Data Structures and Algorithms

In C++

Nathan Warner

Computer Science
Northern Illinois University
February 16, 2023
United States

Contents

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

```
procedure selection_sort(array : list of sortable items, n : length of list)
    i := 0
    while i < n - 1
        min_index ← i
        j := i + 1
        while j < n
            if array[j] < array[min_index]
                min_index ← j
            end if
            j = j + 1
        end while
        swap array[i] and array[min_index]
        i = i + 1
    end while
end procedure
```

1.2 Example

```
int main(int argc, const char* argv[]) {
    int arr[] = {2,4,1,3,5}; int n = 5;

    for (int j=0; j <n-1; ++j) {
        int min = j;
        for (int k=j+1; k <n-1; ++k) {
            if (arr[k] < arr[min]) {
                min = k;
            }
        }
        std::swap(arr[j], arr[min]);
    }
}
```

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

```
procedure insertion_sort(array : list of sortable items, n :
↪ length of list)
    i ← 1
    while i < n
        j ← i
        while j > 0 and array[j - 1] > array[j]
            swap array[j - 1] and array[j]
            j ← j - 1
        end while
        i ← i + 1
    end while
end procedure
```

2.2 Example

```
int main(int argc, const char* argv[]) {
    int arr[] = {2,4,1,3,5};
    int n = 5;

    for (int j=1; j<n; ++j) {
        for (int k=j; k>0; --k) {
            if (arr[k-1] > arr[k]) {
                std::swap(arr[k-1], arr[k]);
            }
        }
    }
}
```

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

```
procedure insertion_sort(array : list of sortable items, n :
 ↪  length of list)
    i ← 1
    while i < n
        temp ← array[i]
        j ← i
        while j > 0 and array[j - 1] > temp
            array[j] ← array[j - 1]
            j ← j - 1
        end while
        array[j] ← temp
        i ← i + 1
    end while
end procedure
```

2.3.2 Example

```
int arr[] = {5,6,4,3,1};
int n = 5;

for (int j=1; j<n; ++j) {
    int tmp = arr[j];
    int k=j;
    for (; k>0; --k) {
        if (arr[k-1] > tmp) {
            arr[k] = arr[k-1];
        } else {
            break;
        }
    }
    arr[k] = tmp;
}
```

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

```
procedure bubble_sort(array : list of sortable items, n : length of list)
    do
        swapped ← false
        i ← 1
        while i < n
            if array[i - 1] > array[i]
                swap array[i - 1] and array[i]
                swapped ← true
            end if
            i ← i + 1
        end while
    while swapped
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

```
int arr[] = {5,6,4,3,1};
int n = 5;

bool swapped;
do {
    swapped = 0;

    for (int i=0; i<n; ++i) {
        if (arr[i-1] > arr[i]) {
            std::swap(arr[i-1], arr[i]);
            swapped = 1;
        }
    }

} while (swapped);

```

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

```
procedure bubble_sort(array : list of sortable items, n : length
↪  of list)
    do
        swapped ← false
        i ← 1
        while i < n
            if array[i - 1] > array[i]
                swap array[i - 1] and array[i]
                swapped ← true
            end if
            i ← i + 1
        end while
        n ← n - 1
    while swapped
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:

```
 do
    last ← 0
    i ← 1
    while i < n
        if array[i - 1] > array[i]
            swap array[i - 1] and array[i]
            last ← i
        end if
        i ← i + 1
    end while
    n ← last
while n > 1
```

3.3.2 Example

```
int last;
do  {
    last = 0;
    int j=1;

    for (; j<n; ++j) {
        if (arr[j-1] > arr[j]) {
            std::swap(arr[j-1], arr[j]);
            last = j;
        }
    }
    n = last;

} 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 Θ, and big Ω 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 $X \leqslant 130$. Similarly, it would also be correct, albeit less useful, to say that $X \leqslant 1,000$ or $X \leqslant 1,000,000$. 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 Ω**: In academia, Ω 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 Ω 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 Θ**: In academia, Θ 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 $\Omega(N)$. Θ 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 Θ 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 $\boldsymbol{\Theta}$

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, wecould take essentially any algorithm, special case some input, and then get a O(1) runtime in thebest 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 $O(n)$ space. If we need a two-dimensional array of size $n \times n$, this will require $O(n^2)$ space.

6.2.1 Constant time

An algorithm has 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 $O(n)$ time and $O(n)$ space

```
// Example 1
int sum(int n) {
if (n <= 0)
    return 0;
else
    return n + sum(n - 1);
}
```

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?

```
// Program 1

for (int i=0; i<A; ++i) {
    cout << arrayA[i];
}

for (int i=0; i<B; ++i) {
    cout << arrayB[i];
}

// Program 2
for (int i=0; i<A; ++i) {
    for (int j=0; j<B; ++j) {
        cout << arrayA[i] << ", " << arrayB[j];
    }
}
```

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 $1 + 2 + 4 + 8 + 16 + \ldots + X$? 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 $X + X/2 + X/4 + X/8 + \ldots + 1$? 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.

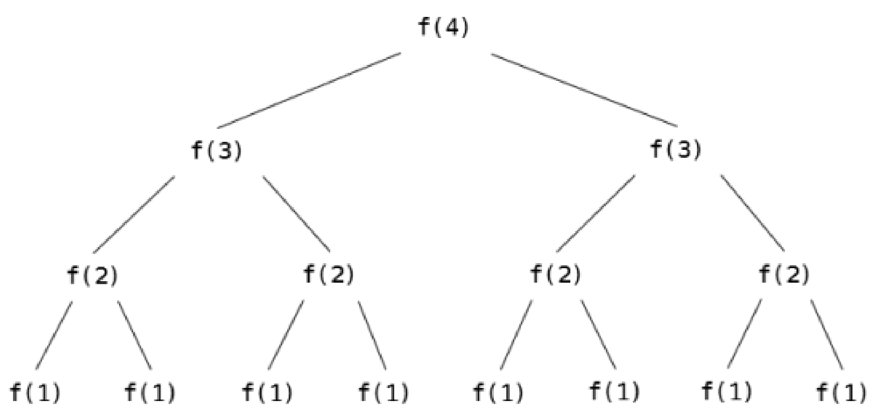
$$\begin{aligned} 2^k &= n \\ &= \log_2 n = k. \end{aligned}$$

6.8 Recursive runtime

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

```
int f(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return f(n-1) + f(n-1);
    }
}

```

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 $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^N$ (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 time each recursive call branches. In this case, this gives us O(2N).

The space complexity of this algorithm will be $O(N)$. Although we have O(2N) 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

```
int arr[] = {6,3,2,1,8};
int n = 5;

for (int iv=n/2; iv>0; iv/=2) {

    for (int i=iv; i<n; ++i) {

        int j;
        int tmp = arr[i];
        for (j = i; j>=iv && arr[j-iv] > tmp; j-=iv) {
            arr[j] = arr[j-iv];
        }
        arr[j] = tmp;
    }
}
```

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

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

8.3.2 Recursive function

```
procedure quicksort(array : list of sortable items, start : first element of list,
end : last element of list)
    if start < end
        pivot_point ← partition(array, start, end)
        quick_sort(array, start, pivot_point - 1)
        quick_sort(array, pivot_point + 1, end)
    end if
end procedure
```

8.3.3 Partition function

```
procedure partition(array : list of sortable items, start : first element of list,
end : last element of list)
    mid ← (start + end) / 2
    swap array[start] and array[mid]

    pivot_index ← start
    pivot_value ← array[start]

    scan ← start + 1
    while scan <= end
        if array[scan] < pivot_value
            pivot_index ← pivot_index + 1
            swap array[pivot_index] and array[scan]
        end if
        scan ← scan + 1
    end while

    swap array[start] and array[pivot_index]

    return pivot_index
end procedure
```

8.4 Examples

```
int partition(int arr[], int start, int end) {
    int pivot_index, pivot_value, mid, scan;

    mid = (start + end) / 2;
    std::swap(arr[start], arr[mid]);

    pivot_index = start;
    pivot_value = arr[start];

    scan = start + 1;

    while (scan <= end) {
        if (arr[scan] < pivot_value) {
            ++pivot_index;
            std::swap(arr[pivot_index], arr[scan]);
        }
        ++scan;
    }
    std::swap(arr[start], arr[pivot_index]);

    return pivot_index;
}

void quicksort(int arr[], int start, int end) {
    int pivot_point;
    if (start < end) {
        pivot_point = partition(arr, start, end);
        quicksort(arr, start, pivot_point - 1);
        quicksort(arr, pivot_point + 1, end);
    }
}

void quicksort(int arr[], int n) {
    quicksort(arr, 0, n-1);
}

int main(int argc, const char* argv[]) {

    int arr[]  = {3,6,1,9,12,7,36,24,18,4};
    int n = 10;

    quicksort(arr,n);

    return EXIT_SUCCESS;
}
```

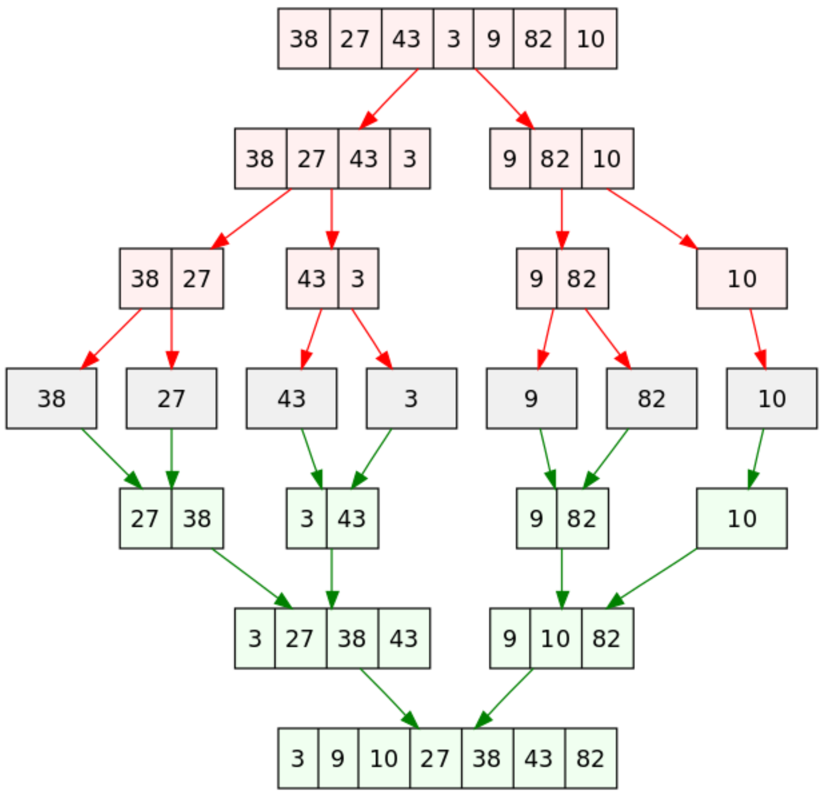
8.5 Complexity

- **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

