

**Data Structures and Algorithms**  
In C++

**Nathan Warner**



**Northern Illinois  
University**

Computer Science  
Northern Illinois University  
February 16, 2023  
United States

## Contents

<b>1</b>	<b>Selection Sort</b>	<b>5</b>
1.1	Psuedocode . . . . .	5
1.2	Example . . . . .	5
1.3	Complexity . . . . .	6
<b>2</b>	<b>Insertion Sort</b>	<b>7</b>
2.1	Psuedocode . . . . .	7
2.2	Example . . . . .	7
2.3	Optimizing Insertion Sort . . . . .	8
2.3.1	Psuedocode . . . . .	8
2.3.2	Example . . . . .	8
2.4	Complexity . . . . .	9
<b>3</b>	<b>Bubble Sort</b>	<b>10</b>
3.1	Psuedocode . . . . .	10
3.2	Example . . . . .	10
3.3	Optimizing Bubble Sort . . . . .	11
3.3.1	Psuedocode . . . . .	11
3.3.2	Example . . . . .	12
3.4	Complexity . . . . .	12
<b>4</b>	<b>Two-Dimensional Array</b>	<b>13</b>
<b>5</b>	<b>Recursion</b>	<b>14</b>
<b>6</b>	<b>Complexity Analysis</b>	<b>15</b>
6.1	Time Complexity . . . . .	15
6.1.1	Common time complexities . . . . .	15

6.1.2	Constant time . . . . .	15
6.1.3	Big O, Big Omega, and Big Theta . . . . .	15
6.1.4	Best Case, Worst Case, and Expected (or Average) Case . . . . .	16
6.2	Space complexity . . . . .	17
6.2.1	Constant time . . . . .	17
6.2.2	Space complexity in recursive algorithms . . . . .	17
6.3	Drop the constants . . . . .	18
6.4	Drop the non-dominant terms . . . . .	18
6.5	Multi-Part Algorithms: Add vs. Multiply . . . . .	18
6.6	Amortized Time . . . . .	19
6.7	Log N Runtimes . . . . .	19
6.8	Recursive runtime . . . . .	20
<b>7</b>	<b>Shell sort</b>	<b>21</b>
7.1	Example . . . . .	21
<b>8</b>	<b>Quick Sort (Recursive)</b>	<b>22</b>
8.1	Base case . . . . .	22
8.2	Pivot selection . . . . .	22
8.3	Psuedocode . . . . .	22
8.3.1	What main calls . . . . .	22
8.3.2	Recursive function . . . . .	23
8.3.3	Partition function . . . . .	23
8.4	Examples . . . . .	24
8.5	Complexity . . . . .	25
<b>9</b>	<b>Merge Sort Algorithm</b>	<b>26</b>
9.1	Psuedocode . . . . .	27
9.2	Example . . . . .	28
<b>10</b>	<b>Binary Heap</b>	<b>29</b>
10.1	Example . . . . .	29
10.2	Insertion . . . . .	30
10.3	Binary heap imbalance . . . . .	30

10.4	Deletion . . . . .	30
10.5	Formulas for a binary heap . . . . .	30
<b>11</b>	<b>Heap Sort</b>	<b>31</b>
11.1	Psuedocode . . . . .	32
11.2	Example . . . . .	34
<b>12</b>	<b>Linear search</b>	<b>35</b>
12.1	The linear search . . . . .	35
<b>13</b>	<b>Binary search</b>	<b>36</b>
<b>14</b>	<b>Array Based Stack Implementation</b>	<b>38</b>
14.1	Data members . . . . .	38
14.2	Member Functions . . . . .	38
14.3	Reference: Vector copy constructor . . . . .	39
14.4	Reference: Vector copy assignment operator . . . . .	39
14.5	Auxiliary: Vector move constructor . . . . .	40
14.6	Array based stack example . . . . .	41
<b>15</b>	<b>Array based stack application: Infix to postfix conversion algorithm</b>	<b>44</b>
15.1	Infix Notation . . . . .	44
15.2	Postfix Notation . . . . .	44
15.3	The algorithm . . . . .	44
15.4	Example . . . . .	46
<b>16</b>	<b>Array based queue</b>	<b>49</b>
16.1	Data members . . . . .	49
16.2	Member Functions . . . . .	49
16.3	Double-Ended Queue . . . . .	50
16.4	Example . . . . .	51
<b>17</b>	<b>Singly-linked list (as a stack)</b>	<b>54</b>
17.1	Structure of a Node . . . . .	54
17.2	Advantages . . . . .	54

17.3	Disadvantages . . . . .	54
17.4	Sample node structure . . . . .	55
17.5	Class to represent a stack . . . . .	55
17.5.1	Data members . . . . .	55
17.5.2	Member Functions . . . . .	55
17.6	Visualization . . . . .	56
17.7	Example (as a stack) . . . . .	57
<b>18</b>	<b>Singly linked list (as a queue)</b>	<b>60</b>
18.1	Data Members . . . . .	60
18.2	Member Functions . . . . .	60
18.3	Visualization . . . . .	61
18.4	Example Code . . . . .	62
<b>19</b>	<b>Hash Table With Linear Probe</b>	<b>65</b>
19.1	Hash table header file . . . . .	66
19.2	Hash table cpp file . . . . .	68
<b>20</b>	<b>Reverse a singly linked list</b>	<b>71</b>
<b>21</b>		<b>72</b>

# Selection Sort

**Concept 1:** The selection sort algorithm sorts an array by repeatedly finding the minimum element (if sorting in ascending order) from the unsorted part of the array and putting it at the end of the sorted part of the array. The algorithm maintains two subarrays in a given array:

- A subarray of already sorted elements.
- A subarray of elements that remain to be sorted.

At the start of the algorithm, the first subarray is empty. In each pass through the outer loop of the selection sort, the minimum element from the unsorted subarray is selected and moved to the end of the sorted subarray.

## 1.1 Psuedocode

```
1  procedure selection_sort(array : list of sortable items, n :  
    ↪ length of list)  
2      i := 0  
3      while i < n - 1  
4          min_index ← i  
5          j := i + 1  
6          while j < n  
7              if array[j] < array[min_index]  
8                  min_index ← j  
9              end if  
10             j = j + 1  
11         end while  
12         swap array[i] and array[min_index]  
13         i = i + 1  
14     end while  
15 end procedure
```

## 1.2 Example

```
1  int main(int argc, const char* argv[]) {  
2      int arr[] = {2,4,1,3,5}; int n = 5;  
3  
4      for (int j=0; j <n-1; ++j) {  
5          int min = j;  
6          for (int k=j+1; k <n-1; ++k) {  
7              if (arr[k] < arr[min]) {  
8                  min = k;  
9              }  
10         }  
11         std::swap(arr[j], arr[min]);  
12     }  
13 }
```

### 1.3 Complexity

- **Time Complexity:**  $O(n^2)$
- **Space Complexity:**  $O(1)$

#### Note:-

The primary advantage of selection sort is that it never makes more than  $O(n)$  swaps, which can be useful if the array elements are large and copying them is a costly operation.

# Insertion Sort

**Concept 2:** The insertion sort algorithm sorts a list by repeatedly inserting an unsorted element into the correct position in a sorted sublist. The algorithm maintains two sublists in a given array:

- A sorted sublist. This sublist initially contains a single element (an array of one element is always sorted).
- A sublist of elements to be inserted one at a time into the sorted sublist.

## 2.1 Psuedocode

```
1  procedure insertion_sort(array : list of sortable items, n :  
    ↪ length of list)  
2      i ← 1  
3      while i < n  
4          j ← i  
5          while j > 0 and array[j - 1] > array[j]  
6              swap array[j - 1] and array[j]  
7              j ← j - 1  
8          end while  
9          i ← i + 1  
10     end while  
11 end procedure
```

## 2.2 Example

```
1  int main(int argc, const char* argv[]) {  
2      int arr[] = {2,4,1,3,5};  
3      int n = 5;  
4  
5      for (int j=1; j<n; ++j) {  
6          for (int k=j; k>0; --k) {  
7              if (arr[k-1] > arr[k]) {  
8                  std::swap(arr[k-1], arr[k]);  
9              }  
10         }  
11     }  
12 }
```



## 2.3 Optimizing Insertion Sort

Performing a full swap of the array elements in each inner for loop iteration is not necessary. Instead, we save the value that we want to insert into the sorted subarray in temporary storage. In place of performing a full swap, we simply copy elements to the right. The saved value can then be inserted into its proper position once that has been located.

This alternative approach can potentially save a considerable number of assignment statements. If  $N$  swaps are performed by the inner loop, the original version of insertion sort requires  $N \cdot 3$  assignment statements to perform those swaps. The improved version listed below only requires  $N + 2$  assignment statements to accomplish the same task.

### 2.3.1 Psuedocode

```
1  procedure insertion_sort(array : list of sortable items, n :  
    ↪ length of list)  
2      i ← 1  
3      while i < n  
4          temp ← array[i]  
5          j ← i  
6          while j > 0 and array[j - 1] > temp  
7              array[j] ← array[j - 1]  
8              j ← j - 1  
9          end while  
10         array[j] ← temp  
11         i ← i + 1  
12     end while  
13 end procedure
```

### 2.3.2 Example

```
1  int arr[] = {5,6,4,3,1};  
2  int n = 5;  
3  
4  for (int j=1; j<n; ++j) {  
5      int tmp = arr[j];  
6      int k=j;  
7      for (; k>0; --k) {  
8          if (arr[k-1] > tmp) {  
9              arr[k] = arr[k-1];  
10             } else {  
11                 break;  
12             }  
13         }  
14         arr[k] = tmp;  
15     }
```

## 2.4 Complexity

- **Time Complexity:**  $O(n^2)$
- **Space Complexity:**  $O(1)$

### Note:-

The primary advantage of insertion sort over selection sort is that selection sort must always scan all remaining unsorted elements to find the minimum element in the unsorted portion of the list, while insertion sort requires only a single comparison when the element to be inserted is greater than the last element of the sorted sublist. When this is frequently true (such as if the input list is already sorted or partially sorted), insertion sort is considerably more efficient than selection sort. The best case input is a list that is already correctly sorted. In this case, insertion sort has  $O(n)$  complexity.

