Algorithms and Data Structures II

Course Notes

Felipe Balbi

March 21, 2021

Contents

Week 1

Key Concepts

- Determine time and memory consumption of an algorithm described using pseudocode
- Determine the growth function of the running time or memory consumption of an algorithm
- Use Big-O, Omega and Theta notations to describe the running time or memory consumption of an algorithm. Learning objectives:

1.001 What is analysis of algorithms?

Analysis allows us to select the best algorithm to perform a given task.

There are three main aspects we generally use to analyse algorithms:

Correctness whether the algorithm performs the given task according to a given specification

Ease of understanding how difficult is it to understand the algorithm

Resource consumption how much memory and how much CPU time does an algorithm consume

Algorithms who perform a given correctly consuming minimum ammount of resources are better candidates than those requiring more resources.

During this cource, emphasis is given to computational resource consumption of algorithms, that is, the amount of memory, CPU time and, perhaps, bandwidth necessary to complete a computation.

Processing requirements (i.e. CPU time) is measured in terms of the number of operations that must be carried out in order to execute the algorithm. This number is important because with lower number operations, naturally, the algorithm executes faster.

Memory requirements, conversely, are measured in terms of the number of memory units required by the algorithm during its execution. This number is important because we can't compute on data that doesn't fit our memory.

In summary, we learn how to analyse algorithms in terms of its CPU and Memory requirements. Based on such analysis, we will be equipped to select the best algorithm given a specific task.

1.002 What is analysis of algorithms?

Please read paragraph 1 of Section 2.2 (p.23) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.004 How to measure/estimate time and space requirements

Suppose we're given the following pseudocode:

```
1: function F(arrays)
2:     for 1 ≤ j <LENGTH(s) do
3:         key ← s[j]
4:         i ← j − 1
5:         while i ≥ 0 ∧ s[i] > key do
6:             s[i + 1] ← s[i]
7:             i ← i − 1
8:         end while
9:         s[i + 1] ← key
10:    end for
11: end function
```

Now we're asked to say how much time and space algorithm **??** needs to execute. How do we go about answering that question?

One may consider an empirical approach of implementing the algorithm in a specific programming language and run it in a specific computer, then measure its runtime and memory consumption in a specific scenario.

One can also consider a more theoretical approach by making some assumptions about the number of operations for each instruction the CPU executes, multiplying by the time required by each instruction and, with that obtaining an estimate for the runtime. For memory requirements, we could look at all new variables created during the execution of the algorithm.

There are pros and cons for either approach:

Approach	Pros	Cons
Empirical	Real/Exact result	Machine-dependant results
	No Need for calculations	Implementation effort
Theoretical	Universal results	Approximate results
	No implementation effort	Calculations effort

During this course, we work with the theoretical approach. There are three aspects we need to understand very well:

The Machine Model know its characteristics well as they affect the results we can obtain.

Assumptions And Simplifications know where assumptions and simplifications cause a deviation from the real world and why.

Calculations calculations will be necessary. Usually simple additions and multiplications.

1.006 The RAM model

The Random-Access Machine Model is a simplified version of a computer machine.

Because a real machine is a very complex structure, we use a model to simplify our work. The model must be simple and yet complete enough to capture enough details as to be relevant. Figure 1 has a visual representation of the model.

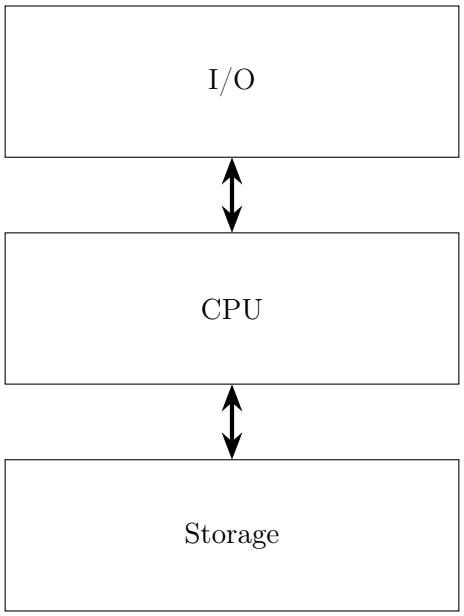


Figure 1: Random-access Machine Model

There are a few assumptions made for this model to work:

Single CPU With a single CPU, all instructions are executed sequentially.

Single Cycle Every simple operation take one time unit (or one cycle) to complete.

Loops/Functions Are Not Simple They are made up of several simple operations.

No Memory Hierarchy Every memory access takes one time unit (or one cycle) to complete. Also we always have exactly as much memory as is needed to run the computation.

We also have one assumption regarding memory consumption:

Simple Variables Uses 1 Memory Position One integer uses 1 memory position while an array of N elements uses N memory positions.

1.007 The Ram Model

Please read pp.23–4 of Section 2.2 from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.009 Counting up time and space units, part 1

We're going the analyse the function shown in listing **??** with the analysis of each line typeset as a comment on that line. In order to get our total, we just add all simple operations together.

```
1: function F1(a, b, c)
2:     max ← a                          ▷ 1 memory read, 1 memory write
3:     if b > max then                  ▷ 1 conditional, 1 comparison, 2 memory reads
4:         max ← b                      ▷ 1 memory read, 1 memory write
5:     end if
6:     if c > max then                  ▷ 1 conditional, 1 comparison, 2 memory reads
7:         max ← c                      ▷ 1 memory read, 1 memory write
8:     end if
9:     return max                       ▷ 1 memory read, 1 return
10: end function
```

Adding up all our memory reads, memory writes, conditionals and conditionals, we get a total of 16 time units. In terms of space, there's only one new variable created, *max*. We have a requirement of only 1 space unit.

1.010 Counting up time and space units, part 2

Let's analyse the linear search algorithm. The algorithm takes 3 arguments, A, N, and x, where A is a 1D array, N is the number of elemnts in A, and x is an integer. The pseudocode is found in algorithm **??**.

```
1: function F2(A, N, x)
2:     for 0 ≤ i < N do
3:         if A[i] = x then      ▷ 1 cond., 1 array access, 1 comparison, 2 memory reads
4:             return i                                  ▷ 1 return, 1 memory read
5:         end if
6:     end for
7:     return −1                                                    ▷ 1 return
8: end function
```

Because the *for* loop is not a simple instruction, we must break it down into simple instructions. A for loop is composed of three main components:

```
1: i ← 0                                                  ▷ 1 memory write
2: if i < N then                          ▷ 1 cond., 2 memory reads, 1 comparison
3:     <instructions>
4:     i ← i + 1                  ▷ 1 memory write, 1 memory read, 1 addition
5: end if
```

Note that the **If** part of the loop takes 4 time units, but runs $N+1$ times, therefore it takes $4 \cdot (N+1)$ time units. Also the increment part of the loop, takes 3 time units and runs N times, therefore it takes $3N$ time units. The total here is $4(N+1)+3N = 7N+5$ time units.

Continuing, we have another 5 time units running N times. Assuming the worst case, only outter-most return statement will execute for exactly 1 time unit.

Adding up all terms we have $7N + 5 + 5N + 1 = 12N + 6$ time units.

In terms of space units, we create a single new variable, i, and therefore our space requirement is 1 space unit.

1.011 Counting up time and space units

Please read about the analysis of insertion sort on pp.24–7 of the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.101 Growth of functions, part 1

Counting up every single time unit is not necessary. After making such large simplifications by using the RAM model, trying to get an exact number of time units is a pointless exercise when all we want to do is compare different algorithms and choose the fastest.

We can look at the running time of two different algorithms for solving the same problem. Figure 2 shows the graph of the running time as the size of the input grows.

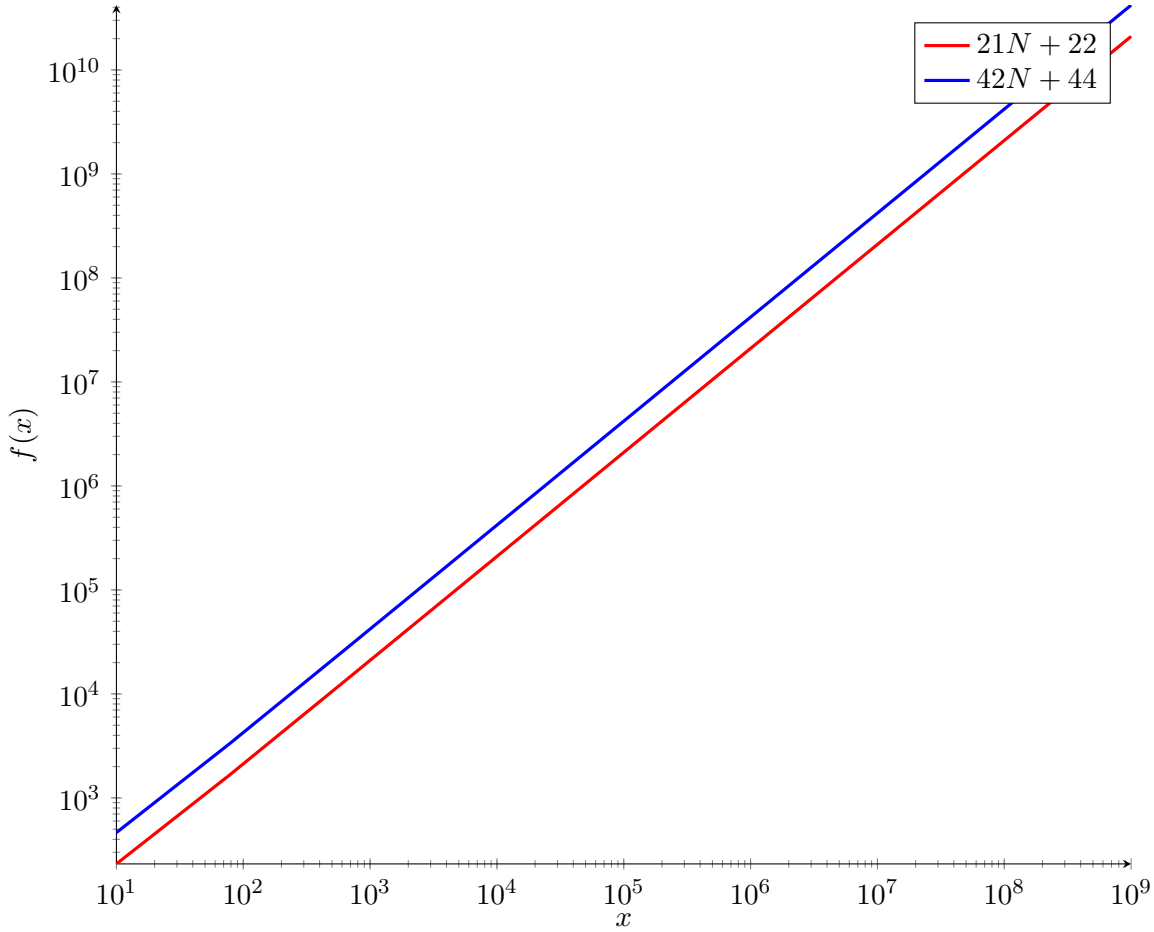


Figure 2: Running Time

Note that the running time grows linearly with the input size. That is, if the input grows 10 times, the running time grows about 10 times as well.

If someone proposes a third algorithm for solving the same problem with running time of $10N^2 + 30$, plotting the new function, we have the graph shown in figure 3.

We can see that the new curve, the one for $10N^2+30$, grows much faster than the other two. The difference is so large that the coefficients are not going to affect the difference as the input size grows.

When comparing algorithms, the growth of the running is sufficient, we don't need to specify coefficients. When analysing asymptotic growth of functions, lower order terms of the function also doesn't affect the function's growth.

For example $N^2 + N \approx N^2$ as N gets larger and larger.

Therefore, when comparing algorithms, we will do the following:

Use Generic Constants e.g. $T(N) = C_1N + C_2$

Growth Of Running Time Ignore constants and lower-order terms

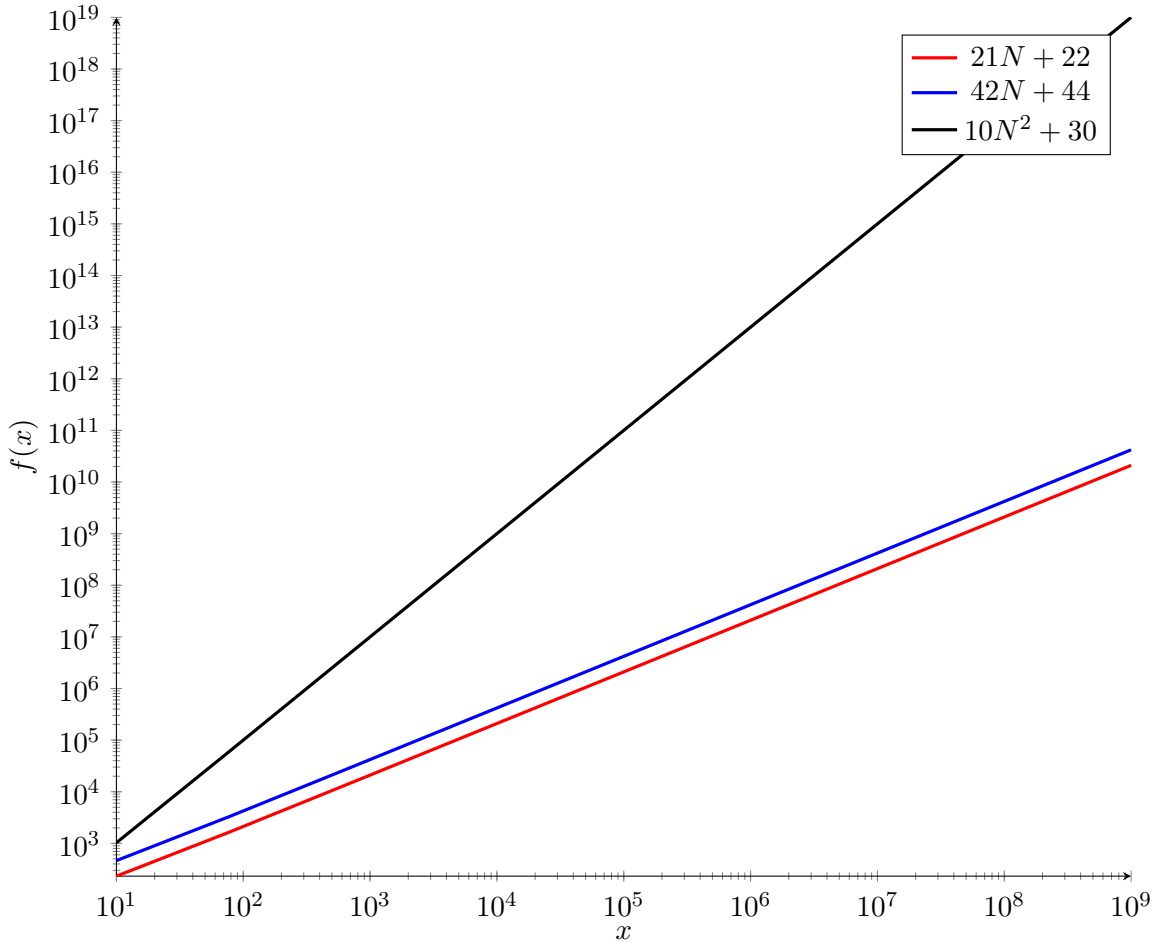


Figure 3: Running Time

Below, we can find a listing of the most common growth functions:

- 1 (constant time)
- $\log N$
- N
- $N \log N$
- N^2
- N^3
- 2^N

Figure 4 depicts each of the growth functions above.

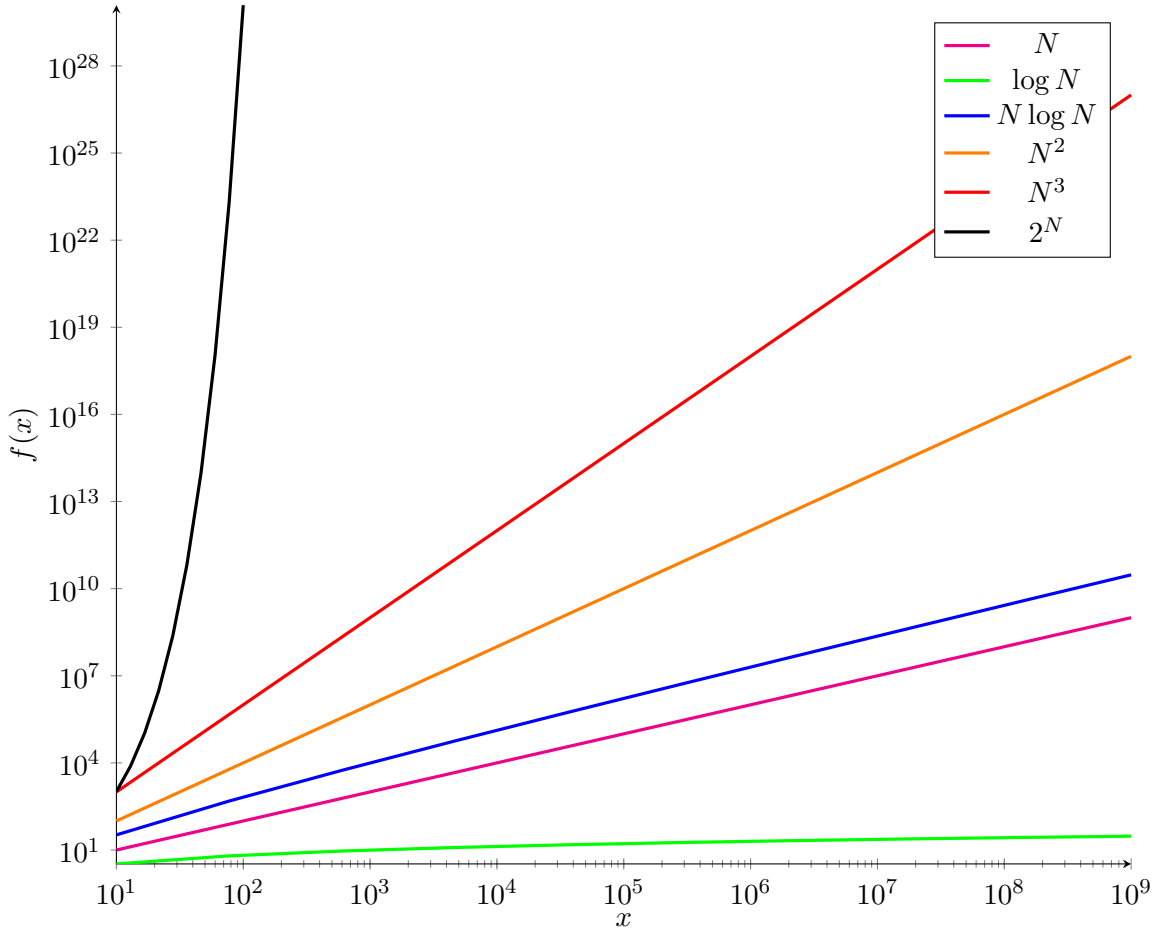


Figure 4: Running Time

1.103 Growth of functions, part 2

The following pseudocode in listing **??**, computes the sum of the diagonal of a square matrix. Instead of counting every memory access and numerical operation, we are checking if the instruction takes constant time or not.

```
1: function SumDiag(A)
2:     sum ← 0                                   ▷ C_0
3:     N ← Length(A[0])                          ▷ C_1 N + C_2
4:     for 0 ≤ i < N do                          ▷ C_3 N + C_4
5:         sum ← sum + A[i, i]                   ▷ C_5 N
6:     end for
7:     return sum                                ▷ C_6
8: end function
```

Adding up all the terms, we get the following expression:

$$\begin{aligned} T(N) &= (C_1 + C_3 + C_5)N + (C_0 + C_2 + C_4 + C_6) \\ &= C_7N + C_8 \\ &= N \end{aligned}$$

1.105 Growth of functions

Please read the sub-section titled 'Order of growth' in Section 2.2 (pp.28–9) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.106 Faster computer versus faster algorithm

Assuming we designed an algorithm to solve a particular problem with a quadratic growth (i.e. $T(N) = N^2$). We will also assume that we have a computer where 1 time unit = 1ns.