```
procedure merge_sort(array : list of sortable items, start : first element of list,
end : last element of list)
    if start < end
        mid ← (start + end) / 2

        merge_sort(array, start, mid)
        merge_sort(array, mid + 1, end)

        merge(array, start, mid, end)
    end if
end procedure

procedure merge(array : list of items to merge, start : first element of first sublist, mid : last element of first sublist,
 end : last element of second sublist)
    vector<int> temp(end - start + 1);

    i ← start
    j ← mid + 1
    k ← 0

    while i <= mid and j <= end
        if array[i] < array[j]
            temp[k] ← array[i]
            i ← i + 1
        else
            temp[k] ← array[j]
            j ← j + 1
        end if
        k ← k + 1
    end while

    while i <= mid
        temp[k] ← array[i]
        i ← i + 1
        k ← k + 1
    end while

    while j <= end
        temp[k] ← array[j]
        j ← j + 1
        k ← k + 1
    end while

    Copy the elements of the vector temp back into array
end procedure
```

9.2 Example

```
void merge(int arr[], int start, int mid, int end) {

    vector<int> temp(end - start + 1);

    int i,j,k;

    i = start;
    j = mid + 1;
    k = 0;

    while (i <= mid && j<= end) {
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            ++i;
        } else {
            temp[k] = arr[j];
            ++j;
        }
        ++k;
    }

    while (i <= mid) {
        temp[k] = arr[i];
        ++i;
        ++k;
    }

    while (j <= end) {
        temp[k] = arr[j];
        ++j;
        ++k;
    }

    for ( i=start, j=0; i<=end; ++i, ++j) {
        arr[i] = temp[j];
    }
}

void merge_sort(int arr[], int start, int end) {

    int mid;
    if (start < end) {
        mid = (start + end) / 2;

        merge_sort(arr, start, mid);
        merge_sort(arr, mid+1, end);

        merge(arr, start, mid, end);
    }
}
```

Binary Heap

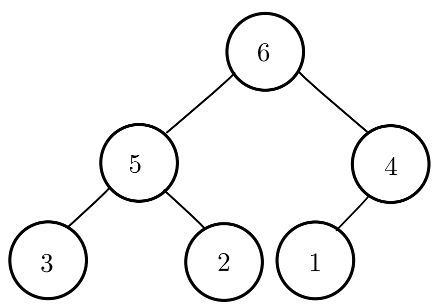
Concept 8: A **binary heap** is a data structure that takes the form of binary tree with two additional constraints

- 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 referes 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 is left and right children are at index positions

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

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

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

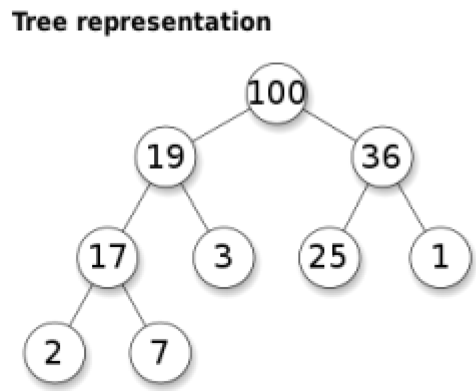
- The index position of the last non-leaf node is given by

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

  Where n is the number of elements in the binary heap

10.6 Derivation: Number of leafs in a complete or almost complete tree

Consider the almost complete binary tree (max heap in this case)



To compute the number of leaf nodes in this tree, we need two quantitys,

1. The number of unfilled nodes at depth d
2. The number of leaf nodes at depth $d-1$

To find the number of unfilled nodes at depth d, we use

$$n - (2^d - 1) = n - 2^d + 1.$$

The quantity $2^d - 1$ returns the number of nodes at depth $d-1$, Hence, subtracting n from this quantity gives the number of leaf nodes at depth d

To find the number of leaf nodes at depth $d-1$, we use

$$\lfloor \frac{2^{d+1} - n - 1}{2} \rfloor.$$

Where d is the depth of the tree, and n is the total number of nodes in the tree. The idea behind this formula is, we find the total number of possible nodes at depth d (2^d), then subtract the number of leaf nodes at depth d to get the number of nodes it will take to completly fill d, divding by 2 gives the number of leaf nodes at depth $d-1$ (flooring the result of course).

The summation of these results gives the total number of leaf nodes L. That is,

$$L = \lfloor \frac{2^{d+1} - n - 1}{2} \rfloor + (n - 2^d + 1).$$

10.7 Algorithm: Verify max heap (assuming complete or almost complete binary tree)

```
    bool is_max_heap(int a[], int n) {

        int depth = log2(n);
        int leaf = (pow(2,(depth + 1)) - n -1) / 2 + (n -
    ↪  pow(2,depth) + 1);

        for (int i=0,j=1,k=2; i<(n-leaf); ++i, j=j+2, k=k+2) {
            if (a[i] < a[j] || a[i] < a[k]) {
                return false;
            }
        }

    }
return true;
```

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

```
procedure heap_sort(array : list of sortable items, n : length of list)
    // end : array subscript
    // Build the heap in array so that largest value is at the
    // root.
    heapify(array, n);

    end = n - 1

    while end > 0
        // array[0] is the root and largest value. The swap
        // moves it in front of the sorted elements.
        swap array[end] and array[0]

        // The heap size is reduced by 1.
        end = end - 1

        // The swap ruined the heap property, so restore it.
        sift_down(array, 0, end);
    end while
end procedure

procedure heapify(array : list of sortable items, n : length of list)
    // start : array subscript
    start = (n - 2) / 2 // Find parent of last element of array
    while start >= 0
        // Sift down the value at subscript 'start' to the proper place
        // such that all values below the start subscript are in max
        // heap order

        sift_down(array, start, n - 1)

        // Go to next parent
        start = start - 1
    end while

    // All elements are now in max heap order
end procedure
```

```
procedure sift_down(array : list of sortable items, start : starting
 subscript of heap, end : ending subscript of heap)
    // root : array subscript
    // largest : array subscript
    // child : array subscript

    // Repair the heap whose root element is at subscript 'start',
    // assuming the heaps rooted at its children are valid

    root = start

    // While the root has at least one child
    while (2 * root + 1) <= end
        child = 2 * root + 1 // Left child of root
        largest = root // Assume root is largest

        // If left child is larger than root, left child is largest
        if array[largest] < array[child]
            largest = child
        end if

        // If there is a right child and it is greater than largest,
        // right child is largest
        if (child + 1) <= end and array[largest] < array[child+1]
            largest = child + 1
        end if

        // If root is largest, no need to continue
        if largest == root
            return
        else
            swap array[root] and array[largest]
            root = largest
        end if

    end while
end procedure
```

11.2 Example

```
#include <utility>
void heapify(int [], int n);
void sift_down(int [], int, int);

void heap_sort(int array[], int n) {
	int end;

	heapify(array, n);

	end = n - 1;

	while(end > 0) {
		std::swap(array[end], array[0]);

		end--;

		sift_down(array, 0, end);
	}
}
void heapify(int array[], int n) {
	int start = (n - 2)/2;

	while(start >= 0) {
		sift_down(array, start, n-1);
		start--;
	}
}
void sift_down(int array[], int start, int end) {
	int root = start;
	int child;
	int largest;

	while(2 * root + 1 <= end) {
		child = 2 * root + 1;
		largest = root;

		if(array[largest] < array[child]) {
			largest = child;
		}
		if(child + 1 <= end && array[largest] <
	↪ array[child + 1]) {
			largest = child + 1;
		}
		if(largest == root) { return; } else {
			std::swap(array[root], array[largest]);
			root = largest;
		}
	}
}
```

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:

```
int main(int argc, const char *argv[]) {

    const int SIZE = 5;
    int arr[SIZE] = {88,67,5,23,19};

    int target = 5;

    for (int i{0}; i <= SIZE + 1; ++i) {
        if (i == SIZE + 1) {
            cout << "Target not in array" << endl;
        }
        if ( arr[i] == target ) {
            cout << "Target [" << target << "] found at index position " << i << endl;
            break;
        }
    }
    return EXIT_SUCCESS;
}
```

```
int linearsearch(int arr[], int size, int target) {
    int index{0}, position{-1};
    bool found = false;

    while (index < size && !found) {
        if (arr[index] == target) {
            position = index;
            found = true;
        }
        ++index;
    }
    return position;
}
```

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 leement 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:

```
int binarysearch(int arr[], int size, int target) {
    int first{0},
        middle,
        last = size -1,
        position{-1};

    bool found = false;

    while (!found && first <= last) {
        middle = (first + last) / 2;
        if (arr[middle] == target) {
            found = true;
            position = middle;
        } else if (target > arr[middle]) {
            first = middle + 1;
        } else {
            last = middle - 1;
        }
    }
    return position;
}
```

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 a 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

```
Vector::Vector(const Vector& other)
{
    // Step 1
    vCapacity = other.vCapacity;
    vSize = other.vSize;

    // Step 2
    if (vCapacity > 0)
    vArray = new int[vCapacity];
    else
    vArray = nullptr;

    // Step 3
    for (size_t i = 0; i < vSize; ++i)
    vArray[i] = other.vArray[i];
}
```

14.4 Reference: Vector copy assignment operator

```
    Vector& Vector::operator=(const Vector& other)
    {
        // Step 1
    if (this != &other)
    {
        // Step 2
        delete[] vArray;

        // Step 3
        vCapacity = other.vCapacity;
        vSize = other.vSize;

        // Step 4
        if (vCapacity > 0)
        vArray = new int[vCapacity];
        else
        vArray = nullptr;


        // Step 5
        for (size_t i = 0; i < vSize; ++i)
        vArray[i] = other.vArray[i];
    }

    // Step 6
    return *this;
}
```

14.5 Auxiliary: Vector move constructor

```
Vector::Vector(Vector&& other)    // rvalue reference to a Vector
{
    // Step 1 - "pilfer" other object's resources
    vCapacity = other.vCapacity;
    vSize = other.vSize;
    vArray = other.vArray;

    // Step 2 - set other object to default state
    other.vCapacity = 0;
    other.vSize = 0;
    other.vArray = nullptr;
}
```

14.6 Array based stack example

```
class mystack {
    // Pointer to dynamically allocated array for stack elements
    char* m_stack = nullptr;
    // Current capacity of the stack
    size_t m_capacity = 0;
    // Current number of elements in the stack
    size_t m_size = 0;

    public:
    // Copy constructor
    mystack(const mystack& x) {
        // Copy capacity and size from source object
        this->m_capacity = x.m_capacity;
        this->m_size = x.m_size;

