CPE 212 - Fundamentals of Software Engineering

•••

Vectors & Heaps

Outline

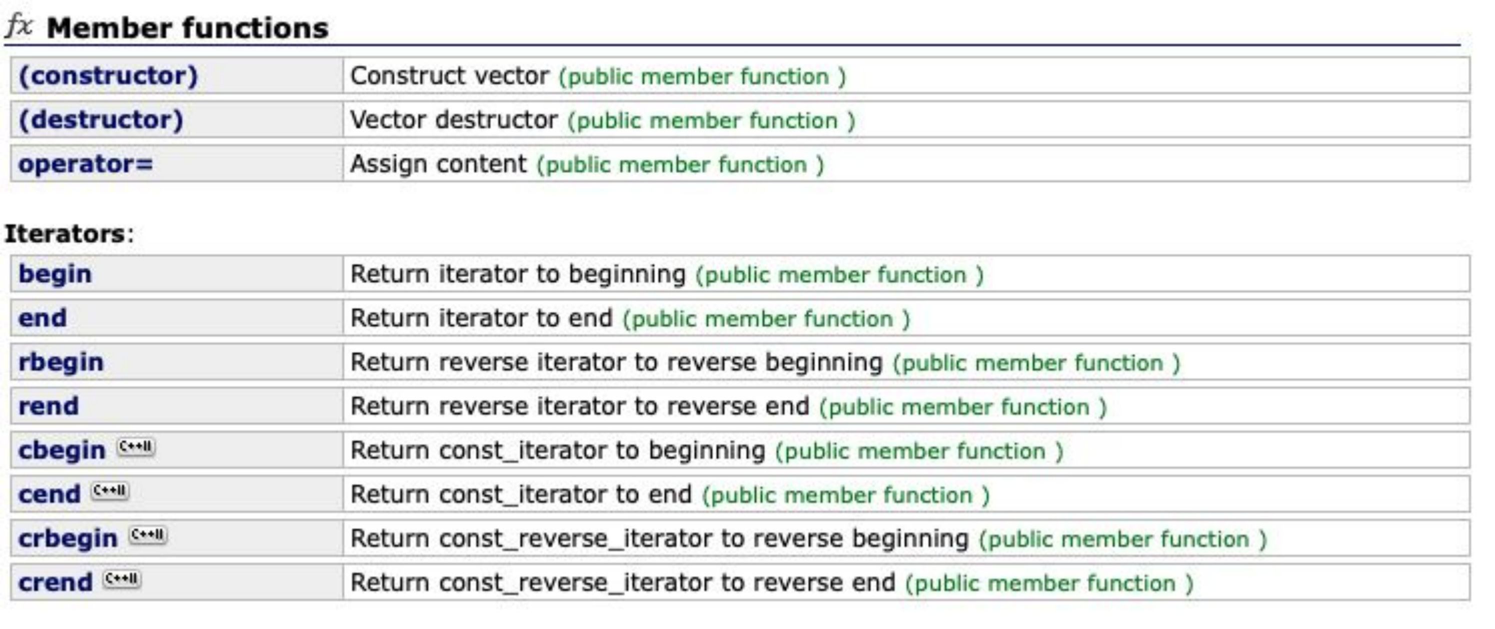
- Vector Introduction
- Using Vectors
- Vector Examples
- Implementing Heaps with Vectors

Project 4 Due
Friday March 13th 11:59pm

C++ Standard Template Library (STL) - Vectors

- Similar to arrays, they use contiguous memory locations for their elements
- The elements can be accessed using offsets on regular pointers just as efficiently as arrays
- Size is dynamic
- Consumes more memory than a array due to its dynamic nature
- Very efficient in access, just like arrays
- Not as efficient as lists at inserting or removing from any place but the end

