((*l1) < (*l2))
17     {
18         list3.push_back(*l1);
19         l1++;
20     }
21     else
22     if((*l2) < (*l1))
23     {
24         list3.push_back(*l2);
25         l2++;
26     }
27     else
28     {
29         list3.push_back(*l1);
30         l1++;
31         l2++;
32     }
33 }
```

```
34
35 while(l1 != list1.end())
36 {
37     list3.push_back(*l1);
38     l1++;
39 }
40
41 while(l2 != list2.end())
42 {
43     list3.push_back(*l2);
44     l2++;
45 }
46
```

Median of Elements in a Linked List (3.3d)

Median of Elements in a Linked List (3.3d)

```
01 //runner technique, tortoise and hare
02 float median(Node* head)
03 {
04     Node* fastptr = head;
05     Node* slowptr = head;
06     while (fastptr->next != NULL && fastptr->next->next != NULL)
07     {
08         slowptr = slowptr->next;
09         fastptr = fastptr->next->next;
10     }
11     return fastptr->next == NULL ? slowptr->value : (slowptr->value + slowptr->next->value)/2.0;
12 }
```

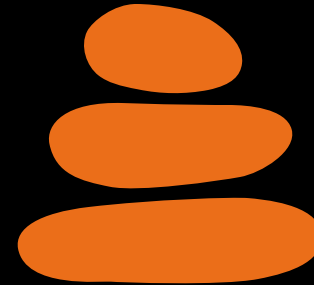
Recommended Resources

- <http://www.cplusplus.com/reference/stl/>
- **Linked List Questions on Edugator**
- <https://stackoverflow.com/questions/5384358/how-does-a-sentinel-node-offer-benefits-over-null>
- [OpenDSA Ch-9.1-9.7](#)
- **Templated Singly Linked List:** https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/lists/list_declaration.html
- <https://www.cplusplus.com/reference/iterator/>
- <https://www.geeksforgeeks.org/input-iterators-in-cpp/>
- <https://onlinegdb.com/Nk9XgDjG->

Stacks

Stack

- Last in First Out (LIFO)



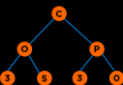
Stack ADT

■ Data

- Items
- Number of Items
- Top

■ Operations

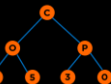
- **push(item):** inserts an element
- **pop():** removes and returns the last inserted element
- **peek():** returns the last inserted element without removing it
- **size():** returns the number of elements stored
- **isEmpty():** indicates whether no elements are stored



Stack Implementation - Array

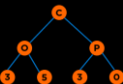
Stack Implementation - Array

Characteristics	
Operations	
Performance	
Benefits	
Drawbacks	



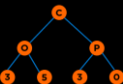
Stack Implementation - Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through top index
Operations	
Performance	
Benefits	
Drawbacks	



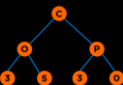
Stack Implementation - Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through top index
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	
Benefits	
Drawbacks	



Stack Implementation - Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through top index
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Benefits	
Drawbacks	



Stack Implementation - Array

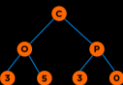
Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through top index
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item) - $O(1)$▪ pop() - $O(1)$▪ peek() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	
Drawbacks	

Stack Implementation - Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through top index
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item) - $O(1)$▪ pop() - $O(1)$▪ peek() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	<ul style="list-style-type: none">▪ Constant time to add and remove elements
Drawbacks	

Stack Implementation - Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through top index
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item) - $O(1)$▪ pop() - $O(1)$▪ peek() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	<ul style="list-style-type: none">▪ Constant time to add and remove elements
Drawbacks	<ul style="list-style-type: none">▪ Fixed size and no random access



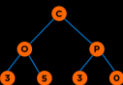
Stack Implementation – Linked List

Characteristics	
Operations	
Performance	
Benefits	
Drawbacks	

Stack Implementation – Linked List

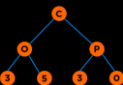
Stack Implementation – Linked List

Characteristics	<ul style="list-style-type: none">▪ Flexible Size▪ Stores Similar Elements▪ Access through top pointer
Operations	
Performance	
Benefits	
Drawbacks	



Stack Implementation – Linked List

Characteristics	<ul style="list-style-type: none">▪ Flexible Size▪ Stores Similar Elements▪ Access through top pointer
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	
Benefits	
Drawbacks	



Stack Implementation – Linked List

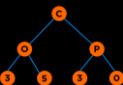
Characteristics	<ul style="list-style-type: none">▪ Flexible Size▪ Stores Similar Elements▪ Access through top pointer
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Benefits	
Drawbacks	

Stack Implementation – Linked List

Characteristics	<ul style="list-style-type: none">▪ Flexible Size▪ Stores Similar Elements▪ Access through top pointer
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item) - $O(1)$▪ pop() - $O(1)$▪ peek() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	
Drawbacks	

Stack Implementation – Linked List

Characteristics	<ul style="list-style-type: none">▪ Flexible Size▪ Stores Similar Elements▪ Access through top pointer
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item) - $O(1)$▪ pop() - $O(1)$▪ peek() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	<ul style="list-style-type: none">▪ Constant time to add and remove elements; Variable size
Drawbacks	



Stack Implementation – Linked List

Characteristics	<ul style="list-style-type: none">▪ Flexible Size▪ Stores Similar Elements▪ Access through top pointer
Operations	<ul style="list-style-type: none">▪ push(item)▪ pop()▪ peek()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ push(item) - $O(1)$▪ pop() - $O(1)$▪ peek() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	<ul style="list-style-type: none">▪ Constant time to add and remove elements; Variable size
Drawbacks	<ul style="list-style-type: none">▪ More memory

Stack in C++ STL

Operations	C++ STL
<ul style="list-style-type: none">▪ <code>push(item)</code>▪ <code>pop()</code>▪ <code>peek()</code>▪ <code>size()</code>▪ <code>isEmpty()</code>	<ul style="list-style-type: none">▪ <code>push(g)</code> – Adds the element 'g' at the top of the stack▪ <code>pop()</code> – Deletes the topmost element of the stack▪ <code>top()</code> – Returns a reference to the topmost element of the stack▪ <code>size()</code> – Returns the size of the stack▪ <code>empty()</code> – Returns whether the stack is empty

Stack in C++ STL: Check Palindrome

Stack in C++ STL: Check Palindrome