        // Allocate memory if capacity is greater than 0
        if (this->m_capacity > 0) {
            this->m_stack = new char[this->m_capacity];
        } else {
            this->m_stack = nullptr;
        }

        // Copy the stack elements from source to this object
        memcpy(this->m_stack, x.m_stack, x.m_size);
    }

    // Copy assignment operator
    mystack& operator=(const mystack& x) {
        // Allocate new memory space for the copy
        this->m_capacity = x.m_capacity;
        this->m_size = x.m_size;

        if (this->m_capacity > 0) {
            this->m_stack = new char[this->m_capacity];
        } else {
            this->m_stack = nullptr;
        }

        // Copy the elements
        memcpy(this->m_stack, x.m_stack, x.m_size);

        return *this; // Return a reference to the current object
    }

    // Returns the current capacity of the stack
    size_t capacity() const {
        return this->m_capacity;
    }

```

```
// Returns the current size of the stack
size_t size() const {
    return this->m_size;
}

// Checks if the stack is empty
bool empty() const {
    return this->m_size == 0;
}

// Clears the stack (does not deallocate memory)
void clear(){
    this->m_size = 0;
}

// Ensures the stack has at least the specified capacity
void reserve(size_t n){
    // Only proceed if the new capacity is greater than the current capacity
    if (n <= this->m_capacity) { return; }

    // Update the capacity
    this->m_capacity = n;
    // Allocate new memory
    char* tmp = new char[this->m_capacity];
    // Copy existing elements to the new memory
    memcpy(tmp, this->m_stack, this->m_size);

    // Delete old stack and update pointer
    delete[] this->m_stack;
    this->m_stack = tmp;
}

// Returns a reference to the top element of the stack
const char& top() const{
    return this->m_stack[(this->m_size)-1];
}

// Adds a new element to the top of the stack
void push(char value){
    // If the stack is full, increase its capacity
    if (this->m_size == this->m_capacity) {
        this->reserve((this->m_capacity == 0 ? 1 : this->m_capacity * 2));
    }

    // Add the new element and increment the size
    this->m_stack[(this->m_size)++] = value;
}

```

```cpp
    // Removes the top element from the stack
    void pop(){
        if (this->m_size > 0) {
            --(this->m_size);
        }
    }

    // Destructor: deallocates the dynamically allocated stack
    ~mystack() {
        delete[] this->m_stack;
    }

    // Friend function to output the contents of the stack to a stream
    friend std::ostream& operator<<(std::ostream& os, const mystack& obj);
};

// Outputs the contents of the stack to a stream
std::ostream& operator<<(std::ostream& os, const mystack& obj) {
    // Iterate through each element in the stack
    for (size_t i = 0; i < obj.m_size; ++i) {
        // Print the element, followed by a comma unless it's the last element
        os << obj.m_stack[i] << (i == (obj.m_size - 1) ? "" : ", ");
    }
    return os;
}
```

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

```cpp
#include <cctype>
#include "inpost.h"
#include "mystack.h"
std::string convert(const std::string& infix) {
    mystack stack; // Create a stack for characters
    std::string postfix = ""; // Create the return string
    // Step through the infix string
    for (auto it = infix.c_str(); *it; ++it) {
        // Check if the character is lowercase
        if (islower(*it)) {
            // Append the current infix character to the return string
            postfix += *it;
            postfix += ' ';
            continue; // Proceed to the next infix character

        // Check if the character is a digit
        } else if (isdigit(*it)) {
            // Keep going to get all the consecutive digits
            while (isdigit(*it)) {
                postfix += *it; // Append to the return string
                ++it; // Proceed to the next character
            }
            postfix += ' '; // Tack on a space
            --it; // Handle the extraneous increment

        // Check if the character is a space
        } else if (isspace(*it)) {
            continue; // Proceed to the next infix character

        // Check if the character is a left parenthesis
        } else if (*it == '(') {
            stack.push(*it); // Push the current infix character onto the stack
            continue; // Proceed to the next infix character

        // Check if the character is a right parenthesis
        } else if (*it == ')') {
            // Loop while the stack is not empty and the character at the top of the stack is not a left parenthesis
            while (stack.size() && stack.top() != '(') {
                postfix+=stack.top(); // Append the character on the top of the stack to the return string
                postfix += ' '; // Tack on a space
                stack.pop(); // Pop the stack
            }
```

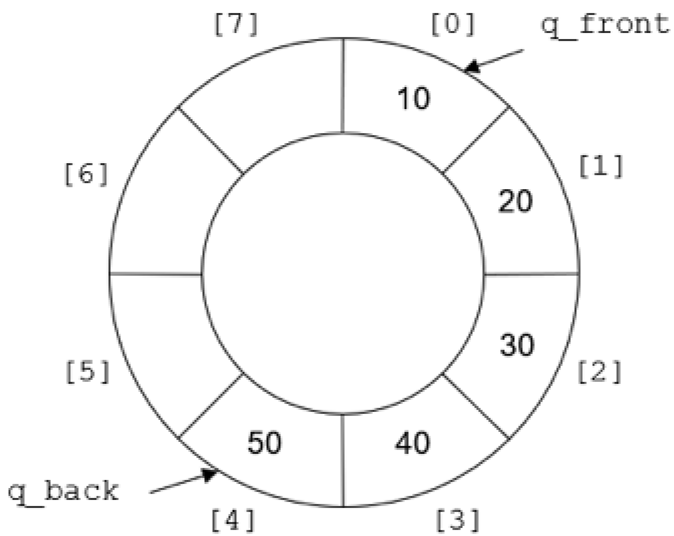
```
            // If the top of the stack is left parenthesis
            if (stack.size()) {
                stack.pop(); // pop the stack
                continue; // Proceed to the next infix character
            }

        // The character is an operator
        } else {

            // 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
            while (stack.size() && (precedence(*it) <= precedence(stack.top()))) {
                postfix += stack.top(); // Append the character on the top of the stack to the return string
                postfix += ' '; // Tack on a space
                stack.pop(); // Pop the stack
            }

            stack.push(*it); // Push the current infix character to the stack
            continue; // Proceed to the next infix character
        }
    }

    // While the stack is not empty
    while (stack.size()) {
        postfix += stack.top(); // Append the character on the top of the stack to the return string
        postfix += ' '; // Tack on a space
        stack.pop(); // Pop the stack
    }

    // Return the result
    return postfix;

}
```

```cpp
unsigned precedence(const char& op) {

    /*
     The operators used, in order of precedence from highest to lowest are.
        1. ~ (Unary negation) and ^ (Exponentiation)
        2. * (Multiplication) and / (Division)
        3. + (Addition) and - (Subtraction)
    */

    switch (op) {
        case '~': case '^':
            return 3;
            break;
        case '*': case '/':
            return 2;
            break;
        case '+': case '-':
            return 1;
            break;
        default:
            return 0;
    }
}
```

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

```
class q {
    int* arr = nullptr;
    size_t m_capacity = 0;
    size_t m_size = 0;
    int m_front = 0;
    int m_back = this->m_capacity-1;

    public:
    q() = default;

    q(const q& other) : m_capacity(other.m_capacity), m_size(other.m_size), m_front(other.m_front), m_back(other.m_back) {
        arr = new int[m_capacity];
        for (size_t i = 0; i < m_size; ++i) {
            // Copy elements starting from m_front and wrapping around as necessary
            arr[(m_front + i) \% m_capacity] = other.arr[(other.m_front + i) \% other.m_capacity];
        }
    }

    q& operator=(const q& other) {
        if (this != &other) { // Protect against self-assignment
            delete[] arr; // Free existing resource
            m_capacity = other.m_capacity;
            m_size = other.m_size;
            m_front = other.m_front;
            m_back = other.m_back;

            arr = new int[m_capacity]; // Allocate new resource
            for (size_t i = 0; i < m_size; ++i) {
                // Copy elements starting from m_front and wrapping around as necessary
                arr[(m_front + i) \% m_capacity] = other.arr[(other.m_front + i) \% other.m_capacity];
            }
        }
        return *this;
    }

    ~q() {
        delete[] arr;
    }

    size_t size() {
        return this->m_size;
    }
```

```
size_t capacity() {
    return this->m_capacity;
}

bool empty() {
    return !this->m_size;
}

void clear() {
    this->m_size = 0;
    this->m_front = 0;
    this->m_back = this->m_capacity-1;
}

int front() {
    return this->arr[this->m_front];
}

int back() {
    if (!empty()) {
        return this->arr[this->m_back];
    }
    return -1;
}

void push(int value) {
    if (m_size == m_capacity) {
        this->reserve((m_capacity > 0 ? m_capacity * 2 : 1));
    }

    m_back = (m_back  + 1) \% m_capacity;

    arr[m_back] = value;
    ++m_size;
}

void pop() {
    if (!empty()) {
        m_front = (m_front + 1) \% m_capacity;
        --m_size;
    }
}

```

```
    void reserve(int n) {
        if (n <= this->m_capacity) {
            return;
        }
        int* tmp = new int[n];

        int i=0;
        int j = this->m_front;

        while (i < this->m_size) {
            tmp[i] = this->arr[j];
            j = (j+1) \% this->m_capacity;
            ++i;
        }
        this->m_capacity = n;
        delete[] this->arr;

        this->arr = tmp;
        this->m_front = 0;
        this->m_back = this->m_size-1;
    }
};
```

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

```
struct node
{
    data-type value;
    node* next;

    node(data-type value, node* next = nullptr)
    {
        this->value = value;
        this->next = next;
    }
};
```

17.5 Class to represent a stack

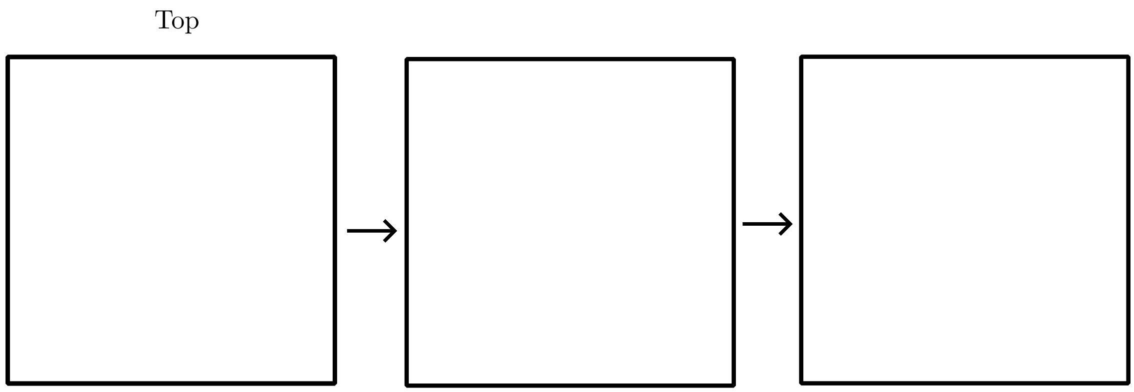
17.5.1 Data members

- **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)