C++ Standard Template Library (STL) - Vectors



fx Member functions

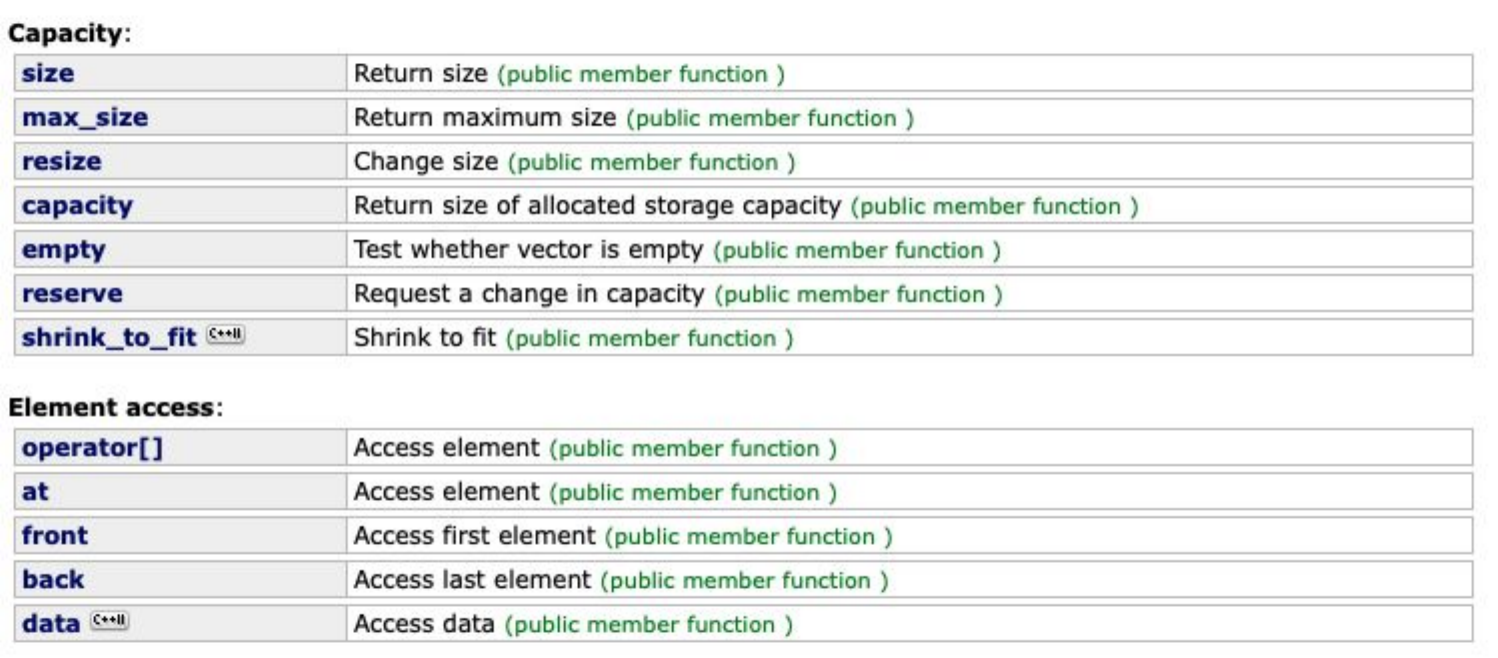
(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Reference: http://www.cplusplus.com/reference/vector/vector/

C++ Standard Template Library (STL) - Vectors



Capacity:

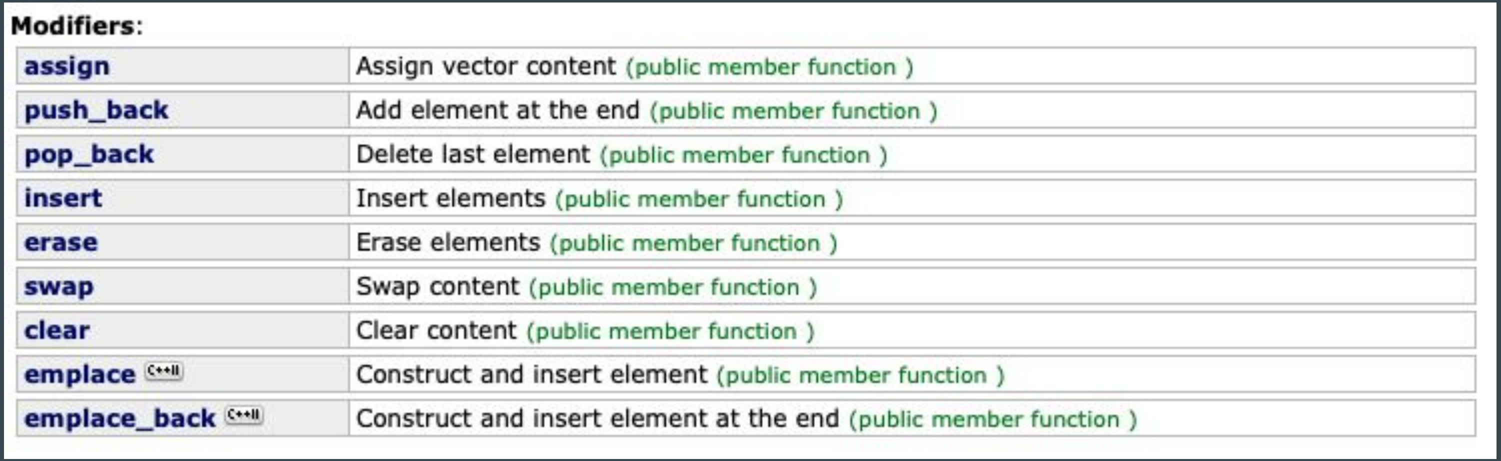
size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Reference: http://www.cplusplus.com/reference/vector/vector/