## Bubble Sort

**Concept 3:** Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

### 3.1 Psuedocode

```
1  procedure bubble_sort(array : list of sortable items, n : length
   ↪   of list)
2      do
3          swapped ← false
4          i ← 1
5          while i < n
6              if array[i - 1] > array[i]
7                  swap array[i - 1] and array[i]
8                  swapped ← true
9              end if
10             i ← i + 1
11         end while
12     while swapped
13 end procedure
```

#### Note:-

If no items are swapped during a pass through the outer loop (i.e., the variable swapped remains false), then the array is already sorted and the algorithm can terminate.

### 3.2 Example

```
1  int arr[] = {5,6,4,3,1};
2  int n = 5;
3
4  bool swapped;
5  do {
6      swapped = 0;
7
8      for (int i=0; i<n; ++i) {
9          if (arr[i-1] > arr[i]) {
10             std::swap(arr[i-1], arr[i]);
11             swapped = 1;
12         }
13     }
14
15 } while (swapped);
16
```

### 3.3 Optimizing Bubble Sort

The bubble sort algorithm can be optimized by observing that the  $n$ -th pass finds the  $n$ -th largest element and puts it into its final place. Therefore the inner loop can avoid looking at the last  $n - 1$  items when running for the  $n$ -th time:

#### 3.3.1 Psuedocode

```
1  procedure bubble_sort(array : list of sortable items, n : length
   ↪   of list)
2      do
3          swapped ← false
4          i ← 1
5          while i < n
6              if array[i - 1] > array[i]
7                  swap array[i - 1] and array[i]
8                  swapped ← true
9              end if
10             i ← i + 1
11         end while
12         n ← n - 1
13     while swapped
14 end procedure
```

It is common for multiple elements to be placed in their final positions on a single pass. In particular, after every pass through the outer loop, all elements after the position of the last swap are sorted and do not need to be checked again. Taking this into account makes it possible to skip over many elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the swapped variable:

```
1      do
2          last ← 0
3          i ← 1
4          while i < n
5              if array[i - 1] > array[i]
6                  swap array[i - 1] and array[i]
7                  last ← i
8              end if
9              i ← i + 1
10         end while
11         n ← last
12     while n > 1
```

### 3.3.2 Example

```
1  int last;
2  do {
3      last = 0;
4      int j=1;
5
6      for (; j<n; ++j) {
7          if (arr[j-1] > arr[j]) {
8              std::swap(arr[j-1], arr[j]);
9              last = j;
10         }
11     }
12     n = last;
13
14 } while (n > 0);
```

### 3.4 Complexity

- **Time Complexity:**  $O(n^2)$
- **Space Complexity:**  $O(1)$

#### Note:-

Other  $O(n^2)$  sorting algorithms, such as insertion sort, generally run faster than bubble sort (even with optimizations) and are no more complex. Therefore, bubble sort is not a practical sorting algorithm. The only significant advantage that bubble sort has over most other sorting algorithms (but not insertion sort), is that the ability to detect that the list is sorted is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only  $O(n)$ .

## Two-Dimensional Array

## Recursion

# Complexity Analysis

## 6.1 Time Complexity

**Concept 4:** Time complexity in algorithms is a way to describe the efficiency of an algorithm in terms of the time it takes to run as a function of the length of the input. It gives us an idea of the growth rate of the runtime of an algorithm as the size of input data increases. Big O notation is a mathematical notation used to express this time complexity, focusing on the worst-case scenario or the upper limit of the algorithm's running time.

Big O notation describes the upper bound of the time complexity, ignoring constants and lower order terms which are less significant for large input sizes. Here are some common Big O notations and their meanings:

### 6.1.1 Common time complexities

The following is a list of the common time complexities, in order from best to worst

Notation	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$	Polynomial
$O(2^n)$	Exponential
$O(n!)$	Factorial

You can also have multiple variables in your runtime. For example, the time required to paint a fence that is  $w$  meters wide and  $h$  meters high could be described as  $O(wh)$ . If you needed  $p$  layers of paint, then you could say that the time is  $O(whp)$

### 6.1.2 Constant time

An  $O(1)$  time complexity, also known as constant time complexity, describes an algorithm where the time to complete does not depend on the size of the input data set.

### 6.1.3 Big O, Big Omega, and Big Theta



Academics use big  $O$ , big  $\Theta$ , and big  $\Omega$  to describe runtimes.

- **Big  $O$**  notation (denoted as  $O$ ) is widely used in academia to describe an upper bound on the time complexity of an algorithm. For instance, an algorithm that prints all the values in an array could be described as  $O(N)$ . However, it could also be described as  $O(N^2)$ ,  $O(N^3)$ , or  $O(2^N)$ , among other possible Big  $O$  notations. The algorithm's execution time is at least as fast as each of these, making them upper bounds on the runtime. This relationship is akin to a less-than-or-equal-to relationship. For example, if Bob is  $X$  years old (assuming no one lives past age 130), then it would be correct to say that  $X \leq 130$ . Similarly, it would also be correct, albeit less useful, to say that  $X \leq 1,000$  or  $X \leq 1,000,000$ . While these statements are technically true, they are not particularly informative. Likewise, a simple algorithm to print the values in an array is  $O(N)$ , but it is also correct to describe it as  $O(N^3)$  or any runtime larger than  $O(N)$ . This illustrates that while multiple Big  $O$  notations can technically describe the time complexity of an algorithm, the most informative description is the one that provides the tightest upper bound.
- **Big  $\Omega$** : In academia,  $\Omega$  is the equivalent concept but for the lower bound. Printing the values in an array is  $\Omega(N)$  as well as  $\Omega(\log N)$  and  $\Omega(1)$ . After all, you know it won't be faster than those runtimes. The  $\Omega$  notation is used to describe the best-case scenario or the minimum amount of time an algorithm will take to complete. It ensures that the algorithm's execution time will not be less than the specified complexity, providing a guarantee on the lower limit of the algorithm's performance.
- **Big  $\Theta$** : In academia,  $\Theta$  notation signifies that an algorithm's time complexity has both an upper and a lower bound. That is, an algorithm is  $\Theta(N)$  if it is both  $O(N)$  and  $\Omega(N)$ .  $\Theta$  notation provides a tight bound on runtime, indicating that the algorithm's execution time grows at a rate directly proportional to the size of the input, neither faster nor slower. This precise characterization makes  $\Theta$  especially useful for describing algorithms where the upper and lower bounds converge to the same complexity, offering a complete understanding of the algorithm's efficiency.

**Note:-**

In the industry, when people refer to big  $O$  notation, they are likely talking about big  $\Theta$

#### 6.1.4 Best Case, Worst Case, and Expected (or Average) Case

We can actually describe our runtime for an algorithm in three different ways. Let's look at this from the perspective of quick sort.

- **Best Case**: If all elements of the array are equal, then quick sort will, on average, just traverse through the array once. This is  $\mathcal{O}(N)$ . (This actually depends slightly on the implementation of quick sort. There are implementations that will run very quickly on a sorted array.)
- **Worst Case**: What if we get really unlucky and the pivot is repeatedly the biggest element in the array? (Actually, this can easily happen. If the pivot is chosen to be the first element in the subarray and the array is sorted in reverse order, we'll have just this situation.) In this case, our recursion doesn't divide the array in half and recursively sort each half, it just shrinks the subarray by one element. We end up with something similar to selection sort and the runtime degenerates to  $\mathcal{O}(N^2)$ .

- **Expected Case:** Usually, though, these wonderful or terrible situations won't happen. Sure, sometimes the pivot will be very low or very high, but it won't happen over and over again. We can expect a runtime of  $\mathcal{O}(N \log N)$ .

We rarely discuss best case time complexity because it's not a very useful concept. After all, we could take essentially any algorithm, special case some input, and then get a  $\mathcal{O}(1)$  runtime in the best case. For many – probably most – algorithms, the worst case and the expected case are the same. Sometimes they're different though and we need to describe both of the runtimes

## 6.2 Space complexity

Time is not the only thing that matters in an algorithm. We might also care about the amount of memory – or space – required by the algorithm. Space complexity is a parallel concept to time complexity. If we need to create an array of size  $n$ , this will require  $\mathcal{O}(n)$  space. If we need a two-dimensional array of size  $n \times n$ , this will require  $\mathcal{O}(n^2)$  space.

### 6.2.1 Constant time

An algorithm has  $\mathcal{O}(1)$  space complexity when the amount of memory it requires does not grow with the size of the input data set. This means the algorithm needs a constant amount of memory space, regardless of how large the input is.

### 6.2.2 Space complexity in recursive algorithms

Stack space in recursive calls counts too. For example, code like this would take  $\mathcal{O}(n)$  time and  $\mathcal{O}(n)$  space

```

1  // Example 1
2  int sum(int n) {
3      if (n <= 0)
4          return 0;
5      else
6          return n + sum(n - 1);
7  }
```

Each of these calls results in a stack frame with a copy of the variable  $n$  being pushed onto the program call stack and takes up actual memory

### 6.3 Drop the constants

It is entirely possible for  $O(n)$  code to run faster than  $O(1)$  code for specific inputs. Big O just describes the rate of increase, not the specific time required

For this reason, we drop the constants in runtimes. An algorithm that one might have described as  $O(2N)$  is actually  $O(N)$ . If you're going to try to count the number of instructions, then you'd have to go to the assembly level and take into account that multiplication requires more instructions than addition, how the compiler would optimize something, and all sorts of other details.

That would be horrendously complicated, so don't even start going down that road. Big O allows us to express how the runtime scales. We just need to accept that it doesn't mean that  $O(N)$  is always better than  $O(N^2)$ .

### 6.4 Drop the non-dominant terms

What do you do about an expression such as  $O(N^2 + N)$ ? That second  $N$  isn't exactly a constant. But it's not especially important. We already said that we drop constants.  $O(N^2 + N^2)$  is  $O(2N^2)$ , and therefore it would be  $O(N^2)$ . If we don't care about the latter  $N^2$  term, why would we care about  $N$ ? We don't.