```cpp
// Define a node structure for use in the mystack class
struct node {
    node* next = nullptr; // Pointer to the next node in the stack
    int value = 0; // The value stored in this node

    node() = default; // Default constructor
    node(node* next, int value) : next(next), value(value) {}; // Constructor initializing members
};

// Define a class to represent a stack using a linked list
class mystack {
    node* stack_top = nullptr; // Pointer to the top node of the stack
    size_t m_size = 0; // Current size of the stack

public:
    // Allow ostream to access private members of mystack for printing
    friend std::ostream& operator<<(std::ostream& os, const mystack& obj);

    // Copy constructor
    mystack(const mystack& x) {
        this->stack_top = nullptr; // Initialize stack_top to nullptr
        this->m_size = x.size(); // Copy size from x
        clear(); // Clear existing content
        clone(x); // Deep copy nodes from x
    }

    // Copy assignment operator
    mystack& operator=(const mystack& x) {
        if (this != &x) { // Check for self-assignment
            this->stack_top = nullptr; // Reset stack_top
            this->m_size = x.size(); // Copy size from x
            clear(); // Clear existing content
            clone(x); // Deep copy nodes from x
        }
        return *this; // Return a reference to the current object
    }
```

```cpp
// Return the current size of the stack
size_t size() const {
    return this->m_size;
}

// Check if the stack is empty
bool empty() const {
    return this->m_size == 0;
}

// Remove the top element from the stack
void pop(){
    node* del = this->stack_top; // Temporary pointer to the top node
    this->stack_top = this->stack_top->next; // Move the top pointer to the next node
    delete del; // Deallocate the removed node
    (this->m_size)--; // Decrement the size of the stack
}

// Clear all elements from the stack
void clear(){
    while (this->stack_top != nullptr) { // While there are nodes in the stack
        this->pop(); // Remove the top node
    }
}

// Access the value of the top element in the stack
const int& top() const{
    return stack_top->value;
}

// Add a new element to the top of the stack
void push(int value){
    node* new_node = new node(this->stack_top, value); // Create a new node with the given value
    this->stack_top = new_node; // Make the new node the top of the stack
    ++this->m_size; // Increment the size of the stack
}
```

```cpp
    // Clone the stack from another mystack object
    void clone(const mystack& obj) {
        if (obj.stack_top == nullptr) { // If the source stack is empty
            this->stack_top = nullptr; // Make the current stack empty
            return;
        }
        stack_top = new node(nullptr, obj.stack_top->value); // Copy the top node
        node* src = obj.stack_top->next; // Pointer to traverse the source stack
        node* dest = stack_top; // Pointer to build the current stack

        while(src != nullptr) { // While there are more nodes to copy
            dest->next = new node(nullptr, src->value); // Copy the node
            dest = dest->next; // Move to the next node
            src = src->next; // Move to the next source node
        }
        this->m_size = obj.m_size; // Copy the size
    }

    // Destructor to clean up the stack
    ~mystack() {
        this->clear(); // Clear the stack
    }
};

// Overload the << operator to print the stack
std::ostream& operator<<(std::ostream& os, const mystack& obj) {
    node* current = obj.stack_top; // Start from the top of the stack

    if (current == nullptr) { return os; }

    while (current != nullptr) { // Iterate through the stack
        os << current->value; // Print the current node's value
        if (current->next != nullptr)
                os << ", "; // If this is not the last node, print a comma and a space
            }
            current = current->next; // Move to the next node
        }
        return os;
}
```

Singly linked list (as a queue)

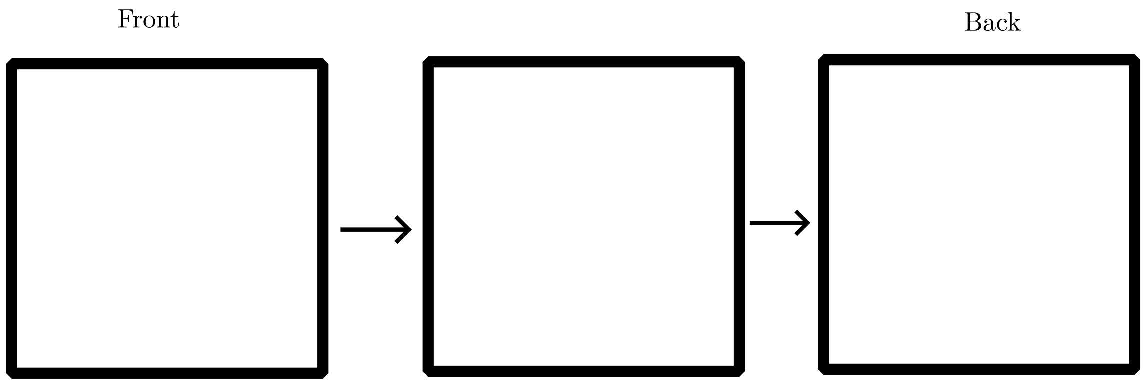
18.1 Data Members

- **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

```
struct node {

    node* next = nullptr;
    int value = 0;

    node() = default;
    node(node* next, int value) : next(next), value(value) {}
};

class qslist {

    node* m_front = nullptr;
    node* m_back = nullptr;
    size_t m_size = 0;



    public:
    qslist() = default;

    qslist(const qslist& other) {
        m_front = m_back = nullptr;
        m_size = 0;
        clone(other);
    }

    qslist& operator=(const qslist& other) {

        if (this != &other) {
            this->clear();
            clone(other);
        }
        return *this;
    }

    ~qslist() {
        this->clear();
    }


    size_t size() const {
        return this->m_size;
    }

    bool empty() const {
        return !this->m_size;
    }
```

```
void clear() {
    while (this->m_size) {
        this->pop();
    }
}

int front() const {
    if (!this->empty()) {
        return this->m_front->value;
    }
    return 0;

}

int back() const {
    if (!this->empty()) {
        return this->m_back->value;
    }
    return 0;
}

void push(int value) {

    node* new_node = new node(nullptr, value);

    if (this->empty()) {
        this->m_front = new_node;
    } else {
        this->m_back->next = new_node;
    }

    this->m_back = new_node;
    ++m_size;

}

void pop() {
    if (m_front == nullptr) {
        return;
    }
    node* del = m_front;
    m_front = m_front->next;

    if (m_front == nullptr) {
        m_back = nullptr;
    }
    delete del;
    --m_size;
}
```

```
    void clone(const qslist& other) {
        node* current = other.m_front;

        while (current != nullptr) {
            this->push(current->value);
            current = current->next;
        }
    }

    friend std::ostream& operator<<(std::ostream& os, const
↪   qslist& obj) {
        node* current = obj.m_front;

        while (current != nullptr) {
            os << current->value << endl;
            current = current->next;
        }
        return os;
    }
};

```

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

```
enum element_state
{
    EMPTY, DELETED, FILLED
};

struct table_element
{
    int key = 0;
    std::string value = "";
    element_state state = EMPTY;
};

class hash_table
{
    friend std::ostream& operator<<(std::ostream&, const
    ↪ hash_table&);

    private:

    static const int TABLE_SIZE = 29;
    table_element table[TABLE_SIZE];

    int hash(int key) const;

    public:

    hash_table() = default;

    bool insert(int key, const std::string& value);
    int find(int key) const;
    bool update(int key, const std::string& value);
    bool erase(int key);
};
```

19.2 Hash table cpp file

```cpp
int hash_table::hash(int key) const
{
    return key % TABLE_SIZE;
}

bool hash_table::insert(int key, const string& value)
{

    int index = hash(key);
    bool haslooped = false;

    while (table[index].state != EMPTY && table[index].state != DELETED) {
        if (index == TABLE_SIZE - 1) {
            index = 0;
            haslooped = true;
        }
        else {
            index++;
        }

        if (index == hash(key) && haslooped) {
            return false;
        }
    }
    table[index].key = key;
    table[index].value = value;
    table[index].state = FILLED;


  return true;
}

int hash_table::find(int key) const
{
    int index = -1;

    index = hash(key);
    bool haslooped = false;

    while (table[index].state != EMPTY && table[index].key != key)
      {
        if (index == TABLE_SIZE - 1)
          {
            index = 0;
            haslooped = true;
          }
        else
          {
            index++;
          }
```

```
        if (index == hash(key) && haslooped)
        {
            return false;
        }
    }
    if (table[index].state == EMPTY)
    {
        return -1;
    }
    else
    {
        return index;
    }
}

bool hash_table::update(int key, const string& value)
{
    int index = find(key);

    if (index == -1)
    {
        return false;
    }
    else
    {
        table[index].value = value;
        return true;
    }
}

bool hash_table::erase(int key)
{
    int index = find(key);

    if (index == -1)
    {
        return false;
    }
    else
    {
        table[index].state = DELETED;
        return true;
    }
}
```

```
ostream& operator<<(ostream& os, const hash_table& obj)
{
    os << "Index  Key    Value\n";
    os << "===========================================================\n";

    for (int i = 0; i < obj.TABLE_SIZE; i++)
    {
        os << setfill(' ') << '[' << setw(2) << right << i << "]  ";

        if (obj.table[i].state == EMPTY)
        os << "EMPTY";
        else if (obj.table[i].state == DELETED)
        os << "DELETED";
        else
        os << setfill('0') << right << setw(4) << obj.table[i].key
        << "   " << setfill(' ') << left << obj.table[i].value;

        os << endl;
    }

    return os;
}
```

Reverse a singly linked list

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

```
void mylist::reverse()
{
    // Temporarys for current, next, and previous
    node* curr = l_front, *prev = nullptr, *next = nullptr;

    while (curr != nullptr) {
        next = curr->next; // assign next to the next node
        curr->next = prev; // Reverse the "arrow" of the current node
        prev = curr; // Advance previous
        curr = next; // Advance current
    }

    l_front = prev; // Assign head of list to new front (at the end of the loop prev will be the top node)
}
```

Sorted Singly-linked list: Insert, remove, remove all

```
void sorted_list::insert(int value)
{
    // Create a new node with value
    node* newnode = new node(value, nullptr);

    // Create traversal and lagging pointers
    node* ptr = l_front, *trail = nullptr;

    // Traverse list, either we get to the end, or we find
    // the correct slot to put the new node
    while (ptr != nullptr && value > ptr->value) {
        // Havent found a spot yet, push pointers forward
        trail = ptr;
        ptr = ptr->next;
    }

    // Value was never greater than any elements, we can insert at the start
    if (trail == nullptr) {
        newnode->next = l_front;
        l_front = newnode;
    // Else, trail will be at the slot we need to insert, and ptr will the node after,
    } else {
        newnode->next = trail->next; // Have newnode next point to node after trail
        trail->next = newnode; // Have trail->next point to newnode
    }
    ++l_size; // Increment the size
}
```

