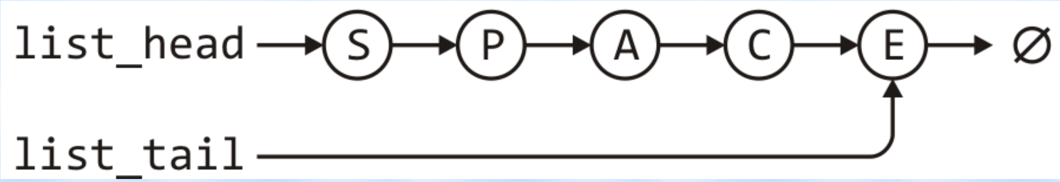
The issue

A significant issue with linked lists (and later, trees) are that node-based data structures require $\Theta(n)$ calls to new

- This requires a call to the operating system requesting a memory allocation

Using an array?

Suppose we store this linked list in an array?




```
list_head = 5;
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		-1	0		3	2	

Using an array?

Rather than using, `-1`, use a constant assigned that value

– This makes reading your code easier

```
list_head = 5;
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		NULLPTR	0		3	2	

A solution

To achieve this, we must create an array of objects that:

- Store the value
- Store the array index where the next entry is stored

```
template <typename Type>
class Single_node {
    private:
        Type node_value;
        next_node int;
    public:
        Type value() const;
        int next() const;
};
```

A solution

Now, memory allocation is done once in the constructor:

```
template <typename Type>
class Single_list {
    private:
        int list_capacity;
        int list_head;
        int list_tail;
        int list_size;
        Single_node<Type> *node_pool;

        static const int NULL;
    public:
        Single_list( int = 16 );
        // member functions
};

const int Single_list::NULLPTR = -1;
```

```
template <typename Type>
Single_list<Type>::Single_list( int n
):
list_capacity( std::max( 1, n ) ),
list_head( NULLPTR ),
list_tail( NULLPTR ),
list_size( 0 ),
node_pool( new Single_node<Type>[n] )
{
    // Empty constructor
}
```

A solution

Question: how do you track unused nodes?

- We could keep a separate container (a stack) which contains the indices of all unused nodes

```
list_head = 5;
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		NULLPTR	0		3	2	

```
stack_size = 3;
```

0	1	2	3	4	5	6	7
7	1	4					

A solution

The stack would be initialized with all the entries

```
list_head = NULLPTR;
list_tail = NULLPTR;
```

0	1	2	3	4	5	6	7

```
stack_size = 8;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

When pushing onto the list, the entry at the top of the stack is used

```
list_head = NULLPTR;
list_tail = NULLPTR;
```

0	1	2	3	4	5	6	7

```
stack_size = 8;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

Now, `push_front( '0' )` would result in

```
list_head = 7;
list_tail = 7;
```

0	1	2	3	4	5	6	7
							0
							NULLPTR

```
stack_size = 7;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

Suppose we call `push_front( 'N' )`

```
list_head = 7;
list_tail = 7;
```

0	1	2	3	4	5	6	7
							O
							NULLPTR

```
stack_size = 7;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

Suppose we call `push_front( 'N' )`

– The next node is at index 6

```
list_head = 6;
list_tail = 7;
```

0	1	2	3	4	5	6	7
						N	O
						7	NULLPTR

```
stack_size = 6;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

Suppose we call `push_back( 'R' )`

```
list_head = 6;
list_tail = 7;
```

0	1	2	3	4	5	6	7
						N	O
						7	NULLPTR

```
stack_size = 6;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

Suppose we call `push_back( 'R' )`

- The next node is at index 5

```
list_head = 6;
list_tail = 5;
```

0	1	2	3	4	5	6	7
					R	N	O
					NULLPTR	7	5

```
stack_size = 5;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

Finally, suppose we call `pop_front()`

```
list_head = 6;
list_tail = 5;
```

0	1	2	3	4	5	6	7
					R	N	O
					NULLPTR	7	5

```
stack_size = 5;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

A solution

Finally, suppose we call `pop_front()`

- The popped node is placed back into the stack

```
list_head = 7;
list_tail = 5;
```

0	1	2	3	4	5	6	7
					R	N	O
					NULLPTR	7	5

```
stack_size = 6;
```

0	1	2	3	4	5	6	7
0	1	2	3	4	6	6	7

A better solution

Problem:

- Our solution requires $\Theta(N)$ additional memory
- In our initial example, the unused nodes are 1, 4 and 7
- How about using these to define a second stack-as-linked-list?

```
list_head = 5;
list_tail = 2;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6		NULLPTR	0		3	2	

A better solution

Problem:

- Our solution requires $\Theta(N)$ additional memory
- In our initial example, the unused nodes are 1, 4 and 7
- How about using these to define a second stack-as-linked-list?

```
list_head = 5;
list_tail = 2;
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	3	2	NULLPTR

- We only need a head pointer for the stack-as-linked-list
  - The top of the stack is the first node

A better solution

Suppose we call `pop_front()`

```
list_head = 5;
list_tail = 2;
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	3	2	NULLPTR

A better solution

Suppose we call `pop_front()`

– The extra node is placed onto the stack

```
list_head = 3;
list_tail = 2;
stack_top = 5;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	1	2	NULLPTR

A better solution

Suppose we now call `push_back( 'D' )`