We might still have a sum in a runtime. For example, the expression  $O(B^2 + A)$  cannot be reduced (without some special knowledge of  $A$  and  $B$ )

### 6.5 Multi-Part Algorithms: Add vs. Multiply

Suppose you have an algorithm that has two steps. When do you multiply the runtimes and when do you add them?

```
1  // Program 1
2
3  for (int i=0; i<A; ++i) {
4      cout << arrayA[i];
5  }
6
7  for (int i=0; i<B; ++i) {
8      cout << arrayB[i];
9  }
10
11 // Program 2
12 for (int i=0; i<A; ++i) {
13     for (int j=0; j<B; ++j) {
14         cout << arrayA[i] << ", " << arrayB[j];
15     }
16 }
```

In the first example, we do  $A$  chunks of work then  $B$  chunks of work. Therefore, the total amount of work is  $O(A+B)$ . In the second example, we do  $B$  chunks of work for each element in  $A$ . Therefore, the total amount of work is  $O(A * B)$

## 6.6 Amortized Time

A C++ vector object allows you to have the benefits of an array while offering flexibility in size. You won't run out of space in the vector since its capacity will grow as you insert elements. A vector is implemented with a dynamic array. When the number of stored in the array hits the array's capacity, the vector class will create a new array with double the capacity and copy all of the elements over to the new array. The old array is then deleted.

How do you describe the runtime of insertion? This is a tricky question. The array could be full. If the array contains  $N$  elements, then inserting a new element will take  $O(N)$  time. You will have to create a new array of capacity  $2N$  and then copy  $N$  elements over. This insertion will take  $O(N)$  time. However, we also know that this doesn't happen very often. The vast majority of the time, insertion will be in  $O(1)$  time.

We need a concept that takes both possibilities into account. This is what amortized time does. It allows us to describe that, yes, this worst case happens every once in a while. But once it happens, it won't happen again for so long that the cost is "amortized."

In this case, what is the amortized time? As we insert elements, we double the capacity when the size of the array is a power of 2. So after  $X$  elements, we double the capacity at array sizes 1, 2, 4, 8, 16, ...,  $X$ . That doubling takes, respectively, 1, 2, 4, 8, 16, 32, 64, ...,  $X$  copies.

What is the sum of  $1 + 2 + 4 + 8 + 16 + \dots + X$ ? If you read this sum left to right, it starts with 1 and doubles until it gets to  $X$ . If you read right to left, it starts with  $X$  and halves until it gets to 1. What then is the sum of  $X + X/2 + X/4 + X/8 + \dots + 1$ ? This is roughly  $2X$ . (It's  $2X - 1$  to be exact, but this is big O notation, so we can drop the constant.).

Therefore,  $X$  insertions take  $O(2X)$  time. The amortized time for each insertion is therefore  $O(1)$ .

## 6.7 Log N Runtimes

We commonly see  $O(\log N)$  in runtimes. Where does this come from?

Let's look at binary search as an example. In binary search, we are looking for an item `search_key` in an  $N$  element sorted array. We first compare `search_key` to the midpoint of the array. If `search_key == array[mid]`, then we return. If `search_key < array[mid]`, then we search on the left side of the array. If `search_key > array[mid]`, then we search on the right side of the array.

We start off with with an  $N$ -element array to search. Then, after a single step, we're down to  $N/2$  elements. One more step, and we're down to  $N/4$  elements. We stop when we either find the value

or we're down to just one element. The total runtime is then a matter of how many steps (dividing  $N$  by 2 each time) we can take until  $N$  becomes 1.

We could look at this in reverse (going from 1 to 16 instead of 16 to 1). How many times can we multiply N by 2 until we get N?

What is k in the expression  $2^k = n$ ? This is exactly what log expresses.

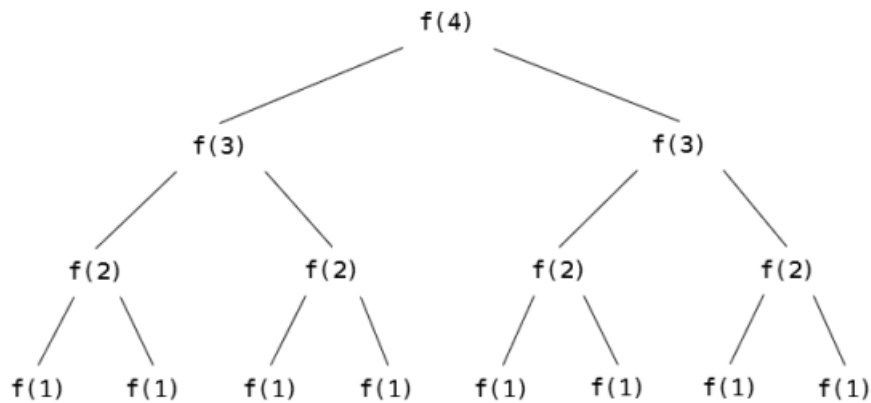
$$\begin{aligned} 2^k &= n \\ &= \log_2 n = k. \end{aligned}$$

## 6.8 Recursive runtime

Here's a tricky one. What's the runtime of this code?

```
1  int f(int n) {  
2      if (n <= 1) {  
3          return 1;  
4      } else {  
5          return f(n-1) + f(n-1);  
6      }  
7  }  
8
```

let's derive the runtime by walking through the code. Suppose we call  $f(4)$ . This calls  $f(3)$  twice. Each of those calls to  $f(3)$  calls  $f(2)$ , until we get down to  $f(1)$ .



How many calls are in this tree?

The tree will have depth N. Each node (i.e., function call) has two children. Therefore, each level will have twice as many calls as the one above it.

Therefore, there will be  $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \dots + 2^N$  (which is  $2^{N+1}-1$ ) nodes.

Try to remember this pattern. When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like  $O(\text{branches}^{\text{depth}})$  where branches is the number of times each recursive call branches. In this case, this gives us  $O(2^N)$ .

The space complexity of this algorithm will be  $O(N)$ . Although we have  $O(2^N)$  function calls in the tree total, only  $O(N)$  exist on the call stack at any given time. Therefore, we would only need to have  $O(N)$  memory available.

## Shell sort

**Concept 5:** Shell sort is an advanced variant of insertion sort. It first sorts elements that are far apart from each other and successively reduces the interval (gap) between the elements to be compared. The idea is to arrange the list of elements into a sequence of incrementally more sorted arrays, which are then finally sorted with a simple insertion sort.

The key concept in Shell sort is the use of an interval to compare elements. Initially, elements far apart are compared and swapped if necessary. As the algorithm progresses, the interval decreases, making the array more and more sorted, until the interval is 1. At an interval of 1, the algorithm is essentially performing a standard insertion sort, but by this time, the array is partially sorted, making the insertion sort more efficient.

### 7.1 Example

```
1  int arr[] = {6,3,2,1,8};
2  int n = 5;
3
4  for (int iv=n/2; iv>0; iv/=2) {
5
6      for (int i=iv; i<n; ++i) {
7
8          int j;
9          int tmp = arr[i];
10         for (j = i; j>=iv && arr[j-iv] > tmp; j-=iv) {
11             arr[j] = arr[j-iv];
12         }
13         arr[j] = tmp;
14     }
15 }
```

## Quick Sort (Recursive)

**Concept 6:** The quicksort algorithm is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this reordering, the pivot is in its final, sorted position. This reordering is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

### 8.1 Base case

The base case of the recursion is an array of size zero or one, which is in order by definition and requires no further sorting.

### 8.2 Pivot selection

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

### 8.3 Psuedocode

#### 8.3.1 What main calls

```
1  procedure quicksort(array : list of sortable items, n : length
   ↪   of list)
2      quicksort(array, 0, n - 1)
3  end procedure
```

### 8.3.2 Recursive function

```
1  procedure quicksort(array : list of sortable items, start :  
   ↪ first element of list,  
2  end : last element of list)  
3      if start < end  
4          pivot_point ← partition(array, start, end)  
5          quick_sort(array, start, pivot_point - 1)  
6          quick_sort(array, pivot_point + 1, end)  
7      end if  
8  end procedure
```

### 8.3.3 Partition function

```
1  procedure partition(array : list of sortable items, start :  
   ↪ first element of list,  
2  end : last element of list)  
3      mid ← (start + end) / 2  
4      swap array[start] and array[mid]  
5  
6      pivot_index ← start  
7      pivot_value ← array[start]  
8  
9      scan ← start + 1  
10     while scan <= end  
11         if array[scan] < pivot_value  
12             pivot_index ← pivot_index + 1  
13             swap array[pivot_index] and array[scan]  
14         end if  
15         scan ← scan + 1  
16     end while  
17  
18     swap array[start] and array[pivot_index]  
19  
20     return pivot_index  
21 end procedure
```



## 8.4 Examples

```
1  int partition(int arr[], int start, int end) {
2      int pivot_index, pivot_value, mid, scan;
3
4      mid = (start + end) / 2;
5      std::swap(arr[start], arr[mid]);
6
7      pivot_index = start;
8      pivot_value = arr[start];
9
10     scan = start + 1;
11
12     while (scan <= end) {
13         if (arr[scan] < pivot_value) {
14             ++pivot_index;
15             std::swap(arr[pivot_index], arr[scan]);
16         }
17         ++scan;
18     }
19     std::swap(arr[start], arr[pivot_index]);
20
21     return pivot_index;
22 }
23
24 void quicksort(int arr[], int start, int end) {
25     int pivot_point;
26     if (start < end) {
27         pivot_point = partition(arr, start, end);
28         quicksort(arr, start, pivot_point - 1);
29         quicksort(arr, pivot_point + 1, end);
30     }
31 }
32
33 void quicksort(int arr[], int n) {
34     quicksort(arr, 0, n-1);
35 }
36
37 int main(int argc, const char* argv[]) {
38
39     int arr[] = {3,6,1,9,12,7,36,24,18,4};
40     int n = 10;
41
42     quicksort(arr,n);
43
44     return EXIT_SUCCESS;
45 }
```