```
01 bool checkPalindrome(string s)
02 {
03     stack<char> stk;
04     int midIndex = s.length()/2;
05
06     if(s.length() < 2)
07         return true;
08
09     for(int i = 0; i < s.length()/2; i++)
10         stk.push(s.at(i));
11
12     if(s.length() % 2 != 0)
13         midIndex += 1;
14
15     for(int i = midIndex; i < s.length(); i++)
16     {
17         if(stk.top() != s.at(i))
18             return false;
19         stk.pop();
20     }
21
22     return true;
23 }
```

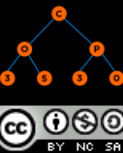
Base case to return true if string has one letter or is an empty string

Pushes first half of string characters on stack

Ignores central element if the size of string is odd

Compare each element in the second half of the string with elements pushed in the stack

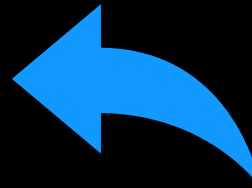
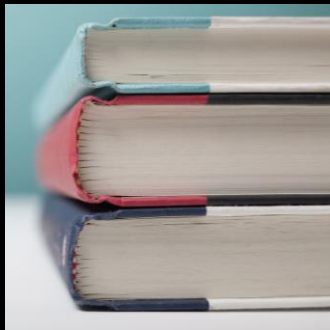
<https://onlinegdb.com/Hy2vC3DVI>



Stack Use Cases

■ Real World

- Plates
- Books
- Pringle Chips



■ Computers

- Function Call Stack
- Evaluate Expressions
- Backtracking
- Balanced Parenthesis
- Undo (CTRL + Z)
- Back button

Stack Use Cases: Call Stack

```
main()
{
    int i=5;
    foo(i);
}

foo(int j)
{
    int k = j+1;
    bar(k);
}

bar(int m)
{
    ...
}
```

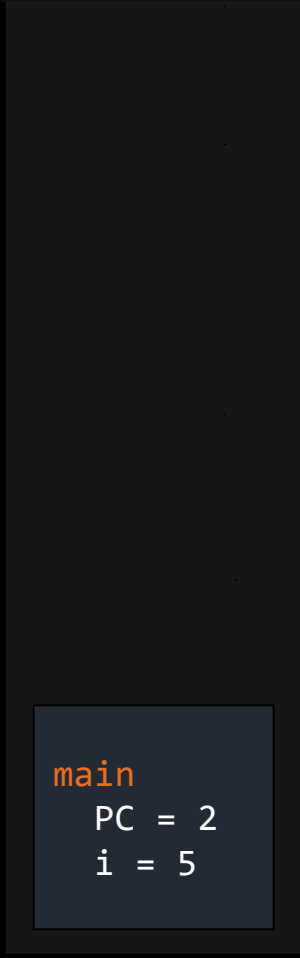


Stack Use Cases: Call Stack

