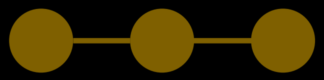
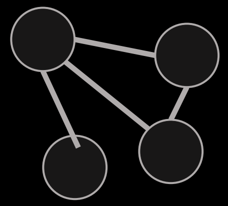
Lists, Stacks and Queues

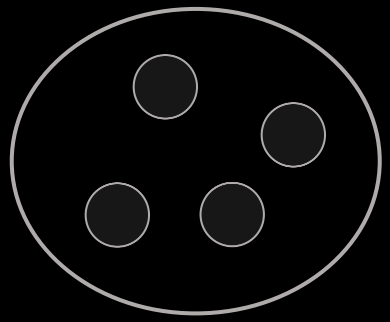
Data Structures & Algorithm

Categories of Data Structures

Linear Ordered	Non-linear Ordered	Not Ordered
Lists	Trees	Sets
Stacks	Graphs	Tables/Maps
Queues		







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

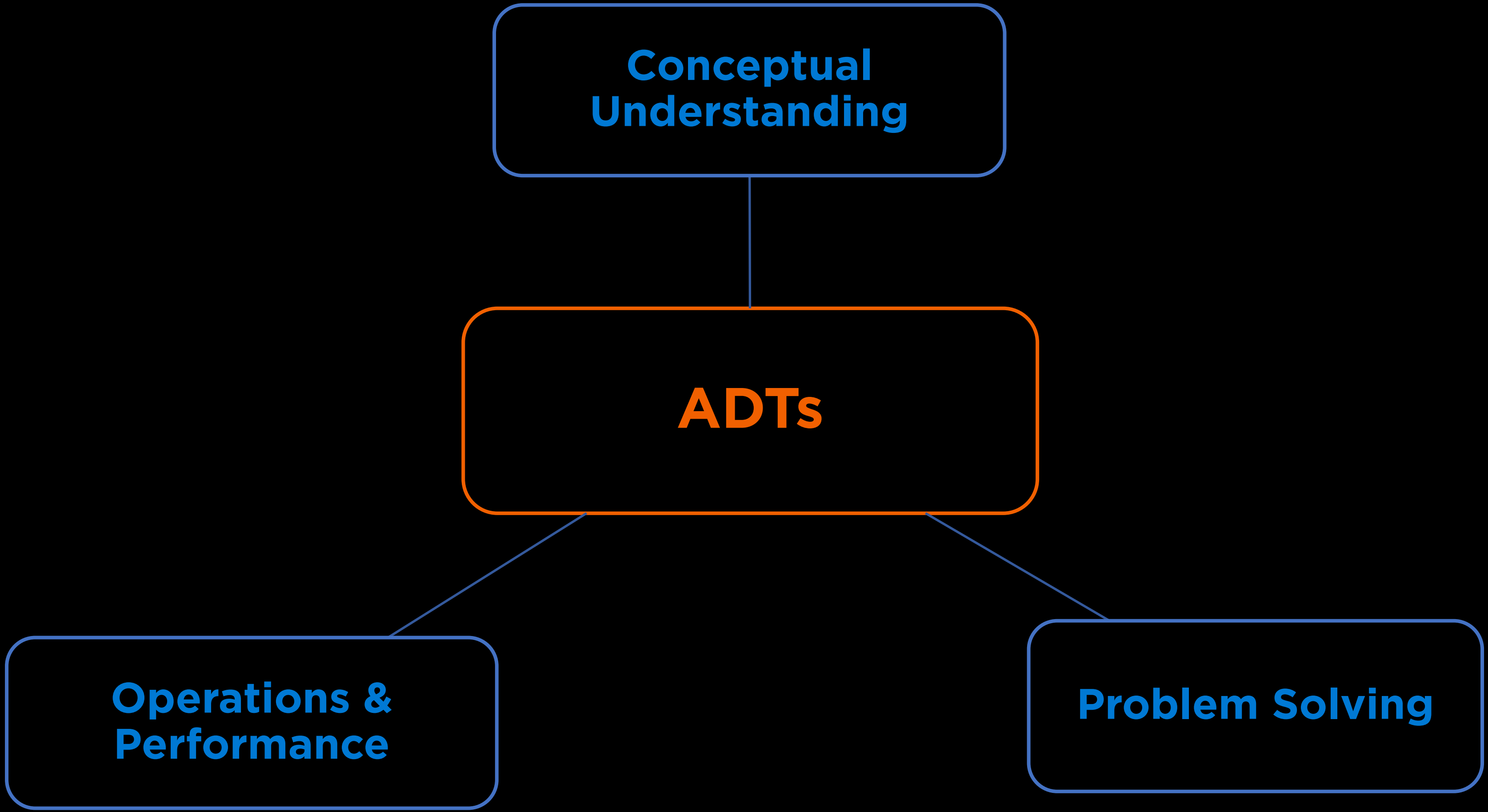
Abstract Data Types

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

<table>
<tr><td rowspan="1">Characteristics</td><td colspan="3">

- Fixed Size
- Stores Similar Elements
- Contiguous Indices
- Elements are stored contiguously in memory
- Allows Random Access

</td></tr>
<tr><td>Operations</td><td colspan="3"></td></tr>
<tr><td rowspan="4">Performance</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Benefits</td><td colspan="3"></td></tr>
<tr><td>Drawbacks</td><td colspan="3"></td></tr>
</table>