The table below shows the running time for different input sizes:

N	N^2
10^1	$0.1\mu S$
10^2	$10\mu S$
10^3	$1mS$
10^4	$100mS$
10^5	$10S$
10^6	$16.7min$
10^7	$27.8hr$
10^8	$116days$

Because of that, we buy a computer which is 10 times faster, which will give us the following table:

N	N^2	N^2 (10x)
10^1	$0.1\mu S$	$0.01\mu S$
10^2	$10\mu S$	$1\mu S$
10^3	$1mS$	$0.1mS$
10^4	$100mS$	$10mS$
10^5	$10S$	$1S$
10^6	$16.7min$	$1.7min$
10^7	$27.8hr$	$2.8hr$
10^8	$116days$	$11.6days$

If we manage to design a new algorithm with a linear growth (i.e. $T(N) = N$), we will get the following table:

N	N^2	N^2 (10x)	N
10^1	$0.1\mu S$	$0.01\mu S$	$10nS$
10^2	$10\mu S$	$1\mu S$	$100nS$
10^3	$1mS$	$0.1mS$	$1\mu S$
10^4	$100mS$	$10mS$	$10\mu S$
10^5	$10S$	$1S$	$0.1mS$
10^6	$16.7min$	$1.7min$	$1mS$
10^7	$27.8hr$	$2.8hr$	$10mS$
10^8	$116days$	$11.6days$	$0.1S$

It's clear that investing in Algorithmic development pays off.

1.108 Faster computer versus faster algorithm

Please read Section 1.2 (p.11–14) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 2

Key Concepts

- Determine time and memory consumption of an algorithm described using pseudocode
- Determine the growth function of the running time or memory consumption of an algorithm
- Use Big-$\mathcal{O}$, Omega and Theta notations to describe the running time or memory consumption of an algorithm.

1.201 Worst and best cases

While computing the running time $T(N)$ of an algorithm as a function of the input size is sufficient for some classes of algorithms, there are other algorithms where the *nature* of the input can also change the running time of the algorithm.

One such example is the **Linear Search** algorithm. Its running time will change according to the input size and the nature of the input.

For example if the value we're looking for is **always** in the first index of the input array, Linear search will run in constant time $\mathcal{O}(1)$ regardless of the input size. If, however, the value we're looking is **never** in the input array, Linear search running grows linearly with the input size.

We can say that the case where the number we're looking is in the first position of the array is the *Best Case* scenario. Conversely, the case where the number we're looking is not in the array is called the *Worst Case* scenario.

1.202 Worst and average cases

Please read p.27 of the guide book, on worst case and average case analysis:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

1.301 Introduction to asymptotic analysis

Asymptotic analysis is the analysis of the growth of a function as the input size grows larger and larger.

As the input size tends to infinity, the constants and lower-order terms are irrelevant as they provide a very small impact in the function growth behavior.

1.303 Big-$\mathcal{O}$ notation

Big-$\mathcal{O}$ Notation gives us an upper bound to a function growth. For any given function, there is a set of functions that can be considered an upper bound. This is exactly what Big-$\mathcal{O}$ notation defines: a set of functions $g(N)$ that can act as a upper bound for the growth of a function $T(N)$.

More formally, Big-$\mathcal{O}$ is defined as:

$$T(N) \in \mathcal{O}(g(N)) \rightarrow C \cdot g(N) \geq T(N) \forall N \geq n_0$$

Where both C and n_0 are positive constants. In figure 5 we show an example function $10N^2 + 15N + 5$ and two possible upper bounds N^2 and $25N^2$.

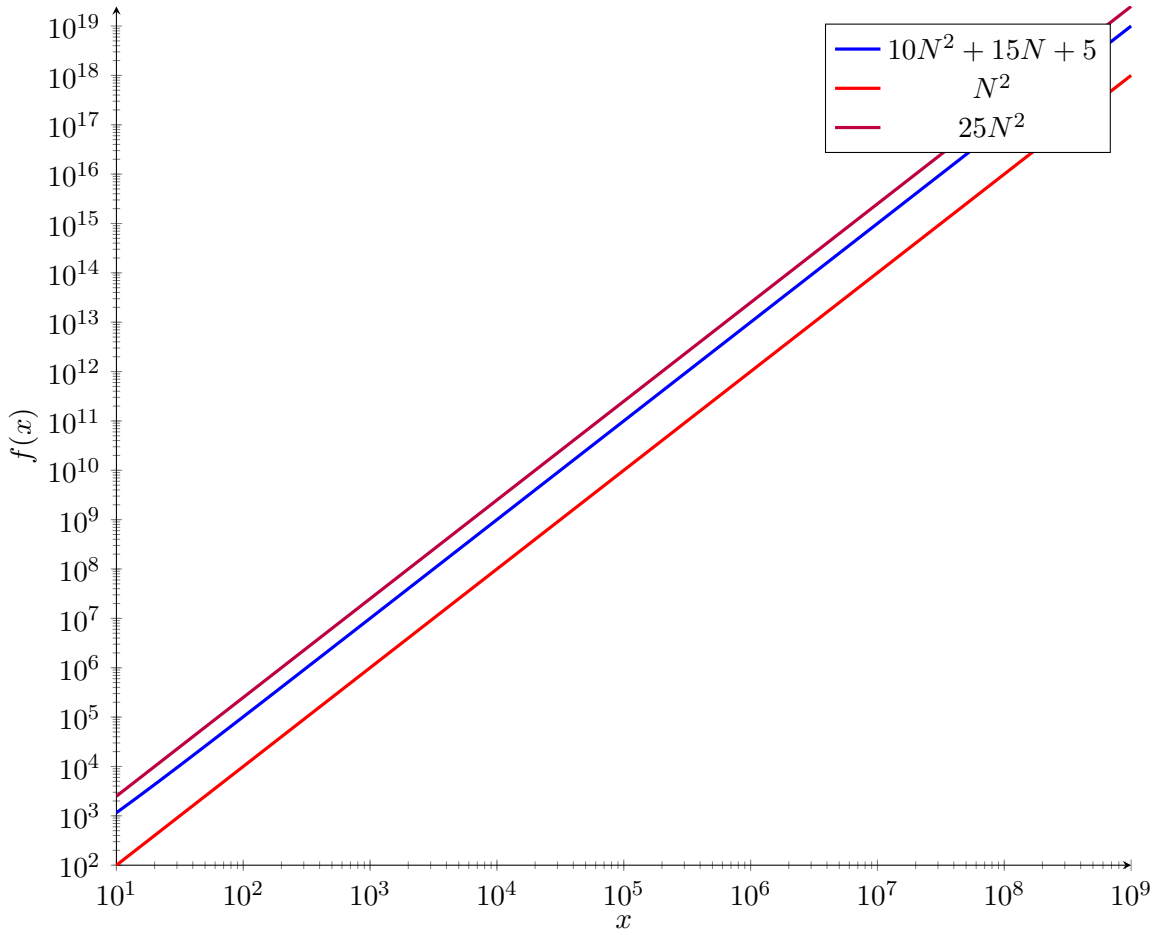


Figure 5: Big-$\mathcal{O}$

We can show the same thing with N^3, N^4, and 2^N. See figure 6 below.

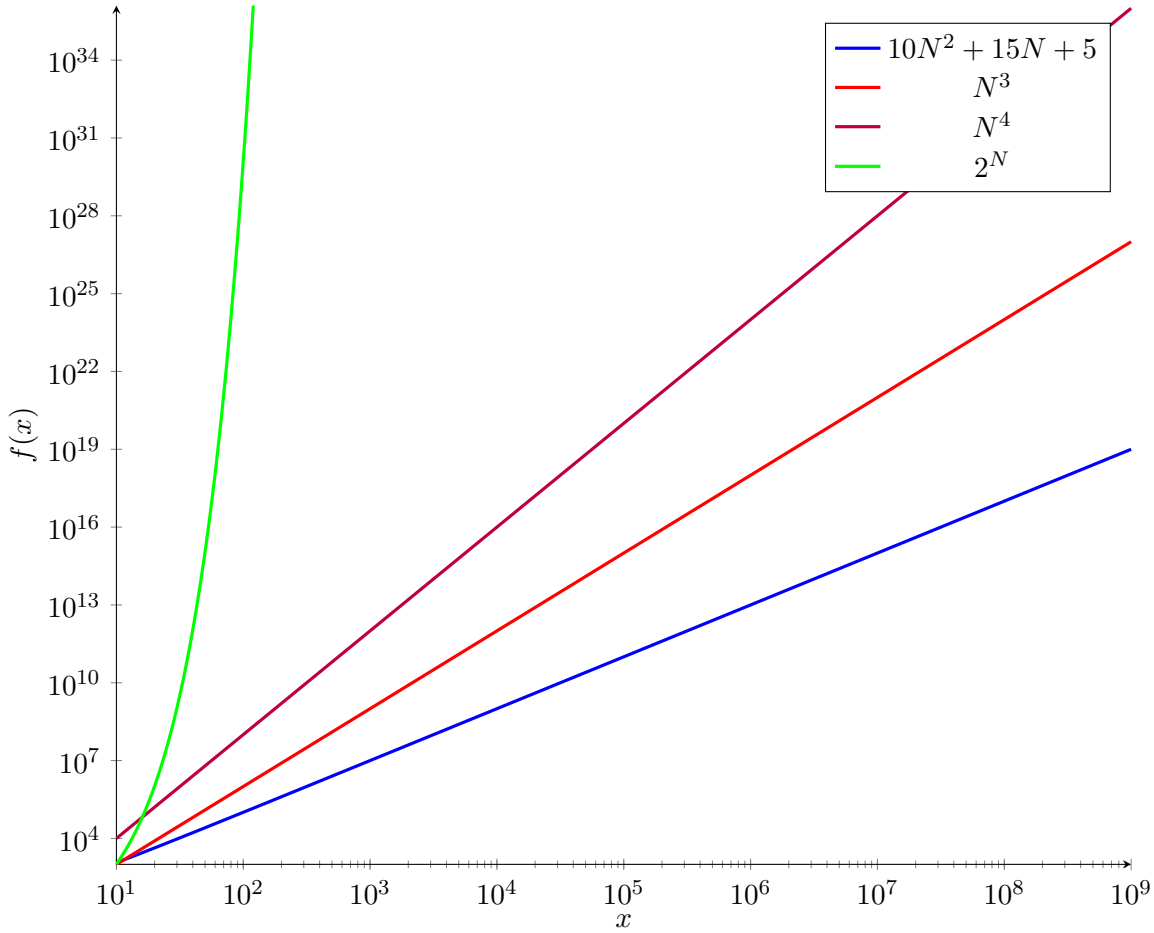


Figure 6: Big-$\mathcal{O}$: N^3, N^4, 2^N

1.305 Omega notation

Big-Ω notation is analogous to Big-$\mathcal{O}$ notation, however instead of looking for upper bounds, we're looking for lower bounds.

Much like Big-$\mathcal{O}$ notation, there are a set of functions that can act as lower bound for a given function. More formally, Big-Ω is defined as:

$$T(N) \in \Omega(g(N)) \rightarrow C \cdot g(N) \leq T(N) \forall N \geq n_0$$

We can produce a similar graph as with Big-$\mathcal{O}$ notation. It's show in figure 7 below.

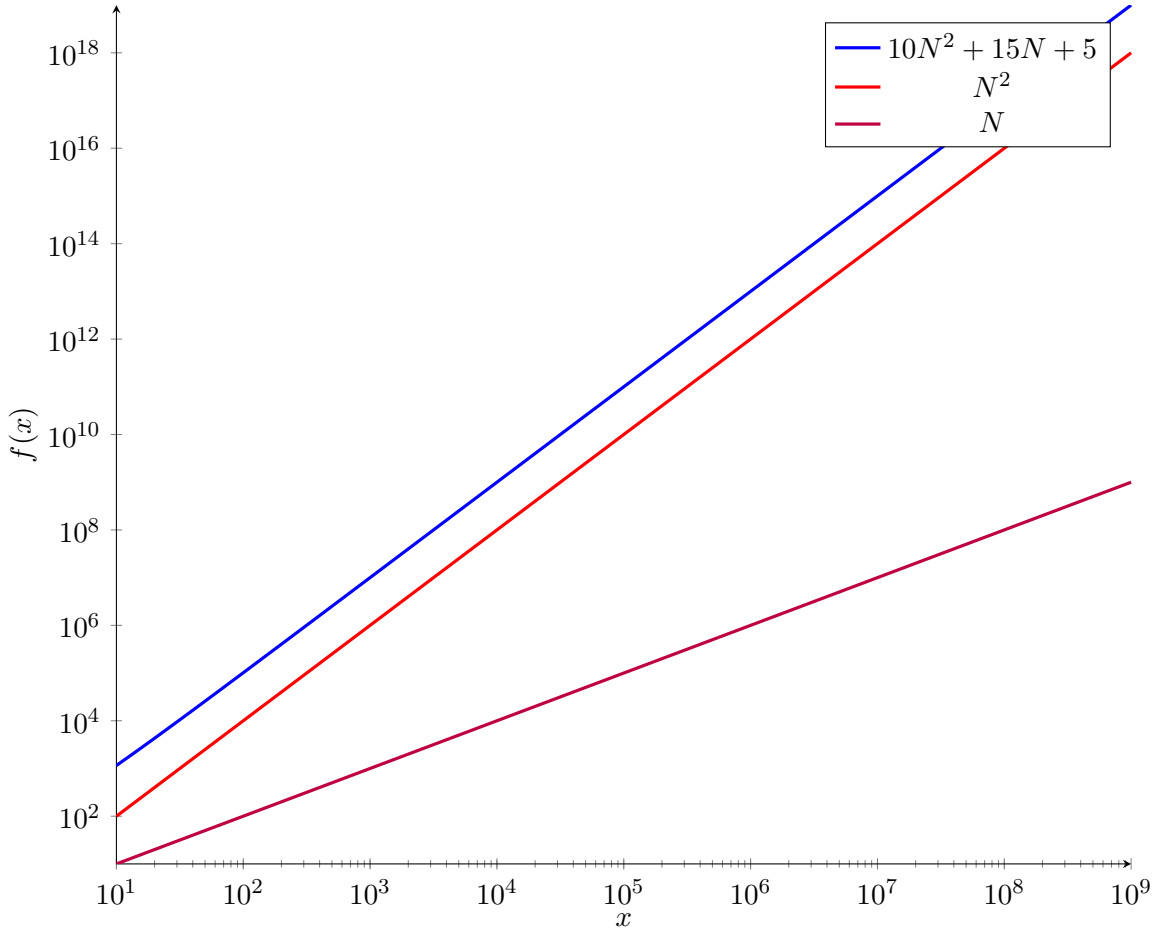


Figure 7: Big-Ω

We can also show that the function $T(N) = 10N^2 + 15N + 5$ is $\Omega(\log N)$ and $\Omega(1)$. See figure 8 below.

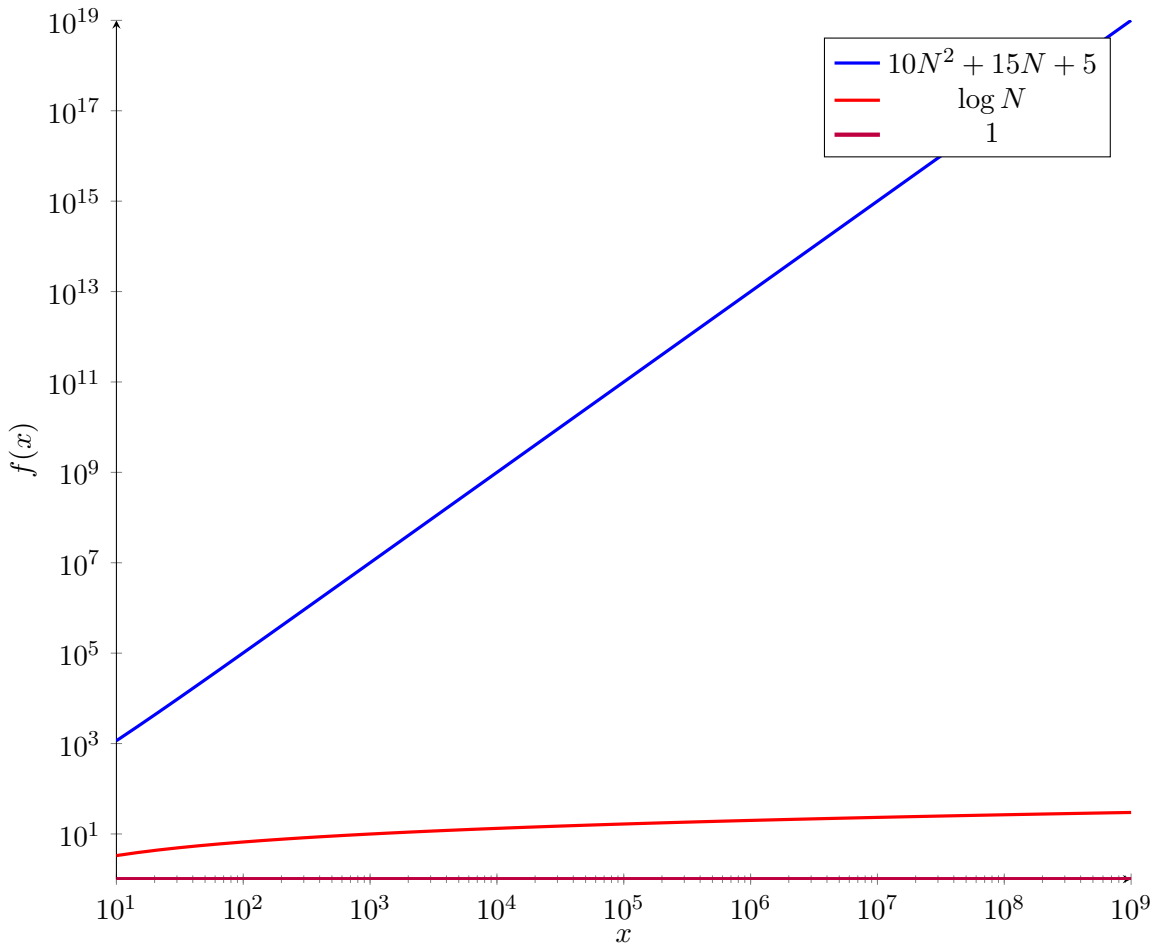


Figure 8: Big-Ω: $\Omega(\log N)$ and $\Omega(1)$

1.307 Theta notation

One drawback of both Big-$\mathcal{O}$ and Big-Ω is that they both refer to a set of functions. This means that when we say that the running time of an algorithm is $\mathcal{O}(N^4)$ it might be that the algorithm grows with N^2 much faster than with N^4, however $\mathcal{O}(N^4)$ is still correct.

With Θ notation, we find a single function that acts as both upper-bound and lower-bound for running time or memory consumption. What we do, in practice, is that we find two different constants c_1 and c_2 such that $c_1 \cdot g(N)$ is a lower bound and $c_2 \cdot g(N)$ is an upper bound. Naturally, $c_1 \leq c_2$.