```
main()
{
    int i=5;
    foo(i);
}

foo(int j)
{
    int k = j+1;
    bar(k);
}

bar(int m)
{
    ...
}
```



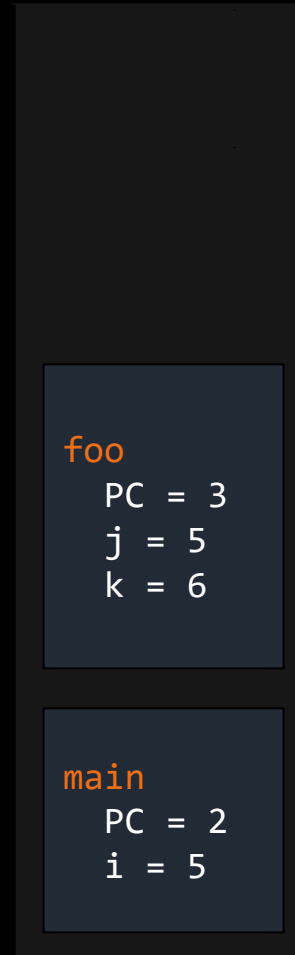
```
main
PC = 2
i = 5
```

Stack Use Cases: Call Stack

```
main()
{
    int i=5;
    foo(i);
}

foo(int j)
{
    int k = j+1;
    bar(k);
}

bar(int m)
{
    ...
}
```

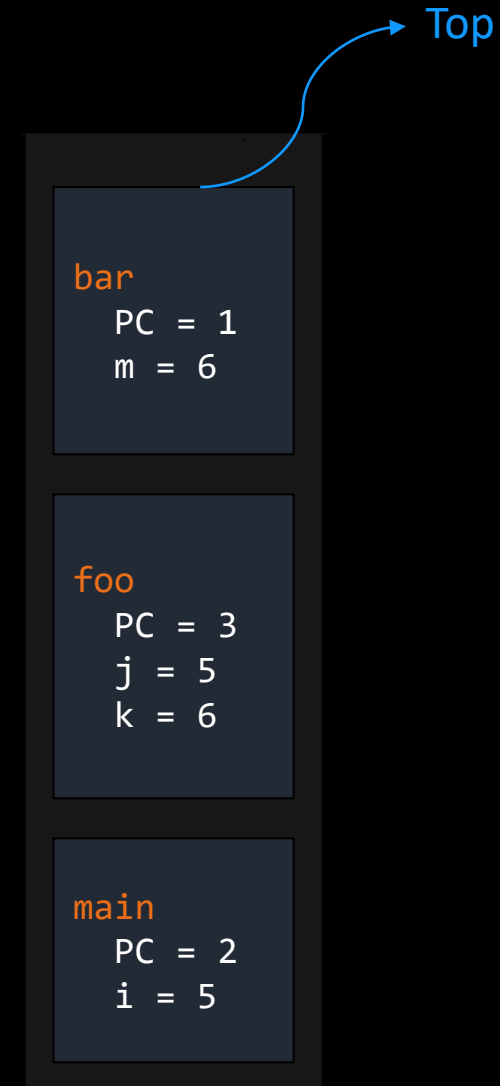


Stack Use Cases: Call Stack