```cpp
void sorted_list::remove_first(int value) {
    node* ptr = l_front;
    node* trail = nullptr;

    while (ptr != nullptr) {
        if (ptr->value == value) {
            if (trail == nullptr) {
                // The node to be removed is the first node
                l_front = ptr->next; // Move the head to the next node
            } else {
                // The node to be removed is not the first node
                trail->next = ptr->next; // Bypass the node to be removed
            }
            delete ptr; // Delete the node to avoid memory leak
            --l_size;   // Decrement the size of the list
            break;      // Exit the loop after removing the first occurrence
        } else {
            trail = ptr;      // Move trail to the current node
            ptr = ptr->next;  // Move to the next node
        }
    }
}
```

```cpp
void sorted_list::remove(int value)
{
    node* ptr = l_front, *trail = nullptr, *temp;

    while (ptr != nullptr) {
        // Check if the current node contains the value to be removed
        if (ptr->value == value) {
            temp = ptr;
            if (trail == nullptr) {
                // Removing the first element
                l_front = ptr->next;
                ptr = l_front;  // Move ptr to the next node in the list
            } else {
                // Link previous node to the next node, skipping the current node
                trail->next = ptr->next;
                ptr = ptr->next;  // Move ptr to the next node in the list
            }
            delete temp;  // Delete the current node
            l_size--;     // Decrement the size of the list
        } else {
            // Move to the next node if the current node does not contain the value to be removed
            trail = ptr;
            ptr = ptr->next;
        }
    }
}
```

Doubly-linked list as a template class

```
#ifndef MYLIST_H
#define MYLIST_H

#include <iostream>
#include <stdexcept>

template<typename T>
struct node {
    T value = T{};
    node<T>* next = nullptr;
    node<T>* prev = nullptr;

    node() = default;
    node(T value, node<T>* next, node<T>* prev) : value(value), next(next), prev(prev) {}
};

template<typename T>
class mylist;

template<typename T>
std::ostream& operator<<(std::ostream& os, const mylist<T>& obj);

template<class T>
class mylist {
    node<T>* m_front = nullptr;
    node<T>* m_back = nullptr;
    size_t m_size = 0;
public:
    mylist() = default;
    ~mylist();
    mylist(const mylist<T>& other);
    mylist& operator=(const mylist<T>& x);
    void clear();
    size_t size() const;
    bool empty() const;
    const T& front() const;
    T& front();
    const T& back() const;
    T& back();
    void push_front(const T& value);
    void push_back(const T& value);
    void pop_front();
    void pop_back();
```

```cpp
    bool operator==(const mylist<T>& rhs) const;
    bool operator<(const mylist<T>& rhs) const;
    void clone(const mylist<T>& other);

    // Friend function
    friend std::ostream& operator<< <T>(std::ostream& os, const mylist<T>& obj);
};


template <typename T>
mylist<T>::~mylist() {
    this->clear();
}

template <typename T>
mylist<T>::mylist(const mylist<T>& other) {
    this->clone(other);
}

template <typename T>
mylist<T>& mylist<T>::operator=(const mylist<T>& x) {

    // Check if they are the same object. In this case, we don't have to do anything
    if (this == &x) {
        return *this;
    }

    // Clone and return the object
    this->clone(x);
    return *this;
}

template <typename T>
void mylist<T>::clear() {

    // Create a node to point to the node that needs to be deleted
    // We also create a next node to point to the next node in the traversal
    node<T>* del = this->m_front,
           * next = nullptr;

    // Traverse the list
    while (del != nullptr) {
        next = del->next; // Forward the next pointer
        delete del; // Delete the current node
        del = next; // Advance
    }
```

```
    // Reset list to default state
    m_front = nullptr;
    m_back = nullptr;
    m_size = 0;
}

template <typename T>
size_t mylist<T>::size() const {
    return this->m_size;
}

template<typename T>
bool mylist<T>::empty() const {
    return !(this->m_size);
}

template <typename T>
const T& mylist<T>::front() const {

    // Check if the list is empty
    if (this->empty()) {
        throw std::underflow_error("underflow exception on call to front()");
    }
    // Return the reference
    return this->m_front->value;
}

template<typename T>
T& mylist<T>::front() {

    // Check if the list is empty
    if (this->empty()) {
        throw std::underflow_error("underflow exception on call to front()");
    }

    // Return the reference
    return this->m_front->value;
}


template <typename T>
const T& mylist<T>::back() const {
    if (this->empty()) {
        throw std::underflow_error("underflow exception on call to back()");
    }

    return this->m_back->value;
}
```

```cpp
template <typename T>
T& mylist<T>::back() {
    if (this->empty()) {
        throw std::underflow_error("underflow exception on call to back()");
    }
    return this->m_back->value;
}


template <typename T>
void mylist<T>::push_front(const T& value) {

    // Create a new node
    node<T>* newnode = new node<T>(value, nullptr, nullptr);

    // If the list is empty, we make this new node the front and back
    if (this->empty()) {
        m_back = newnode;

    // If the list is not empty, make this new node point to the current front
    } else {
        newnode->next = m_front;
        m_front->prev = newnode; // make the current front point back to the newnode
    }
    m_front = newnode;
    ++m_size;
}


template <typename T>
void mylist<T>::push_back(const T& value) {

    // Create a newnode
    node<T>* newnode = new node<T>(value, nullptr, nullptr);

    // If the list is empty, we make this new node the front and back
    if (this->empty()) {
        m_front = newnode;

    // We do the opposite of push_front here
    } else {
        newnode->prev = m_back;
        m_back->next = newnode;
    }
    m_back = newnode;
    ++m_size;
}
```

```
template <typename T>
void mylist<T>::pop_front() {

    // Throw an error if the list is empty
    if (this->empty()) {
        throw(std::underflow_error("underflow exception on call to pop_front()"));

    // Else, we pop the front node
    } else {

        // Create pointer to front node
        node<T>* del = m_front;

        // Advance front node
        m_front = m_front->next;

        // Ensure we are assigning states to a valid node
        if (m_front != nullptr) {
            m_front->prev = nullptr;
        }

        // Delete the node
        delete del;
    }
    --m_size;
}


template <typename T>
void mylist<T>::pop_back() {

    // Throw an error if the list is empty
    if (this->empty()) {
        throw(std::underflow_error("underflow exception on call to pop_back()"));

    // Else, we pop the back node
    } else {
        node<T>* del = m_back;

        m_back = m_back->prev;

        if (m_back != nullptr) {
            m_back->next = nullptr;
        }
        delete del;
    }
    --m_size;
}
```

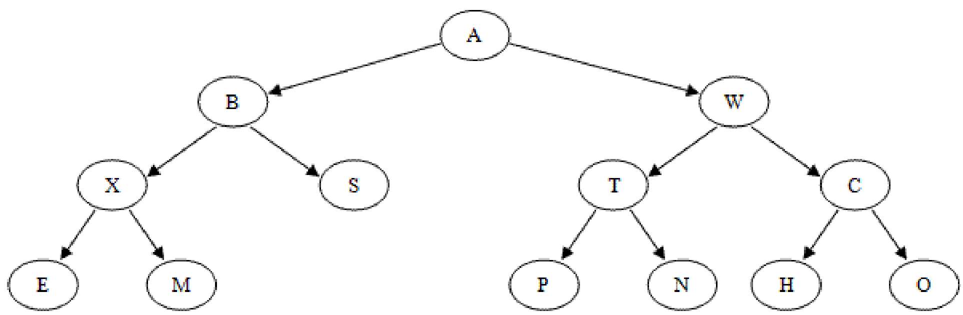
```
template<typename T>
bool mylist<T>::operator==(const mylist<T>& rhs) const {
    if (this->m_size != rhs.m_size) {
        return false;
    }
    node<T>* l_curr, *r_curr;
    l_curr = this->m_front,
    r_curr = rhs.m_front;

    while (l_curr != nullptr) {

        if (l_curr->value != r_curr->value) {
            return false;
        }
        l_curr = l_curr->next,
        r_curr = r_curr->next;
    }
    return true;
}

template <typename T>
bool mylist<T>::operator<(const mylist<T>& rhs) const {
    size_t smallest_size = (this->size() >= rhs.size() ? rhs.size() : this->size());

    node<T>* l_start, *r_start;
    l_start = this->m_front,
    r_start = rhs.m_front;

    for (size_t i = 0; i<smallest_size; ++i) {
        if (l_start->value < r_start->value ) {
            return true;
        } else if (l_start->value > r_start->value) {
            return false;
        } else {
            l_start = l_start->next;
            r_start = r_start->next;
        }
    }
    return (this->size() < rhs.m_size ? true : false);
}
```

```cpp
template <typename T>
void mylist<T>::clone(const mylist<T>& other) {

    // Clear the list
    this->clear();

    node<T>* curr = other.m_back;
    while (curr != nullptr) {
        this->push_front(curr->value);
        curr = curr->prev;
    }
}

template <typename T>
std::ostream& operator<<(std::ostream& os, const mylist<T>& obj)
↪  {

    node<T>* curr = obj.m_front;

    // Traverse the list, outputing values
    while (curr != nullptr) {
        os << curr->value << " ";
        curr = curr->next;
    }