C++ Standard Template Library (STL) - Vectors



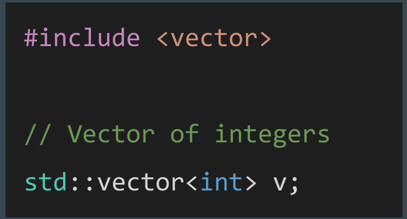
Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

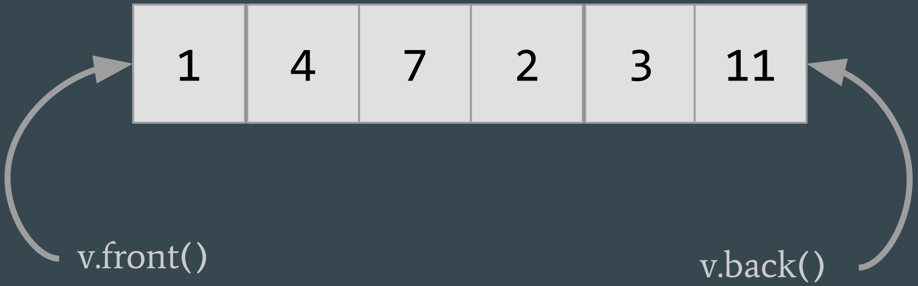
Reference: http://www.cplusplus.com/reference/vector/vector/

Vector - Initialization

- Must include the vector library
- Initialization must use the std:: notation or you can provide the “using namespace std”

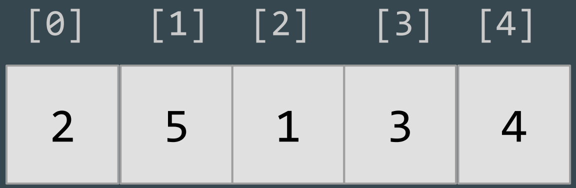


```cpp
#include <vector>

// Vector of integers
std::vector<int> v;
```



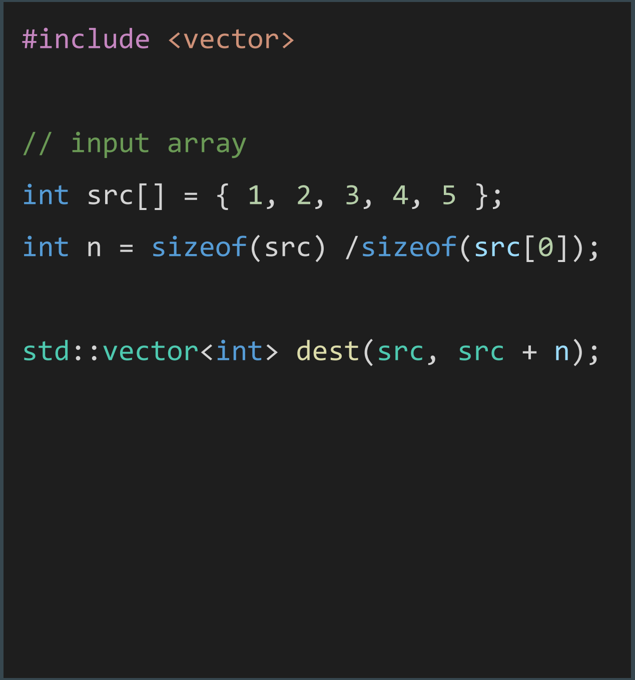
Vector - Adding elements

```
#include <vector>


// Vector of integers
std::vector<int> v;


v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);
```