```
main()
{
    int i=5;
    foo(i);
}

foo(int j)
{
    int k = j+1;
    bar(k);
}

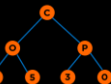
bar(int m)
{
    ...
}
```



Stack Use Cases: Balanced Parenthesis

Algorithm for method isBalanced

1. Create an empty stack of characters
2. Assume that the expression is balanced (balanced is **true**).
3. Set index to 0
4. while balanced is **true** and index < the expression's length
 5. Get the next character in the data string
 6. if the next character is an opening parenthesis
 7. Push it onto the stack.
 8. else if the next character is a closing parenthesis
 9. Pop the top of the stack
 10. if stack was empty or its top does not match the closing parenthesis
 11. Set balanced to **false**
 12. Increment index
13. Return **true** if balanced is **true** and the stack is empty



Stack Use Cases: Expression Evaluation

- Dijkstra's two-stack algorithm

- **Value:** push on the *value* stack
- **Operator:** push on the *operator* stack
- **Left parenthesis, (:** ignore
- **Right parenthesis,) :** pop operator and two values; push the result of applying that operator to the values on the *value* stack

Stack Use Cases: Expression Evaluation

- **Postfix Expression: Removes Parenthesis**

Postfix Expression	Infix Expression	Value
$\boxed{4 \ 7 \ *}$	$4 * 7$	28
$\boxed{4 \ \boxed{7 \ 2 \ +} \ *}$	$4 * (7 + 2)$	36
$\boxed{4 \ 7 \ *} \ 20 \ -$	$(4 * 7) - 20$	8
$3 \ \boxed{\boxed{4 \ 7 \ *} \ 2 \ /} \ +$	$3 + ((4 * 7) / 2)$	17

Stack Use Cases: Postfix Evaluation

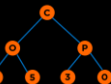
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
10. push the result onto the stack
11. pop the stack and return the result



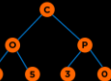
Mentimeter

Menti.com

8544 3577

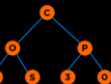
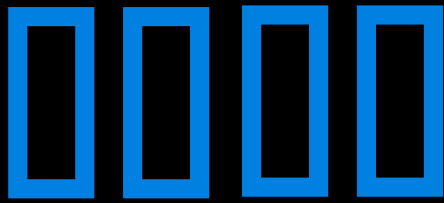


Queues



Queue

- First in First Out (FIFO)



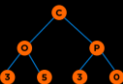
Queue ADT

- **Data**

- **Items**
- **Number of Items**
- **Front and Back**

- **Operations**

- **enqueue(item):** **inserts an element to the back of the queue**
- **dequeue():** **removes the element from the front**
- **size():** **returns the number of elements stored**
- **isEmpty():** **indicates whether no elements are stored**

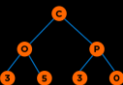
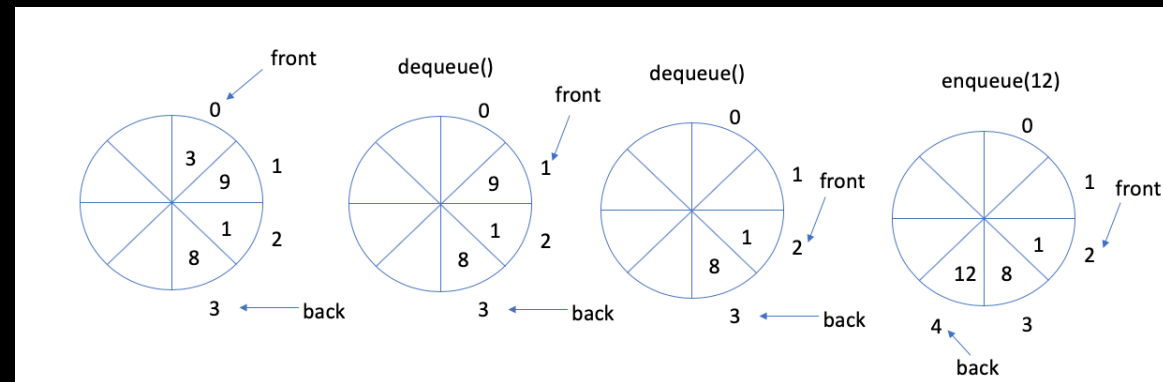
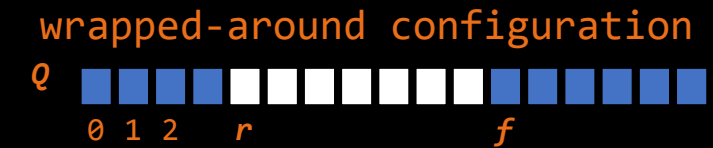


Queue Implementation - Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through front index
Operations	<ul style="list-style-type: none">▪ enqueue(item)▪ dequeue()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ enqueue(item) - $O(1)$▪ dequeue() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	<ul style="list-style-type: none">▪ Constant time to add and remove elements
Drawbacks	<ul style="list-style-type: none">▪ Limited space▪ Rightward drift problem

Queue Implementation – Circular Array