List – Example: Array

<table>
<tr><td>Characteristics</td><td colspan="3">▪ Fixed Size
▪ Stores Similar Elements
▪ Contiguous Indices
▪ Elements are stored contiguously in memory
▪ Allows Random Access</td></tr>
<tr><td>Operations</td><td colspan="3">▪ Adding – Beginning, Middle, End
▪ Removal – Beginning, Middle, End</td></tr>
<tr><td rowspan="4">Performance</td><td></td><td>Add</td><td>Remove</td></tr>
<tr><td>Beginning</td><td></td><td></td></tr>
<tr><td>End</td><td></td><td></td></tr>
<tr><td>Middle</td><td></td><td></td></tr>
<tr><td>Benefits</td><td colspan="3"></td></tr>
<tr><td>Drawbacks</td><td colspan="3"></td></tr>
</table>

List – Example: Array

<table>
<tr><td>Characteristics</td><td colspan="3">▪ Fixed Size
▪ Stores Similar Elements
▪ Contiguous Indices
▪ Elements are stored contiguously in memory
▪ Allows Random Access</td></tr>
<tr><td>Operations</td><td colspan="3">▪ Adding – Beginning, Middle, End
▪ Removal – Beginning, Middle, End</td></tr>
<tr><td rowspan="4">Performance</td><td></td><td>Add</td><td>Remove</td></tr>
<tr><td>Beginning</td><td>O(n)</td><td>O(n)</td></tr>
<tr><td>End</td><td>O(1)</td><td>O(1)</td></tr>
<tr><td>Middle</td><td>O(n)</td><td>O(n)</td></tr>
<tr><td>Benefits</td><td colspan="3"></td></tr>
<tr><td>Drawbacks</td><td colspan="3"></td></tr>
</table>

List – Example: Array

<table>
<tr><td rowspan="1">Characteristics</td><td colspan="3">▪ Fixed Size
▪ Stores Similar Elements
▪ Contiguous Indices
▪ Elements are stored contiguously in memory
▪ Allows Random Access</td></tr>
<tr><td>Operations</td><td colspan="3">▪ Adding – Beginning, Middle, End
▪ Removal – Beginning, Middle, End</td></tr>
<tr><td rowspan="4">Performance</td><td></td><td>Add</td><td>Remove</td></tr>
<tr><td>Beginning</td><td>O(n)</td><td>O(n)</td></tr>
<tr><td>End</td><td>O(1)</td><td>O(1)</td></tr>
<tr><td>Middle</td><td>O(n)</td><td>O(n)</td></tr>
<tr><td>Benefits</td><td colspan="3">▪ Constant Access Time, arr[i] = arr + (i*sizeOf(type))</td></tr>
<tr><td>Drawbacks</td><td colspan="3"></td></tr>
</table>

List – Example: Array

<table>
<tr><td rowspan="1">Characteristics</td><td colspan="3">

Fixed Size
Stores Similar Elements
Contiguous Indices
Elements are stored contiguously in memory
Allows Random Access

</td></tr>
<tr><td>Operations</td><td colspan="3">

Adding – Beginning, Middle, End
Removal – Beginning, Middle, End

</td></tr>
<tr><td rowspan="4">Performance</td><td></td><td>Add</td><td>Remove</td></tr>
<tr><td>Beginning</td><td>O(n)</td><td>O(n)</td></tr>
<tr><td>End</td><td>O(1)</td><td>O(1)</td></tr>
<tr><td>Middle</td><td>O(n)</td><td>O(n)</td></tr>
<tr><td>Benefits</td><td colspan="3">

Constant Access Time, arr[i] = arr + (i*sizeOf(type))