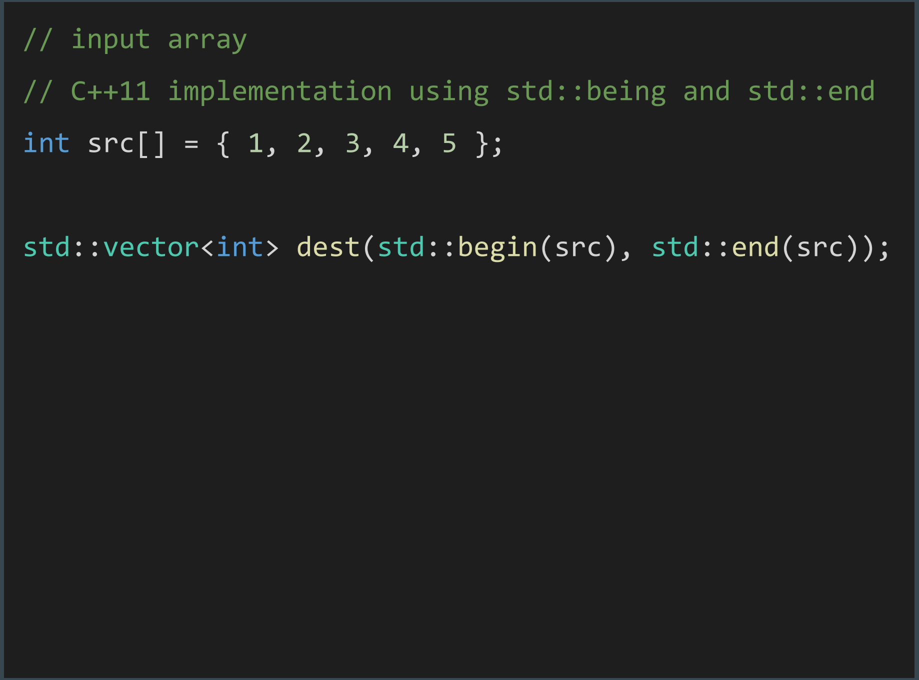


[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

Vector - Copy array into vector



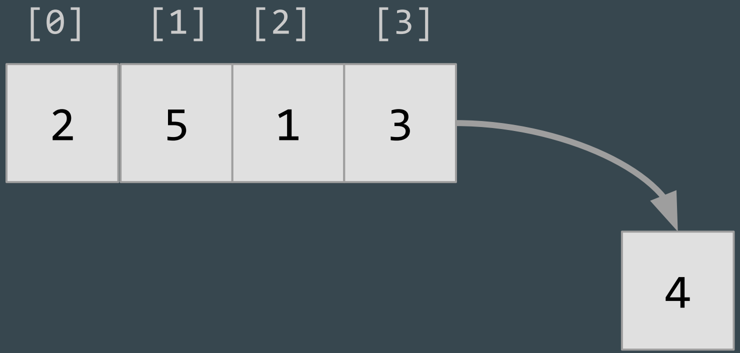
```
#include <vector>

// input array
int src[] = { 1, 2, 3, 4, 5 };
int n = sizeof(src) /sizeof(src[0]);

std::vector<int> dest(src, src + n);
```



```
// input array
// C++11 implementation using std::being and std::end
int src[] = { 1, 2, 3, 4, 5 };

std::vector<int> dest(std::begin(src), std::end(src));
```

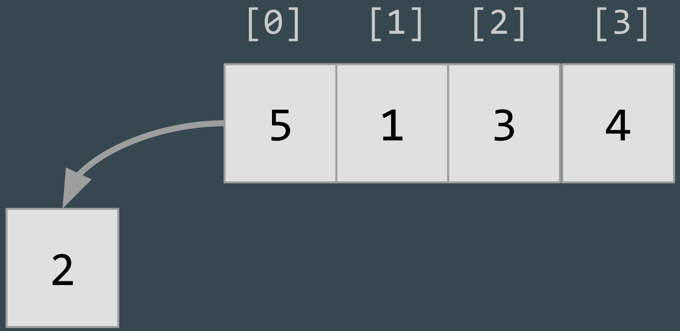
Vector - Removing elements

```cpp
#include <vector>


// Vector of integers
std::vector<int> v;


v.pop_back();
```