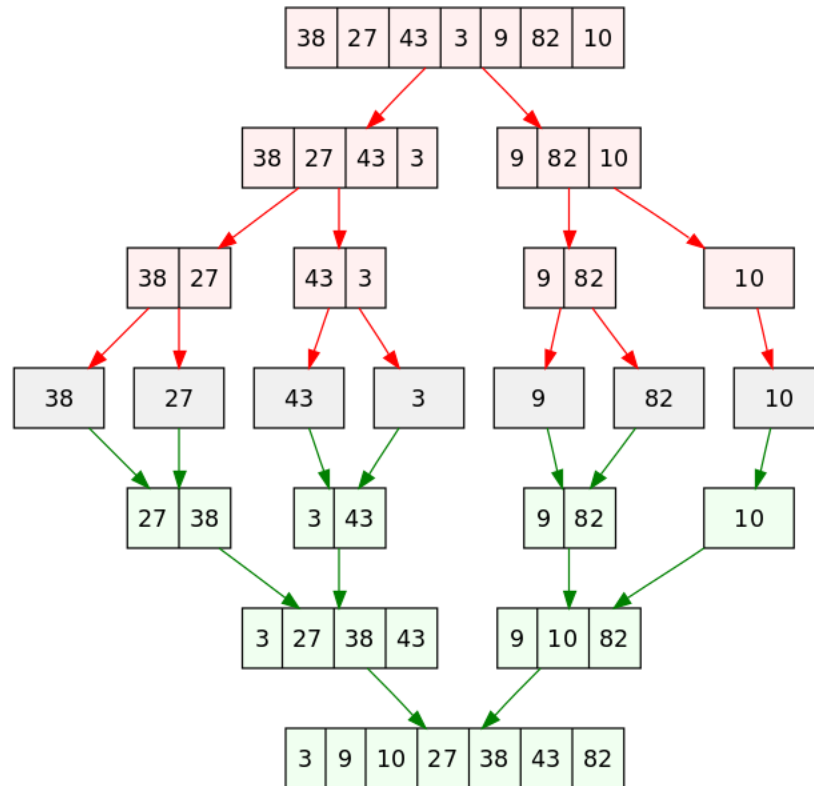
## 8.5 Complexity

- **Time Complexity:**  $O(n \log n)$
- **Space Complexity:**  $O(\log n)$

# Merge Sort Algorithm

**Concept 7:** Merge sort works as follows:

- Divide the unsorted list into  $n$  sublists, each containing one element. A list of one element is sorted by definition.
- Repeatedly merge sorted sublists (called "runs") to produce longer runs until there is only one run remaining. This is the sorted list



This algorithm makes use of a variable length temporary array, which is most easily represented in C++ using the **vector** class from the standard library. When implementing the algorithm, include the following code at the top of the source file:

## 9.1 Psuedocode

```
1  procedure merge_sort(array : list of sortable items, start :  
   ↪ first element of list,  
2  end : last element of list)  
3      if start < end  
4          mid ← (start + end) / 2  
5  
6          merge_sort(array, start, mid)  
7          merge_sort(array, mid + 1, end)  
8  
9          merge(array, start, mid, end)  
10     end if  
11 end procedure  
12  
13 procedure merge(array : list of items to merge, start : first  
   ↪ element of first sublist, mid : last element of first  
   ↪ sublist,  
14 end : last element of second sublist)  
15     vector<int> temp(end - start + 1);  
16  
17     i ← start  
18     j ← mid + 1  
19     k ← 0  
20  
21     while i <= mid and j <= end  
22         if array[i] < array[j]  
23             temp[k] ← array[i]  
24             i ← i + 1  
25         else  
26             temp[k] ← array[j]  
27             j ← j + 1  
28         end if  
29         k ← k + 1  
30     end while  
31  
32     while i <= mid  
33         temp[k] ← array[i]  
34         i ← i + 1  
35         k ← k + 1  
36     end while  
37  
38     while j <= end  
39         temp[k] ← array[j]  
40         j ← j + 1  
41         k ← k + 1  
42     end while  
43  
44     Copy the elements of the vector temp back into array  
45 end procedure
```

## 9.2 Example

```
1  void merge(int arr[], int start, int mid, int end) {
2
3      vector<int> temp(end - start + 1);
4
5      int i,j,k;
6
7      i = start;
8      j = mid + 1;
9      k = 0;
10
11     while (i <= mid && j<= end) {
12         if (arr[i] < arr[j]) {
13             temp[k] = arr[i];
14             ++i;
15         } else {
16             temp[k] = arr[j];
17             ++j;
18         }
19         ++k;
20     }
21
22     while (i <= mid) {
23         temp[k] = arr[i];
24         ++i;
25         ++k;
26     }
27
28     while (j <= end) {
29         temp[k] = arr[j];
30         ++j;
31         ++k;
32     }
33
34     for ( i=start, j=0; i<=end; ++i, ++j) {
35         arr[i] = temp[j];
36     }
37 }
38
39 void merge_sort(int arr[], int start, int end) {
40
41     int mid;
42     if (start < end) {
43         mid = (start + end) / 2;
44
45         merge_sort(arr, start, mid);
46         merge_sort(arr, mid+1, end);
47
48         merge(arr, start, mid, end);
49     }
50 }
```

# Binary Heap

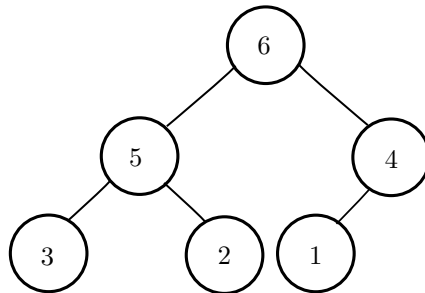
**Concept 8:** A **binary heap** is a data structure that takes the form of binary tree with two additional constraints

- The binary tree must be complete or almost complete; that is, all levels of the tree, except possibly the last (deepest) one, are fully filled. If the last level of the tree is not complete, the nodes of that level are filled from left to right.
- The key stored in each node is either greater than or equal to or less than or equal to the keys in the node's children.

A binary heap where the parent key is greater than or equal to the child keys is called a **max-heap**; a heap where the parent key is less than or equal to the child keys is called a **min-heap**.

Because a binary heap is always a complete or almost complete binary tree, the tree nodes can be efficiently stored in an array with no wasted space. The top-level node (or root) of the tree is stored in the first element of the array. Then, for each node in the tree that is stored at subscript  $k$ , the node's left child can be stored at subscript  $2k + 1$  and the right child can be stored at subscript  $2k + 2$ .

## 10.1 Example



## 10.2 Insertion

When placing nodes, we first place the root. From there, we add from left to right.

When adding new nodes to the head, we place them at the bottom. However, this may lead to a violation in the trees order. In this case, we compare the newly inserted node to its parent, swapping them if necessary, we do this until the node is in the correct position.

## 10.3 Binary heap imbalance

It is not always possible to have a balanced heap. That is, for each level of the tree, each side has the same number of nodes. To account for this, we allow the left sub tree to hold one more than the right sub tree.

## 10.4 Deletion

Because the binary heap is designed to give us access to the minimum element (min-head), or maximum element (max-head), we can only delete the root node.

Once we delete the root node, you may notice that the shape is disrupted. That is, we no longer have a root node. To solve this, we first get the tree back to its correct shape, and then focus on the invariance.

**Note:** Invariance in a binary heap refers to the property that ensures the status of the min or max heap.

## 10.5 Formulas for a binary heap

- For any node at index  $i$  its left and right children are at index positions

$$\begin{aligned} 2i + 1 & \text{ (left child)} \\ 2i + 2 & \text{ (right child).} \end{aligned}$$

- For any node at index  $i$  its parent is at index position

$$\frac{i - 1}{2}.$$

- The index position of the last non-leaf node is given by

$$\frac{n - 2}{2}.$$

Where  $n$  is the number of elements in the binary heap

# Heap Sort

**Concept 9: Heapsort** is a comparison-based sorting algorithm that uses an implicit binary heap. Heapsort can be thought of as an improved selection sort: like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to find the largest element more quickly in each step.

The heapsort algorithm can be divided into two parts:

- In the first part, the elements of an unsorted array are rearranged to create a binary heap (a max-heap if the array is to be sorted in ascending order).
- In the second part, a sorted array is created by repeatedly swapping the largest element from the heap (the root of the heap) with the last element of the heap and then decrementing the heap size (which effectively removes that element from the heap). The heap is updated after each removal to recreate the max-heap property. Once all elements have been removed from the heap, the result is a sorted array.



## 11.1 Psuedocode

```
1  procedure heap_sort(array : list of sortable items, n : length
   ↪ of list)
2      // end : array subscript
3      // Build the heap in array so that largest value is at the
4      // root.
5      heapify(array, n);
6
7      end = n - 1
8
9      while end > 0
10         // array[0] is the root and largest value. The swap
11         // moves it in front of the sorted elements.
12         swap array[end] and array[0]
13
14         // The heap size is reduced by 1.
15         end = end - 1
16
17         // The swap ruined the heap property, so restore it.
18         sift_down(array, 0, end);
19     end while
20 end procedure
21
22 procedure heapify(array : list of sortable items, n : length of
   ↪ list)
23     // start : array subscript
24     start = (n - 2) / 2 // Find parent of last element of array
25     while start >= 0
26         // Sift down the value at subscript 'start' to the
   ↪ proper place
27         // such that all values below the start subscript are in
   ↪ max
28         // heap order
29
30         sift_down(array, start, n - 1)
31
32         // Go to next parent
33         start = start - 1
34     end while
35
36     // All elements are now in max heap order
37 end procedure
```

```

1  procedure sift_down(array : list of sortable items, start :
   ↪   starting
2      subscript of heap, end : ending subscript of heap)
3      // root : array subscript
4      // largest : array subscript
5      // child : array subscript
6
7      // Repair the heap whose root element is at subscript
   ↪   'start',
8      // assuming the heaps rooted at its children are valid
9
10     root = start
11
12     // While the root has at least one child
13     while (2 * root + 1) <= end
14         child = 2 * root + 1 // Left child of root
15         largest = root // Assume root is largest
16
17         // If left child is larger than root, left child is
   ↪   largest
18         if array[largest] < array[child]
19             largest = child
20         end if
21
22         // If there is a right child and it is greater than
   ↪   largest,
23         // right child is largest
24         if (child + 1) <= end and array[largest] < array[child+1]
25             largest = child + 1
26         end if
27
28         // If root is largest, no need to continue
29         if largest == root
30             return
31         else
32             swap array[root] and array[largest]
33             root = largest
34         end if
35
36     end while
37 end procedure

```

## 11.2 Example

```
1  #include <utility>
2  void heapify(int [], int n);
3  void sift_down(int [], int, int);
4
5  void heap_sort(int array[], int n) {
6      int end;
7
8      heapify(array, n);
9
10     end = n - 1;
11
12     while(end > 0) {
13         std::swap(array[end], array[0]);
14
15         end--;
16
17         sift_down(array, 0, end);
18     }
19 }
20 void heapify(int array[], int n) {
21     int start = (n - 2)/2;
22
23     while(start >= 0) {
24         sift_down(array, start, n-1);
25         start--;
26     }
27 }
28 void sift_down(int array[], int start, int end) {
29     int root = start;
30     int child;
31     int largest;
32
33     while(2 * root + 1 <= end) {
34         child = 2 * root + 1;
35         largest = root;
36
37         if(array[largest] < array[child]) {
38             largest = child;
39         }
40         if(child + 1 <= end && array[largest] <
↪ array[child + 1]) {
41             largest = child + 1;
42         }
43         if(largest == root) { return; } else {
44             std::swap(array[root], array[largest]);
45             root = largest;
46         }
47     }
48 }
```

# Linear search

## 12.1 The linear search

The linear search is very simple, it uses a loop to sequentially step through an array, starting with the first element.