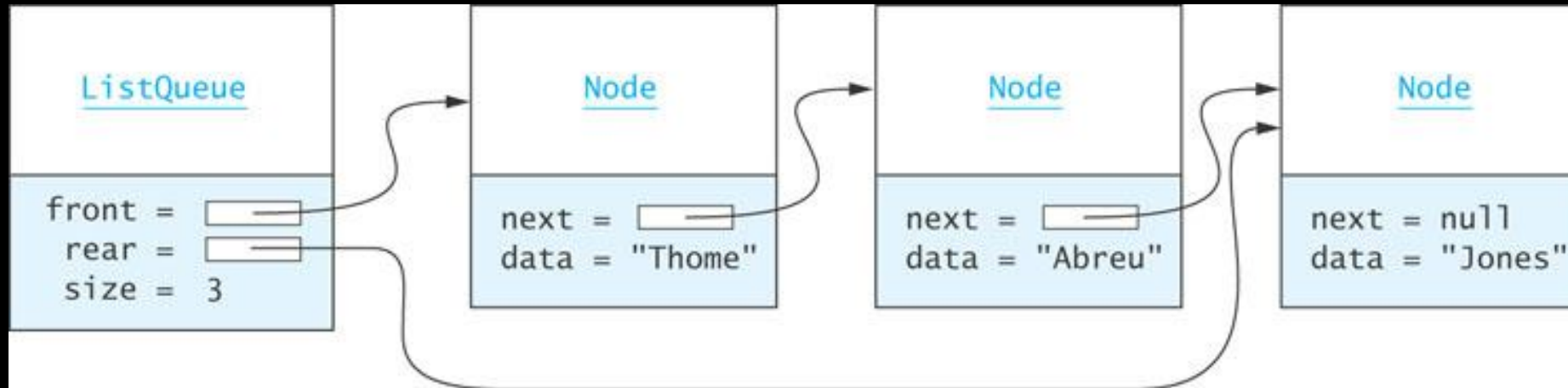
Use an array of size N in a circular fashion
Two variables keep track of the front and rear
 f index of the front element
 r index immediately past the rear element
Array location r is kept empty



Queue Implementation – Circular Array

Characteristics	<ul style="list-style-type: none">▪ Fixed Size▪ Stores Similar Elements▪ Access through front index
Operations	<ul style="list-style-type: none">▪ enqueue(item)▪ dequeue()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ enqueue(item) - $O(1)$▪ dequeue() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	<ul style="list-style-type: none">▪ Constant time to add and remove elements
Drawbacks	<ul style="list-style-type: none">▪ Limited space

Queue Implementation - Linked List



Queue Implementation – Linked List

Characteristics	<ul style="list-style-type: none">▪ Flexible Size▪ Stores Similar Elements▪ Access through front pointer
Operations	<ul style="list-style-type: none">▪ enqueue(item)▪ dequeue()▪ size()▪ isEmpty()
Performance	<ul style="list-style-type: none">▪ enqueue(item) - $O(1)$▪ dequeue() - $O(1)$▪ size() - $O(1)$▪ isEmpty() - $O(1)$
Benefits	<ul style="list-style-type: none">▪ Constant time to add and remove elements; Variable size
Drawbacks	<ul style="list-style-type: none">▪ More memory

Queue in C++ STL

Operations	C++ STL
<ul style="list-style-type: none">▪ <code>enqueue(item)</code>▪ <code>dequeue()</code>▪ <code>size()</code>▪ <code>isEmpty()</code>	<ul style="list-style-type: none">▪ <code>push(g)</code> – Adds the element 'g' at the end of the queue▪ <code>pop()</code> – Deletes the first element of the queue▪ <code>size()</code> – Returns the size of the queue▪ <code>empty()</code> – Returns whether the queue is empty▪ <code>front()</code> – Returns a reference to the first element of the queue▪ <code>back()</code> – Returns a reference to the last element of the queue

<http://www.cplusplus.com/reference/queue/>

Queue Use Cases

■ Real World

- Buying Tickets
- Drive thru at fast food chains
- Appointments



■ Computers

- Print Queue
- Task Scheduling by OS
- Packet Forwarding by Routers

The screenshot shows a Windows XP print queue window. The title bar reads 'HP LaserJet 4050 Series PS - Use Printer Offline'. The menu bar includes 'Printer', 'Document', 'View', and 'Help'. The main area contains a table with the following data:

Document Name	Status	Owner	Pages	Size	Submitted	P
Microsoft Word - Queues_Paul_1007.doc		Paul Wolfgang	52	9.75 MB	1:53:18 PM 10/7/2003	
Microsoft Word - Stacks.doc		Paul Wolfgang	46	9.05 MB	1:53:57 PM 10/7/2003	
Microsoft Word - Trees2.doc		Paul Wolfgang	54	38.4 MB	1:54:41 PM 10/7/2003	

At the bottom of the window, a status bar indicates '3 document(s) in queue'.

Recommended Resources

- <http://www.cplusplus.com/reference/stl/>
- Stacks and Queue Questions on Edugator
- [OpenDSA Ch-9.8-9.14](#)