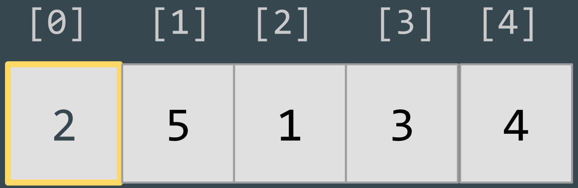
Vector - Removing elements

```
#include <vector>


// Vector of integers
std::vector<int> v;


v.erase(v.begin());
```



Vector - Getting first element

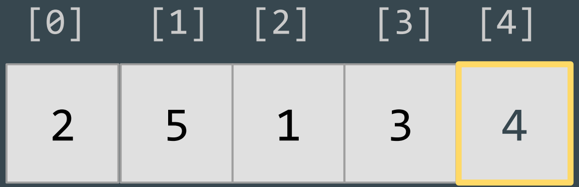
```cpp
#include <vector>

// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

v.front();
```



[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

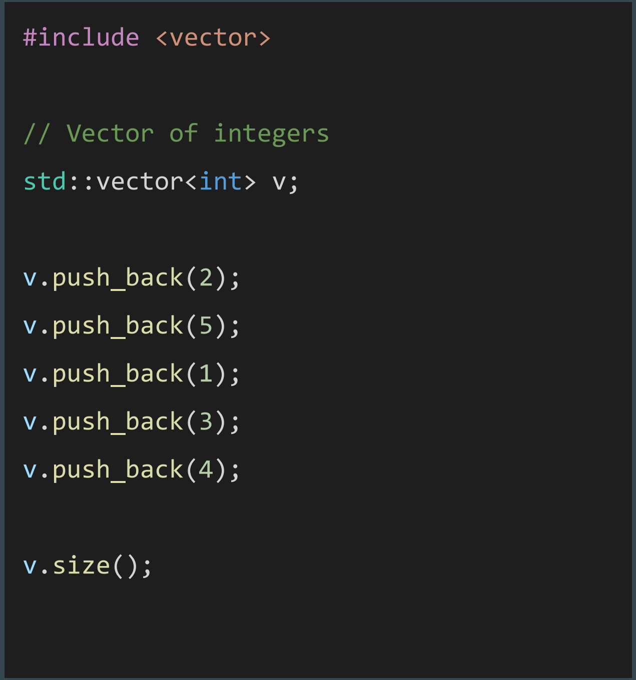
Vector - Getting last element

```
#include <vector>

// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

v.back();
```

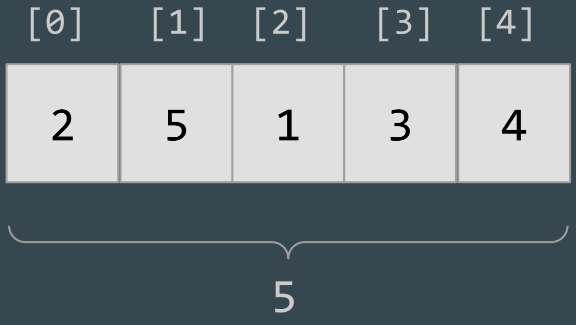


[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

Vector - Getting size of vector



```cpp
#include <vector>

// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

v.size();
```



Vector - Checking for empty

```cpp
#include <vector>

// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

v.empty();
```

[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

False

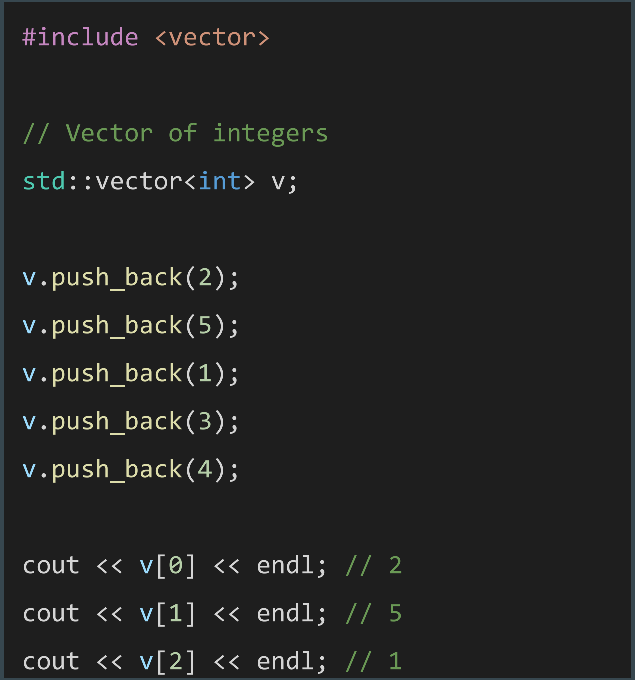
Vector - Checking for empty

```
#include <vector>


// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

v.empty();
```