Figure 9 depicts this:

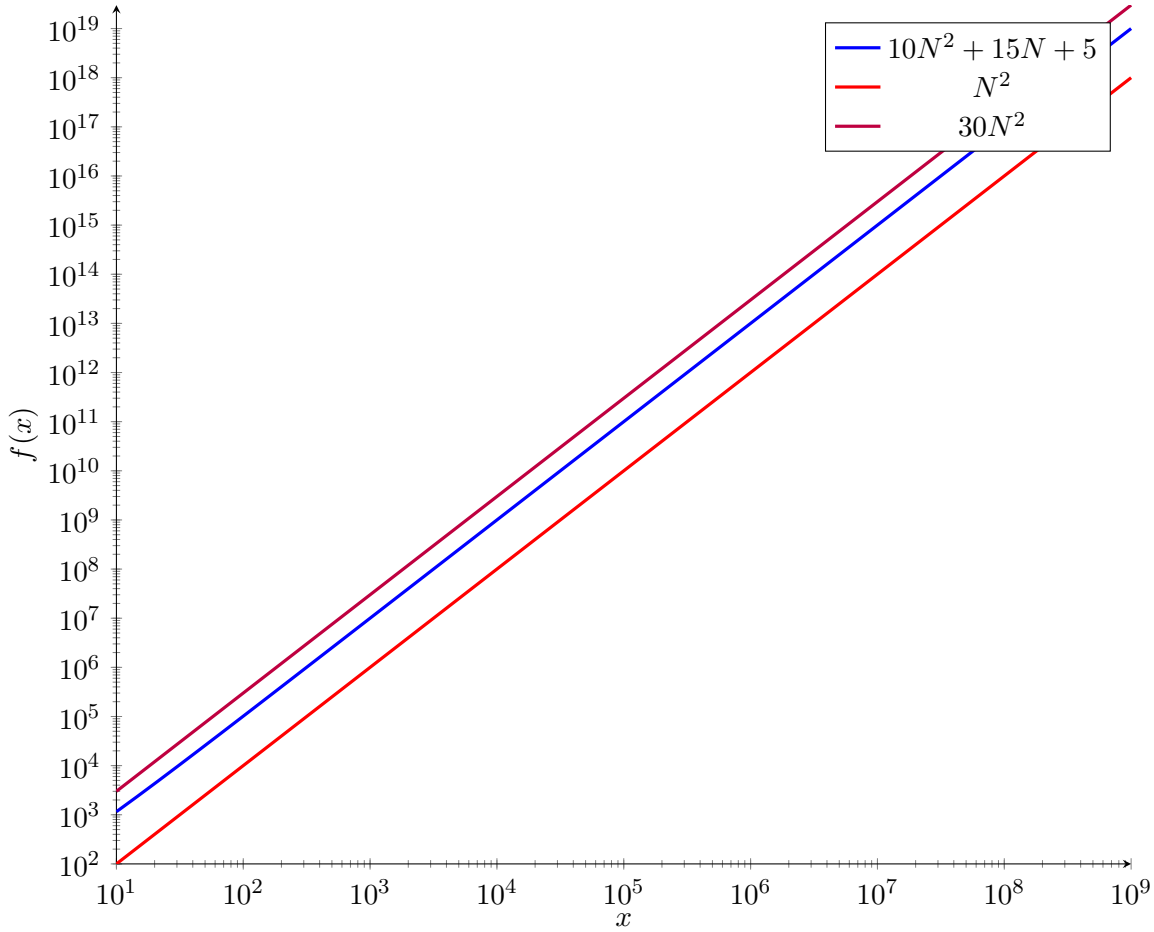


Figure 9: Big-Θ

What we can see is figure 9 is that if $g(N) = N^2$ is multiplied by 1, then it can act as a lower-bound, while if it's multiplied by 30, then it can act as an upper-bound. Therefore $c_1 = 1$ and $c_2 = 30$.

More formally, Big-*Theta* notation is defined as follows

$$T(N) \in \Theta(g(N)) \rightarrow \begin{cases} c_1 \cdot g(N) \geq T(N) \forall N \geq n_0 \\ c_2 \cdot g(N) \leq T(N) \forall N \geq n_0 \end{cases}$$

1.309 Asymptotic notation

Please read Section 3.1 (pp.43–52) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 3

Key Concepts

- Trace and write recursive algorithms
- Write the recursive version of an iterative algorithm using pseudocode
- Calculate the time complexity of recursive algorithms.

2.001 Introduction to recursion

During this week we learn about recursion. The topic is divided into three parts:

1. Understanding Recursion
2. Creating Recursion
3. Analysing Recursion

Recursion happens when an algorithm calls itself. For example, listing 1 is a recursive algorithm:

Algorithm 1 A Simple Recursive Algorithm

```
1: function HELLO
2:     PRINT("hello")          ▷ Print "hello" on the screen
3:     HELLO                   ▷ Recursive call
4: end function
```

The algorithm shown in listing 1 is infinitely recursive, meaning it will never stop with the recursive calls. This is the result of a badly designed recursive algorithm.

2.002 The structure of recursive algorithms

We can modify the previous algorithm so it doesn't recurse infinitely. Algorithm 2 shows the new version of the algorithm.

Algorithm 2 A Better Recursive Algorithm

```
1: function HELLO(n)
2:     if n = 0 then                          ▷ If n = 0...
3:         return                             ▷ We're done
4:     end if
5:     PRINT("hello")                         ▷ Print "hello" on the screen
6:     HELLO(n − 1)                           ▷ Recursive call approaching base case
7: end function
```

The `if` statement in algorithm 2 is called the *Base Case*. We use it to stop the recursion.

As a rule of thumb, recursive algorithms should always include at least one base case and a recursive call approaching the base case.

2.004 Tracing a recursive algorithm

Tracing a recursive algorithm lets us understand what task is accomplished by the algorithm. Algorithm 3 below will be used to demonstrate this.

Algorithm 3 Tracing a recursive algorithm

```
1: function F(a, b)
2:     if b = 0 then
3:         return a
4:     end if
5:     return F(a + 1, b − 1)
6: end function
```

It's clear from the code listing that the base case triggers when b is equal to 0. We can also see that in the recursive call, we're getting closer to 0 by decrementing b by 1 unit. At the same time b is decremented, a is incremented by the same amount.

We can start tracing this algorithm with inputs $2, 2$ respectively for a and b. The first time the algorithm runs, it checks if $b = 0$. Because that check evaluates to false, we move on to the recursive call and change a to 3 and b to 1.

In the recursive call we check if $b = 0$; it isn't, then we move to the recursive call by changing a to 4 and b to 0.

In this new recursive call we check if $b = 0$, it is, then we return the value of a which is 4. That value trickles all the way back to the first call.

In summary, this recursive algorithm calculates $a + b$.

2.101 From iteration to recursion

An iterative algorithm is one that uses a loop to repeat a set of instructions. A recursive algorithm repeats a set of instructions by calling itself.

Algorithm 4 and 5 achieve the same thing, that is printing the numbers from n down to 0. One is iterative while the other is recursive.

Algorithm 4 Iterative Count Down

```
1: function IterCountDown(n)
2:     for i ← n downto 0 do
3:         Print(n)
4:     end for
5: end function
```

Algorithm 5 Recursive Count Down

```
1: function RecCountDown(n)
2:     if n < 0 then
3:         return
4:     end if
5:     Print(n)
6:     RecCountDown(n − 1)
7: end function
```

Both of these algorithms need an initial value, a condition to stop or continue repetition, and a method for updating the value of the variable we're using otherwise we will never stop repeating.

2.103 Writing a recursive algorithm, part 1

When writing a recursive algorithm, we should first treat the recursive call as a black box, for which we only know the result.

By doing that, we limit the amount of information we need to keep track of in order to understand what's happening.

This means that each call is responsible for a small part of the job, with everything being delegated to the recursive call.

2.104 Writing a recursive algorithm, part 2

Applying the technique from the previous section in a recursive linear search algorithm.

The small part the algorithm is going to execute is checking if the value we're looking for is in the last element of the array, if it is we're done, if it isn't, we'll delegate the search in the remaining part of the array.

This would result in an implementation like the one shown in algorithm 6.

Note that we if the value of N is less than 0, we know that we have consumed the entire array or we received an empty array to start with. Therefore, the item wasn't in the array, so we return *FALSE*.

Algorithm 6 Recursive Linear Search

```
1: function RecLinearSearch(A, N, x)
2:     if N < 0 then
3:         return FALSE
4:     end if
5:     if A[N − 1] = x then
6:         return TRUE
7:     end if
8:     return RecLinearSearch(A, N − 1, x)
9: end function
```

Moreover, we're always checking the final value of the array, pointed to by $A[N-1]$. If the value we're searching for is in that position, we return it.

If, however, the value is not there, we recursively call ourselves to process the remaining part of the array. This causes us to reduce N by one at least recursive call and, thus, approximate the base case of an empty array.

Week 4

Key Concepts

- Trace and write recursive algorithms
- Write the recursive version of an iterative algorithm using pseudocode
- Calculate the time complexity of recursive algorithms.

2.201 Time complexity of recursive algorithms

The time complexity of an algorithm is the asymptotic number of simple operations executed by the algorithm. We can apply the same analysis to recursive algorithms.

As an example, we use the `Factorial` function whose pseudocode is shown in listing 7:

Algorithm 7 Factorial Function

```
1: function FACTORIAL(n)
2:     if n ≤ 1 then
3:         return 1
4:     end if
5:     return n× FACTORIAL(n − 1)
6: end function
```

We can annotate this algorithm with the cost of each line, seen below in listing 8

Algorithm 8 Factorial Function Annotated With Cost

```
1: function FACTORIAL(n)                          ▷ T(N)
2:     if n ≤ 1 then                              ▷ C_0
3:         return 1
4:     end if
5:     return n× FACTORIAL(n − 1)                 ▷ C_4 + T(N − 1)
6: end function
```

With that we can extract the expression:

$$T(N) = C_0 + C_4 + T(N-1)$$
$$T(N) = C_5 + T(N-1)$$

We can see that the running time of $T(N)$ depends on the running time of the $T(N-1)$, we refer to this type of equation as *Recurrence Equation.*

2.203 Solving recurrence equations

The main problem with a recurrence equation is that we don't have an explicit expression for the running time of an algorithm.

To solve a recurrence equation we follow a two-step process:

1. Find a value of N for which $T(N)$ is known. Usually, this can be achieved with the running time of the best-case scenario input.
2. Expand the right side of the recurrence equation until you can't replace the known value of $T(N)$ on it anymore.

For example, looking back at algorithm 7 we can see that the best case is achieved when the number 1 is our argument. In such a case, the conditional statement evaluates to true which causes the algorithm to immediately return. Both instructions, i.e the `if` and the `return` execute in constant time, therefore our best case runs in constant time. We conclude that $T(1) = C$. With that in mind, we can start to expand the right side of the expression:

$$
\begin{aligned}
T(N) &= C_5 + T(N-1) \\
T(N) &= C_5 + C_5 + T(N-2) \\
T(N) &= C_5 + C_5 + C_5 + T(N-3) \\
T(N) &= C_5 + C_5 + C_5 + C_5 + T(N-4) \\
T(N) &= C_5 + C_5 + C_5 + C_5 + \ldots + T(1) \\
T(N) &= C_5 + C_5 + C_5 + C_5 + \ldots + C \\
T(N) &= (N-1)C_5 + C
\end{aligned}
$$

Now that the recurrence equation is known, we can do an asymptotic analysis for $T(N)$:

Big-$\mathcal{O}$ $\mathcal{O}(N), \mathcal{O}(N^2), \mathcal{O}(N^3), \ldots$

Big-Ω $\Omega(N), \Omega(\log N), \Omega(1), \ldots$

Big-Θ $\Theta(N)$

2.301 The master theorem

The Master Theorem is a simpler way of executing asymptotic analysis, however it can't be applied to every recurrence equation.

In order to apply the Master Theorem, the recurrence equation must be of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$.

When the Master Theorem can be applied, there are three cases to take into account:

1. $f(n) < n^{\log_b a}$

   In this case, $T(n) = \Theta(n^{\log_b a})$

2. $f(n) = n^{\log_b a}$

   In this case, $T(n) = \Theta(n^{\log_b a} \log n)$

3. $f(n) > n^{\log_b a}$

   For this case to be applicable, there is one extra requirement to be met: $a \cdot f(\frac{n}{b}) \leq c$, where $c < 1$ and n is large. In this case, $T(n) = \Theta(f(n))$

2.303 Recursive algorithms and their analysis

Please read:

- Section 2.3 (pp.29–37), only if you are familiar with Mergesort. If not, we will review this section again later
- Chapter 4, pp.65–113 (except section 4.6)

from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 5

Key Concepts

- Identify the different approaches of different comparison sorting algorithms
- Implement different comparison sorting algorithms
- Calculate the time complexity of different comparison sorting algorithms

3.001 Comparison and non comparison sorting algorithms

Sorting algorithms can be split into two main categories: Comparison Sorts and Non-comparison Sorts.

We can quickly build a simple tree showing the main algorithms:

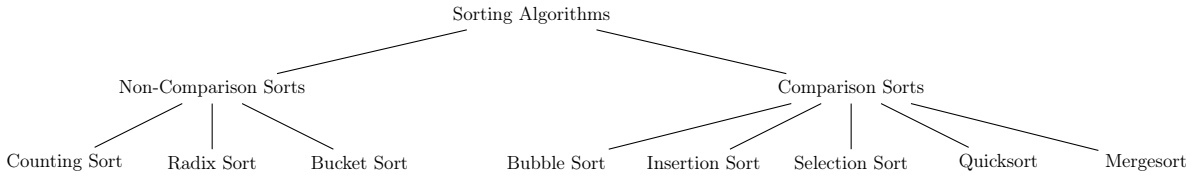


Figure 10: Sorting Algorithms

The difference between them is that comparison sorts will compare two elements to decide the order, while non-comparison sorts will not.

Comparison sorts have a limit on their worst-case time complexity; they can never be faster than $N \log N$ while non-comparison sorts do not suffer from this limitation.

Table 0.1 below provides a summary of worst- and best-case time complexity of the comparison sorts listed above.

Table 0.1: Comparison Sorts Complexity

Algorithm	**Worst-case**	**Best-case**
Bubble	$\Theta(N^2)$	$\Theta(N)
Insertion	$\Theta(N^2)$	$\Theta(N)$
Selection	$\Theta(N^2)$	$\Theta(N^2)$
Quicksort	$\Theta(N^2)$	$\Theta(N \log N)$
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$

3.004 Bubble sort: Pseudocode

We can see bubble sort pseudocode in algorithm 9.

Algorithm 9 Bubble Sort

```
1: function BubbleSort(A, N)
2:     swapped ← true
3:     while swapped do
4:         swapped ← false
5:         for 0 ≤ i < N − 1 do
6:             if A[i] > A[i + 1] then
7:                 Swap(A[i], A[i + 1])
8:                 swapped ← true
9:             end if
10:        end for
11:        N ← N − 1
12:    end while
13:    return A
14: end function
```

3.102 Insertion sort: Pseudocode

We can see insertion sort pseudocode in algorithm 10.

Algorithm 10 Insertion Sort

```
1: function InsertionSort(A, N)
2:     for 1 ≤ j < N − 1 do
3:         ins ← A[j]
4:         i ← j − 1
5:         while i ≥ 0 ∧ ins < A[i] do
6:             A[i + 1] ← A[i]
7:             i ← i − 1
8:         end while
9:         A[i + 1] ← ins
10:    end for
11:    return A
12: end function
```

3.104 Insertion sort

Please read Sections 2.1 (pp.16–22) and 2.2 (pp.23–9) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

3.105 Selection sort: Pseudocode

We can see selection sort pseudocode in algorithm 11.

Algorithm 11 Selection Sort

```
1: function SelectionSort(A, N)
2:     for 1 ≤ i < N − 1 do
3:         min ← PosMin(A, i, N − 1)
4:         Swap(A[i], A[min])
5:     end for
6:     return A
7: end function
```

Week 6

Key Concepts

- Identify the different approaches of different comparison sorting algorithms
- Implement different comparison sorting algorithms
- Calculate the time complexity of different comparison sorting algorithms

3.202 Quicksort: Pseudocode

Quicksort is a comparison sorting algorithm that's very simple to implement if we use recursion.

Listing 12 shows the pseudocode for Quicksort.

Algorithm 12 Quicksort

```
1: function QUICKSORT(A, low, high)
2:     if low < high then
3:         p ←PARTITION(A, low, high)
4:         QUICKSORT(A, low, p − 1)
5:         QUICKSORT(A, p + 1, high)
6:     end if
7: end function
```

We see that Quicksort calls itself twice during its execution. It does this by partitioning the input array into two smaller arrays of roughly half the size. The function `Partition` is responsible for selecting a *pivot*, moving all numbers lower than *pivot* to the left side of the array, and moving the *pivot* to its final position.

As any recursive algorithm, Quicksort requires a base case. In the pseudocode above, the base case is implicit in the `else` part of the `if` condition. Note that if $low \geq high$ the algorithm will stop executing.

We should write the pseudocode for the `Partition` function. The requirements are:

1. Select number in position `high` as the pivot
2. Move all numbers lower than pivot to the left of the array
3. Return the pivot

The pseudocode may look like the one shown in listing 13.

The `Swap` function is a simple helper to swap the i^{th} and j^{th} elements of the array A. Its pseudocode is shown in listing 14.

Algorithm 13 Partition

```
1: function PARTITION(A, low, high)
2:     p ← A[high]
3:     i ← low
4:     for low ≤ j < high do
5:         if A[j] ≤ p then
6:             SWAP(A, i, j)
7:             i ← i + 1
8:         end if
9:     end for
10:    SWAP(A, i, high)
11:    return i
12: end function
```

Algorithm 14 Swap

```
1: function SWAP(A, i, j)
2:     t ← A[i]
3:     A[i] ← A[j]
4:     A[j] ← t
5: end function
```

3.204 Quicksort

Please read the introduction to Chapter 7, Section 7.1 (pp.170–3) and Section 7.2 (p.174–8) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Available from here.

3.302 Mergesort: Pseudocode

Mergesort is another comparison sorting algorithm that's very easy to implement if we use recursion. There are several possible implementations of Mergesort depending on:

1. The data structure used
2. The way the merge part works

During this section, we use the Array data structure and an out-of-place merge, which means that we will allocate extra memory during the merge operation. This gives us a better time complexity.

Listing 15 contains the pseudocode for Mergesort.

Much like Quicksort, the base case for Mergesort is implicit in the `if` condition. We can see that whenever $l \geq h$ the algorithm won't do anything and simply return.

Algorithm 15 Mergesort

```
1: function MergeSort(A, l, h)
2:     if l < h then                                  ▷ Should continue?
3:         mid ← ⌊(h+l)/2⌋                            ▷ Midpoint calculation
4:         MergeSort(A, l, mid)                       ▷ Sort left half
5:         MergeSort(A, mid + 1, h)                   ▷ Sort right half
6:         Merge(A, l, mid, h)                        ▷ Merge left and right halves
7:     end if
8: end function
```

The midpoint between l and h is calculated by line 3. Right after calculating the midpoint, we execute our first recursive call to Mergesort. This will try to sort the left half of the array, this can seen in line 4. What follows is a recursive call to Mergesort to operate on the right side of the array, as seen in line 5. When this complete, both halves of the array will be sorted. The only thing left to do is to merge both halves maintaining the order. This can seen in line 6.