</td></tr>
<tr><td>Drawbacks</td><td colspan="3">

Expensive for adding/removing elements from front

</td></tr>
</table>

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;
}
```

https://onlinegdb.com/rk6AxHfsU

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

https://onlinegdb.com/rk6AxHfsU

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	▪ Consists of Nodes ○ 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

<table>
<tr><td>Characteristics</td><td colspan="3">▪ Consists of Nodes
○ Data
○ Pointer to Next Node
▪ Stores Similar Elements
▪ Elements are linked in memory but stored non-contiguously
▪ Does not allow Random Access</td></tr>
<tr><td>Operations</td><td colspan="3">▪ 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)</td></tr>
<tr><td rowspan="4">Performance</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Benefits</td><td colspan="3"></td></tr>
<tr><td>Drawbacks</td><td colspan="3"></td></tr>
</table>

List – Example: Single Linked List

<table>
<tr><td rowspan="1">Characteristics</td><td colspan="3">

- Consists of Nodes
  - Data
  - Pointer to Next Node
- Stores Similar Elements
- Elements are linked in memory but stored non-contiguously
- Does not allow Random Access

</td></tr>
<tr><td>Operations</td><td colspan="3">

- 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)

</td></tr>
<tr><td rowspan="4">Performance</td><td>PushFront</td><td>PushBack</td><td>AddBefore</td></tr>
<tr><td>PopFront</td><td>PopBack</td><td>AddAfter</td></tr>
<tr><td>TopFront</td><td>TopBack</td><td></td></tr>
<tr><td>Find</td><td>Erase</td><td>Empty</td></tr>
<tr><td>Benefits</td><td colspan="3"></td></tr>
<tr><td>Drawbacks</td><td colspan="3"></td></tr>
</table>

List – Example: Single Linked List

Characteristics	▪ Consists of Nodes ○ Data ○ Pointer to Next Node ▪ Stores Similar Elements ▪ Elements are linked in memory but stored non-contiguously ▪ Does not allow Random Access		
Operations	▪ 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	▪ Consists of Nodes ○ Data ○ Pointer to Next Node ▪ Stores Similar Elements ▪ Elements are linked in memory but stored non-contiguously ▪ Does not allow Random Access		
Operations	▪ 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	▪ Adding/Removing in front is faster, O(1)		
Drawbacks			

List – Example: Single Linked List

<table>
<tr><td rowspan="1">Characteristics</td><td colspan="3">

- Consists of Nodes
  - Data
  - Pointer to Next Node
- Stores Similar Elements
- Elements are linked in memory but stored non-contiguously