Example:

```
1  int main(int argc, const char *argv[]) {
2
3      const int SIZE = 5;
4      int arr[SIZE] = {88,67,5,23,19};
5
6      int target = 5;
7
8      for (int i{0}; i <= SIZE + 1; ++i) {
9          if (i == SIZE + 1) {
10             cout << "Target not in array" << endl;
11         }
12         if ( arr[i] == target ) {
13             cout << "Target [" << target << "] found at index
↪ position " << i << endl;
14             break;
15         }
16     }
17     return EXIT_SUCCESS;
18 }
```

```
1  int linearsearch(int arr[], int size, int target) {
2      int index{0}, position{-1};
3      bool found = false;
4
5      while (index < size && !found) {
6          if (arr[index] == target) {
7              position = index;
8              found = true;
9          }
10         ++index;
11     }
12     return position;
13 }
```

One drawback to the linear search is its potential inefficiency, its quite obvious to notice

that for large arrays, the linear search will take a long time, if an array has 20,000 elements, and the target is at the end, then the search will have to compare 20,000 elements.

## Binary search

The binary search algorithm is a clever approach to searching arrays. Instead of testing the array's first element, the algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the array), will be found somewhere in the first half of the array. If it is less, then the desired value, it will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

### Note:-

The binary search algorithm requires the array to be sorted.

Example:

```
1  int binarysearch(int arr[], int size, int target) {
2      int first{0},
3          middle,
4          last = size - 1,
5          position{-1};
6
7      bool found = false;
8
9      while (!found && first <= last) {
10         middle = (first + last) / 2;
11         if (arr[middle] == target) {
12             found = true;
13             position = middle;
14         } else if (target > arr[middle]) {
15             first = middle + 1;
16         } else {
17             last = middle - 1;
18         }
19     }
20     return position;
21 }
```

Powers of twos are used to calculate the max number of comparisons the binary search will make on an array. Simply find the smallest power of 2 that is greater than or equal to the number of elements in the array. For example:

$$\begin{aligned}n &= 50,000 \\2^{15} &= 32,768 \\2^{16} &= 65,536.\end{aligned}$$

Thus, there are a maximum of 16 comparisons for an array of size 50,000

## Array Based Stack Implementation

**Concept 10:** An array-based stack in C++ is a linear data structure that follows the Last In, First Out (LIFO) principle. This means the last element added to the stack will be the first one to be removed. Stacks can be implemented using various underlying data structures, but using an array is one of the most straightforward methods. In this implementation, the array holds the stack elements, and an integer variable (often named `top`) tracks the index of the last element inserted into the stack.

### 14.1 Data members

- **stk\_array** - Stack array pointer. A pointer to the data type of the items stored in the stack; points to the first element of a dynamically-allocated array.
- **stk\_capacity** - Stack capacity. The number of elements in the stack array.
- **stk\_size** - Stack size. The number of items currently stored in the stack. The top item in the stack is always located at subscript `stk_size - 1`. Member Functions

### 14.2 Member Functions

- **Default constructor:** Sets stack to initial empty state. The stack capacity and stack size should be set to 0. The stack array pointer should be set to `nullptr`.
- **size()** Returns the stack size.
- **capacity()** Returns the stack capacity.
- **empty()** Returns true if the stack size is 0; otherwise, false.
- **clear()** Sets the stack size back to 0. Does not deallocate any dynamic storage.
- **top()** Returns the top item of the stack (`stk_array[stk_size - 1]`).
- **push()** Inserts a new item at the top of the stack.
- **pop()** Removes the top item from the stack.
- **Copy constructor** Similar to the copy constructor for the example Vector class in the notes on dynamic storage allocation.
- **Copy assignment operator** Similar to the copy assignment operator for the example Vector class in the notes on dynamic storage allocation.
- **Destructor** Deletes the stack array.
- **reserve()** Reserves additional storage for the stack array.

### 14.3 Reference: Vector copy constructor

```
1  Vector::Vector(const Vector& other)
2  {
3      // Step 1
4      vCapacity = other.vCapacity;
5      vSize = other.vSize;
6
7      // Step 2
8      if (vCapacity > 0)
9          vArray = new int[vCapacity];
10     else
11         vArray = nullptr;
12
13     // Step 3
14     for (size_t i = 0; i < vSize; ++i)
15         vArray[i] = other.vArray[i];
16 }
```

### 14.4 Reference: Vector copy assignment operator

```
1  Vector& Vector::operator=(const Vector& other)
2  {
3      // Step 1
4      if (this != &other)
5      {
6          // Step 2
7          delete[] vArray;
8
9          // Step 3
10         vCapacity = other.vCapacity;
11         vSize = other.vSize;
12
13         // Step 4
14         if (vCapacity > 0)
15             vArray = new int[vCapacity];
16         else
17             vArray = nullptr;
18
19
20         // Step 5
21         for (size_t i = 0; i < vSize; ++i)
22             vArray[i] = other.vArray[i];
23     }
24
25     // Step 6
26     return *this;
27 }
```



## 14.5 Auxiliary: Vector move constructor

```
1  Vector::Vector(Vector&& other)    // rvalue reference to a Vector
2  {
3      // Step 1 - "pilfer" other object's resources
4      vCapacity = other.vCapacity;
5      vSize = other.vSize;
6      vArray = other.vArray;
7
8      // Step 2 - set other object to default state
9      other.vCapacity = 0;
10     other.vSize = 0;
11     other.vArray = nullptr;
12 }
```

## 14.6 Array based stack example

```
1  class mystack {
2      // Pointer to dynamically allocated array for stack elements
3      char* m_stack = nullptr;
4      // Current capacity of the stack
5      size_t m_capacity = 0;
6      // Current number of elements in the stack
7      size_t m_size = 0;
8
9      public:
10     // Copy constructor
11     mystack(const mystack& x) {
12         // Copy capacity and size from source object
13         this->m_capacity = x.m_capacity;
14         this->m_size = x.m_size;
15
16         // Allocate memory if capacity is greater than 0
17         if (this->m_capacity > 0) {
18             this->m_stack = new char[this->m_capacity];
19         } else {
20             this->m_stack = nullptr;
21         }
22
23         // Copy the stack elements from source to this object
24         memcpy(this->m_stack, x.m_stack, x.m_size);
25     }
26
27     // Copy assignment operator
28     mystack& operator=(const mystack& x) {
29         // Allocate new memory space for the copy
30         this->m_capacity = x.m_capacity;
31         this->m_size = x.m_size;
32
33         if (this->m_capacity > 0) {
34             this->m_stack = new char[this->m_capacity];
35         } else {
36             this->m_stack = nullptr;
37         }
38
39         // Copy the elements
40         memcpy(this->m_stack, x.m_stack, x.m_size);
41
42         return *this; // Return a reference to the current object
43     }
44
45     // Returns the current capacity of the stack
46     size_t capacity() const {
47         return this->m_capacity;
48     }
49 }
```

```

1  // Returns the current size of the stack
2  size_t size() const {
3      return this->m_size;
4  }
5
6  // Checks if the stack is empty
7  bool empty() const {
8      return this->m_size == 0;
9  }
10
11 // Clears the stack (does not deallocate memory)
12 void clear(){
13     this->m_size = 0;
14 }
15
16 // Ensures the stack has at least the specified capacity
17 void reserve(size_t n){
18     // Only proceed if the new capacity is greater than the
    ↪ current capacity
19     if (n <= this->m_capacity) { return; }
20
21     // Update the capacity
22     this->m_capacity = n;
23     // Allocate new memory
24     char* tmp = new char[this->m_capacity];
25     // Copy existing elements to the new memory
26     memcpy(tmp, this->m_stack, this->m_size);
27
28     // Delete old stack and update pointer
29     delete[] this->m_stack;
30     this->m_stack = tmp;
31 }
32
33 // Returns a reference to the top element of the stack
34 const char& top() const{
35     return this->m_stack[(this->m_size)-1];
36 }
37
38 // Adds a new element to the top of the stack
39 void push(char value){
40     // If the stack is full, increase its capacity
41     if (this->m_size == this->m_capacity) {
42         this->reserve((this->m_capacity == 0 ? 1 :
    ↪ this->m_capacity * 2));
43     }
44
45     // Add the new element and increment the size
46     this->m_stack[(this->m_size)++] = value;
47 }
48

```

```

1      // Removes the top element from the stack
2      void pop(){
3          if (this->m_size > 0) {
4              --(this->m_size);
5          }
6      }
7
8      // Destructor: deallocates the dynamically allocated stack
9      ~mystack() {
10         delete[] this->m_stack;
11     }
12
13     // Friend function to output the contents of the stack to a
↵    stream
14     friend std::ostream& operator<<(std::ostream& os, const
↵    mystack& obj);
15 };
16
17 // Outputs the contents of the stack to a stream
18 std::ostream& operator<<(std::ostream& os, const mystack& obj) {
19     // Iterate through each element in the stack
20     for (size_t i = 0; i < obj.m_size; ++i) {
21         // Print the element, followed by a comma unless it's
↵    the last element
22         os << obj.m_stack[i] << (i == (obj.m_size - 1) ? "" : ",
↵    ");
23     }
24     return os;
25 }

```

## Array based stack application: Infix to postfix conversion algorithm

**Concept 11:** In computer science, the conversion of an expression from infix notation to postfix notation is a well-known problem that can be efficiently solved using an array-based stack. This process is crucial in computer science because computers can more easily evaluate expressions in postfix notation (also known as Reverse Polish Notation, RPN) than in infix notation.