We must write the pseudocode for the `Merge` function. The requirements are:

1. Copy the already sorted elements between l and mid into a new array called L.
2. Copy the already sorted elements between $mid + 1$ and r into a new array called R.
3. Compare first elements of L and R, smallest goes back into A. Repeat until both L and R are empty.

A possible implementation of `Merge` is provided in listing 16.

We should also calculate the worst- and best-case time complexity of Mergesort.

3.305 Mergesort

Please read Sections 2.3.1 (pp.30–34) and 2.3.2 (pp.34–37) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Algorithm 16 Merge

```
1: function MERGE(A, l, mid, h)
2:     L ← A[l ... mid]
3:     R ← A[mid + 1 ... h]
4:     i ← 0
5:     j ← 0
6:     k ← l
7:     while i ≤ mid ∧ j < (h − mid) do
8:         if L[i] ≤ R[j] then
9:             A[k] ← L[i]
10:            i ← i + 1
11:        else
12:            A[k] ← R[j]
13:            j ← j + 1
14:        end if
15:        k ← k + 1
16:    end while
17:    while i ≤ mid do
18:        A[k] ← L[i]
19:        i ← i + 1
20:        k ← k + 1
21:    end while
22:    while j < (h − mid do
23:        A[k] ← R[j]
24:        j ← j + 1
25:        k ← k + 1
26:    end while
27: end function
```

Week 7

Key Concepts

- Identify the different approaches of different non-comparison sorting algorithms
- Implement different non-comparison sorting algorithms
- Calculate the time complexity of different non-comparison sorting algorithms

4.001 The limits of comparison sorts

We have, thus far, reviewed 5 comparison sorts:

Bubble Sort compares pairs of elements and swaps them if they are in the wrong order

Insertion Sort finds the correct position in which to insert the next unsorted element in the array

Select Sort selects the minimum value in the unsorted part of the array and stores it at the beginning of the unsorted part

Quicksort recursively selects a pivot and stores it in its final correct position

Merge Sort divides the array in halves until individual elements are left which are then merged back in sorted order

During analysis of the worst-case time complexity of these algorithms, we found out that they will never perform better than $\Theta(N \log N)$, where N is the number of elements in the array.

We can get an idea for why this is the case with a simple thought exercise. Let's assume that we have an array with N unsorted numbers. There are exactly $N!$ ways of arranging the numbers in the array. Among all the different arrangements, only 1 is the correct order. Now, the question we're asking is "what is the **maximum number of comparisons** a sorting algorithm must do to find the correct arragement of numbers?"

Taking a 3-element array 11 as an example:

The first comparison happens between $A[0]$ and $A[1]$, essentially we're testing if the 0^{th} element is smaller than the 1^{st} element. If the answer is *yes*, the elements are already sorted. If the answer is *no*, then we need to put them in the correct order. This will continue until we process the entire array. We can build a decision tree 12 of all these cases:

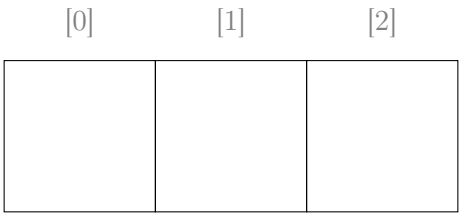


Figure 11: 3-element Array

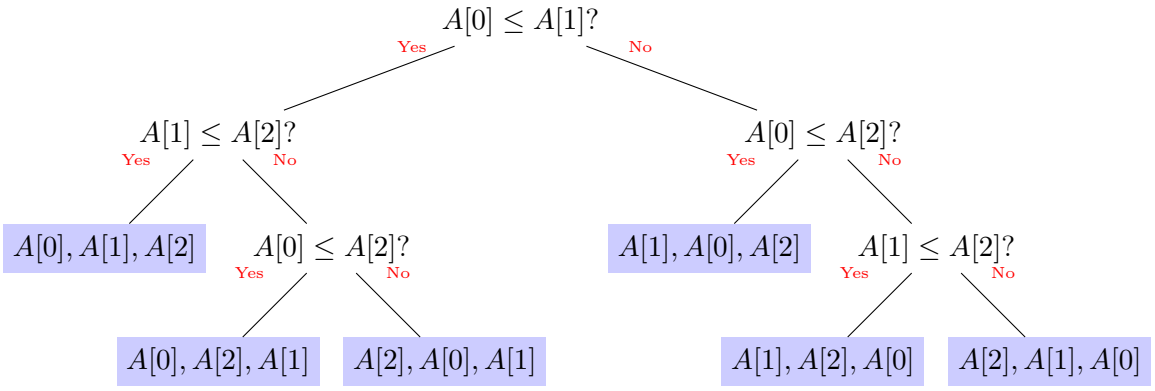


Figure 12: Decision Tree

From decision tree 12, we can conclude that:

1. there are exactly $N!$ leaves in the tree (the ones colored blue);
2. the maximum number of comparisons is the length of the longest path in the tree;
3. there are at most 2^L leaves in this tree.

We also know that actual number of leaves cannot be greater than the maximum number of possible leaves, therefore:

$$
\begin{aligned}
N! &\leq 2^L && \text{N! is at most } 2^L \\
\log N! &\leq \log 2^L && \text{Applying log to both sides} \\
\frac{\log N!}{\log 2} &\leq L && \text{Dividing by } \log 2 \\
L &= \Omega(\log N!) && \text{Applying asymptotic notation} \\
N! &\approx N^N && \text{Stirling's approximation} \\
L &= \Omega(\log N^N) && \text{Substituting } N! \text{ for } N^N \\
L &= \Omega(N \log N) && \text{Logarithm rule}
\end{aligned}
$$

4.002 Lower bounds for commparison sorts

Please read the introduction to Chapter 8 and Section 8.1 (pp.191–3) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

4.101 Counting sort: Introduction

Counting sort is a non-comparison sort with a linear worst-case running time.

Let's assume we're sorting numbers within the range 0 through 9, both inclusive. We can build a frequency array of ten items where the value at the index k is the number of times the number k appears in the set of numbers we're trying to sort.

Like shown in figure 13:

1	2	0	0	0	1	3	1	1	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figure 13: Counting Sort: Array C

Figure 13 tells us that the number 0 appears once in the set of numbers, the number two is not part of the set of nubmers, and the number 6 appears three times.

Given array C, we can find out what the array of sorted numbers looks like, that's shown in figure 14 below:

0	1	1	5	6	6	6	1	1

Figure 14: Counting Sort: Array R

Array R is sorted and we never did a single comparison to produce it. All we did was visit every element in Array C and place as many copies as indicated in R.

Counting sort works in a very similar fashion. Given an input argument A (an unsorted array) we must:

1. Create array C
    a) Find maximum value in array A
    b) Create array C with (k+1) elements, where k is the maximum value in A
    c) Traverse array A and update frequencies in C
2. Create array R

a) Create array R with the same length as A

b) Traverse array C and copy corresponding elements as many times as required.

Figure 15 below shows a depiction of the process:

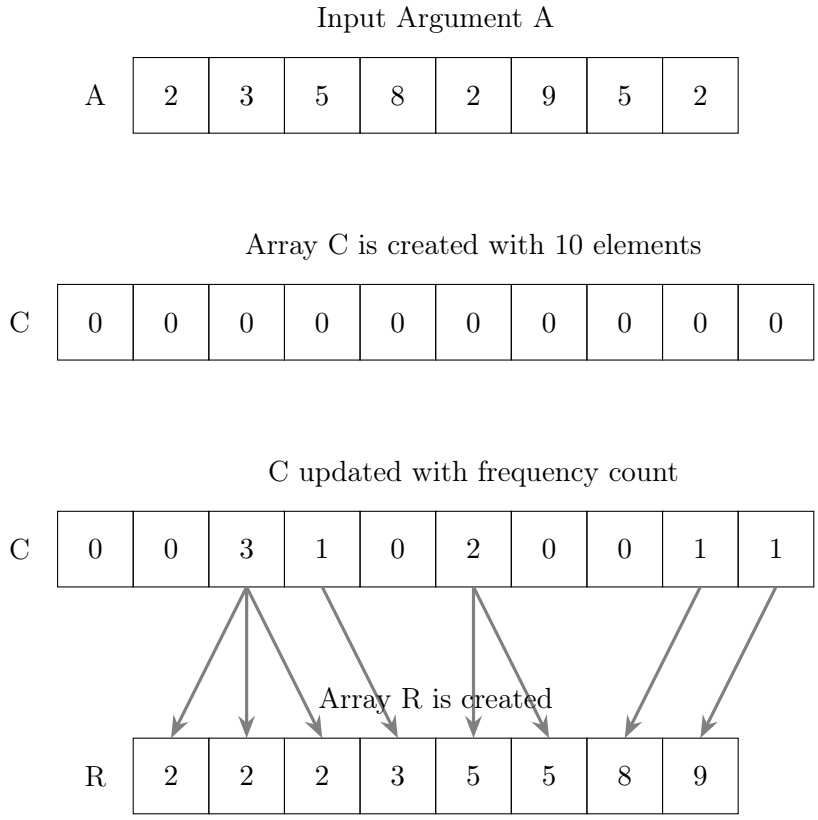


Figure 15: Counting Sort Execution

While the counting sort algorithm has a very desirable linear time complexity, there two main drawbacks:

It can only sort integer numbers Because the algorithm uses the array indices to represent the numbers to sort and indices must be integer

C must have as many elements as the *max* + 1 The extra memory used by array C can be significant if the maximum number in the set of numbers to represent is too big. For example, what happens if we have to sort an array with 10 elements where the maximum is 10^9?

4.102 Counting sort: Pseudocode

The pseudocode for Counting is as shown in listing 17.

Algorithm 17 Counting Sort Pseudocode

```
1: function CountingSort(A, k)
2:     C ← new Vector[k + 1]                    ▷ We assume vector is zero-initialized
3:     R ← new Vector[Length(A)]
4:     pos ← 0
5:     for 0 ≤ j < Length(A) do
6:         C[A[j]] ← C[A[j]] + 1
7:     end for
8:     for 0 ≤ i < k + 1 do
9:         for pos ≤ r < pos + C[i] do
10:            R[r] ← i
11:        end for
12:        pos ← r
13:    end for
14:    return R
15: end function
```

The algorithms receives as input an unsorted array A and the maximum value stored in array A k. Arrays C and R are allocated and zero-initialized. A variable *pos* used to indicate the position in array R is also declared and initialized to zero.

The first *for* loop is responsible for counting frequencies of numbers in array A and storing them in array C. The second *for* loop is responsible for updating array R with the final sorted numbers.

4.104 Counting sort

Please read Section 8.2 (p.194–6) from the guide book

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 8

Key Concepts

- Identify the different approaches of different non-comparison sorting algorithms
- Implement different non-comparison sorting algorithms
- Calculate the time complexity of different non-comparison sorting algorithms

4.201 Radix sort

Radix sort employs a novel technique of sorting. It will sort its input numbers by splitting the number in digits and sort the digits in steps. First it will sort the least-significant digits, then then sort the next and the next until all digits are sorted.

One caveat of Radix sort is that the algorithm used to sort the digits **must** be a stable sort, meaning that once the least-significant digit is sorted, sorting the second-significant digit will preserve the original sorted order of the least-significant digits.

Counting sort, luckily, is a stable sort. We can use it for the digit sorting part of Radix sort.

Radix sort runs for as many iterations as there are digits in the numbers, i.e., if we're sorting 3-digit numbers, Radix sort will make three passes through numbers.

We can visualize Radix sort in figure 16.

Note that from one pass to another, the relative position of elements remain. In other words, after the first pass, 157 will always come before 457, even though we're running other passes of counting sort along the way.

Using counting sort to sort the digits results in Radix sort exhibiting the Time Complexity of $\Theta(d(N+k))$ where d is the number of digits, N is the number of numbers and k is the maximum value of digits. Moreover, $\Theta(N+k)$ is the time complexity of counting sort, therefore Radix sort has a time complexity of $\Theta(d \cdot g(N))$ where $g(n)$ is the time complexity of the sorting algorithm used to sort the digits.

One extra good aspect of Radix sort, is that it also works for sorting numbers containing decimal/fractional digits.

The basic minimal pseudocode from Radix sort is shown in listing 18.

4.203 Radix sort

Please read Section 8.3 (pp.197–9) from the guide book

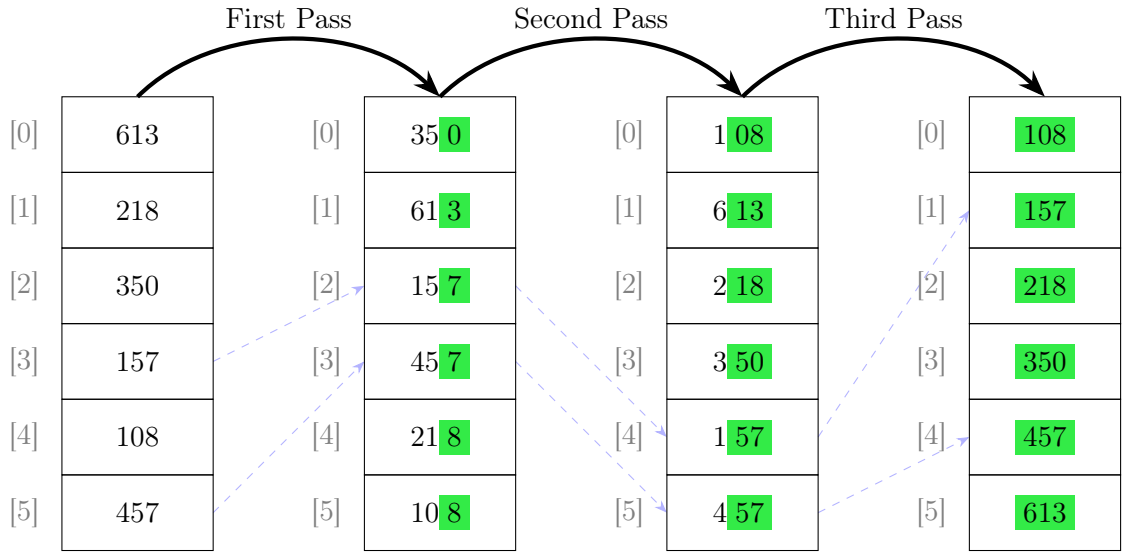


Figure 16: Radix Sort

Algorithm 18 Radix Sort

```
1: function RadixSort(A, d)
2:     for 0 ≤ j < d do
3:         CountingSort(A[j])                ▷ Sort A on digit j
4:     end for
5: end function
```

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

4.301 Bucket sort

A simple analogy for Bucket sort is when we want to sort a pile of coins. Generally, we don't compare one coin against another to sort them, we would pick a random coin and place it in a stack according to its value. After doing that to every coin, we would be left with n stacks of coins, all sorted.

Bucket sort works in a similar fashion. In the best case, we will have one copy of each number which will result in each number going to a different bucket. The array will, therefore, be sorted after n operations.

Figure 17 shows a depiction of this idea.

To calculate the buckets in figure 17 we divide each by 100 and take the floor of that, e.g. $\left\lfloor \frac{137}{100} - 1 \right\rfloor = 0$.

Bucket sort will behave well if the numbers in the input array are uniformly distributed. In case they aren't, we could fall into a bad case where all numbers fall into the same

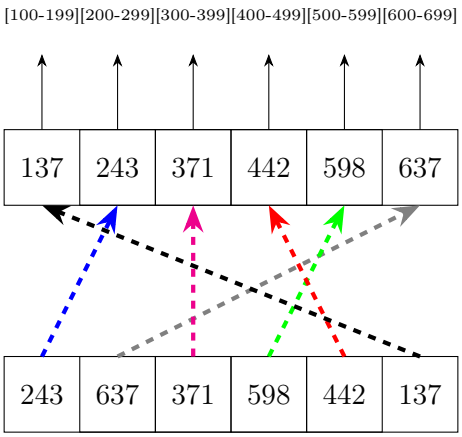


Figure 17: Bucket Sort

bucket.

In a normal situation where more than one number can fall into the same bucket, we must find a way to accomodate more than one number into the same position in the array. After that, we must sort the numbers inside every bucket before copying them back to the original array.

The first challenge can be solved with a linked list (future topic). The second challenge, i.e. sorting the elements in a linked list, can be solved with any other sorting algorithm. We could either use a comparison sort (insertion sort seems to be common) or a non-comparison sort, where we could apply counting sort, radix sort or recursively call bucket sort itself.

A simplified pseudocode of Bucket sort is shown in listing 19.

Algorithm 19 Bucket Sort

```
1:  function BucketSort(A, N, max)
2:      Buckets ← newArray[N]                          ▷ New array of size N
3:      for 0 ≤ i < N do
4:          Buckets[i] ← empty linked list             ▷ New list in each element of Buckets
5:      end for
6:      for 0 ≤ i < N do
7:          Buckets[⌊(A[i]·N)/(max+1)⌋] ← A[i]          ▷ Add A[i] to correct list
8:      end for
9:      for 0 ≤ i < N do
10:         Sort(Buckets[i])                           ▷ Sort each list
11:     end for
12:     for 0 ≤ i < N do
13:         Copy(Buckets[i], A)                        ▷ Copy list Buckets[i] back to A
14:     end for
15: end function
```

The time complexity of Bucket sort is $T(N) = C_4 \cdot N + T(\textit{Sorting linked lists})$.

4.303 Bucket sort

Please read Section 8.4 (pp.200–) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 9

Key Concepts

- Describe the different methods used to search for data
- Describe different collision resolution methods
- Implement a hash table with linear probing collision resolution.

5.001 What is hashing?

Hashing is the process of transforming a sequence of alphanumeric characters to a value. The algorithm responsible for the conversion process is referred to as a *Hash Function*.

This function receives a sequence of characters as input and returns a hash value. We can use any function that's able to transform the input into a value. For example, we can use the sum of the ASCII values of the input character:

$$\text{cat1} = 99 + 97 + 116 + 49 = 361$$

Hash functions are computationally cheap to apply, but computationally expensive to reverse given a hash value. Because of that, they are also referred to as *one-way functions*.

Hashing has important applications in security and information retrieval.

We can employ hashing on a password input to avoid transmitting the actual password outside of the users' computer. The example given above – i.e. that of summing the ASCII values of the input characters – is not robust enough for real-world applications. Different passwords containing the same characters in a different order will result in the same hash value. We want hash values to be unique. There are, however, much more secure hashing functions such as Secure Hash Algorithms (SHA).

Hashing functions can also be used for content verification. Assuming we will transmit sensitive information through the network, how can we guarantee that the data being sent has not been tampered with along the way?

While we can't stop a malicious agent from tampering with the data, we can provide means for detecting that the data hasn't been modified. We can achieve that by hashing the information before sending it and transmitting both the hash value and the information through the network. In case the information is modified somehow, the hash value won't match. More information about this application here.

Another important application of Hash functions is to enable *Fast Searching*. Say we want to verify if particular number is stored in an array. One could implement Linear

Search and check every position in the array; however, given the size of the input array, the worst case can take a long time.

Another option is to hash the numbers before inserting them into the array. The hash value will tell us where to store the number in the array. This means that when we want to search for the number, we can hash it again to get the position in the array where it should have been stored. This means that our search can be completed in $\mathcal{O}(1)$. This is the basic idea of a *Hash Table*.

5.003 Hash tables

To motivate the discussion of Hash Tables, we will define our searching problem as follows:

- the input data will be an array of numbers and a number to search for.
- the algorithm must give a result (true or false)

With this in mind, we will discuss three possible solution to the searching problem before introducing Hash Tables as a possible fourth solution.