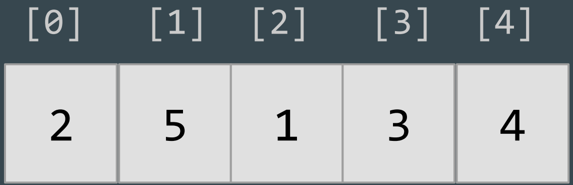
[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

False

Vector - Element access



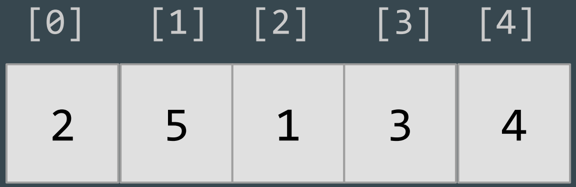
```
#include <vector>

// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

cout << v[0] << endl; // 2
cout << v[1] << endl; // 5
cout << v[2] << endl; // 1
```



[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

Vector - Element access

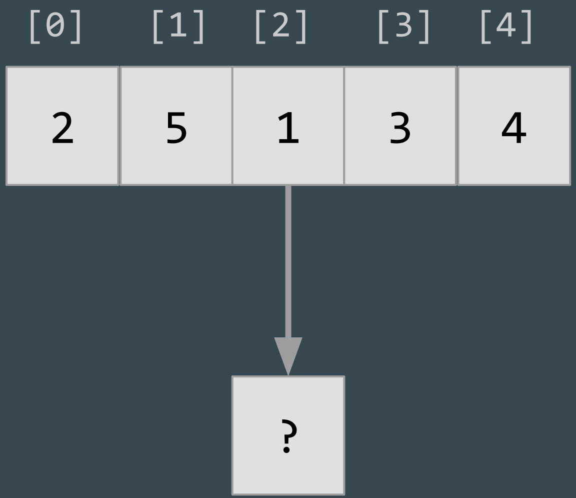
```
#include <vector>

// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

cout << v[0] << endl; // 2
cout << v[1] << endl; // 5
cout << v[2] << endl; // 1
```



[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

Vector - Element access

```
#include <vector>

// Vector of integers
std::vector<int> v;

v.push_back(2);
v.push_back(5);
v.push_back(1);
v.push_back(3);
v.push_back(4);

v.clear();
v.empty();  // True
```