### 15.1 Infix Notation

In infix notation, operators are written between the operands they operate on, e.g.,  $A + B$ . While this notation is straightforward for human readers, it requires that the computer understand precedence rules and parentheses to evaluate expressions correctly.

### 15.2 Postfix Notation

In postfix notation, the operator follows all of its operands, e.g.,  $AB+$ . This arrangement eliminates the need for parentheses to dictate order of operations; the order of the operators in the expression does the job instead. Evaluation of postfix expressions can be performed straightforwardly using a stack, making it very attractive for computer processing.

### 15.3 The algorithm

The algorithm for converting an infix expression to a postfix expression using an array-based stack involves the following steps:

1. Create a stack for storing characters and an empty string for the postfix expression.
2. Iterate through each character of the infix expression.
  - (a) If the current character is a lowercase letter, append it to the postfix string followed by a space, and continue to the next character.
  - (b) If the current character is a digit, append all consecutive digits to the postfix string as part of the same number, add a space after the last digit, and then continue to the next character.
  - (c) If the current character is a space, simply continue to the next character without doing anything.
  - (d) If the current character is a left parenthesis '(', push it onto the stack, and continue to the next character.
  - (e) If the current character is a right parenthesis ')', repeatedly pop from the stack and append to the postfix string each character until a left parenthesis '(' is encountered. Pop the left parenthesis from the stack but do not append it to the postfix string. Add a space after each popped character.
  - (f) If the current character is an operator, pop from the stack and append to the postfix string all operators that have greater or equal precedence than the current operator. Add a space after each popped operator. Then, push the current operator onto the stack.

3. After the infix expression has been fully processed, pop and append all remaining operators from the stack to the postfix string, adding a space after each one.
4. Return the resulting postfix string.

The **precedence** function assigns a numerical precedence level to operators, with unary negation and exponentiation having the highest precedence, followed by multiplication and division, and then addition and subtraction with the lowest precedence.

## 15.4 Example

```
1  #include <cctype>
2  #include "inpost.h"
3  #include "mystack.h"
4  std::string convert(const std::string& infix) {
5      mystack stack; // Create a stack for characters
6      std::string postfix = ""; // Create the return string
7      // Step through the infix string
8      for (auto it = infix.c_str(); *it; ++it) {
9          // Check if the character is lowercase
10         if (islower(*it)) {
11             // Append the current infix character to the return
12             ↪ string postfix += *it;
13             postfix += ' ';
14             continue; // Proceed to the next infix character
15
16             // Check if the character is a digit
17         } else if (isdigit(*it)) {
18             // Keep going to get all the consecutive digits
19             while (isdigit(*it)) {
20                 postfix += *it; // Append to the return string
21                 ++it; // Proceed to the next character
22             }
23             postfix += ' '; // Tack on a space
24             --it; // Handle the extraneous increment
25
26             // Check if the character is a space
27         } else if (isspace(*it)) {
28             continue; // Proceed to the next infix character
29
30             // Check if the character is a left parenthesis
31         } else if (*it == '(') {
32             stack.push(*it); // Push the current infix character
33             ↪ onto the stack
34             continue; // Proceed to the next infix character
35
36             // Check if the character is a right parenthesis
37         } else if (*it == ')') {
38             // Loop while the stack is not empty and the
39             ↪ character at the top of the stack is not a left parenthesis
40             while (stack.size() && stack.top() != '(') {
41                 postfix+=stack.top(); // Append the character on
42                 ↪ the top of the stack to the return string
43                 postfix += ' '; // Tack on a space
44                 stack.pop(); // Pop the stack
45             }
46         }
```

```

1         // If the top of the stack is left parenthesis
2         if (stack.size()) {
3             stack.pop(); // pop the stack
4             continue; // Proceed to the next infix character
5         }
6
7         // The character is an operator
8     } else {
9
10        // While the stack is not empty, and the precedence
    ↪ of the current infix character is <= the precedence of the
    ↪ character at the top of the stack
11        while (stack.size() && (precedence(*it) <=
    ↪ precedence(stack.top()))) {
12            postfix += stack.top(); // Append the character
    ↪ on the top of the stack to the return string
13            postfix += ' '; // Tack on a space
14            stack.pop(); // Pop the stack
15        }
16
17        stack.push(*it); // Push the current infix character
    ↪ to the stack
18        continue; // Proceed to the next infix character
19    }
20 }
21
22 // While the stack is not empty
23 while (stack.size()) {
24     postfix += stack.top(); // Append the character on the
    ↪ top of the stack to the return string
25     postfix += ' '; // Tack on a space
26     stack.pop(); // Pop the stack
27 }
28
29 // Return the result
30 return postfix;
31
32 }

```



```

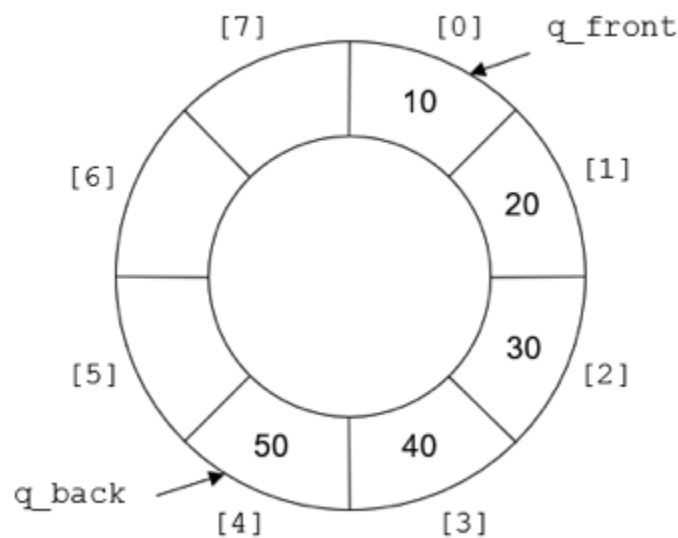
1  unsigned precedence(const char& op) {
2
3      /*
4      The operators used, in order of precedence from highest to
5      ↪ lowest are.
6          1. ~ (Unary negation) and ^ (Exponentiation)
7          2. * (Multiplication) and / (Division)
8          3. + (Addition) and - (Subtraction)
9      */
10     switch (op) {
11         case '~': case '^':
12             return 3;
13             break;
14         case '*': case '/':
15             return 2;
16             break;
17         case '+': case '-':
18             return 1;
19             break;
20         default:
21             return 0;
22     }
23 }

```

## Array based queue

**Concept 12:** An array-based queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, where elements are added (enqueued) at the rear end and removed (dequeued) from the front end. It uses an array to store the elements.

In C++, we could implement an array-based queue as a class. To conserve space, we'll implement it as a "circular queue", an array in which the last position is logically connected back to the first position to make a circle. This is sometimes also called a "ring buffer".



### 16.1 Data members

- **q\_array** - Queue array pointer. A pointer to the data type of the items stored in the queue; points to the first element of a dynamically-allocated array.
- **q\_capacity** - Queue capacity. The number of elements in the queue array.
- **q\_size** - Queue size. The number of items currently stored in the queue.
- **q\_front** - Queue front. The subscript of the front (or head) item in the queue.
- **q\_back** - Queue back. The subscript of the back (or rear or tail) item in the queue.

### 16.2 Member Functions

- **Default constructor** Sets queue to initial empty state. The queue capacity and queue size should be set to 0. The queue array pointer should be set to nullptr. q\_front should be set to 0, while q\_back is set to q\_capacity - 1.
- **size()** Returns the queue size.
- **capacity()** Returns the queue capacity.

- **empty()** Returns true if the queue size is 0; otherwise, false.
- **clear()** Sets the queue size back to 0 and resets `q_front` and `q_back` to their initial values. Does not deallocate any dynamic storage or change the queue capacity.
- **front()** Returns the front item of the queue (`q_array[q_front]`).
- **back()** Returns the back item of the queue (`q_array[q_back]`).
- **push()** Inserts a new item at the back of the queue.
- **pop()** Removes the front item from the queue.
- **Copy constructor** Similar to the copy constructor for the example Vector class in the notes on dynamic storage allocation. A key difference is that we cannot assume that the items in the queue are stored in elements 0 to `q_size - 1` the way we can in the Vector or an array-based stack. It is therefore necessary to copy the entire queue array.
- **Copy assignment operator** Similar to the copy assignment operator for the example Vector class in the notes on dynamic storage allocation. A key difference is that we cannot assume that the items in the queue are stored in elements 0 to `q_size - 1` the way we can in the Vector or an array-based stack. It is therefore necessary to copy the entire queue array.
- **Destructor** Deletes the queue array.
- **reserve()** Reserves additional storage for the queue array. The process of copying the original array contents into the new, larger array is complicated by the fact that the exact locations of the queue items within the queue array are unknown and that there is no guarantee that `q_front` is less than `q_back`.

**Note:-**

the `push()` operation described here is frequently called "enqueue" while the `pop()` operation is frequently called "dequeue".

### 16.3 Double-Ended Queue

We can also easily implement a double-ended queue using an array. The `push()` operation becomes `push_back()` while the `pop()` operation becomes `pop_front()`. No other changes to the code previously described are required. The following two operations can be added to insert an item at the front of the double-ended queue and to remove an item from the back of the double-ended queue.

- **push\_front()** Inserts a new item at the front of the queue.
- **pop\_back()** Removes the back item from the queue.