The first solution to this problem is Linear Search. As shown in figure 18, this algorithm will visit every position of the array to check if the number we're looking for is there or not.

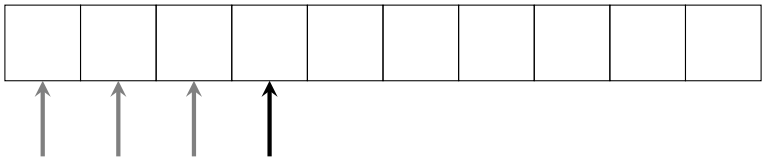


Figure 18: Linear Search

This algorithm will return either when we find the number we're looking for (in which case, it returns *true*) or we reach the end of the array (in which case, it returns *false*).

The pseudocode for this algorithm is shown in listing 20.

Algorithm 20 Linear Search

```
1: function LinearSearch(A, N, x)
2:     for 0 ≤ i < N do
3:         if A[i] = x then
4:             return true
5:         end if
6:     end for
7:     return false
8: end function
```

In the worst-case, this algorithm will run in $T(N) = \Theta(N)$. In the best-case, Linear Search is $T(N) = \Theta(1)$. In terms of memory, Linear Search is $S(N) = \Theta(1)$.

The second solution to this problem is Binary Search. Binary search requires the array to be sorted, but with that we can complete the search in $T(N) = \Theta(\log N)$ in worst-case and $T(N) = \Theta(1)$ in the best-case. In terms of memory, Binary Search has different behavior if it's a recursive version or iterative version.

In the recursive version we have $S(N) = \Theta(N)$ in the worst-case and $S(N) = \Theta(1)$ in the best-case. The iterative version is always $S(N) = \Theta(1)$.

The pseudocode for the recursive version of Binary Search is shown in listing 21.

Algorithm 21 Binary Search

```
1: function BinarySearch(A, low, high, x)
2:     if low > high then
3:         return −1
4:     end if
5:     mid ← ⌊(low+(high−low))/2⌋
6:     if A[mid] = x then
7:         return mid
8:     end if
9:     if A[mid] > x then
10:        return BinarySearch(A, low, mid − 1, x)
11:    end if
12:    if A[mid] < x then
13:        return BinarySearch(A, mid + 1, high, x)
14:    end if
15: end function
```

The third solution for the search problem is called *Direct Addressing*. The idea is to use the index of the array to represent a number. When we want to search for the number, we check if the array at the number's index contains a 0 or a 1. The array of 1s and 0s created for this algorithm is referred to as *Bit Vector*.

The pseudocode for direct addressing is shown in listing 22.

Algorithm 22 Direct Addressing

```
1: function DirectAddrSearch(B, x)
2:     return B[x]
3: end function
```

The time complexity of this solution is $T(N) = \Theta(1)$. Space complexity is $S(N) = \Theta(k)$ where k is the maximum value stored in the original array.

Finally, we reach to the *Hash Table* solution. Its behavior is similar to Direct Addressing, but requires far less memory. Similarly to Direct Addressing, we create another array to store the numbers. When the new array is uninitialized, it's filled with -1.

What we do, is that we transform each number into an index in the hash table using a hash function. To search for the number, the algorithm is similar to Direct Addressing,

but before indexing the table with argument, we run the argument through the same hash function as shown in listing 23.

Algorithm 23 Hash Table Search

```
1: function HashSearch(H, x)
2:     i ← h(x)
3:     if H[i] = x then
4:         return true
5:     end if
6:     return false
7: end function
```

Much like Direct Addressing, Hash Table Search has a time complexity of $T(N) = \Theta(1)$ and space complexity of $S(N) = \Theta(M)$ where M is the number of elements in the hash table.

The table table below summarizes the information.

Algorithm	**Time Complexity**	**Space Complexity**
Linear Search (iterative)	$\Theta(N)$ (worst) $\Theta(1)$ (best)	$\Theta(1)$
Binary Search (iterative)	$\Theta(\log N)$ (worst) $\Theta(1)$ (best)	$\Theta(1)$
Binary Search (recursive)	$\Theta(\log N)$ (worst) $\Theta(1)$ (best)	$\Theta(1)$ (best) $\Theta(\log N)$ (worst)
Direct Addressing	$\Theta(1)$	$\Theta(k)$ (k is max value)
Hash Table (We know the numbers)	$\Theta(1)$	$\Theta(M)$ (M numbers in hash table)

5.101 Collisions in hash tables

What problems can arise when we don't know the numbers to be placed in the Hash Tables? One of the possible problems is that of collisions, which is happens when more than one input number map to the same location in the Hash Table.

Collisions can't be avoided, but there are ways to deal with them.

One possible method is known as *Extend And Re-hash*. In essence, we must enlarge the hash table then come up with a new hashing function to re-hash all elements.

This process consists of 3 steps:

1. Enlarge the Hash Table

   This is so it can fit more items. It corresponds to the "extend" part of the method's name.

2. Modify the reduction function

The part of the hash function which we modify during step 2 is known as the reduction function because it *reduces* the input values to the range of values permitted by the hash table. When we increase the number of buckets in the hash function, we must update the reduction function.

3. Re-hash numbers already stored in the hash table

   Now that the reduction function has been updated, all numbers must be re-hashed and moved to their new correct locations. It corresponds to the "re-hash" part of the method's name.

This collision resolution method can be applied in two ways:

Reactive Executed after a collision occurs

Proactive Executed after utilisation of the hash table reaches a threshold

The second resolution method is known as *Linear Probing*. It consists of searching for the next available bucket to store the number in collision.

The third and final collision resolution method is known as *Separate Chaining*. It works by creating a chain of buckets at each position of the hash table. As numbers are added to buckets, they each occupy a different position in the chain. The advantage of chains is that they can grow and shrink on demand.

When it comes to asymptotic analysis, we have:

Best-case No collisions happen. In this case we have INSERT = $\Theta(1)$, SEARCH = $\Theta(1)$ and DELETE = $\Theta(1)$.

Worst-case Everything collides. In the case of Separate Chaining we have INSERT = $\Theta(1)$, SEARCH = $\Theta(N)$ and DELETE = $\Theta(N)$.

5.103 Hashing

Please read Chapter 11, pp.253–285 (except sections 11.3.3 and 11.5) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 10

Key Concepts

- Describe the different methods used to search for data
- Describe different collision resolution methods
- Implement a hash table with linear probing collision resolution.

5.301 End of Topic 5

We have reviewed one of the most common uses of Hashing: fast searching using a hash table.

During this week, we should be preparing our midterm assignment and nothing more.

Week 13

Key Concepts

- Describe linear data structures and its operations using pseudocode
- Understand array and linked list based implementations of stacks and queues
- Implement a sorted linked list.

7.001 Introduction to data structures

During the first half of the module, we focussed on the study of algorithms. During the second half, we focus on the study of data structures.

We will study the following data structures:

- Lists, Stacks, Queues
- Trees
- Heaps
- Graphs

A Data Structure is a container of data where data is organized in a specific way. As an example, lists are linear data structures because data is organized linearly, i.e. one element follows the other, like shown figure 19 below.

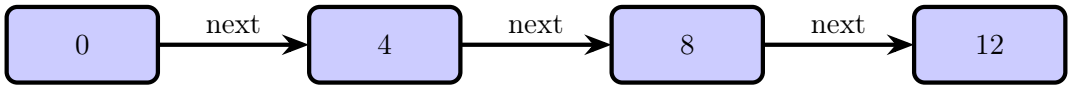


Figure 19: List

Trees and Heaps, on the other hand are organized in a hierarchical way, like the one shown in figure 20 below.

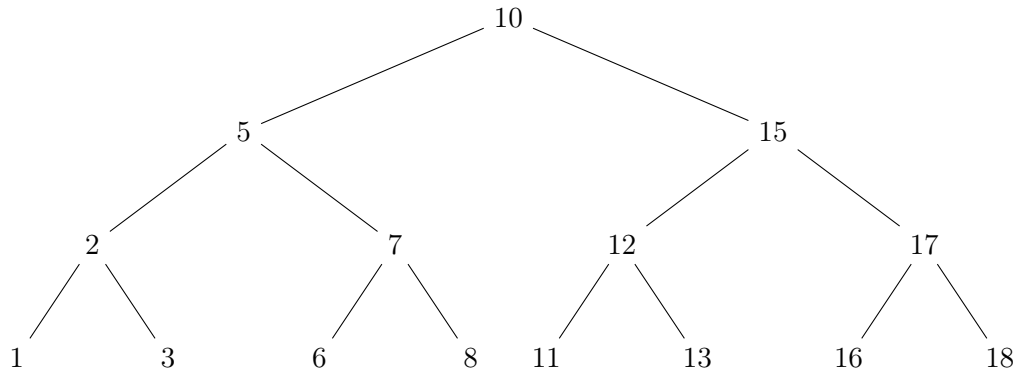


Figure 20: Tree

Every data structure has a set of operations associated with them which allows us to access and manipulate the data stored in them. As an example, Hash Tables have the operations *insert*, *search*, and *remove* associated with them.

7.003 Linked lists: Introduction

Much like a one-dimensional array, a Linked List is a linear data structure. However, unlike a one-dimensional array, a Linked List does not require a contiguous block of memory. Each element points to the next one using a pointer. This means that elements of a linked list can be located anywhere in memory so long as we update the *next* pointer of the previous element to point to the new one.

A *pointer* is, simply put, a memory address. Each element of the linked list must store, not only the data, but also a pointer to the next element, i.e. the memory address of the next element.

To access the next element we say that we *dereference* the pointer. This should be easy to understand, a memory address is a reference to a data, much like the index of a book is a reference the content we're looking for. *Dereferencing*, therefore, is accessing the memory address (or the page on the book) that contains the data we want to access.

Whenever we create a linked list, we must hold a reference to the first element (commonly referred to as the *head* of the list) otherwise we won't be able to recover any data. Moreover, the first time the list created, it contains nothing, therefore the *head* elements points to a special address known as *NULL*. The same *NULL* is used as a list terminator.

7.101 Linked lists: Insert operation

After our big overview of linked lists, we start studying its operations. The first operation we will study is insertion.

Before looking at the insertion pseudocode, let us define the representation of linkked lists in memory. Figure 19 is the simplest abstract representation of a list. It depicts the order of elements and arrows play the role of the pointer portion. Albeit being a

very good representation for depicting the contents of the list, it lacks important details necessary to understand the pseudocode.

Therefore, figure 21 shows an improved representation of the list and its elements.

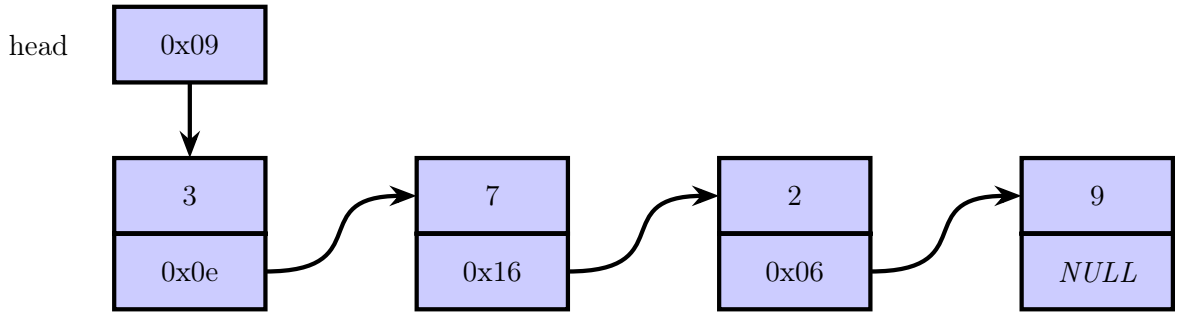


Figure 21: Improved List Representation

Inserting an element into a linked list involves traversing the linked list until we find the location in the list we want to insert the element.

As we can see, each element in a list is composed of two parts:

- A data part, shown as the top square
- A pointer part, shown as the bottom square

We will refer to each element in a list as a *Node*. In our pseudocode, when we refer to the data part we will write *node.data*, similarly the address of the next element will be written as *node.next*.[1]

We're ready to look at the pseudocode 24 of the *insert* function. The first step is to allocate a new Node to contain the new item. After the Node is initialized, we must link it to the list. This part changes depending on where we want to inser the element.

If we want to insert the element at the beginning of the list, first we must make the new Node point to the element currently pointed to by head. The final step would be change head to point to the new Node. Note that if we change the order of these two operations, i.e. changing head first, we would loost the reference to the node originally pointed to by head. In other words, we would loose the reference to number in figure 21.

Algorithm 24 Linked List Insert

```
1: function INSERT(head, x)
2:     newNode ← newNode(x)
3:     newNode.next ← head
4:     head ← newNode
5: end function
```

[1] We're differing from the lecture, which uses the -> operator, simply because the . is more common in LaTeX. It's also a little easier to type.

Inserting to the beginning of the list has a time complexity $T(N) = \Theta(1)$ because we will always a constant number of operations to carry out the insertion.

There are two other methods of inserting into a list:

Inserting at the end In this case, we will always traverse the list until we find an node whose *next* pointer is *NULL*, this means we have found the end of the list, then we make this node's *next* point to the new Node.

Inserting at an arbitrary position In this case, we must traverse the list until we find our arbitrary position containing node *pos*, modify *newNode.next* to point to *pos.next*, modify *pos.next* to point to *newNode*.

7.103 Linked lists: Delete operation

The second operation available in linked list is the *delete* operation. It gives us the ability to remove a node from a list. Essentially, we will reverse the steps done in listing 24.

The first thing we need to do is *bypass* the node to be removed, that is *prev.next* = *tmp.next*. It should look similar to figure 22 below:

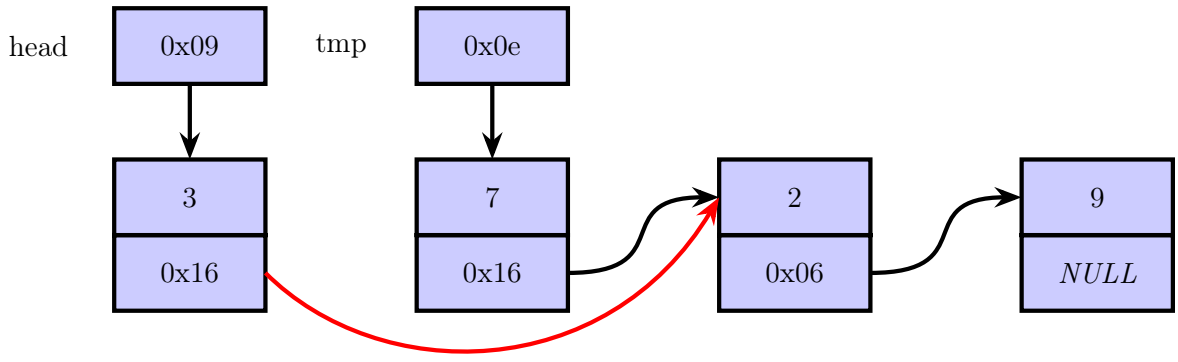


Figure 22: Linked List Delete

After the node to be removed is *bypassed* we can free the memory originally allocated for it. Listing 25 shows the pseudocode for deleting an item from the list.

7.105 Linked lists: Summary

The complexity of the main operations associated with a linked list, insert, delete, and search, is described below.

Starting with insert, its complexity depends on where we're inserting the new node. There are three cases, as below:

Beginning $T(N) = \Theta(1)$

Algorithm 25 Linked List Delete

```
1: function DELETE(list, x)
2:     Nodetmp ← head
3:     Nodeprev ← NULL
4:     if tmp = NULL then
5:         return                                   ▷ Nothing to delete
6:     else
7:         if tmp.data = x then
8:             head ← tmp.next
9:             return list
10:        else
11:            prev ← tmp
12:            tmp ← tmp.next
13:            while tmp ≠ NULL do
14:                if tmp.data = x then
15:                    prev.next ← tmp.next
16:                    return list
17:                end if
18:                prev ← tmp
19:                tmp ← tmp.next
20:            end while
21:        end if                  ▷ If we get here, x was not found in the list
22:    end if
23: end function
```

End $T(N) = \Theta(N)$

Arbitrary Location There are two possibilities

Best case $T(N) = \Theta(1)$

Worst case $T(N) = \Theta(N)$

In the case of delete, we have to look at the best case and worst case. The best case happens when the node to be deleted is the first node in the list and the worst case happens when the node to be deleted is at the end of the list.

Best case $T(N) = \Theta(1)$

Worst case $T(N) = \Theta(N)$

Search will always have the same complexity as the Linear Search algorithm. This is because we must visit node n before we get the address of node $n + 1$. Therefore the time complexity of Searching a linked list is always the same as Linear Search algorithm, which is:

Best case $T(N) = \Theta(1)$

Worst case $T(N) = \Theta(N)$

Note that it's the same time complexity as deleting a node. The reason for this is that in order to delete a node, we must first search for it.

There are a few types of linked lists which we can build:

Doubly Linked List Each node points to the next and previous nodes.

Circular Linked List The last node points to the first node, instead of *NULL*.

7.108 Linked lists

Please read the definition of data structures on p.9 and then section 10.2 (pp.236–41) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 14

Key Concepts

- Describe linear data structures and its operations using pseudocode
- Understand array and linked list based implementations of stacks and queues
- Implement a sorted linked list.

7.201 Stacks: Introduction

The *Stack* is another example of a linear data structure. It behaves much like a stack (e.g. of books) in the physical world. Objects can only be inserted at the top of the stack and removed from the top of the stack.

Because of this behavior, the insertion and removal operations have special names for them, *push* and *pop* respectively. Two other operations are *isEmpty()* (which returns **true** when the stack is empty) and *peek* (which returns the value of the element at the top).

Whenever we want to query the content of the stack, only the top of the stack is accessible. To get the next element, we must, first, *pop* the top element.

While they seem limited at first, stacks have very important applications in Computer Science. For example, Stacks are used to check for matching curly braces ({) when parsing source code[1], implementation of Reverse Polish Notation calculators[2], procedure calls[3], and countless other applications.

7.203: Stacks: Implementation

Stacks are so common that they're part of the standard library of virtually every programming language.

Because a stack is a linear data structure, it can be implemented on top of other linear data structures, such as an array of a linked list. Arrays are peculiar because they have

[1] Whenever a { is found, an element is pushed onto the stack. Whenever the matching } is found, that element is popped. If we reach the end of the statement with a non-empty stack, we have an error.

[2] Whenever an operand is entered, push it onto the stack. Whenever an operator is entered, pop the correct amount of operands off the stack, execute the operation and push the result back onto the stack.

[3] Whenever a different procedure must be called, context of the calling procedure (i.e. the content of CPU registers) is pushed onto the stack. Upon returning from the called procedure, context is popped from the stack and restored onto respective registers.

a static size, therefore we either end up with unused memory (and that's wasteful) or we run out of space, in which case we could take one of three different paths:

1. Stop accepting new elements;
2. Allocate bigger array and copy elements from small to big array before inserting new element; or
3. Corrupt memory due to overflow of the stack space[4]

Whenever we want to use an array or a linked list to implement a stack, we must enforce the access rules of the stack. Using the example of arrays, a push would be implemented with the algorithm shown in listing 26 (note that *top* is initialized to -1 to signify an empty stack):

Algorithm 26 Stack: Push

```
1: function PUSH(x)
2:     top ← top + 1
3:     A[top] ← x
4: end function
```

This doesn't take into consideration the fact that we can run out of space in the array. As mentioned before we can stop accepting new elements (see 27), or allocate a bigger array and move elements over (see 28) or do nothing (and introduce a possible bug).