- Does not allow Random Access

</td></tr>
<tr><td>Operations</td><td colspan="3">

- 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)

</td></tr>
<tr><td rowspan="4">Performance</td><td>PushFront – O(1)</td><td>PushBack – O(n)</td><td>AddBefore – O(n)</td></tr>
<tr><td>PopFront – O(1)</td><td>PopBack – O(n)</td><td>AddAfter – O(1)</td></tr>
<tr><td>TopFront – O(1)</td><td>TopBack – O(n)</td><td></td></tr>
<tr><td>Find – O(n)</td><td>Erase – O(n)</td><td>Empty – O(1)</td></tr>
<tr><td>Benefits</td><td colspan="3">

- Adding/Removing in front is faster, O(1)

</td></tr>
<tr><td>Drawbacks</td><td colspan="3">

- Expensive for random access, O(n)
- TopBack, PushBack, PopBack and AddBefore are expensive

</td></tr>
</table>

List : Single Linked List with Tail

Characteristics	▪ Consists of Nodes and has a Tail
	  ○ Data
	  ○ Pointer to Next Node
	▪ Stores Similar Elements
	▪ Elements are linked in memory but stored non-contiguously
	▪ Does not allow Random Access		

Operations	▪ 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	▪ Solves the issue of pushing back and getting last element		
Drawbacks	▪ Expensive for PopBack and AddBefore		

List : Doubly Linked List with Tail

Characteristics	▪ Consists of Nodes ○ 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	▪ 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	▪ Solves the issue of PopBack and AddBefore		
Drawbacks	▪ Extra memory		

List – Example: Circular Linked List

- **Single Circular**

- **Doubly Circular**

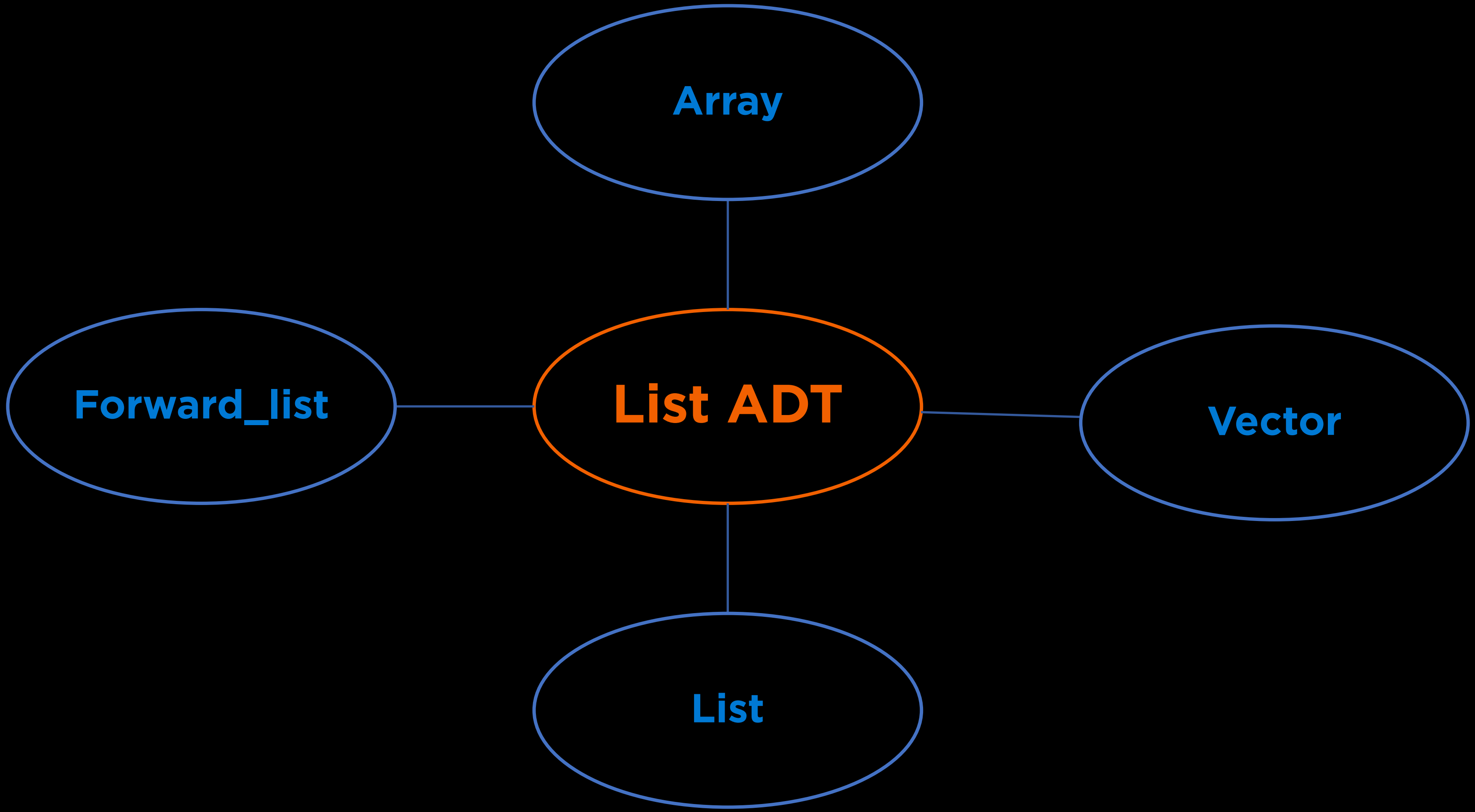
Array vs Linked List: Space

- Space
  - List size
    - **Array implementation** requires an estimate of the size when the list is created, or requires reallocation
    - **Linked implementation** can grow as needed
  - Element size
    - **Array implementation** requires storage of only the item
    - **Linked implementation** requires storage of the item and a pointer

Array vs Linked List: Time

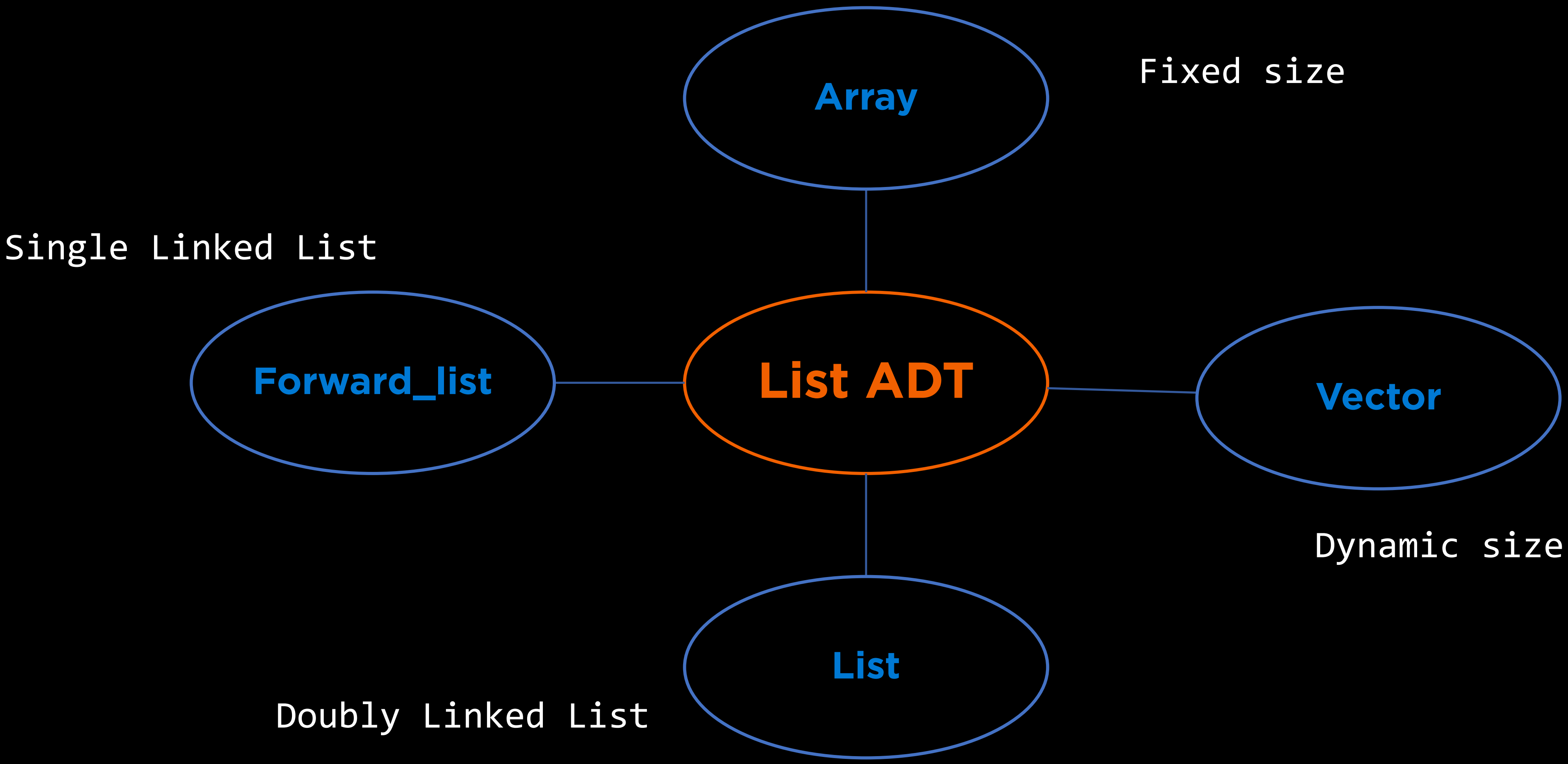
- Time
  - Access
    - **Array implementation** allows constant time access to any index
    - **Linked implementation** requires stepping through the list to access an item
  - Adding and Removing
    - **Array implementation** requires elements to be moved
    - **Linked implementation** insertion or removal is constant time if the iterator is where you want it

Lists in C++ Standard Template Library



http://www.cplusplus.com/reference/stl/

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