## 16.4 Example

```
1  class q {
2      int* arr = nullptr;
3      size_t m_capacity = 0;
4      size_t m_size = 0;
5      int m_front = 0;
6      int m_back = this->m_capacity-1;
7
8      public:
9      q() = default;
10
11     q(const q& other) : m_capacity(other.m_capacity),
↪ m_size(other.m_size), m_front(other.m_front),
↪ m_back(other.m_back) {
12         arr = new int[m_capacity];
13         for (size_t i = 0; i < m_size; ++i) {
14             // Copy elements starting from m_front and wrapping
↪ around as necessary
15             arr[(m_front + i) \% m_capacity] =
↪ other.arr[(other.m_front + i) \% other.m_capacity];
16         }
17     }
18
19     q& operator=(const q& other) {
20         if (this != &other) { // Protect against self-assignment
21             delete[] arr; // Free existing resource
22             m_capacity = other.m_capacity;
23             m_size = other.m_size;
24             m_front = other.m_front;
25             m_back = other.m_back;
26
27             arr = new int[m_capacity]; // Allocate new resource
28             for (size_t i = 0; i < m_size; ++i) {
29                 // Copy elements starting from m_front and
↪ wrapping around as necessary
30                 arr[(m_front + i) \% m_capacity] =
↪ other.arr[(other.m_front + i) \% other.m_capacity];
31             }
32         }
33         return *this;
34     }
35
36     ~q() {
37         delete[] arr;
38     }
39
40     size_t size() {
41         return this->m_size;
42     }
```

```

1  size_t capacity() {
2      return this->m_capacity;
3  }
4
5  bool empty() {
6      return !this->m_size;
7  }
8
9  void clear() {
10     this->m_size = 0;
11     this->m_front = 0;
12     this->m_back = this->m_capacity-1;
13 }
14
15 int front() {
16     return this->arr[this->m_front];
17 }
18
19 int back() {
20     if (!empty()) {
21         return this->arr[this->m_back];
22     }
23     return -1;
24 }
25
26 void push(int value) {
27     if (m_size == m_capacity) {
28         this->reserve((m_capacity > 0 ? m_capacity * 2 : 1));
29     }
30
31     m_back = (m_back + 1) \% m_capacity;
32
33     arr[m_back] = value;
34     ++m_size;
35 }
36
37 void pop() {
38     if (!empty()) {
39         m_front = (m_front + 1) \% m_capacity;
40         --m_size;
41     }
42 }
43

```

```

1     void reserve(int n) {
2         if (n <= this->m_capacity) {
3             return;
4         }
5         int* tmp = new int[n];
6
7         int i=0;
8         int j = this->m_front;
9
10        while (i < this->m_size) {
11            tmp[i] = this->arr[j];
12            j = (j+1) \% this->m_capacity;
13            ++i;
14        }
15        this->m_capacity = n;
16        delete[] this->arr;
17
18        this->arr = tmp;
19        this->m_front = 0;
20        this->m_back = this->m_size-1;
21    }
22 };

```

## Singly-linked list (as a stack)

**Concept 13:** A singly linked list is a linear data structure that consists of a sequence of elements, where each element is contained in a "node." The list is called "singly" linked because each node points to the next node in the sequence, forming a single chain of nodes. Unlike arrays, the elements in a singly linked list are not stored in contiguous memory locations; instead, each node contains a reference (or pointer) to the next node, allowing for dynamic memory usage and flexibility in adding or removing elements.

### 17.1 Structure of a Node

Each node in a singly linked list typically has two components:

- **Data:** The actual value or information that the node represents.
- **Next:** A pointer (or reference) to the next node in the list.

### 17.2 Advantages

- **Dynamic Size:** Unlike arrays, singly linked lists can easily grow or shrink in size, making efficient use of memory.
- **Ease of Insertion/Deletion:** Adding or removing elements from a singly linked list does not require shifting elements, as in the case of arrays, making these operations potentially more efficient.

### 17.3 Disadvantages

- **Sequential Access:** Elements in a singly linked list can only be accessed sequentially, starting from the first node. This makes access times slower compared to arrays, which offer constant time access.
- **Extra Memory:** Each node requires extra memory for the pointer, in addition to the data it holds.
- **No Backward Traversal:** Since each node only points to the next node, it's not possible to traverse the list backward without additional structures or references.

Singly linked lists are a fundamental data structure, useful in scenarios where dynamic memory allocation is needed and the benefits of easy insertion/deletion outweigh the costs of slower access times and extra memory usage for pointers.

## 17.4 Sample node structure

```
1  struct node
2  {
3      data-type value;
4      node* next;
5
6      node(data-type value, node* next = nullptr)
7      {
8          this->value = value;
9          this->next = next;
10     }
11 };
```

## 17.5 Class to represent a stack

### 17.5.1 Data members

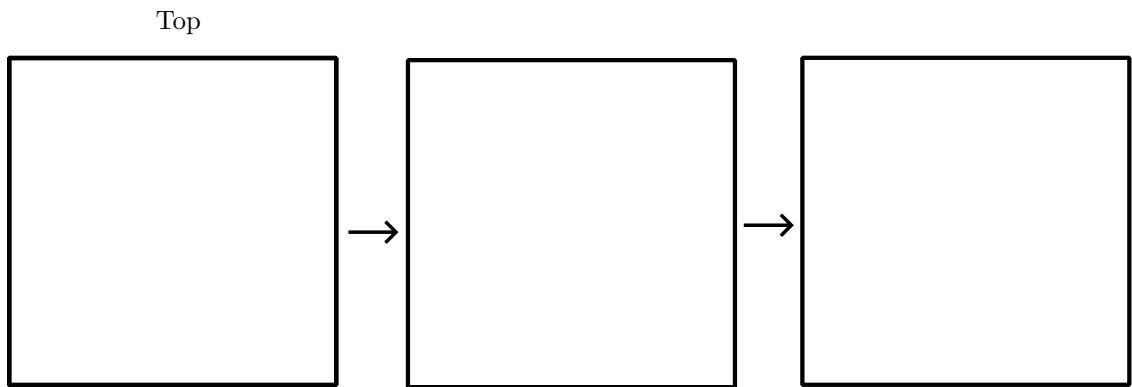
- **stk\_top** - Stack top pointer. Pointer to the top (first) node in the linked list.
- **stk\_size** - Number of items currently stored in the stack.

### 17.5.2 Member Functions

- **Default constructor** Sets stack to initial empty state. The stack top pointer should be set to nullptr. The stack size should be set to 0.
- **size()** Returns the stack size.
- **empty()** Returns true if the stack size is 0; otherwise, false.
- **clear()** We can easily set the stack back to the empty state by repeatedly calling pop() until the stack is empty.
- **top()** Returns the top item of the stack (stk\_top->value).
- **push()** Inserts a new item at the top of the stack.
- **pop()** Removes the top item from stack.
- **Copy Constructor**
- **Copy Assignment Operator**
- **Destructor** We can delete all of the dynamic storage for the stack by calling the clear() member function.
- **clone()** Copies the linked list from the stack x to this object.



## 17.6 Visualization



## 17.7 Example (as a stack)

```
1 // Define a node structure for use in the mystack class
2 struct node {
3     node* next = nullptr; // Pointer to the next node in the
    ↪ stack
4     int value = 0; // The value stored in this node
5
6     node() = default; // Default constructor
7     node(node* next, int value) : next(next), value(value) {};
    ↪ // Constructor initializing members
8 };
9
10 // Define a class to represent a stack using a linked list
11 class mystack {
12     node* stack_top = nullptr; // Pointer to the top node of the
    ↪ stack
13     size_t m_size = 0; // Current size of the stack
14
15 public:
16     // Allow ostream to access private members of mystack for
    ↪ printing
17     friend std::ostream& operator<<(std::ostream& os, const
    ↪ mystack& obj);
18
19     // Copy constructor
20     mystack(const mystack& x) {
21         this->stack_top = nullptr; // Initialize stack_top to
    ↪ nullptr
22         this->m_size = x.size(); // Copy size from x
23         clear(); // Clear existing content
24         clone(x); // Deep copy nodes from x
25     }
26
27     // Copy assignment operator
28     mystack& operator=(const mystack& x) {
29         if (this != &x) { // Check for self-assignment
30             this->stack_top = nullptr; // Reset stack_top
31             this->m_size = x.size(); // Copy size from x
32             clear(); // Clear existing content
33             clone(x); // Deep copy nodes from x
34         }
35         return *this; // Return a reference to the current object
36     }
```

```

1  // Return the current size of the stack
2  size_t size() const {
3      return this->m_size;
4  }
5
6  // Check if the stack is empty
7  bool empty() const {
8      return this->m_size == 0;
9  }
10
11 // Remove the top element from the stack
12 void pop(){
13     node* del = this->stack_top; // Temporary pointer to the top
    ↳ node
14     this->stack_top = this->stack_top->next; // Move the top
    ↳ pointer to the next node
15     delete del; // Deallocate the removed node
16     (this->m_size)--; // Decrement the size of the stack
17 }
18
19 // Clear all elements from the stack
20 void clear(){
21     while (this->stack_top != nullptr) { // While there are
    ↳ nodes in the stack
22         this->pop(); // Remove the top node
23     }
24 }
25
26 // Access the value of the top element in the stack
27 const int& top() const{
28     return stack_top->value;
29 }
30
31 // Add a new element to the top of the stack
32 void push(int value){
33     node* new_node = new node(this->stack_top, value); // Create
    ↳ a new node with the given value
34     this->stack_top = new_node; // Make the new node the top of
    ↳ the stack
35     ++this->m_size; // Increment the size of the stack
36 }