Algorithm 27 Stack: Push with block

```
1: function PUSH(x)
2:     if top = SIZE(A) − 1 then
3:         return
4:     end if
5:     top ← top + 1
6:     A[top] ← x
7: end function
```

Algorithm 28 Stack: Push with extend

```
1: function PUSH(x)
2:     if top = SIZE(A) − 1 then
3:         EXTENDANDCOPY(A)
4:     end if
5:     top ← top + 1
6:     A[top] ← x
7: end function
```

[4] `https://en.wikipedia.org/wiki/Stack_buffer_overflow`

In the case of 28, the time complexity of the *push* operation grows from $\Theta(1)$ to $\Theta(N)$ because we must copy all elements over to the new, bigger array before inserting a new element.

Moving on to the *pop* operation, its algorithm is shown in listing 29. All operations in *pop* take a constant time, therefore its time complexity is $\Theta(1)$.

Algorithm 29 Stack: Pop

```
1: function Pop
2:     if top = −1 then
3:         return
4:     end if
5:     top ← top − 1
6: end function
```

The next operation is *peek*, shown in listing 30. Similarly to *pop*, all operations in *peek* take a constant time, which makes its time complexity $\Theta(1)$.

Algorithm 30 Stack: Peek

```
1: function Peek
2:     if top = −1 then
3:         return −1
4:     end if
5:     return A[top]
6: end function
```

The last operation is *isEmpty*, shown in listing 31. Much like the previous two operations, all statements in *isEmpty* take a constant time and its time complexity is also $\Theta(1)$.

Algorithm 31 Stack: isEmpty

```
1: function IsEmpty
2:     if top = −1 then
3:         return true
4:     end if
5:     return false
6: end function
```

The linked list implementation of stacks is analogous to that of the array implementation. In listings 32, 33, 34, 35 we show linked list versions of *push*, *pop*, *peek*, and *isEmpty* respectively. Every operatio has time complexity of $\Theta(1)$.

Algorithm 32 Stack: Push (Linked List)

```
1: function PUSH(x)
2:     n ← newNode
3:     n.data ← x
4:     n.next ← top
5:     top ← n
6: end function
```

Algorithm 33 Stack: Pop (Linked List)

```
1: function POP
2:     if top = NULL then
3:         return
4:     end if
5:     top ← top.next
6: end function
```

Algorithm 34 Stack: Peek (Linked List)

```
1: function PEEK
2:     if top = NULL then
3:         return −1
4:     end if
5:     return top.data
6: end function
```

Algorithm 35 Stack: isEmpty (Linked List)

```
1: function ISEMPTY
2:     if top = NULL then
3:         return true
4:     end if
5:     return false
6: end function
```

7.301 Queues: Introduction

The queue is the final linear data structure that we will study. A queue may seem like witchcraft at first  , however it's far from it. It behaves exactly like a queue in real life: people queue by standing at the end of the queue and are served from the front of the queue. Once served, they are removed from the queue.

We say that queues behave in a **FIFO** (standing for *First In, First Out*) manner.

The operations associated with a queue are:

1. Enqueue
2. Dequeue
3. Peek
4. isEmpty

Much like stacks, queues have several applications. Many of which refer to processing requests in the order they come.

7.303 Queues: Array-based implementation

Similarly to a stack, a queue can be implemented with arrays or linked lists. Many of the concerns with array-based stacks, apply to array-based queues as well.

Considering array-based implementation for now, we initialize the *front* (sometimes referred to as *head*) and *tail* pointers to -1 to signify an empty queue. To enqueue an element, we move the *tail* ahead by 1 position. When the queue is initialy empty, *front* must also be moved by 1. We should the algorithm in listing 36.

Algorithm 36 Queue: Enqueue

```
1: function ENQUEUE(x)
2:     if (tail + 1) mod N = front then
3:         return −1
4:     end if
5:     if ISEMPTY then
6:         front ← 0
7:         tail ← 0
8:     else
9:         tail ← (tail + 1) mod N
10:    end if
11:    A[tail] ← x
12: end function
```

The dequeue operation is analogous to enqueue. It is shown in listing 37.

Algorithm 37 Queue: Dequeue

```
1: function DEQUEUE
2:     if ISEMPTY then
3:         return
4:     end if
5:     if front = tail then
6:         front ← −1
7:         tail ← −1
8:     else
9:         front ← (front + 1) mod N
10:    end if
11: end function
```

Algorithm 38 Queue: Peek

```
1: function PEEK
2:     if front = −1 then
3:         return −1
4:     end if
5:     return A[front]
6: end function
```

Peek is, also, a very simple algorithm. All we have to do is return the value of the element at the *front* if the list is not empty. The algorithm is shown in listing 38.

The algorithm for *isEmpty* is the simplest of them all. We just need to return **true** or **false** depending if the list is empty or not. Listing 39 shows the algorithm.

Algorithm 39 Queue: isEmpty

```
1: function ISEMPTY
2:     if front = −1 then
3:         return true
4:     end if
5:     return false
6: end function
```

The time complexity of all operations is $\Theta(1)$.

7.305 Queues: List-based implementation

To implement a queue with a linked list, we need one extra pointer for the tail. Without it, one of the operations would have to traverse the entire list before executing its role (enqueueing or dequeueing).

Traversing the list has a time complexity of $\Theta(N)$. With the second pointer, we can guarantee operations in $\Theta(1)$.

Listings 40, 41, 42, and 43 show the operations *enqueue*, *dequeue*, *peek* and *isEmpty* respectively. All operations have a time complexity of $\Theta(1)$.

Algorithm 40 Queue: Enqueue (Linked List)

```
1:  function ENQUEUE(x)
2:      n ← newNode
3:      x.data ← x
4:      if front = NULL ∧ tail = NULL then
5:          front ← n
6:          tail ← n
7:      else
8:          tail.next ← n
9:          tail ← n
10:     end if
11: end function
```

Algorithm 41 Queue: Dequeue (Linked List)

```
1:  function DEQUEUE
2:      if front = NULL ∧ tail = NULL then
3:          return
4:      end if
5:      if front = tail then
6:          front ← NULL
7:          tail ← NULL
8:      else
9:          front ← front.next
10:     end if
11: end function
```

7.307 Stacks and queues

Please read Section 10.1 (pp.232–5) from the guide book:

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Algorithm 42 Queue: Peek (Linked List)

```
1: function PEEK
2:     if front = NULL ∧ tail = NULL then
3:         return −1
4:     else
5:         return front.data
6:     end if
7: end function
```

Algorithm 43 Queue: isEmpty (Linked List)

```
1: function ISEMPTY
2:     if front = NULL ∧ tail = NULL then
3:         return true
4:     end if
5:     return false
6: end function
```

Week 15

Key Concepts

- Understand how to implement a tree
- Describe and trace different types of binary tree traversals using pseudocode
- Describe and trace binary search tree operations using pseudocode.

8.001 Trees: Introduction

Moving on to the study of non-linear data structures, we will study Trees. A tree is a data structure that looks like the one shown in figure 23.

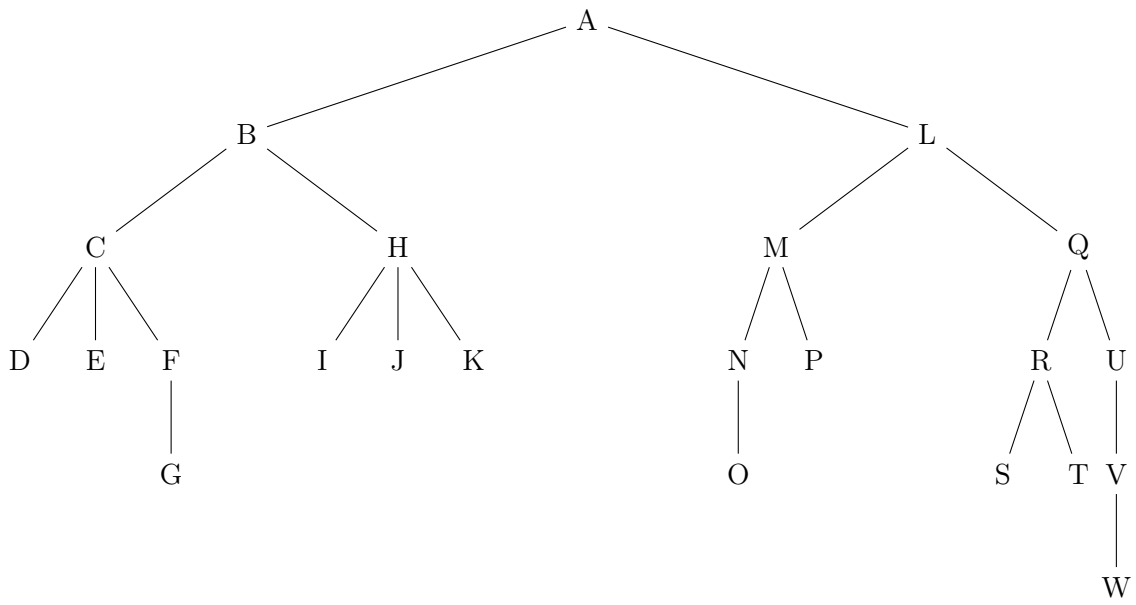


Figure 23: A Generic Tree Data Structure

The node at the very top of the tree (node A in figure 23) is called the **root** of the tree. The nodes at the very end $(D, E, G, I, J, K, O, P, S, T, W)$ are called the **leaves** of the tree, the edges from one node to another are called the **branches** of the tree.

We also have a parent-child relationship between nodes. We say that node A is the parent of both B and L, while H is the parent of I, J, and K.

All nodes from a particular node back to the root are referred to as the ancestors of the node. For example, in figure 24 we marked all ancestors of W with the color red. Similarly, all the node below a node are called their descendants.

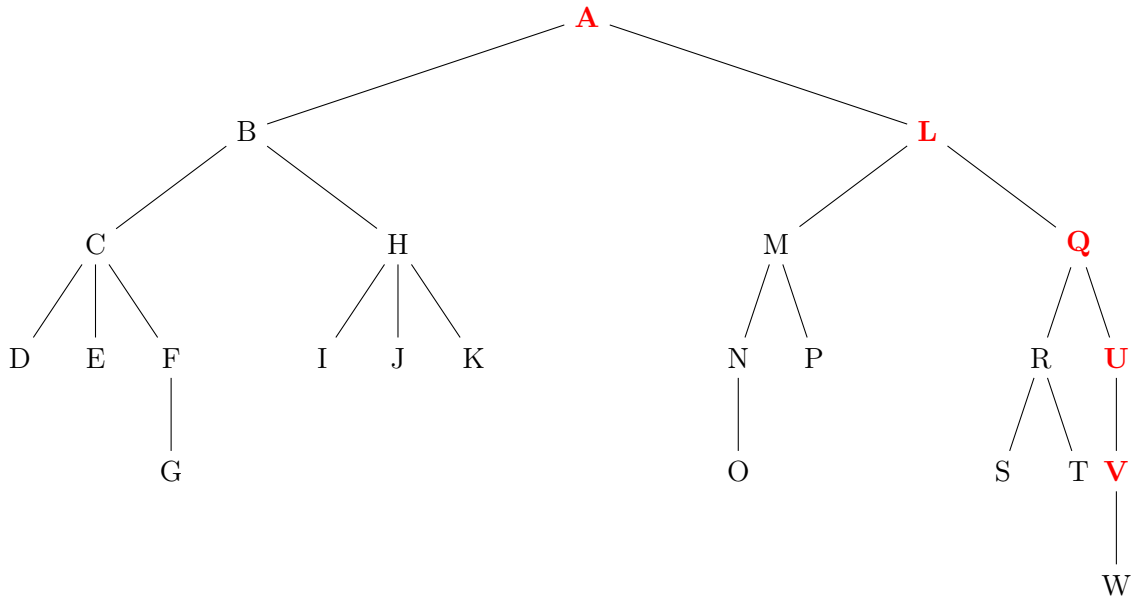


Figure 24: The Ancestors of W

Every node in the tree also has a depth, which is the number of branches between the root and that particular node (e.g. the depth of W is 5). A tree also has a height, which is measured from the root of the tree to deepest leaf. The tree in figure 23 has a height of 5.

When the number of children of every node in a tree is constrained to a maximum of two, that tree is known as a *Binary Tree*, likewise when the number of children is constrained to a maximum of three, that tree is called a *Ternary Tree*. When the number of children is constrained to a maximum of four, the tree is called a *Quaternary Tree* and so on. More generally, we refer to a *m-ary Tree*[1] when the number of children for every node is constrained to a maximum of m.

When every node in the tree has exactly m children, we say that we have a full m-ary tree. For example in figure 25 we see a Full Binary Tree of 3 levels.

In the special case of Binary Trees, because it only has two children, we refer to those as *Left Child* and *Right Child*. We can also talk about the *Left Subtree* and *Right Subtree*. In figure 26 we show the right subtree of A.

Note that trees have a rather recursive nature. After we choose a path (left or right) we have another tree to process.

[1] Unary, Binary, Ternary, Quaternary, Quinary, Senary, Septenary, Octonary, Novenary, Denary, Undenary, Duodenary, Ternidenary, and so on come from Latin cardinal numbers.

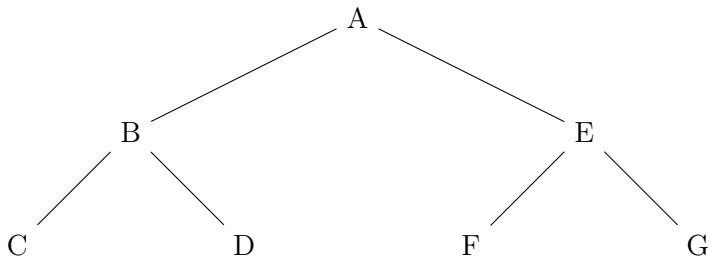


Figure 25: Full Binary Tree

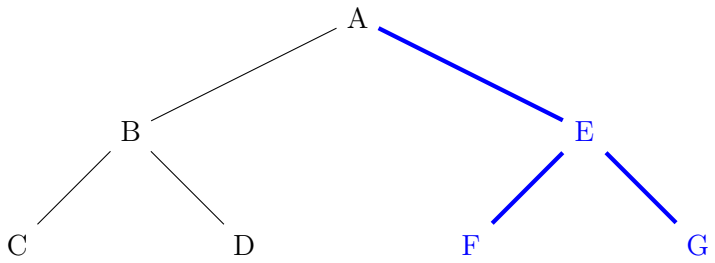


Figure 26: Full Binary Tree

8.003 Binary trees: Implementation

During this lecture we look into two possible implementations of binary trees: using pointers (similar to linked lists) and using arrays.

When we use pointers, we must define our *Node* type to contain three pieces of information:

value the data to be stored in the node

left a pointer to the left child

right a pointer to the right child

A visualization of such a tree is shown in figure 27 below.

Another of implementing a Binary Tree, is using an array. When choosing this form, the element at level 0 is stored at index 0; i.e. the root of the tree is in index 0 of the array.

The elements at level 1, therefore, will be stored in indices 1 and 2 of the array and, elements of level 2 will be in indices 3, 4, 5, and 6 and so on. To indicate the absence of a node in a particular index, we store a number that's outside the range of acceptable numbers. For example, if we're dealing with non-negative numbers, storing -1 could indicate the absence of a node.

In general, elements at level k are stored using 2^k positions starting at index $2^k - 1$.

The benefit of using an array, is that we will use less memory; however there's a drawback that arrays have a fixed size.

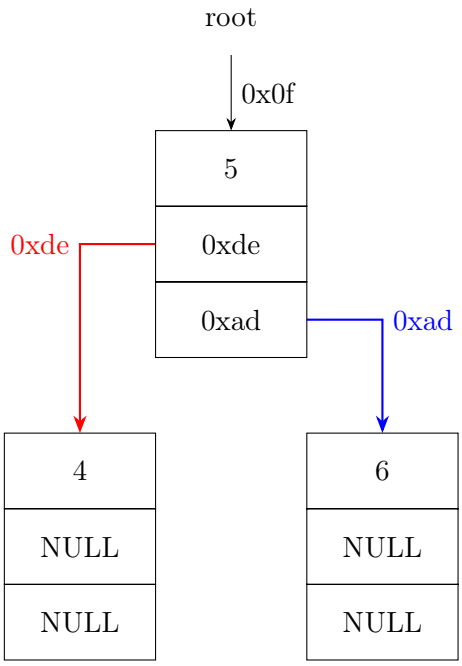


Figure 27: Binary Tree Visualization

8.101 Binary tree traversal: Introduction

Traversing a binary tree is required during insertion, deletion, and searching of a specific node. There are two main approaches to tree traversal:

Breadth-First Traversal Visit all sibblings of a node before visiting their descendants

Depth-First Traversal Visit all descendants of a node before visiting their sibblings

To give an idea of the differences of both traversal methods, follow the numbers in the nodes of the trees in figures 28 and 29 in ascending order.

There are three different types of depth-first traversal:

Pre-order Root node is the first node to vist

In-order Root node is visited in the middle of the traversal

Post-order Root node is the last node to visit

This means that our example of Depth-First Traversal in figure 29, is an example of Pre-order Depth-First Traversal.

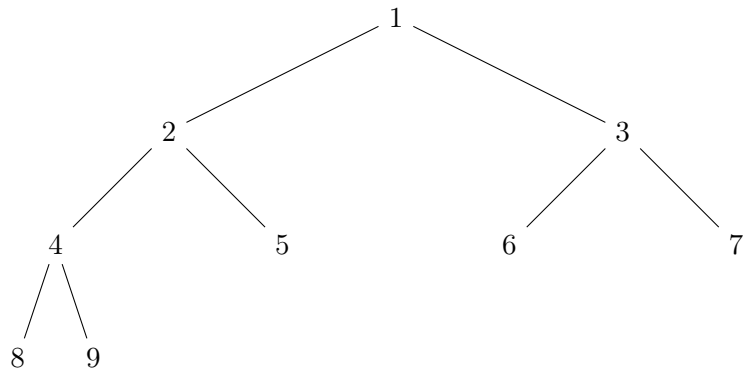


Figure 28: Breadth-First Traversal

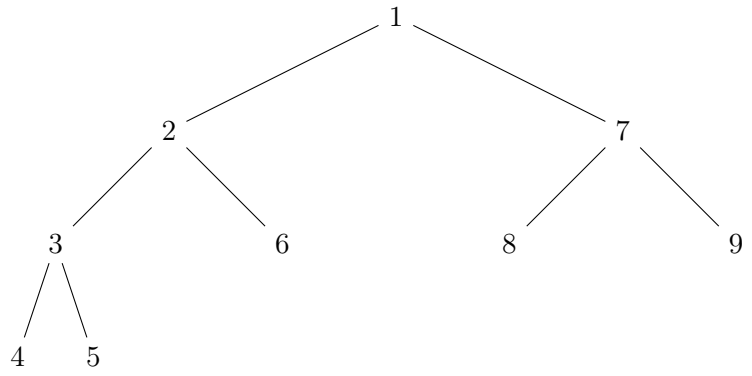


Figure 29: Depth-First Traversal

8.102 Depth-first traversal

Listings 44, 45, and 46 refer to Pre-order, In-order, and Post-order traversal respectively. After reading through those, it should be clear what is meant by the order in which we visit the root of the tree.

8.104 Breadth-first traversal

The only missing point is Breadth-First Traversal. This algorithm is simpler in iterative form than in recursive form[2], so we study the iterative version.