```
list<int> mylist (4,100);    // [100, 100, 100, 100]
mylist.push_front (200);     // [200, 100, 100, 100, 100]
for(list<int>::iterator it = mylist.begin(); it != mylist.end(); ++it)
      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

<table>
<tr><th colspan="4">category</th><th>properties</th><th>valid expressions</th></tr>
<tr><td colspan="4" rowspan="2">all categories</td><td>copy-constructible, copy-assignable and destructible</td><td><code>X b(a);</code>
<code>b = a;</code></td></tr>
<tr><td>Can be incremented</td><td><code>++a</code>
<code>a++</code></td></tr>
<tr><td rowspan="10">Random Access</td><td rowspan="6">Bidirectional</td><td rowspan="5">Forward</td><td rowspan="2">Input</td><td>Supports equality/inequality comparisons</td><td><code>a == b</code>
<code>a != b</code></td></tr>
<tr><td>Can be dereferenced as an rvalue</td><td><code>*a</code>
<code>a->m</code></td></tr>
<tr><td>Output</td><td>Can be dereferenced as an lvalue
(only for mutable iterator types)</td><td><code>*a = t</code>
<code>*a++ = t</code></td></tr>
<tr><td rowspan="2"></td><td>default-constructible</td><td><code>X a;</code>
<code>X()</code></td></tr>
<tr><td>Multi-pass: neither dereferencing nor incrementing affects dereferenceability</td><td><code>{ b=a; *a++; *b; }</code></td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td><code>--a</code>
<code>a--</code>
<code>*a--</code></td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Supports arithmetic operators + and -</td><td><code>a + n</code>
<code>n + a</code>
<code>a - n</code>
<code>a - b</code></td></tr>
<tr><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td><code>a < b</code>
<code>a > b</code>
<code>a <= b</code>
<code>a >= b</code></td></tr>
<tr><td>Supports compound assignment operations += and -=</td><td><code>a += n</code>
<code>a -= n</code></td></tr>
<tr><td>Supports offset dereference operator ([])</td><td><code>a[n]</code></td></tr>
</table>

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

Iterators: Forward

```cpp
// Forward Lists - support forward iterators: https://www.cplusplus.com/reference/forward_list/forward_list/

std::forward_list<int> flist_container {1, 2, 3, 4, 5};

for(auto it = flist_container.begin(); it != flist_container.end(); it++)
{
    std::cout << *it << std::endl;
    std::cout << *(--it);  //This line will throw an error as iterator does not support going backward
    std::cout << *(it + 2);  //This line will throw an error as iterator does not support random access
    break;
 }
```

https://onlinegdb.com/UpmaNAbTd

Iterators: Bidirectional

```
// Lists - support bidirectional iterators: https://www.cplusplus.com/reference/list/list/
std::list<int> list_container {1, 2, 3, 4, 5};

for(auto it = list_container.begin(); it != list_container.end(); it++)
{
    std::cout << *it << " ";
    std::cout << *(--it) << std::endl;  //This line is undefined behavior
    //std::cout << *(it + 2);  //This line will throw an error as iterator does not support random access
    break;
 }
```

https://onlinegdb.com/UpmaNAbTd

Iterators: Random Access

```
// Vectors - support random access iterators: https://cplusplus.com/reference/vector/vector/

std::vector<int> vector_container {1, 2, 3, 4, 5};

for(auto it = vector_container.begin(); it != vector_container.end(); it++)
{
    std::cout << *it << " ";
    std::cout << *(--it) << " ";  //This line is undefined behavior
    std::cout << *(it + 2);
     break;
 }
```

https://onlinegdb.com/UpmaNAbTd

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)

```
int arr1[] = {16,22,77,129};
int arr2[] = {1,2,7,29,77,155,166};

//Creating Lists from Arrays
list<int> list1 (arr1, (arr1 + (sizeof(arr1) / sizeof(int))) );
list<int> list2 (arr2, (arr2 + (sizeof(arr2) / sizeof(int))) );

//Final Merged Sorted List
list<int> list3;

list<int>::iterator l1 = list1.begin();
list<int>::iterator l2 = list2.begin();

while(l1 != list1.end() && l2 != list2.end())
{
    if((*l1) < (*l2))
    {
        list3.push_back(*l1);
        l1++;
    }
    else
    if((*l2) < (*l1))
    {
        list3.push_back(*l2);
        l2++;
    }
    else
    {
        list3.push_back(*l1);
        l1++;
        l2++;
    }
}

while(l1 != list1.end())
{
    list3.push_back(*l1);
    l1++;
}

while(l2 != list2.end())
{
    list3.push_back(*l2);
    l2++;
}
```

https://onlinegdb.com/rkMcymjoI

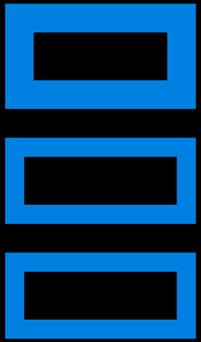
Recommended Resources

- http://www.cplusplus.com/reference/stl/
- Linked List Questions on Stepik
- 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	▪ Fixed Size ▪ Stores Similar Elements ▪ Access through top index
Operations	
Performance	
Benefits	
Drawbacks	

Stack Implementation - Array

Characteristics	▪ Fixed Size ▪ Stores Similar Elements ▪ Access through top index
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	
Benefits	
Drawbacks	