    // Return the stream
    return os;
}
#endif
```

Binary Trees

Concept 15: A binary tree consists of a finite set of nodes that is either empty, or consists of one specially designated node called the root of the binary tree, and the elements of two disjoint binary trees called the left subtree and right subtree of the root.

Note that the definition above is recursive: we have defined a binary tree in terms of binary trees. This is appropriate since recursion is an innate characteristic of tree structures.



23.1 Binary Tree Terminology

Tree terminology is generally derived from the terminology of family trees (specifically, the type of family tree called a lineal chart).

- Each root is said to be the parent of the roots of its subtrees.
- Two nodes with the same parent are said to be siblings; they are the children of their parent.
- The root node has no parent.
- A great deal of tree processing takes advantage of the relationship between a parent and its children, and we commonly say a directed edge (or simply an edge) extends from a parent to its children. Thus edges connect a root with the roots of each subtree. An undirected edge extends in both directions between a parent and a child.
- Grandparent and grandchild relations can be defined in a similar manner; we could also extend this terminology further if we wished (designating nodes as cousins, as an uncle or aunt, etc.).

- The number of subtrees of a node is called the degree of the node. In a binary tree, all nodes have degree 0, 1, or 2.
- A node of degree zero is called a terminal node or leaf node.
- A non-leaf node is often called a branch node.
- The degree of a tree is the maximum degree of a node in the tree. A binary tree is degree 2.

- A directed path from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leqslant i < k$. An undirected path is a similar sequence of undirected edges. The length of this path is the number of edges on the path, namely $k - 1$ (i.e., the number of nodes - 1). There is a path of length zero from every node to itself. Notice that in a binary tree, there is exactly one path from the root to each node.
- The level or depth of a node with respect to a tree is defined recursively: the level of the root is zero; and the level of any other node is one higher than that of its parent. Or to put it another way, the level or depth of a node n_i is the length of the unique path from the root to n_i.
- The height of n_i is the length of the longest path from n_i to a leaf. Thus, all leaves in the tree are at height 0.
- The height of a tree is equal to the height of the root. The depth of a tree is equal to the level or depth of the deepest leaf; this is always equal to the height of the tree.
- If there is a directed path from n_1 to n_2, then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1.

23.2 Special types

There are a few special forms of binary tree worth mentioning.

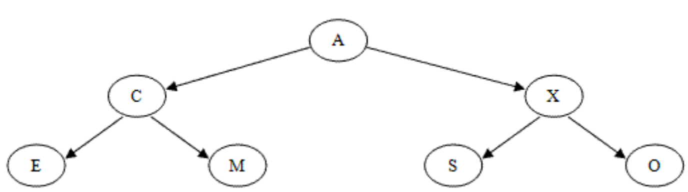
23.2.1 Strictly (full) binary tree

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one. A strictly binary tree with N leaves always contains $2N - 1$ nodes.

23.3 Complete binary tree

A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.

The total number of nodes in a complete binary tree of depth d equals $2^{d+1} - 1$. Since all leaves in such a tree are at level d, the tree contains 2^d leaves and, therefore, $2^d - 1$ internal nodes.



23.4 Almost complete binary tree

A binary tree of depth d is an almost complete binary tree if:

- Each leaf in the tree is either at level d or at level $d-1$.
- For any node n_d in the tree with a right descendant at level d, all the left descendants of n_d that are leaves are also at level d.

An almost complete strictly binary tree with N leaves has $2N-1$ nodes (as does any other strictly binary tree). An almost complete binary tree with N leaves that is not strictly binary has $2N$ nodes. There are two distinct almost complete binary trees with N leaves, one of which is strictly binary and one of which is not.

There is only a single almost complete binary tree with N nodes. This tree is strictly binary if and only if N is odd.

Some texts do not make a distinction between complete and almost complete binary trees, considering both to be complete binary trees.

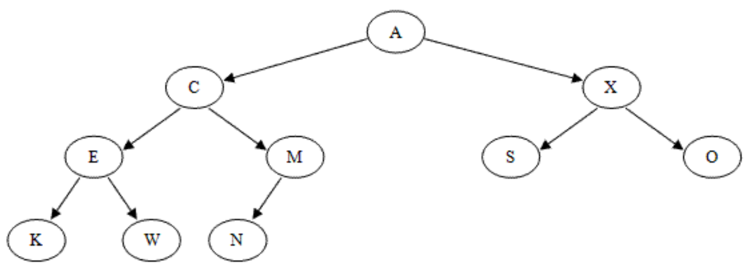


Figure 3

23.5 Mathematical formulae

23.5.1 Complete binary tree

1. **Number of Nodes at Level** i:
$$N_i = 2^i$$
2. **Total Number of Nodes**:
$$N = 2^0 + 2^1 + 2^2 + \ldots + 2^d = 2^{d+1} - 1$$
3. **Number of Leaf Nodes**:
$$L = 2^d$$
4. **Maximum Number of Nodes for Given Depth**:
$$N_{\max} = 2^{d+1} - 1$$

5. **Depth from Total Nodes**:

$$d = \lfloor \log_2(N+1) \rfloor - 1$$

6. **Relationship Between Number of Leaf Nodes (L) and Total Number of Nodes (N)**:

$$N = 2L - 1$$

7. **Height of the Tree**:

$$h = d$$

23.5.2 Strictly (full) binary tree

1. **Relationship Between the Number of Leaf Nodes (L) and Internal Nodes (I)**:

$$L = I + 1$$

2. **Total Number of Nodes (N)**: Given L as the number of leaf nodes and I as the number of internal nodes, the total number of nodes can be expressed as:

$$N = 2L - 1$$
$$N = 2I + 1.$$

3. **Depth of the Tree (d)**: The depth of a strictly binary tree, considering the total number of nodes:

$$d \geqslant \log_2(N+1) - 1$$

This becomes an equality when N is the total number of nodes in a strictly binary tree.

4. **Finding L or I from N**: Given the total number of nodes (N), the number of leaf nodes (L) and internal nodes (I) can be calculated as:

$$L = \frac{N+1}{2}$$

$$I = \frac{N-1}{2}$$

5. **Minimum Number of Leaf Nodes ($L_{\mathbf{min}}$)**: In the context of the initial request, this item doesn't directly apply to strictly binary trees as described here since all nodes in such trees have either two or no children. However, for completeness in a general context where this might be considered:

$$L_{\min} = 2^{d-1}$$

Note: The above formula for $L_{\min}$ generally applies to complete binary trees, not strictly binary trees, indicating the minimum number of leaf nodes at the last level of a complete binary tree. In strictly binary trees, every non-leaf node has exactly two children, making the structure and formula application different.

23.5.3 Almost complete

- An almost complete binary tree with N leaves that is not strictly binary has $2N$ nodes
- An almost complete binary tree with N nodes is strictly binary if and only if N is odd

23.6 Properties of binary trees

1. **Parent of a Node**: For 0-based indexing, the formula to find the parent node is given by
$$\text{Parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$$
2. **Left Child of a Node**: For 0-based indexing:
$$\text{Left Child}(i) = 2i + 1$$
3. **Right Child of a Node**: For 0-based indexing:
$$\text{Right Child}(i) = 2i + 2$$
4. **Depth of the Node**: The depth of a node in a binary tree is determined by the number of edges from the node to the tree's root. While there's no simple arithmetic formula for finding a node's depth in all cases, for complete binary trees represented in an array, the depth can often be inferred from the node's index.
5. **Height of the Tree**: The height of a binary tree is the number of edges in the longest path from the root to a leaf. For a perfectly balanced binary tree, the height can be related to the total number of nodes (N) as:
$$h = \lfloor \log_2(N+1) \rfloor$$
For general binary trees, the height must be computed by traversing the tree.

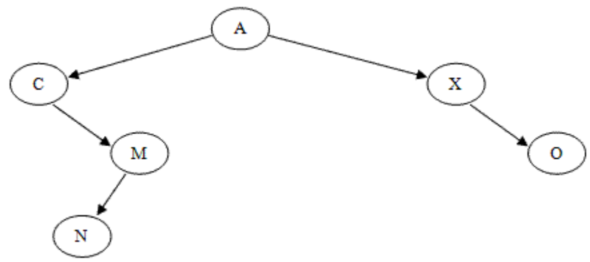
23.7 Representing binary trees in memory

For a complete or almost complete binary tree, storing the binary tree as an array may be a good choice.

One way to do this is to store the root of the tree 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. For example, the almost complete binary tree shown in Figure 3 can be stored in an array like this:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	C	X	E	M	S	O	K	W	N

However, if this scheme is used to store a binary tree that is not complete or almost complete, we can end up with a great deal of wasted space in the array. For example, the following binary tree

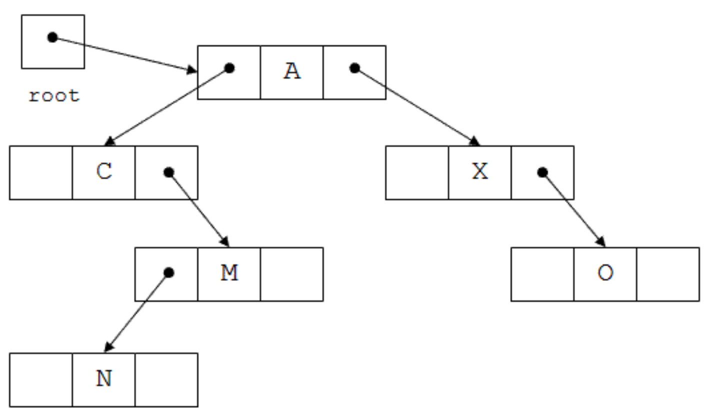


Would be stored using this technique like this:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	C	X		M		O			N

23.7.1 Linked representation

If a binary tree is not complete or almost complete, a better choice for storing it is to use a linked representation similar to the linked list structures covered earlier in the semester:



Each tree node has two pointers (usually named left and right). The tree class has a pointer to the root node of the tree (labeled root in the diagram above).

Any pointer in the tree structure that does not point to a node will normally contain the value nullptr. A linked tree with N nodes will always contain N + 1 null links.

23.8 Binary Tree traversals

Concept 16: Traversal is a common operation performed on data structures. It is the process in which each and every element present in a data structure is ”visited” (or accessed) at least once. This may be done to display all of the elements or to perform an operation on all of the elements.

For example, to traverse a singly-linked list, we start with the first (front) node in the list and proceed forward through the list by following the next pointer stored in each node until we reach the end of the list (signified by a next pointer with the special value nullptr). A doubly-linked list can also be traversed in reverse order, starting at the last (back) node in the list and proceeding backwards down the list by following the prev pointer stored in each node, stopping when we reach the beginning of the list (signified by a prev pointer with the special value nullptr). Arrays can likewise easily be traversed either forward or backward, simply by starting with either the first or last element and then incrementing or decrementing a subscript or pointer to the array element.

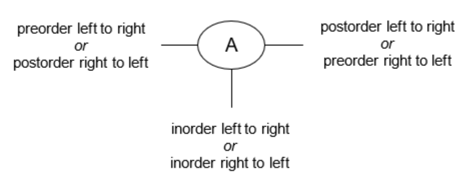
On the other hand, binary trees can be traversed in multiple ways. These notes describe four different traversals:

- preorder
- inorder
- postorder
- level order

23.8.1 The ”tick trick”

Concept 17: This is a handy trick for figuring out by hand the order in which a binary tree's nodes will be ”visited” for the preorder, inorder, and postorder traversals.

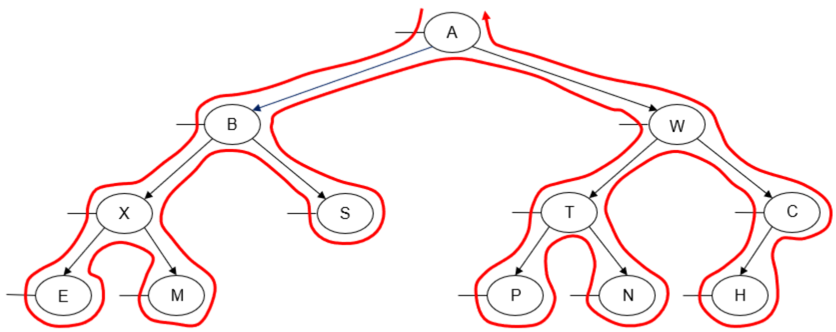
1. Draw a line or tick mark on one of the sides or the bottom of each node in the tree. Where you draw the mark depends on which traversal you are attempting to perform, as shown in the diagram below:
2. Draw a line or tick mark on one of the sides or the bottom of each node in the tree. Where you draw the mark depends on which traversal you are attempting to perform, as shown in the diagram below:



The point at which the path you've drawn around the binary tree intersects the tick mark is the point at which that node will be ”visited” during the traversal.

23.8.2 Preorder traversal

In a preorder traversal of a binary tree, we "visit" a node and then traverse both of its subtrees. Usually, we traverse the node's left subtree first and then traverse the node's right subtree.



Printing the value of each node as we "visit" it, we get the following output:

A B X E M S W T P N C H

23.8.3 Preorder traversal recursive algorithm