To illustrate the behavior we want is that every time we visit a node, we add its children to a queue of *pending* nodes, then we just follow the list from left to right, removing nodes as we visit them.

Listing 47 shows the pseudocode for Breadth-First Traversal.

[2]Every recursive algorithm can be transformed into iterative and *vice versa*.

Algorithm 44 Pre-Order Depth-First Traversal

```
1: function PreOrder(T)
2:     if ¬IsEmpty(T) then
3:         Visit(Root(T))
4:         PreOrder(Left(T))
5:         PreOrder(Right(T))
6:     end if
7: end function
```

Algorithm 45 In-Order Depth-First Traversal

```
1: function InOrder(T)
2:     if ¬IsEmpty(T) then
3:         InOrder(Left(T))
4:         Visit(Root(T))
5:         InOrder(Right(T))
6:     end if
7: end function
```

Algorithm 46 Post-Order Depth-First Traversal

```
1: function PostOrder(T)
2:     if ¬IsEmpty(T) then
3:         PostOrder(Left(T))
4:         PostOrder(Right(T))
5:         Visit(Root(T))
6:     end if
7: end function
```

Algorithm 47 Breadth-First Traversal

```
1: function BreadthFirst(root)
2:     Q ← new Queue
3:     EnqueueIf(Q, root)
4:     while ¬IsEmpty(Q) do
5:         t ← Peek(Q)
6:         Visit(t)
7:         EnqueueIf(Q, Left(t))
8:         EnqueueIf(Q, Right(t))
9:         Dequeue(Q)
10:    end while
11: end function
12: function EnqueueIf(Q, t)
13:    if ¬Null(t) then
14:        Enqueue(Q, t)
15:    end if
16: end function
```

Week 16

Key Concepts

- Understand how to implement a tree
- Describe and trace different types of binary tree traversals using pseudocode
- Describe and trace binary search tree operations using pseudocode.

8.201 Binary search trees (BSTs)

In order to motivate the topic of *Binary Search Trees*, let us first review Linked Lists.

Linked Lists are linear data structures setup in a way that the n^{th} node points to the $(n+1)^{th}$. In practice, this means that we must visit every node in order; i.e. because each node can be anywhere in memory, there's no way to find the midpoint of the list because we don't have a reference for it readily available.

If we wanted to apply Binary Search on a linked list, we would have to traverse the entire list just to find its midpoint. Clearly, this is far wasteful and completely defeats the purpose of employing Binary Search.

A solution to this problem is to use a tree structure such as the one shown in figure 30.

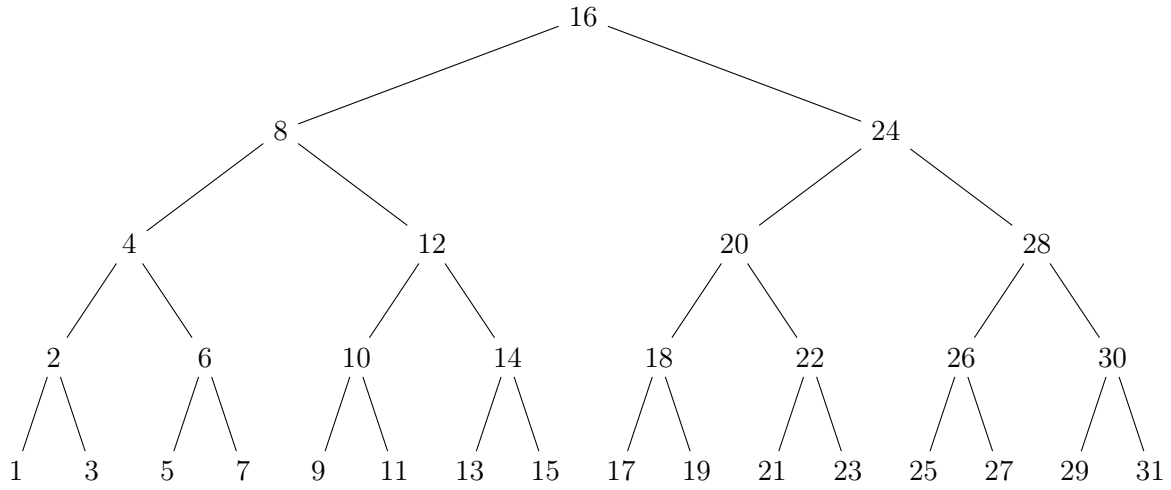


Figure 30: Binary Search Tree

As we can see from figure 30 all nodes to the left of any node, are less than that node and all nodes to its right are greater. Every Binary Tree that guarantees these two requirements is referred to as a *Binary Search Tree*.

It's up to us to guarantee that, during insertion, nodes are inserted in a sorted manner, so that all nodes to the left are smaller and all nodes to right are greater.

8.203 BST: Insert

Given a new node, how do we insert it into an existing BST without breaking its requirements?

Before looking at the pseudocode, let's think about what are the possible cases that can arise while trying to insert a new node into the BST. They are:

Root is *NULL* This is easy, we create the node, assign the value and return.

Value is less than Root's value This is easy, we insert at the left node.

Value is greater than Root's value This is also easy, we insert at the right node.

Essentially, whenever *Root* is *NULL*, we have reached our base case. The other two cases just reduce the problem to a smaller problem that aproximates the base case. Listing 48 shows the pseudocode for BST insertion.

Algorithm 48 Binary Search Tree Insertion

```
1: function INSERT(root, x)
2:     if root = NULL then
3:         n ← newNode
4:         n.data ← x
5:         root ← n
6:     else if x < root.data then
7:         INSERT(root.left, x)
8:     else
9:         INSERT(root.right, x)
10:    end if
11: end function
```

This algorithm exploits the recursive nature of the BST in order to perform insertion. The time complexity of BST Insertion is $\Theta(\log N)$ where N is the number of nodes in the BST.

8.301 BST: Search

Searching, is very similar to Insertion, with the difference that instead of adding an element in the tree, we're only looking for it.

Re-using all three cases and updating for search operation we have:

Root is *NULL* This is easy, just return false

Value is less than Root's value This is easy, it must be at the left

Value is greater than Root's value This is also easy, it must be at the right

We have to add a new case. What should we do if the value is exactly the one we're looking for? We should return true. Therefore, our updated cases are:

Root is *NULL* This is easy, just return false

Value is equal to Root's value This is easy, just return true

Value is less than Root's value This is easy, it must be at the left

Value is greater than Root's value This is also easy, it must be at the right

Listing 49 shows the pseudocode:

Algorithm 49 Binary Search Tree Searching

```
1: function SEARCH(root, x)
2:     if root = NULL then
3:         return false
4:     else if x = root.data then
5:         return true
6:     else if x < root.data then
7:         SEARCH(root.left, x)
8:     else
9:         SEARCH(root.right, x)
10:    end if
11: end function
```

Ideally, with a perfectly balanced tree, searching will take $\Theta(\log N)$, however, if the tree is not balanced, this may not be the case.

8.303 BST: Delete

Deleting a node from a BST is a more complex affair. Using our figure 30, what happens if we remove the number 12 from the tree? Which node should take its place, 10 or 14? How do we connect them together back into the main tree?

The simple case happens when the node we want to delete has no children, i.e. it's a leaf node. In that case, we simply delete the node and return.

When has a single child, it's also somewhat simple. We delete the node and replace its child in its location.

When the node has two children, however, things get more involved. In essence, we delete the node and replace with the maximum value in the left subtree or the minimum

value in the right subtree. To illustrate the steps, figures 31, 32, 33, and 34, show the steps necessary to remove node 12 from our tree.

To summarize, there are three steps to perform the delete operation:

1. Find the **minimum value of the left** subtree or the **maximum value of the right** subtree
2. Copy the value found to the location of the node to be deleted
3. Delete the minimum/maximum value

The pseudocode for this operation is shown in listing 50.

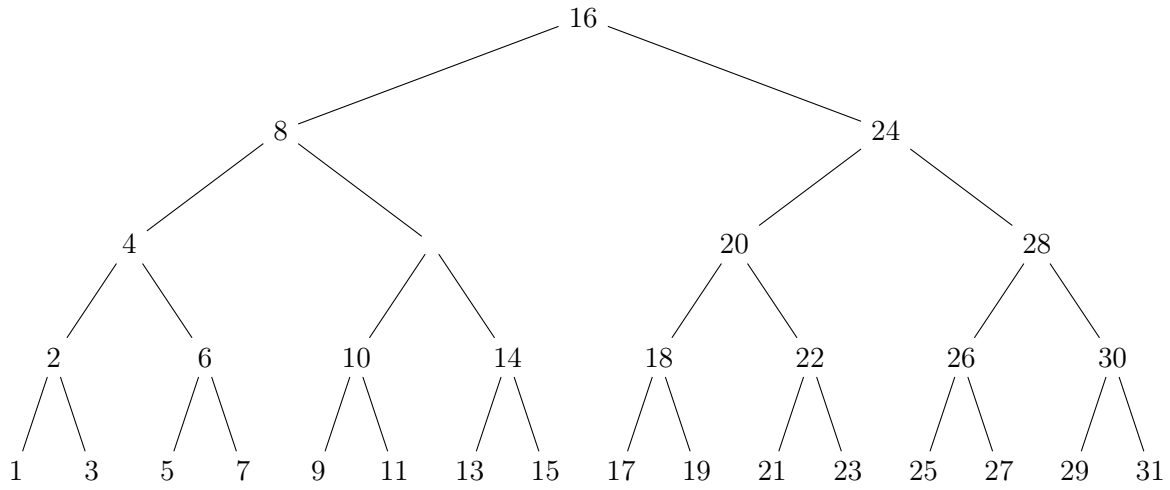


Figure 31: Binary Search Tree Deletion

8.305 Binary search trees

Please read the introduction to Chapter 12 and Sections 12.1 (pp.286–8), 12.2 (pp.289–92) and 12.3 (pp.294–8) from the guide book:

- Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Algorithm 50 Binary Search Tree Deletion

```
1: function DELETE(root, x)
2:     if root = NULL then
3:         return NULL
4:     else if x < root.data then
5:         root.left ← DELETE(root.left, x)
6:     else if x > root.data then
7:         root.right ← DELETE(root.right, x)
8:     else
9:         if root.left = NULL ∧ root.right = NULL then
10:            root ← NULL
11:            return root
12:        else if root.left = NULL then
13:            root ← root.right
14:            return root
15:        else if root.right = NULL then
16:            root ← root.left
17:            return root
18:        else
19:            tmp ← GETMINRIGHT(root.right)
20:            root.data ← tmp.data
21:            root.right ← DELETE(root.right, root.data)
22:        end if
23:    end if
24: end function
```

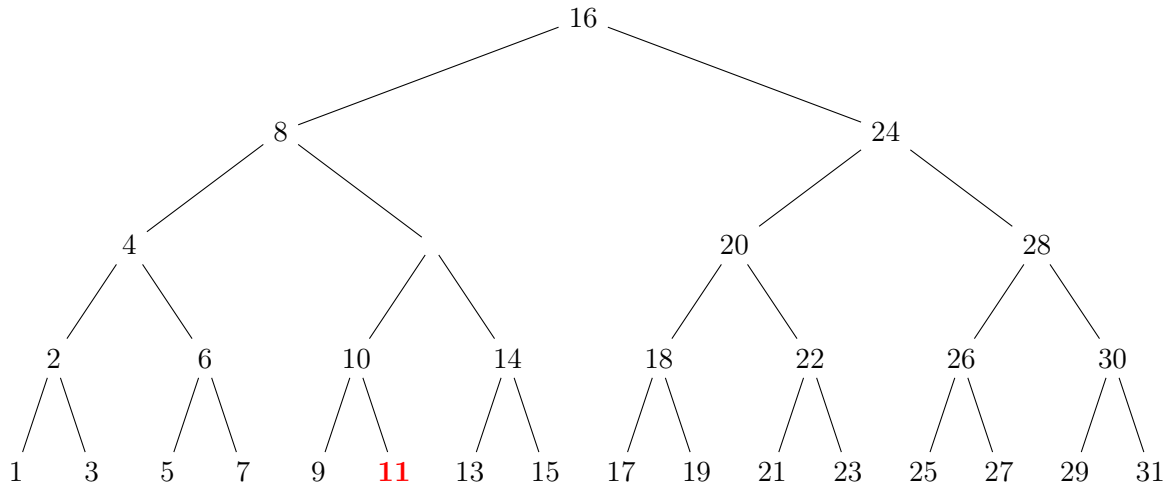


Figure 32: Binary Search Tree Deletion

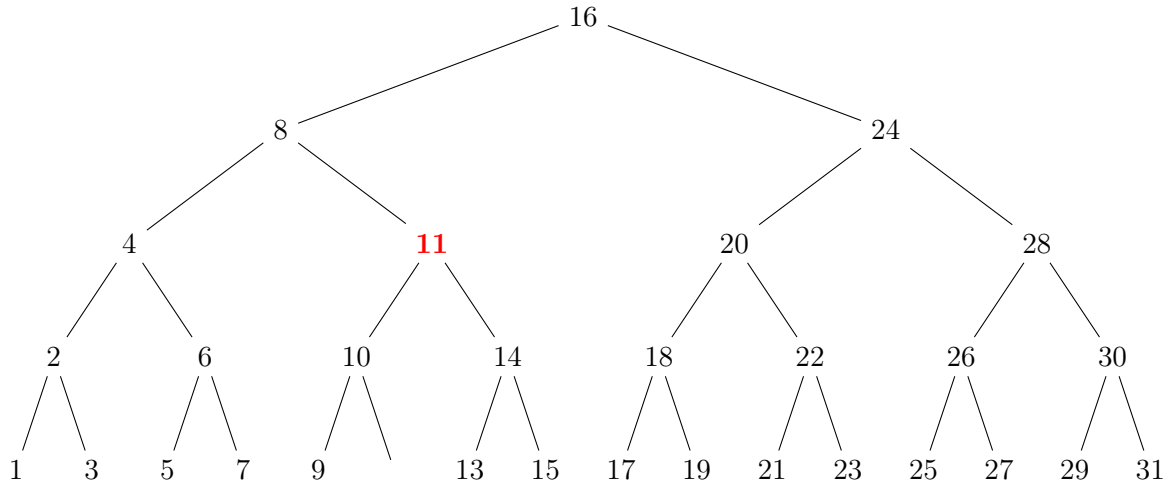


Figure 33: Binary Search Tree Deletion

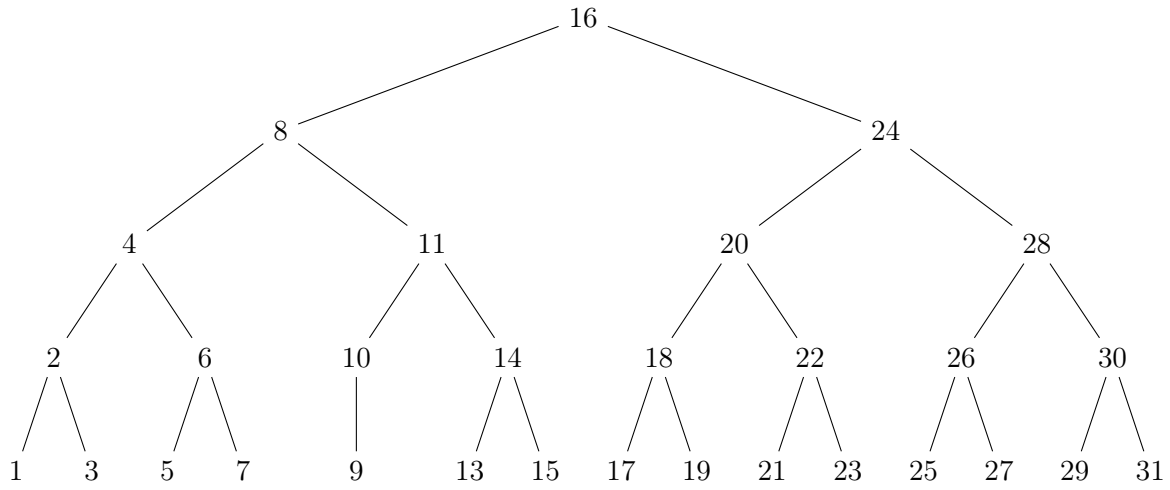


Figure 34: Binary Search Tree Deletion

Week 17

Key Concepts

- Check heap and shape properties
- Describe heap operations using pseudocode
- Implement heapsort using a heap

9.001 Heaps: Introduction

A heap is a Tree-like data structure that satisfies two properties:

Heap Property The value of the parent must be in relation to the value of its children. If the heap is a max-heap, then the value stored in any parent is always greater than the values stored in its children. In case we have a min-heap, the property is analogous.

Shape Property The tre must be a perfect triangle shape or a triangle shape plus a left-aligned rectangle in the base. The perfect triangle refers to a tree with all levels full and the rectangle refers to the last level not being full.

The tree shown in figure 35 is an example of a max-heap.

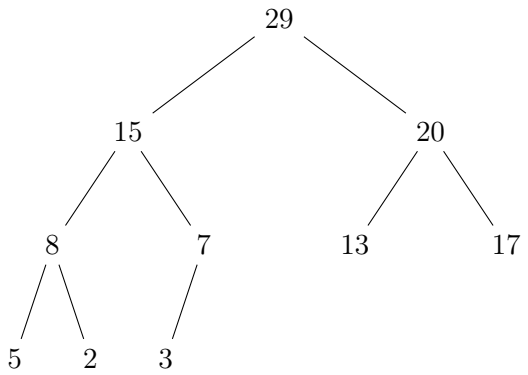


Figure 35: Heap

In figure 35 above, we have an example of a Binary Heap, where every node has at most 2 children. However, any tree can be a heap, meaning we have Ternary Heaps, Quaternary Heaps, or *n-ary* Heaps, where nodes in the tree have at most n children.

9.003 Heaps: Implementation

As discussed previously, trees can be implemented using Arrays of memory pointers. In the case of memory pointers, every node in a tree is composed of three fields:

Data the value of the node

Left a pointer to the left sub-tree

Right a pointer to the right sub-tree

A visualization of a tree implemented using memory pointers is shown in figure 27. This way of representing a tree is referred to as an *Explicit Representation*, because the memory addresses of the children are explicitly set in the parent node.

When using arrays, position 0 in the array stores the root node, or level 0. Level 1 follows in the positions 1 and 2, while level 2 takes up positions 3, 4, 5, and 6. In general, elements of level k take positions $2^k - 1$ to $2^{k+1} - 2$. This way of representing a tree is referred to as an *Implicit Representation*, because the memory addresses of the nodes are implicitly given by their position in the array.

We will use the array representation which introduces the challenge of finding an expression to index parent and children given a position k in the array. To help our understanding, we build a table showing the relationships:

Node	**Parent**	**Left**	**Right**
0	–	1	2
1	0	3	4
2	0	5	6
3	1	7	8
4	1	9	10
5	2	11	12
6	2	13	14
7	3	15	17
...	...	...	...
k	$\lfloor \frac{k-1}{2} \rfloor$	$2k+1$	$2k+2$

Based on the algebraic expressions we extracted we can implement our acessors as shown in code listing 51, 52, and 53.

Algorithm 51 Parent

```
1: function PARENT(k)
2:     return ⌊(k−1)/2⌋
3: end function
```

We use these basic operations in the next lecture to manipulate the data stored in a Heap.