Stack Implementation - Array

Characteristics	▪ **Fixed Size** ▪ **Stores Similar Elements** ▪ **Access through top index**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Benefits	
Drawbacks	

Stack Implementation - Array

Characteristics	▪ **Fixed Size** ▪ **Stores Similar Elements** ▪ **Access through top index**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) – O(1) ▪ pop() - O(1) ▪ peek() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	
Drawbacks	

Stack Implementation - Array

Characteristics	▪ **Fixed Size** ▪ **Stores Similar Elements** ▪ **Access through top index**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) – O(1) ▪ pop() - O(1) ▪ peek() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	▪ Constant time to add and remove elements
Drawbacks	

Stack Implementation - Array

Characteristics	▪ **Fixed Size** ▪ **Stores Similar Elements** ▪ **Access through top index**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) – O(1) ▪ pop() - O(1) ▪ peek() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	▪ Constant time to add and remove elements
Drawbacks	▪ Fixed size and no random access

Stack Implementation – Linked List

Characteristics	
Operations	
Performance	
Benefits	
Drawbacks	

Stack Implementation – Linked List

Stack Implementation – Linked List

Characteristics	▪ Flexible Size ▪ Stores Similar Elements ▪ Access through top pointer
Operations	
Performance	
Benefits	
Drawbacks	

Stack Implementation – Linked List

Characteristics	▪ **Flexible Size** ▪ **Stores Similar Elements** ▪ **Access through top pointer**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	
Benefits	
Drawbacks	

Stack Implementation – Linked List

Characteristics	▪ Flexible Size ▪ Stores Similar Elements ▪ Access through top pointer
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Benefits	
Drawbacks	

Stack Implementation – Linked List

Characteristics	▪ **Flexible Size** ▪ **Stores Similar Elements** ▪ **Access through top pointer**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) – O(1) ▪ pop() - O(1) ▪ peek() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	
Drawbacks	

Stack Implementation – Linked List

Characteristics	▪ **Flexible Size** ▪ **Stores Similar Elements** ▪ **Access through top pointer**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) – O(1) ▪ pop() - O(1) ▪ peek() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	▪ Constant time to add and remove elements; Variable size
Drawbacks	

Stack Implementation – Linked List

Characteristics	▪ **Flexible Size** ▪ **Stores Similar Elements** ▪ **Access through top pointer**
Operations	▪ push(item) ▪ pop() ▪ peek() ▪ size() ▪ isEmpty()
Performance	▪ push(item) – O(1) ▪ pop() - O(1) ▪ peek() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	▪ Constant time to add and remove elements; Variable size
Drawbacks	▪ More memory

Stack in C++ STL

Operations	C++ STL
▪ **push(item)** ▪ **pop()** ▪ **peek()** ▪ **size()** ▪ **isEmpty()**	▪ push(g) – Adds the element ‘g’ at the top of the stack ▪ pop() – Deletes the topmost element of the stack ▪ top() – Returns a reference to the topmost element of the stack ▪ size() – Returns the size of the stack ▪ empty() – Returns whether the stack is empty

http://www.cplusplus.com/reference/stack/stack/

Stack in C++ STL: Check Palindrome

Stack in C++ STL: Check Palindrome

```
bool checkPalindrome(string s)
{
    stack<char> stk;
    int midIndex = s.length()/2;

    if(s.length() < 2)
        return true;

    for(int i = 0; i < s.length()/2; i++)
        stk.push(s.at(i));

    if(s.length() % 2 != 0)
         midIndex += 1;

    for(int i = midIndex; i < s.length(); i++)
    {
        if(stk.top() != s.at(i))
            return false;
        stk.pop();
    }

    return true;
}
```

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

Pushes first half of string characters on stack

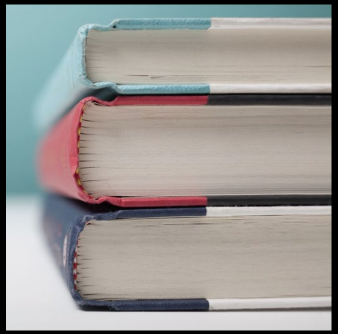
Ignores central element if the size of string is odd

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



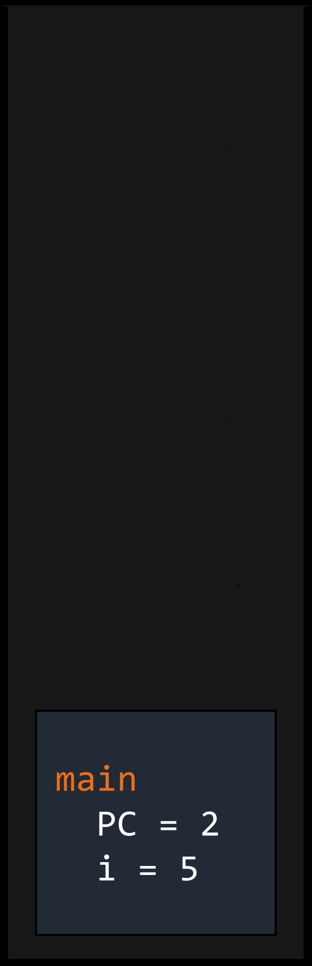
- **Computers**
  - Function Call Stack
  - Evaluate Expressions
  - Backtracking
  - Balanced Parenthesis
  - Undo (CTRL + Z)
  - Back button