```
list_head = 3;
list_tail = 2;
stack_top = 5;
```

0	1	2	3	4	5	6	7
A		E	P		S	C	
6	4	NULLPTR	0	7	1	2	NULLPTR

A better solution

Suppose we now call `push_back( 'D' )`

- We pop the node off of the top of the stack

```
list_head = 3;
list_tail = 5;
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		D	C	
6	4	5	0	7	NULLPTR	2	NULLPTR

A better solution

Suppose we finally call `pop_front()` again

```
list_head = 3;
list_tail = 5;
stack_top = 1;
```

0	1	2	3	4	5	6	7
A		E	P		D	C	
6	4	5	0	7	NULLPTR	2	NULLPTR

A better solution

Suppose we finally call `pop_front()` again

– The node containing `'P'` is pushed back onto the stack

```
list_head = 0;
list_tail = 5;
stack_top = 3;
```

0	1	2	3	4	5	6	7
A		E	P		D	C	
6	4	5	1	7	NULLPTR	2	NULLPTR

A better solution

In this case, our data structure would be initialized to:

```
list_head = NULLPTR;
list_tail = NULLPTR;
stack_top = 0;
```

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	NULLPTR

A solution

Our class would look something like:

```
template <typename Type>
class Single_list {
    private:
        int list_head;
        int list_tail;
        int list_size;
        int list_capacity;
        Single_node<Type> *node_pool;
        int stack_top;

        static const int NULL;
    public:
        Single_list( int = 16 );
        // member functions
};

const int Single_list::NULLPTR = -1;
```

```
template <typename Type>
Single_list<Type>::Single_list( int n ):
list_head( NULLPTR ),
list_tail( NULLPTR ),
list_size( 0 ),
list_capacity( std::max( 1, n ) ),
node_pool( new Single_node<Type>[n] ),
stack_top( 0 ) {
    for ( int i = 1; i < capacity(); +
+i ) {
        node_pool[i - 1].next = i;
    }

    node_pool[capacity() - 1] = NULLPTR;
}
```

Analysis

This solution:

- Requires only three more member variable than our linked list class
- It still requires $\mathrm{O}(N)$ additional memory over an array
- All the run-times are identical to that of a linked list
- Only one call to new, as opposed to $\Theta(n)$
- There is a potential for up to $\mathrm{O}(N)$ wasted memory

Question: What happens if we run out of memory?

Reallocation of memory

Suppose we start with a capacity N but after a while, all the entries have been allocated

– We can double the size of the array and copy the entries over

```
list_head = 6;
list_tail = 4;
list_size = 8;
list_capacity = 8;
stack_top = NULLPTR;
```

0	1	2	3	4	5	6	7
C	R	U	T	R	U	S	T
7	2	0	1	NULLPTR	4	3	5

Reallocation of memory

Suppose we start with a capacity N but after a while, all the entries have been allocated

- We can double the size of the array and copy the entries over
- Only the stack needs to be updated and the old array deleted

```
list_head = 6;
list_tail = 4;
list_size = 8;
list_capacity = 16;
stack_top = 8;
```

0	1	2	3	4	5	6	7
C	R	U	T	R	U	S	T
7	2	0	1	NULLPTR	4	3	5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	R	U	T	R	U	S	T								
7	2	0	1	NULLPTR	4	3	5	9	10	11	12	13	14	15	NULLPTR

Reallocation of memory

Now `push_back( 'E' )` would use the next location

```
list_head = 6;
list_tail = 4;
list_size = 8;
list_capacity = 16;
stack_top = 8;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	R	U	T	R	U	S	T								
7	2	0	1	NULLPTR	4	3	5	9	10	11	12	13	14	15	NULLPTR

Reallocation of memory

Now `push_back( 'E' )` would use the next location

```
list_head = 6;
list_tail = 8;
list_size = 9;
list_capacity = 16;
stack_top = 9;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C	R	U	T	R	U	S	T	E							
7	2	0	1	8	4	3	5	NULLPTR	10	11	12	13	14	15	NULLPTR

Reallocation of memory

If at some point, we decide it is desirable to reduce the memory allocated, it might be easier to just insert the entries into a newer and smaller table

```
list_head = 4;
list_tail = 5;
list_size = 4;
list_capacity = 16;
stack_top = 7;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		T		D	A					A					
14	9	5	0	10	NULLPTR	8	12	NULLPTR	3	2	13	1	15	11	6

Reallocation of memory

If at some point, we decide it is desirable to reduce the memory allocated, it might be easier to just insert the entries into a newer and smaller table

```
list_head = 4;
list_tail = 5;
list_size = 4;
list_capacity = 16;
stack_top = 7;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		T		D	A					A					
14	9	5	0	10	NULLPTR	8	12	NULLPTR	3	2	13	1	15	11	6

0	1	2	3	4	5	6	7
D	A	T	A				
1	2	3	NULLPTR	5	4	3	NULLPTR

Reallocation of memory

If at some point, we decide it is desirable to reduce the memory allocated, it might be easier to just insert the entries into a newer, and smaller table

– Now, delete the old array and update the member variables

```
list_head = 0;
list_tail = 3;
list_size = 4;
list_capacity = 8;
stack_top = 4;
```

0	1	2	3	4	5	6	7
D	A	T	A				
1	2	3	NULLPTR	5	4	3	NULLPTR