```
procedure preorder(p : pointer to a tree node)
    if p != nullptr
        Visit the node pointed to by p
        preorder(p->left)
        preorder(p->right)
    end if
end procedure
```

Note:-

On the initial call to the preorder() procedure, we pass it the root of the binary tree. To convert the pseudocode above to a right-to-left traversal, just swap left and right so that the right subtree is traversed before the left subtree.

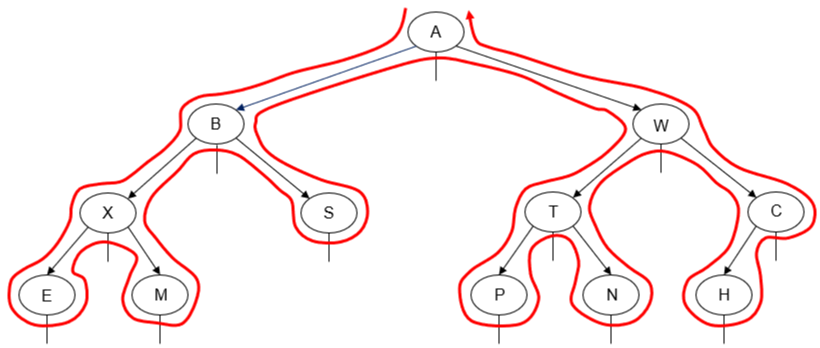
23.8.4 Preorder traversal iterative algorithm

Preorder traversal can also be performed using a non-recursive or iterative algorithm. In order to backtrack up the tree from a node to its parent, we use a stack.

```
procedure iterative_preorder()
    // root : pointer to the root node of the tree (nullptr if tree is empty)
    // p    : pointer to a tree node
    // s    : a stack of pointers to tree nodes

    // Start at the root of the tree.
    p ← root

    // While p is not nullptr or the stack is not empty...
    while p != nullptr or s is not empty

        // Go all the way to the left.
        while p != nullptr
            Visit the node pointed to by p

            // Place a pointer to the node on the stack before
            // traversing the node's left subtree.
            s.push(p)
            p ← p->left
        end while

        // p must be nullptr at this point, so backtrack one level.
        p ← s.top()
        s.pop()

        // We have visited the node and its left subtree, so
        // now we traverse the node's right subtree.
        p ← p->right

    end while
end procedure
```

23.8.5 Inorder Traversal

In an inorder traversal of a binary tree, we traverse one subtree of a node, then ”visit” the node, and then traverse its other subtree. Usually, we traverse the node’s left subtree first and then traverse the node’s right subtree.



Printing the value of each node as we ”visit” it, we get the following output:

E X M B S A P T N W H C

23.8.6 Recursive Inorder Traversal Algorithm

```
procedure inorder(p : pointer to a tree node)
    if p != nullptr
        inorder(p->left)
        Visit the node pointed to by p
        inorder(p->right)
    end if
end procedure
```

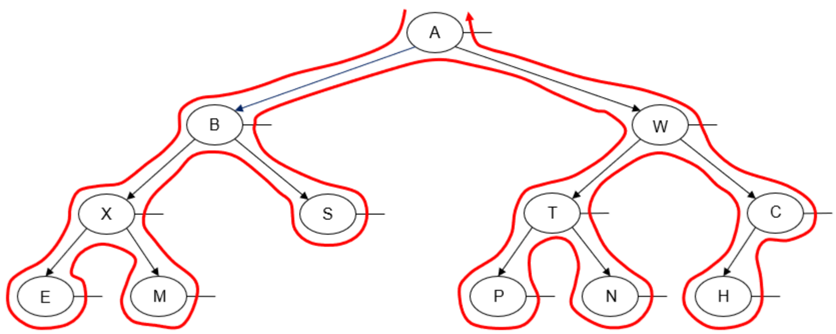
23.8.7 Iterative Inorder Traversal Algorithm

```
procedure iterative_inorder()
    // root : pointer to the root node of the tree (nullptr if tree is empty)
    // p    : pointer to a tree node
    // s    : a stack of pointers to tree nodes

    // Start at the root of the tree.
    p ← root

    // While p is not nullptr or the stack is not empty...
    while p != nullptr or s is not empty

        // Go all the way to the left.
        while p != nullptr

            // Place a pointer to the node on the stack before
            // traversing the node's left subtree.
            s.push(p)
            p ← p->left
        end while

        // p must be nullptr at this point, so backtrack one level.
        p ← s.top()
        s.pop()

        // We have visited this node's left subtree, so now we
        // visit the node.
        Visit the node pointed to by p

        // We have visited the node and its left subtree, so
        // now we traverse the node's right subtree.
        p ← p->right

    end while
end procedure
```

23.8.8 Postorder Traversal

In a postorder traversal of a binary tree, we traverse both subtrees of a node, then "visit" the node. Usually we traverse the node's left subtree first and then traverse the node's right subtree.

Here's an example of a left-to-right postorder traversal of a binary tree:



Printing the value of each node as we "visit" it, we get the following output:

E M X S B P N T H C W A

Note:-

the left-to-right postorder traversal is a mirror image of the right-to-left preorder traversal, while the right-to-left postorder traversal is a mirror image of the left-to-right preorder traversal.

23.8.9 Recursive Postorder Traversal Algorithm

```
procedure postorder(p : pointer to a tree node)
    if p != nullptr
        postorder(p >left)
        postorder(p->right)
        Visit the node pointed to by p
    end if
end procedure
```

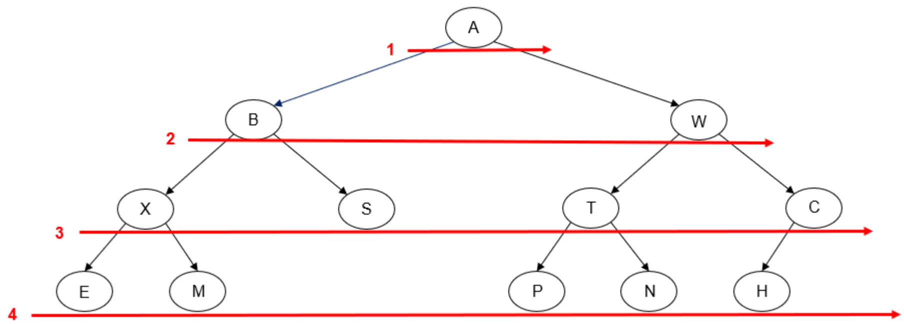
23.8.10 Iterative Postorder Traversal Algorithm

Postorder traversal can also be performed using a non-recursive or iterative algorithm. This is a trickier algorithm to write than the iterative preorder or inorder traversals, since we will need to backtrack from a node to its parent twice. Some sources solve this problem (poorly, in my opinion) by using two different stacks. Donald Knuth's The Art of Computer Programming has a more efficient version of the algorithm that maintains an extra pointer to the node that was last visited and uses it to distinguish between backtracking to a node after traversing its left subtree versus backtracking to a node after traversing its right subtree.

```
procedure iterative_postorder()
    // root         : pointer to the root node of the tree (nullptr if tree is empty)
    // p            : pointer to a tree node
    // last_visited : pointer to the last tree node visited
    // s            : a stack of pointers to tree nodes

    // Start at the root of the tree.
    last_visited ← nullptr
    p ← root

    while p != nullptr and last_visited != root

        // Go all the way to the left.
        while p != nullptr and p != last_visited

            // Place a pointer to the node on the stack before
            // traversing the node's left subtree.
            s.push(p)
            p ← p->left
        end while

        // p must be nullptr at this point, so backtrack one
        // level.
        p ← s.top()
        s.pop()

        // If this node has no right child or we've already traversed
        // its right subtree...
        if p->right == nullptr or p->right == last_visited

            Visit the node pointed to by p

            // Mark this node as the last visited.
            last_visited ← p
        else
            // Otherwise, traverse the node's right subtree.
            s.push(p)
            p ← p->right
        end if
    end while
end procedure
```

23.8.11 Level Order Traversal

In a level order traversal of a binary tree, we traverse all of the tree nodes on level 0, then all of the nodes on level 1, etc. The ”tick trick” does not work for this traversal, but there’s no real need for it, since the order the nodes will be traversed is easy to determine by hand.



Printing the value of each node as we ”visit” it, we get the following output:

A B W X S T C E M P N H

23.8.12 Iterative Level Order Traversal Algorithm

Level order traversal can be performed using a non-recursive or iterative algorithm. As a given level is traversed, a queue is used to store pointers to the nodes on the next level.