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)
          {
              …
          }
```



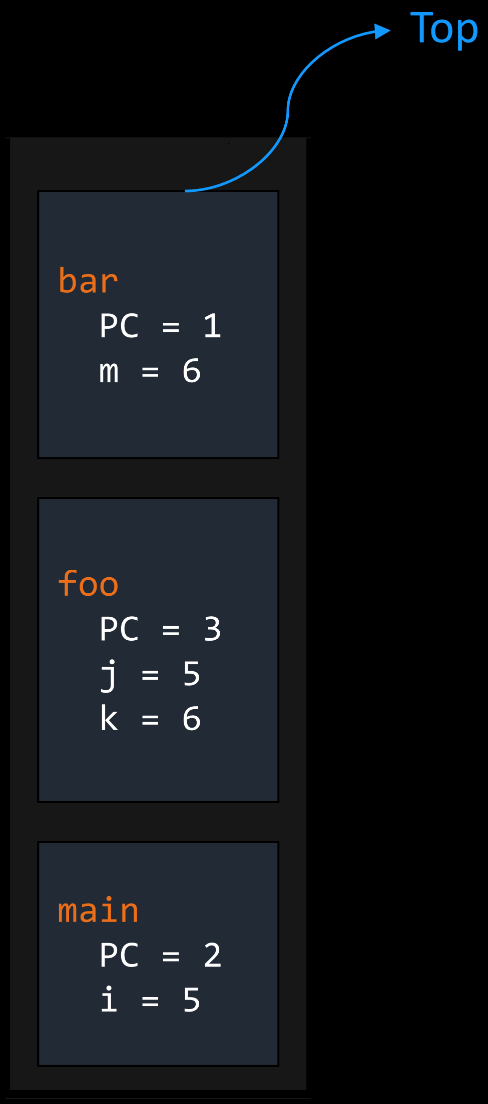
Stack Use Cases: Call Stack

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

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

          bar(int m)
          {
              …
          }
```

```
foo
  PC = 3
  j = 5
  k = 6

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: 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
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

Stack Use Cases: Postfix Evaluation

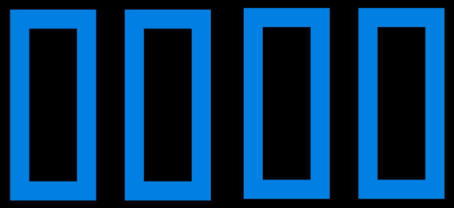
```
create an empty stack of integers
while there are more tokens
  get the next token
  if the first character of the token is a digit
     push the token on the stack
  else if the token is an operator
     pop the right operand off the stack
     pop the left operand off the stack
     evaluate the operation
     push the result onto the stack
pop the stack and return the result
```

4	7	*	20	-

Queues

Queue

- **First in First Out (FIFO)**



Queue ADT

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

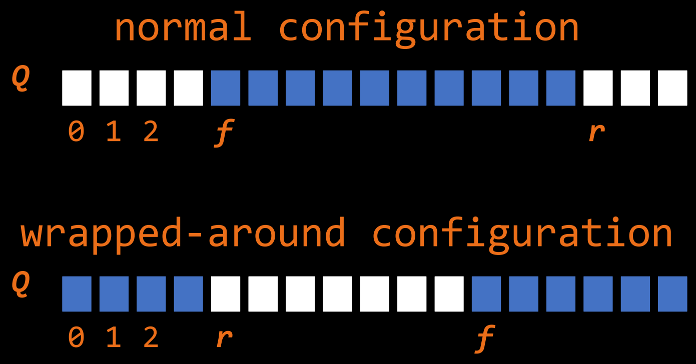
- **Operations**
  - enqueue(item): inserts an element to the back of the queue
  - dequeue(): removes the element from the front
  - size(): returns the number of elements stored
  - isEmpty(): indicates whether no elements are stored

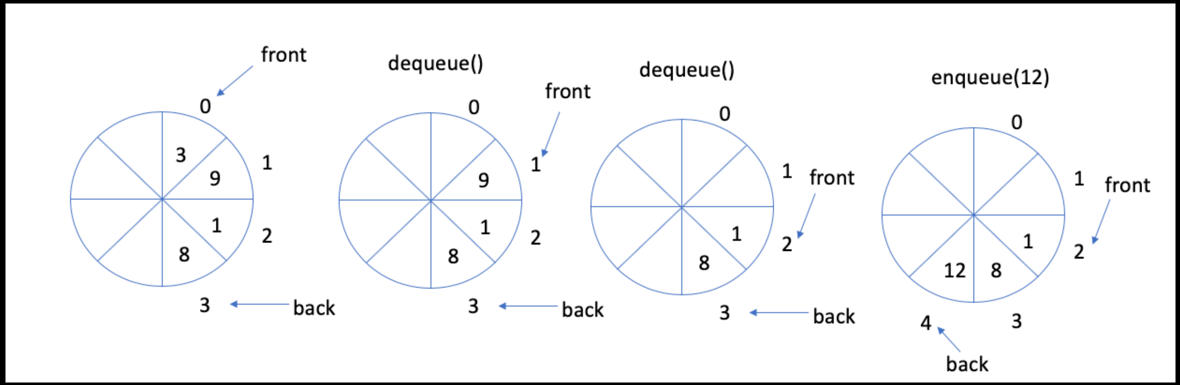
Queue Implementation - Array

Characteristics	▪ **Fixed Size** ▪ **Stores Similar Elements** ▪ **Access through front index**
Operations	▪ enqueue(item) ▪ dequeue() ▪ size() ▪ isEmpty()
Performance	▪ enqueue(item) - O(1) ▪ dequeue() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	▪ Constant time to add and remove elements