[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

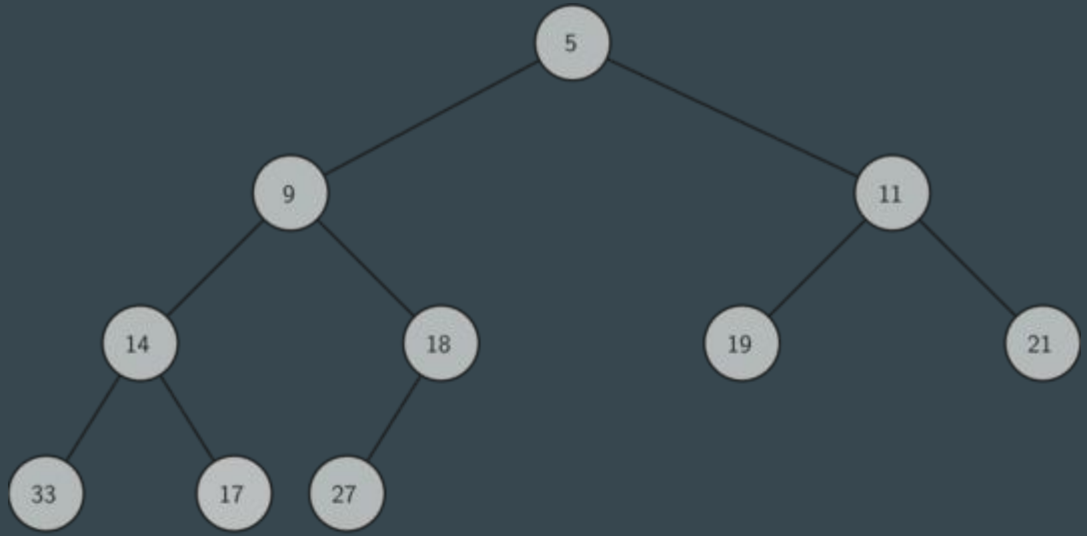
Heap ADT

- Heap
  - A complete binary tree, each of whose elements satisfies the heap ordering property
    - Min-heap : the value of each node is greater than or equal to the value of its parent, with the min-value element at the root
    - Max-heap : the value of each node is less than or equal to the value of its parent, with the max-value element at the root
  - Shape property & order property
  - A heap is not a sorted structure and can be regarded as partially ordered.

Complete Binary Tree

- Complete Binary Tree
  - A complete binary tree of height *h* is full down to height *h* - 1.
  - Example:
    - Height = 5
    - Full from height = 1 to height = 4
  - When a node at height 4 has children all nodes at the same height and to it's left have two children each
  - When a node at height 4 has one child it's a left child
  - All nodes at *h* - 2 and above have two children each

Heap Order Property

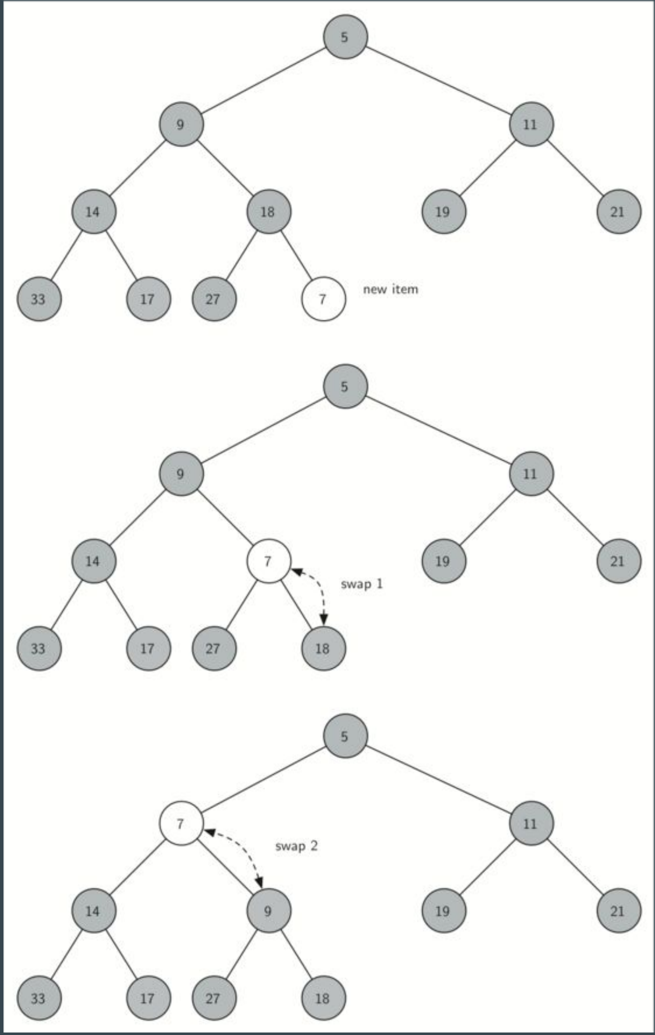


0	5	9	11	14	18	19	21	33	17	27	
0	1	2	3	4	5	6	7	8	9	10	11

https://runestone.academy/runestone/books/published/cppds/Trees/BinaryHeapImplementation.html