```
procedure iterative_level_order()
    // root : pointer to the root node of the tree (nullptr if tree is empty)
    // p    : pointer to a tree node
    // q    : a queue of pointers to tree nodes

    // If tree is empty, return.
    if root == nullptr
        return
    end if

    q.push(root)

    while q is not empty

        // Remove front item in the queue and visit it.
        p ← q.front()
        q.pop()
        Visit the node pointed to by p

        // Insert left child of p into queue.
        if p->left != nullptr
            q.push(p->left)
        end if

        // Insert right child of p into queue.
        if p->right != nullptr
            q.push(p->right)
        end if
    end while
end procedure
```

23.8.13 Recursive Level Order Traversal Pseudocode

```
    procedure level_order()
    // root : pointer to the root node of the tree (nullptr if
↪   tree is empty)
    // h    : computed height of the tree (i.e., number of levels
    // i    : loop counter

    h ← height(root);

    i ← 1
    while i <= h
        print_level(root, i)
        i ← i + 1
     end while
end procedure

procedure print_level(p : pointer to a tree node, level : level
↪   number)
    if p == nullptr
        return

    if level == 1
        Visit the node pointed to by p
    else if level > 1
        print_level(p->left, level-1)
        print_level(p->right, level-1)
    end if
end procedure

procedure height(p : pointer to a tree node)
    // l_height : computed height of node's left subtree
    // r_height : computed height of node's right subtree

    if p == nullptr
        return 0
    end if

    l_height ← height(p->left)
    r_height ← height(p->right)

    if l_height > r_height
        return l_height + 1
    else
        return r_height + 1
    end if
end procedure
```

Binary search tree

Concept 18: A binary search tree, sometimes called an ordered or sorted binary tree is a binary tree in which nodes are ordered in the following way:

1. each node contains a key (and optionally also an associated value)
2. the key in each node must be greater than or equal to any key stored in its left subtree, and less than or equal to any key stored in its right subtree. Depending on the application, duplicate keys may or may not be allowed.

Performing a left-to-right inorder traversal of a binary search tree will ”visit” the nodes in ascending key order, while performing a right-to-left inorder traversal will ”visit” the nodes in descending key order.

Binary search trees are a common choice for implementing several abstract data types, including Ordered Set, Ordered Multi-Set, Ordered Map, and Ordered Multi-Map. These ADTs have three main operations:

- Insertion of elements
- Deletion of elements
- Find / lookup an element

24.1 Binary Search Tree Insertion

Insertion into a binary search tree can be coded either iteratively or recursively.

Note:-

If the tree is empty, the new element is inserted as the root node of the tree. Otherwise, the key of the new element is compared to the key of the root node to determine whether it must be inserted in the root's left subtree or its right subtree. This process is repeated until a null link is found or we find a key equal to the key we are trying to insert (if duplicate keys are disallowed). The new tree node is always inserted as a leaf node.

24.1.1 Iterative Insertion into a Binary Search Tree Pseudocode

```
procedure insert(key : a key to insert, value : a value to insert)
    // root     : pointer to the root node of the tree (nullptr if tree is empty)
    // t_size   : tree size
    // p        : pointer to a tree node
    // parent   : pointer to the parent node of p (nullptr if p points to the root node)
    // new_node : pointer used to create a new tree node

    // Start at the root of the tree.
    p ← root
    parent ← nullptr

    // Search the tree for a null link or a duplicate key (if duplicates are disallowed).
    while p != nullptr and key != p->key
        parent ← p
        if key < p->key
            p ← p->left
        else
            p ← p->right
        end if
    end while

    // If duplicates are disallowed, signal that insertion has failed.
    if p != nullptr
        return false
    end if

    // Otherwise, create a tree node and insert it as a new leaf node.
    Create a new tree node new_node to contain key and value

    if parent == nullptr
        root ← new_node
    else
        if new_node->key < parent->key
            parent->left ← new_node
        else
            parent->right ← new_node
        end if
    end if

    t_size ← t_size + 1

    // If duplicates are disallowed, signal that insertion has succeeded.
    return true

end procedure
```

If keys are inserted into a binary search tree in sorted order, they will always end up being inserted in the same subtree. The result is referred to as a degenerate binary search tree and is effectively a linked list. This has a negative impact on the complexity of the binary search tree operations (see Complexity below). One way to prevent this problem is with a self-balancing binary search tree such as an AVL tree or a red-black tree. Both data structures are outside the scope of this course.

24.2 Binary Search Tree Deletion

Deletion of a node with a specified key from a binary search tree can also be coded either iteratively or recursively. Pseudocode for an iterative version of the algorithm is shown below.

24.2.1 Iterative Deletion from a Binary Search Tree Pseudocode

```
procedure remove(key : key to remove from the tree)
    // root           : pointer to the root of the binary search
↪   tree
    // t_size         : tree size
    // p              : pointer to the node to delete from the
↪   tree
    // parent         : pointer to the parent node of the node
↪   to delete from the tree (or
    //                  nullptr if deleting the root node)
    // replace        : pointer to node that will replace the
↪   deleted node
    // replace_parent : pointer to parent of node that will
↪   replace the deleted node

    // Start at the root of the tree and search for the key to
↪   delete.
    p ← root
    parent ← nullptr
    while p != nullptr and key != p->key
        parent ← p
        if key < p->key
            p ← p->left
        else
            p ← p->right
        end if
    end while

    // If the node to delete was not found, signal failure.
    if p == nullptr
        return false
    end if

    if p->left == nullptr
        // Case 1a: p has no children. Replace p with its right
↪   child
        // (which is nullptr).
        //   - or -
        // Case 1b: p has no left child but has a right child.
↪   Replace
        // p with its right child.
        replace ← p->right
    else if p->right == nullptr
        // Case 2: p has a left child but no right child.
↪   Replace p
        // with its left child.
        replace ← p->left
    else
```

```
        // Case 3: p has two children. Replace p with its inorder predecessor.

        // Go left...
        replace_parent ← p
        replace ← p->left

        // ...then all the way to the right.
        while replace->right != nullptr
            replace_parent ← replace
            replace ← replace->right
        end while

        // If we were able to go to the right, make the replacement node's
        // left child the right child of its parent. Then make the left child
        // of p the replacement's left child.
        if replace_parent != p
            replace_parent->right ← replace->left
            replace->left ← p->left
        end if

        // Make the right child of p the replacement's right child.
        replace->right ← p->right
    end if

    // Connect replacement node to the parent node of p (or the root if p has no parent).
    if parent == nullptr
        root ← replace
    else
        if p->key < parent->key
            parent->left ← replace
        else
            parent->right ← replace
        end if
    end if

    // Delete the node, decrement the tree size, and signal success.
    Delete the node pointed to by p
    t_size ← t_size - 1

    return true
end procedure
```

24.2.2 Deletion cases

1. Node to delete has no left child

When a node we want to delete has no left child, we replace the deleted node with its right child. If the node to delete also has no right child, it will be replaced with nullptr.

if the node we want to delete does have a right child, the deleted node is replaced with that right child.

2. Node to delete has no right child

When a node we want to delete has no right child, we replace the deleted node with its left child.

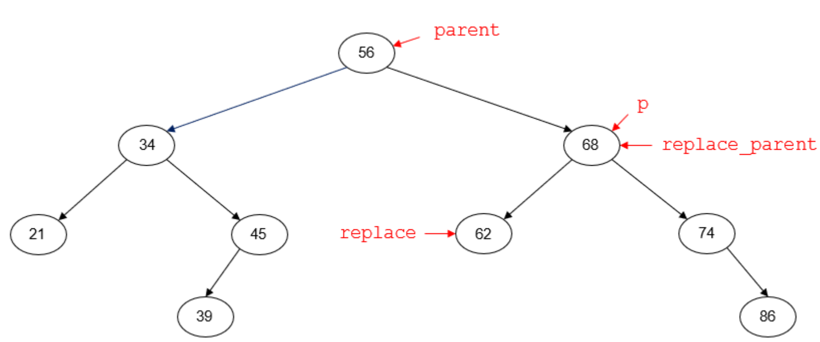
3. Node to delete has two children

When a node to delete has two children, we replace the deleted node with its inorder predecessor. (Replacing the node with its inorder successor would also work, but we have to pick one or the other when we code the algorithm.) To find the inorder predecessor of a node with two children, we go to its left and then all the way to the right.

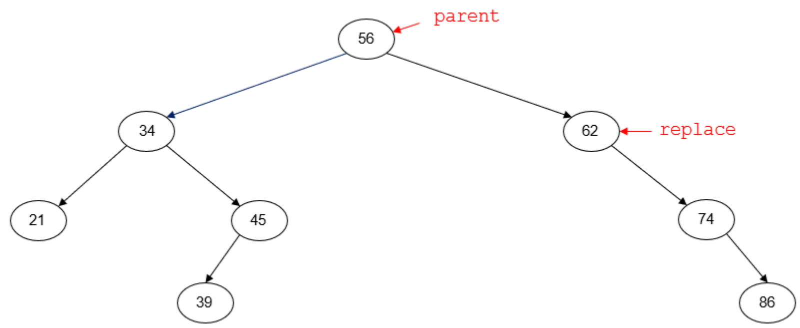
Sometimes after going left we may be unable to go right, because the left child of p has no right child. In that case, the left child of p is its inorder predecessor.

For example, suppose that we want to delete the node with key 68. Prior to deleting the node, the tree will look like the following diagram. p points to the node to be deleted (68). parent points to the parent node of p (56). replace points to the left child of p (62), which is its inorder predecessor. replace_parent points to the same node as p (68), which tells us that after going left we were unable to go to the right.

We know in this situation that the node pointed to by replace is the left child of p, so we don't need to worry about dealing with that. The node pointed to by p also has a right child. Since the node pointed to by replace currently has no right child of its own (remember, we were unable to go to the right), the right child of the node pointed to by p can become its new right child.



After deletion, the tree will look like this:

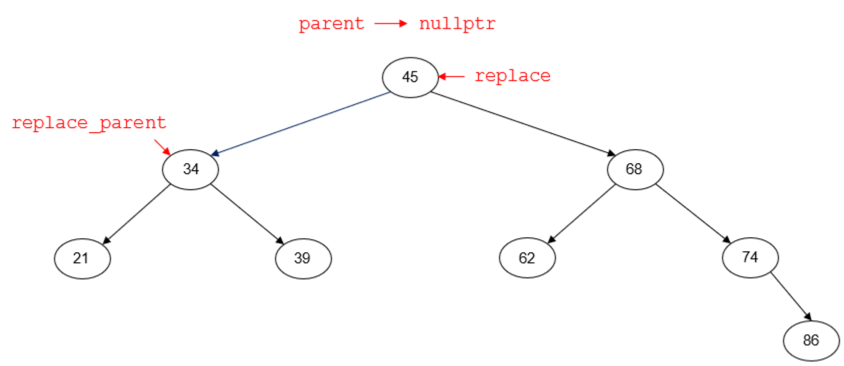


If the left child of p has a right child, we need to continue going to the right until we reach a node with no right child. That node will be the inorder predecessor of p.

For example, suppose that we want to delete the node with key 56. Prior to deleting the node, the tree will look like the following diagram. p points to the node to be deleted (56). parent is nullptr; the node with key 56 is the root node of the tree and has no parent node. replace points to the inorder predecessor of p (45). replace_parent points to the parent node of replace (34).

In this situation, we have a couple more links that need to be set. The node pointed to by replace has no right child, but it might have a left child. That left child will become the right child of replace_parent, taking the place of the node pointed to by replace.

The node pointed to by p definitely has both a left child and a right child - if it didn't, we wouldn't be in the code for this case! Those children need to become the children of the node pointed to by replace.



24.3 Binary Search Tree Find / Lookup

```
procedure find(key : a key for which to search)
    // root   : pointer to the root node of the tree (nullptr if tree is empty)
    // p      : pointer to a tree node

    // Start at the root of the tree.
    p ← root

    // Search the tree for a null link or a matching key.
    while p != nullptr and key != p->key
        if key < p->key
            p ← p->left
        else
            p ← p->right
        end if
    end while

    // p either points to the node with a matching key or is nullptr if
    // the key is not in the tree.
    return p

end procedure
```