```

```

1      // Clone the stack from another mystack object
2      void clone(const mystack& obj) {
3          if (obj.stack_top == nullptr) { // If the source stack
↳ is empty
4              this->stack_top = nullptr; // Make the current stack
↳ empty
5              return;
6          }
7          stack_top = new node(nullptr, obj.stack_top->value); //
↳ Copy the top node
8          node* src = obj.stack_top->next; // Pointer to traverse
↳ the source stack
9          node* dest = stack_top; // Pointer to build the current
↳ stack
10
11         while(src != nullptr) { // While there are more nodes to
↳ copy
12             dest->next = new node(nullptr, src->value); // Copy
↳ the node
13             dest = dest->next; // Move to the next node
14             src = src->next; // Move to the next source node
15         }
16         this->m_size = obj.m_size; // Copy the size
17     }
18
19     // Destructor to clean up the stack
20     ~mystack() {
21         this->clear(); // Clear the stack
22     }
23 };
24
25 // Overload the << operator to print the stack
26 std::ostream& operator<<(std::ostream& os, const mystack& obj) {
27     node* current = obj.stack_top; // Start from the top of the
↳ stack
28
29     if (current == nullptr) { return os; }
30
31     while (current != nullptr) { // Iterate through the stack
32         os << current->value; // Print the current node's value
33         if (current->next != nullptr)
34             os << ", "; // If this is not the last node,
↳ print a comma and a space
35         }
36         current = current->next; // Move to the next node
37     }
38     return os;
39 }

```

## Singly linked list (as a queue)

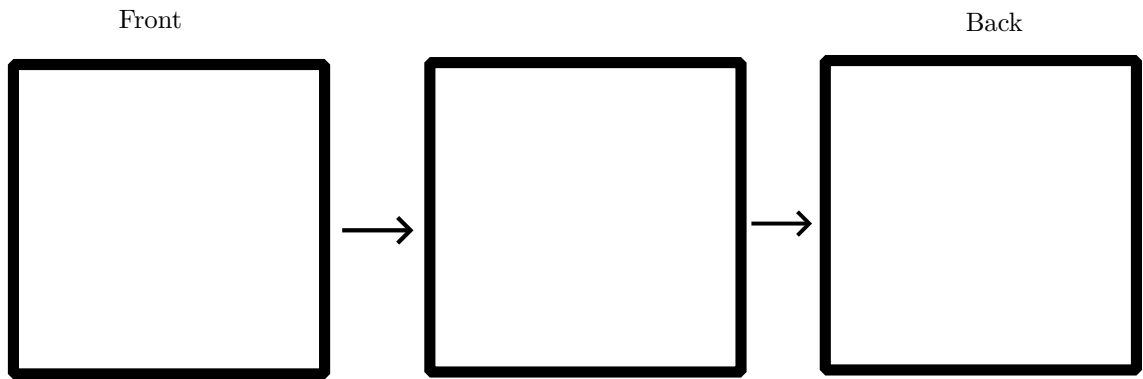
### 18.1 Data Members

- **q\_front** - pointer to front (first) node in the list.
- **q\_back** - pointer to back (last) node in the list.
- **q\_size** - Number of items currently stored in queue.

### 18.2 Member Functions

- **Default constructor** Sets queue to initial empty state. The queue front pointer and the queue back pointer should be set to nullptr. The queue size should be set to 0.
- **size()** Returns the queue size.
- **empty()** Returns true if the queue size is 0; otherwise, false.
- **clear()** We can easily set the queue back to the empty state by repeatedly calling pop() until the queue is empty.
- **front()** Returns the front item of the queue (q\_front->value).
- **back()** Returns the front item of the queue (q\_back->value).
- **push()** Inserts a new item at rear of queue.
- **pop()** Removes the front item from queue.
- **Copy Constructor**
- **Copy Assignment Operator**
- **Destructor**
- **clone()**

### 18.3 Visualization



## 18.4 Example Code

```
1  struct node {
2
3      node* next = nullptr;
4      int value = 0;
5
6      node() = default;
7      node(node* next, int value) : next(next), value(value) {}
8  };
9
10 class qlist {
11
12     node* m_front = nullptr;
13     node* m_back = nullptr;
14     size_t m_size = 0;
15
16
17
18     public:
19     qlist() = default;
20
21     qlist(const qlist& other) {
22         m_front = m_back = nullptr;
23         m_size = 0;
24         clone(other);
25     }
26
27     qlist& operator=(const qlist& other) {
28
29         if (this != &other) {
30             this->clear();
31             clone(other);
32         }
33         return *this;
34     }
35
36     ~qlist() {
37         this->clear();
38     }
39
40
41     size_t size() const {
42         return this->m_size;
43     }
44
45     bool empty() const {
46         return !this->m_size;
47     }
```

```

1  void clear() {
2      while (this->m_size) {
3          this->pop();
4      }
5  }
6
7  int front() const {
8      if (!this->empty()) {
9          return this->m_front->value;
10     }
11     return 0;
12 }
13
14
15 int back() const {
16     if (!this->empty()) {
17         return this->m_back->value;
18     }
19     return 0;
20 }
21
22 void push(int value) {
23
24     node* new_node = new node(nullptr, value);
25
26     if (this->empty()) {
27         this->m_front = new_node;
28     } else {
29         this->m_back->next = new_node;
30     }
31
32     this->m_back = new_node;
33     ++m_size;
34 }
35
36
37 void pop() {
38     if (m_front == nullptr) {
39         return;
40     }
41     node* del = m_front;
42     m_front = m_front->next;
43
44     if (m_front == nullptr) {
45         m_back = nullptr;
46     }
47     delete del;
48     --m_size;
49 }

```



```

1      void clone(const qslist& other) {
2          node* current = other.m_front;
3
4          while (current != nullptr) {
5              this->push(current->value);
6              current = current->next;
7          }
8      }
9
10     friend std::ostream& operator<<(std::ostream& os, const
↪ qslist& obj) {
11         node* current = obj.m_front;
12
13         while (current != nullptr) {
14             os << current->value << endl;
15             current = current->next;
16         }
17         return os;
18     }
19 };
20

```

## Hash Table With Linear Probe

**Concept 14:** A hash table, also known as a hash map, is a data structure that implements an associative array, also called a dictionary, which maps keys to values. An associative array stores a set of (key, value) pairs and allows insertion, deletion, and lookup (search), with the constraint of unique keys.

A hash table uses a hash function to compute an index or subscript (also called a hash value) of an element (or slot) in the array. A key and value to be inserted are stored at this location in the array. During lookup, the key is hashed and the resulting index indicates where the corresponding value is stored.

Ideally, the hash function will assign each key to a unique slot in the hash table array, but most hash table designs employ an imperfect hash function, which might cause **collisions** where the hash function generates the same index for more than one key. Such collisions can be resolved using a number of different strategies; the technique used to resolve collisions on this assignment is called linear probe, which is a variation on the linear search algorithm. This algorithm is described in the member function descriptions below

## 19.1 Hash table header file

```
1  enum element_state
2  {
3      EMPTY, DELETED, FILLED
4  };
5
6  struct table_element
7  {
8      int key = 0;
9      std::string value = "";
10     element_state state = EMPTY;
11 };
12
13 class hash_table
14 {
15     friend std::ostream& operator<<(std::ostream&, const
16     ↪ hash_table&);
17
18     private:
19
20     static const int TABLE_SIZE = 29;
21     table_element table[TABLE_SIZE];
22
23     int hash(int key) const;
24
25     public:
26
27     hash_table() = default;
28
29     bool insert(int key, const std::string& value);
30     int find(int key) const;
31     bool update(int key, const std::string& value);
32     bool erase(int key);
33 };
34
```



## 19.2 Hash table cpp file

```
1  int hash_table::hash(int key) const
2  {
3      return key % TABLE_SIZE;
4  }
5
6  bool hash_table::insert(int key, const string& value)
7  {
8
9      int index = hash(key);
10     bool haslooped = false;
11
12     while (table[index].state != EMPTY && table[index].state !=
↪ DELETED) {
13         if (index == TABLE_SIZE - 1) {
14             index = 0;
15             haslooped = true;
16         }
17         else {
18             index++;
19         }
20
21         if (index == hash(key) && haslooped) {
22             return false;
23         }
24     }
25     table[index].key = key;
26     table[index].value = value;
27     table[index].state = FILLED;
28
29
30     return true;
31 }
32
33 int hash_table::find(int key) const
34 {
35     int index = -1;
36
37     index = hash(key);
38     bool haslooped = false;
39
40     while (table[index].state != EMPTY && table[index].key !=
↪ key)
41     {
42         if (index == TABLE_SIZE - 1)
43         {
44             index = 0;
45             haslooped = true;
46         }
47         else
48         {
49             index++;
50         }
```

```

1         if (index == hash(key) && haslooped)
2         {
3             return false;
4         }
5     }
6     if (table[index].state == EMPTY)
7     {
8         return -1;
9     }
10    else
11    {
12        return index;
13    }
14 }
15
16 bool hash_table::update(int key, const string& value)
17 {
18     int index = find(key);
19
20     if (index == -1)
21     {
22         return false;
23     }
24     else
25     {
26         table[index].value = value;
27         return true;
28     }
29 }
30
31 bool hash_table::erase(int key)
32 {
33     int index = find(key);
34
35     if (index == -1)
36     {
37         return false;
38     }
39     else
40     {
41         table[index].state = DELETED;
42         return true;
43     }
44 }

```

```

1 ostream& operator<<(ostream& os, const hash_table& obj)
2 {
3     os << "Index  Key    Value\n";
4     os << "=====」
↳  =====\n";
5
6     for (int i = 0; i < obj.TABLE_SIZE; i++)
7     {
8         os << setfill(' ') << '[' << setw(2) << right << i << "]"
↳      ";
9
10        if (obj.table[i].state == EMPTY)
11            os << "EMPTY";
12        else if (obj.table[i].state == DELETED)
13            os << "DELETED";
14        else
15            os << setfill('0') << right << setw(4) <<
↳      obj.table[i].key
16            << " " << setfill(' ') << left << obj.table[i].value;
17
18        os << endl;
19    }
20
21    return os;
22 }
23

```

## Reverse a singly linked list

For this, we just add a `reverse()` method in the singly-linked list class file

```
1  void mylist::reverse()
2  {
3      // Temporarys for current, next, and previous
4      node* curr = l_front, *prev = nullptr, *next = nullptr;
5
6      while (curr != nullptr) {
7          next = curr->next; // assign next to the next node
8          curr->next = prev; // Reverse the "arrow" of the current
↪      node
9          prev = curr; // Advance previous
10         curr = next; // Advance current
11     }
12
13     l_front = prev; // Assign head of list to new front (at the
↪     end of the loop prev will be the top node)
14 }
```



## Doubly-linked list as a template class