Heap Operations - Insert

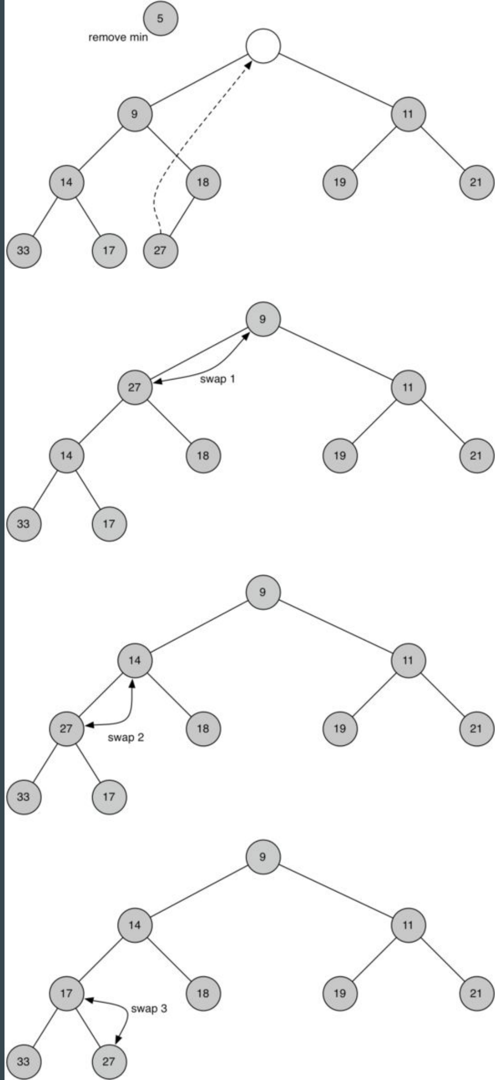
- Insert method adds the element to the end of the vector using the push_back method
- Using the reheapUp and swap methods we can compare the newly inserted node with its parent moving it up the tree
- Remember the following relationships for a given parent *p*
  - Left Child : $2p$
  - Right Child: $2p + 1$
  - Parent of node *n*: $n/2$



https://runestone.academy/runestone/books/published/cppds/Trees/BinaryHeapImplementation.html

Heap Operations - Remove

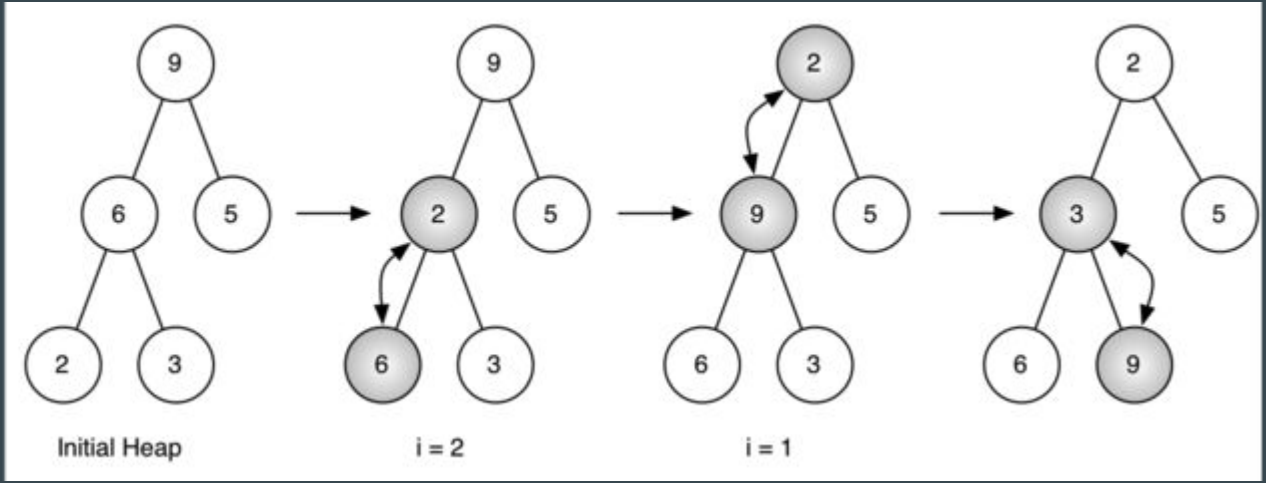
- This is a min-heap so we will define the delMin method that deletes the first item in the vector
- Deleting the head node we promote the last node to the top
- Using the reheapDown method we swap the current node with its smallest child less than the root



https://runestone.academy/runestone/books/published/cppds/Trees/BinaryHeapImplementation.html

Heap Operations - Build

- This method takes in a vector and builds a min-heap with the given values
- Instead of iterating through the list and building it one at a time you use the reheapDown method to sort it in its original form



https://runestone.academy/runestone/books/published/cppds/Trees/BinaryHeapImplementation.html