Drawbacks	▪ 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



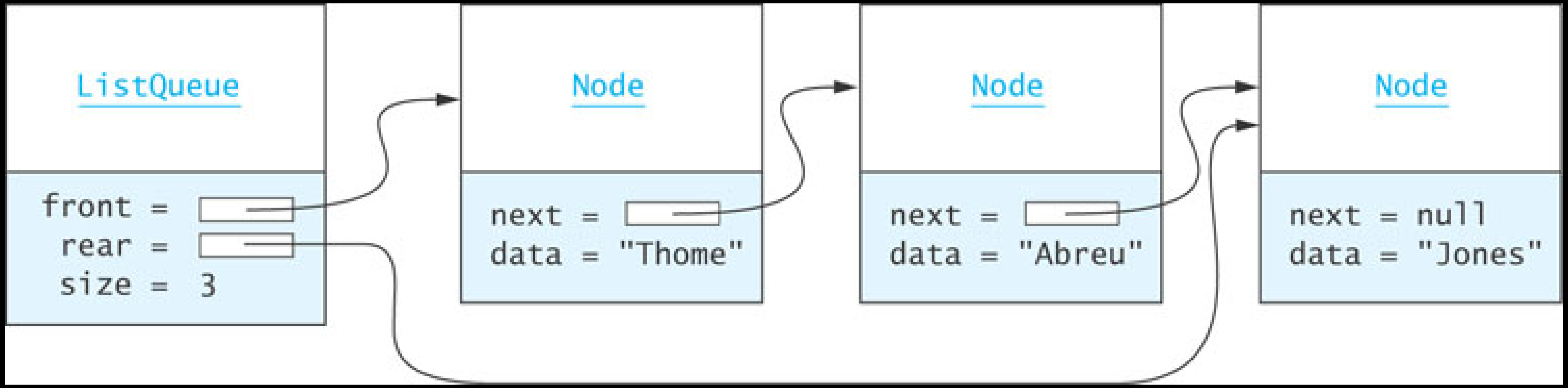


https://stepik.org/lesson/354738/step/1?unit=338782

Queue Implementation – Circular Array

Characteristics	▪ **Fixed Size** ▪ **Stores Similar Elements** ▪ **Access through front index**
Operations	▪ enqueue(item) ▪ dequeue() ▪ size() ▪ isEmpty()
Performance	▪ enqueue(item) - O(1) ▪ dequeue() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	▪ Constant time to add and remove elements
Drawbacks	▪ Limited space

Queue Implementation – Linked List



Queue Implementation – Linked List

Characteristics	▪ **Flexible Size** ▪ **Stores Similar Elements** ▪ **Access through front pointer**
Operations	▪ enqueue(item) ▪ dequeue() ▪ size() ▪ isEmpty()
Performance	▪ enqueue(item) - O(1) ▪ dequeue() - O(1) ▪ size() - O(1) ▪ isEmpty() - O(1)
Benefits	▪ Constant time to add and remove elements; Variable size
Drawbacks	▪ More memory

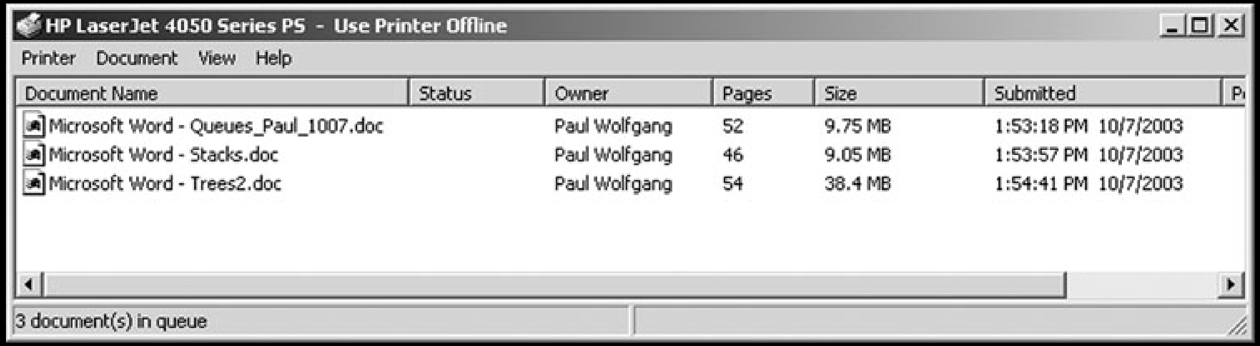
Queue in C++ STL

Operations	C++ STL
▪ **enqueue(item)** ▪ **dequeue()** ▪ **size()** ▪ **isEmpty()**	▪ push(g) – Adds the element ‘g’ at the end of the queue ▪ pop() – Deletes the first element of the queue ▪ size() – Returns the size of the queue ▪ empty() – Returns whether the queue is empty ▪ front() – Returns a reference to the first element of the queue ▪ back() – 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



HP LaserJet 4050 Series PS - Use Printer Offline

Printer Document View Help

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	

3 document(s) in queue

Recommended Resources

- http://www.cplusplus.com/reference/stl/
- Stacks and Queue Questions on Stepik
- OpenDSA Ch-9.8-9.14