Algorithm 52 Left

```
1: function LEFT(k)
2:     return 2k + 1
3: end function
```

Algorithm 53 Right

```
1: function RIGHT(k)
2:     return 2k + 2
3: end function
```

9.005 Heaps: Insert (element by element)

The insertion operation lets us build a heap from scratch. We will use a max-heap to illustrate the algorithm, but it works for min-heap as well with minor modifications. We use an auxiliary variable *size* to count the number of elemnts currently in the heap. This variable also makes the job of finding the next available position much easier as it will always be given by $size - 1$

We start with an empty heap as shown in figure 36 below:

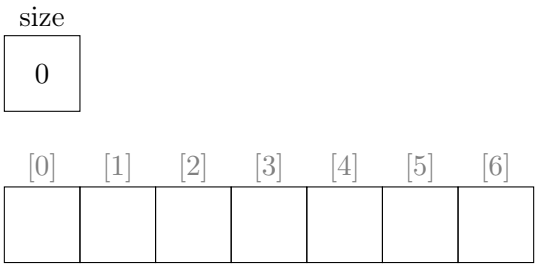


Figure 36: Empty Heap

If we want to insert the number 23 on the heap, there's nothing to do other than inserting 23 at the root of the heap. Therefore, our heap changes like shown in figure 37 below:

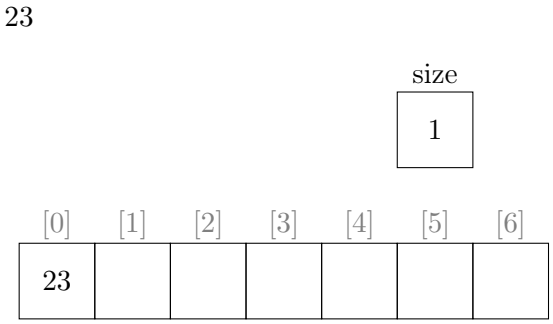


Figure 37: One Element

Assuming we want to insert the number 14, we have to make sure the shape property is maintained. Therefore, we must insert the element at the left child, as shown in figure 38:

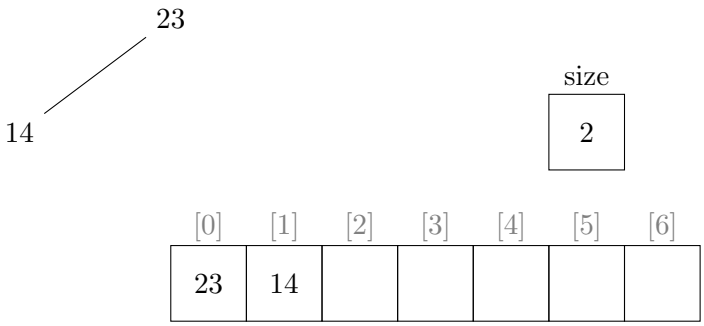


Figure 38: Two Elements

After insertion, we must check if the heap property is satisfied. Is $23 > 14$, then answer is true, therefore we move on. The next insertion is the number 37. To satisfy the shape property, we insert 37 at the next available space: the right child of 23, shown below in figure 39:

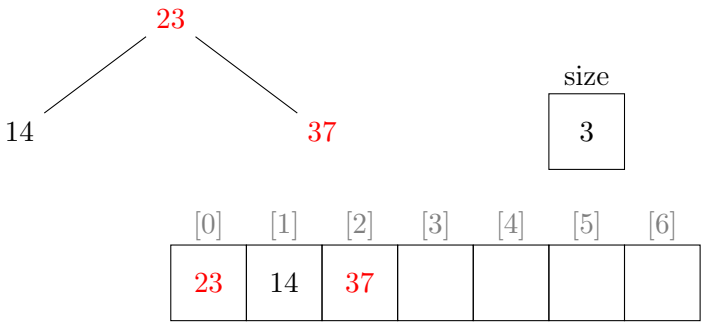


Figure 39: Three Elements

This time, however, the heap property of the max heap is not maintained because $23 < 37$. We must swap 23 and 37 to satisfy the heap property, which results in the heap shown in figure 40 below:

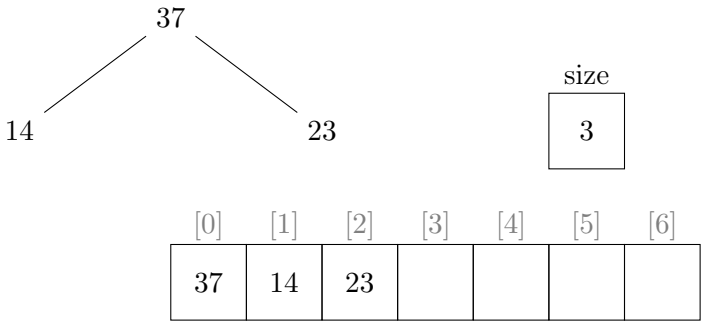


Figure 40: Three Elements (Swapped)

Let's insert the number 42 into our heap. To comply with the shape property, we must insert it into the next available space in our heap as shown in figure 41 below:

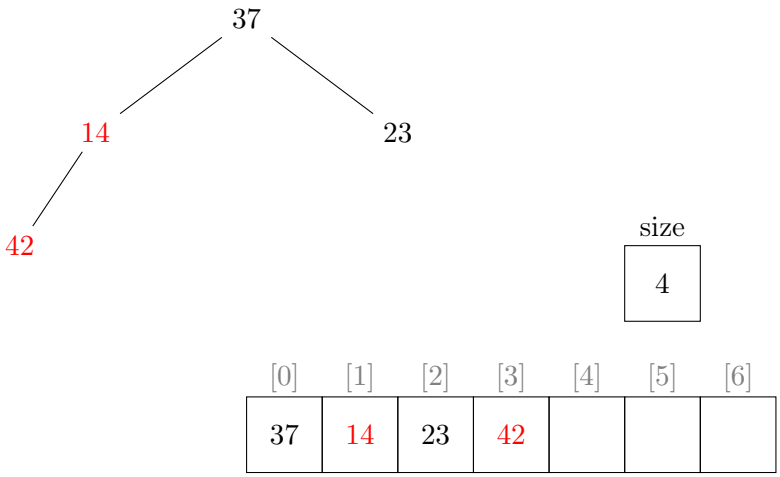


Figure 41: Four Elements

Because the heap property is violated, we swap 42 and 14, resulting in the heap shown in figure 42 below:

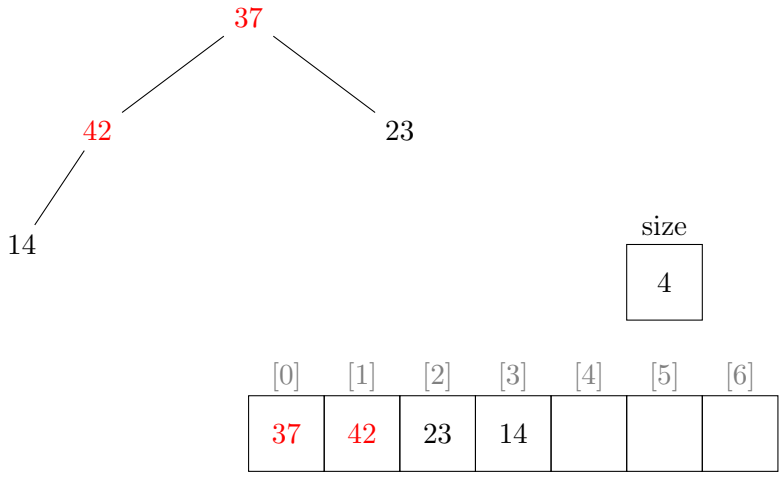


Figure 42: Four Elements (Swap 1)

We solved the first violation of the heap property, but introduced another one. We must swap 37 and 42. The result is shown in figure 43 below:

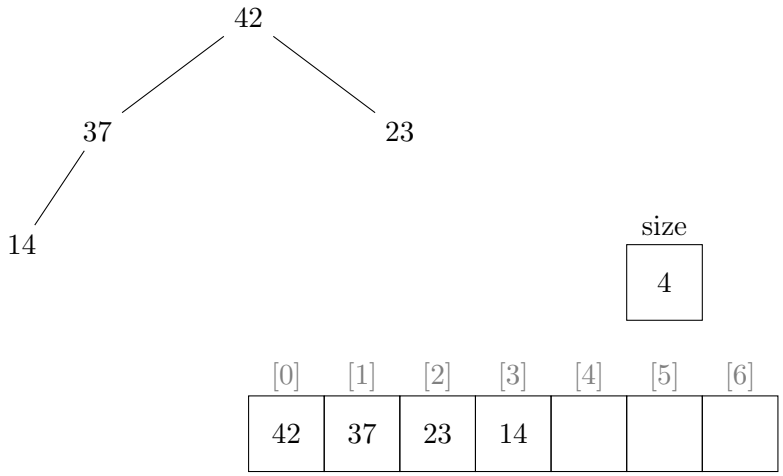


Figure 43: Four Elements (Swap 2)

In the general case, the checks for the heap property stop when the node being bubbled up has no parents (meaning it's the root of the heap) or the value of the parent already satisfies the heap property.

The pseudocode of the insert operation is given in listing 54 below:

9.007 Insert: Deletion (extract maximum)

The converse of the insert operation is the delete operation. Unlike regular trees, in a Heap we always remove the root node. This means that in a max-heap, we remove the

Algorithm 54 Heap Insertion

```
1: function INSERT(heap, k)
2:     pos ← heapSize
3:     heap[pos] ← k
4:     heapSize ← heapSize + 1
5:     while pos > 0 ∧ heap[PARENT(pos)] < heap[pos] do
6:         SWAP(heap[PARENT(pos)], heap[pos])
7:         pos ← PARENT(pos)
8:     end while
9: end function
```

maximum element using an operation called *Extract Max* and in a min-heap, we remove the minimum element using an operation called *Extract Min*.

While it's possible to devise an algorithm that is capable of removing any given element from the heap, that algorithm will not be efficient due to the nature of the partial sorting of the heap. Moreover, heaps were designed to support a sorting algorithm known as *Heap Sort*, where the operation *Extract Max* (or *Extract Min*) is of great importance.

After removing the root node we need to choose a new root element that maintains both properties of the heap. To illustrate the problem, we will use the Heap shown in figure 44.

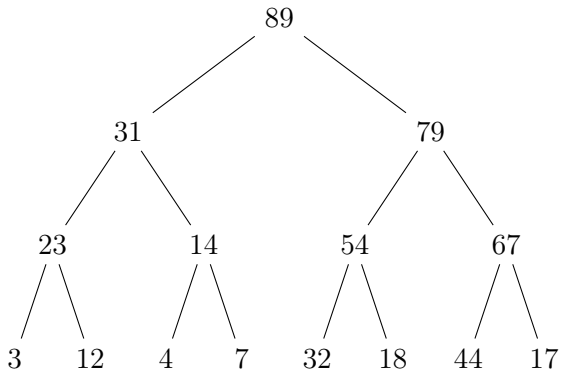


Figure 44: Max Heap

To carry out the deletion process, we will use a process similar to the deletion of a node in a Binary Search Tree. That is, instead of deleting the number 89, we will copy the contents of another node over 89 and delete that node instead.

The easiest node to be deleted is the most recently added node in the heap as shown in figure 45. Therefore, we copy that node over 89, which results in the heap shown in figure 46.

After overwritting the root, we can delete the node storing number 17, as shown in figure 47.

However, while the resulting heap complies with the shape property, it violates the

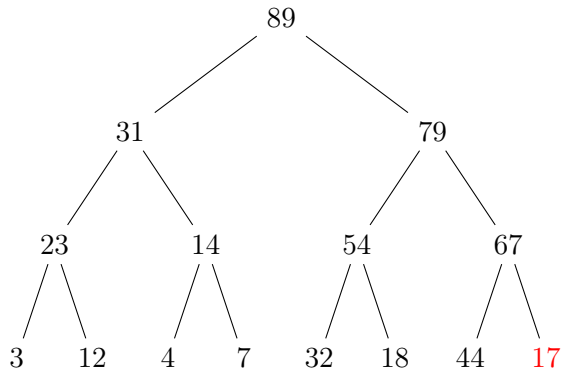


Figure 45: Max Heap Deletion: Choose Node

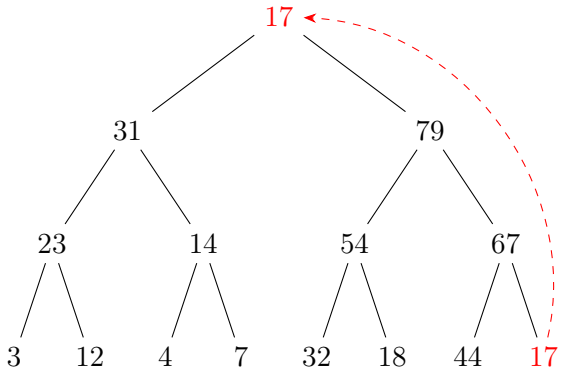


Figure 46: Max Heap Deletion: Overwrite Root

heap property. We must *Heapify*[1] the heap to correct that violation. Therefore, we start by swapping the root with its largest child, as shown in figure 48.

The resulting heap is still not a max-heap, so we heapify the subtree rooted at the number 17, as shown in figure 49.

The heap is still not a max-heap, therefore we heapify the subtree rooted at node 17, as shown in figure 50.

The resulting heap satisfies both max-heap properties and is, therefore, a valid heap. Deletion of the largest element is complete.

The pseudocode for the *Extract Max* operation is shown in listing 55 below:

The pseudocode for *Max Heapify* is as shown in listing 56:

[1]To *Heapify* a tree is to apply swaps until it satisfy the heap property. For a max-heap, we will swap the root with its largest child and repeat the process for the child's subtree.

Algorithm 55 Extract Max

```
1: function ExtractMax(heap)
2:     max ← heap[0]
3:     heap[0] ← heap[heapSize − 1]
4:     heapSize ← heapSize − 1
5:     MaxHeapify(heap, 0)
6:     return max
7: end function
```

Algorithm 56 Max Heapify

```
1: function MaxHeapify(heap, root)
2:     largest ← IndexLargestNode(root)
3:     if largest ≠ root then
4:         Swap(heap[largest], heap[root])
5:         MaxHeapify(heap, largest)
6:     end if
7: end function
8: function IndexLargestNode(heap, root)
9:     if root ≥ heapSize then
10:        return root
11:    else if heap[Left(root)] > heap[Right(root)] then
12:        return Left(root)
13:    else
14:        return Right(root)
15:    end if
16: end function
```

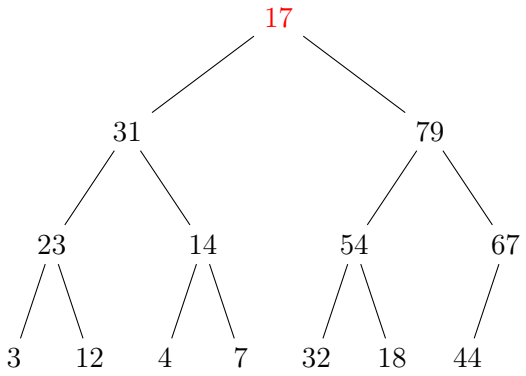


Figure 47: Max Heap Deletion: Remove Leaf

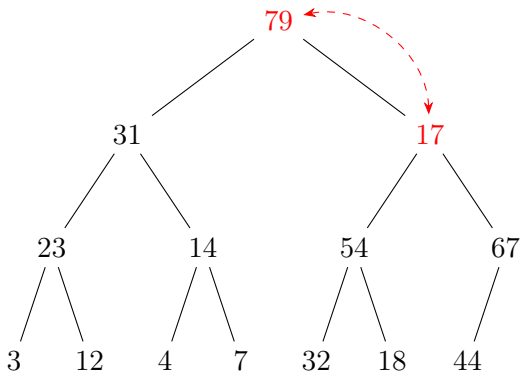


Figure 48: Max Heap Deletion: Heapify Pass 1

9.101 Heaps: Build in place

In case we already have a Binary Tree, we can transform it into a Max Heap. We are assuming that the Binary Tree satisfies the Shape Property but not the Heap Property.

One method for converting a Binary Tree into a Max Heap is by running Breadth First Search on the Binary Tree and upon visiting each node of the BST, call the Max Heap Insert operation to insert the element into a secondary array. Because we use a secondary array for the heap, we refer to this as an *Out-Of-Place Algorithm.*

Another method for conversion, is to rearrange the array with successive swaps. In this case, we're using an *In-Place Algorithm.*

To illustrate, we will use the BST shown in figure 51:

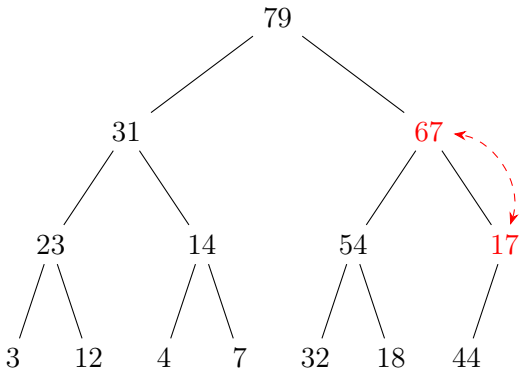


Figure 49: Max Heap Deletion: Heapify Pass 2

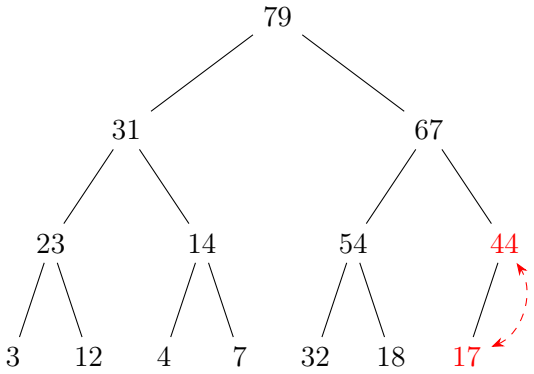


Figure 50: Max Heap Deletion: Heapify Pass 3

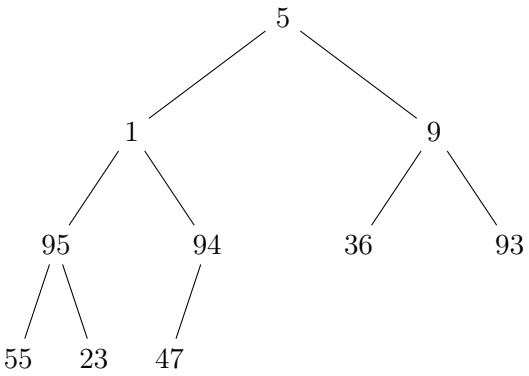


Figure 51: Heapify BST

We start with the level right above the leaves. Starting at the node 94, there's nothing to be done, because 94 is already largest than 47.

Moving to the node 95, there's nothing to do there either because 95 is larger than both its children. This level is done, so we move one level up, starting at node 9. We

must heapify that because the subtree rooted at 9 is not a max heap, as shown in figure 52:

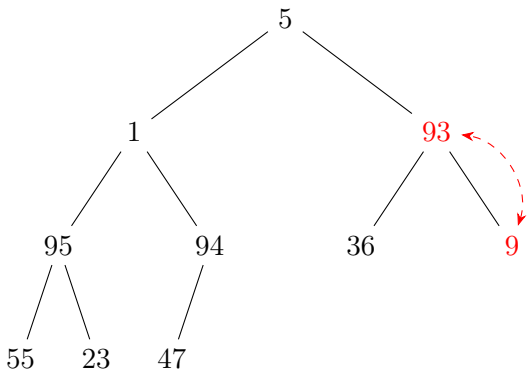


Figure 52: Heapify BST

We move one the subtree rooted at 1 and heapify it as shown in figure 53.

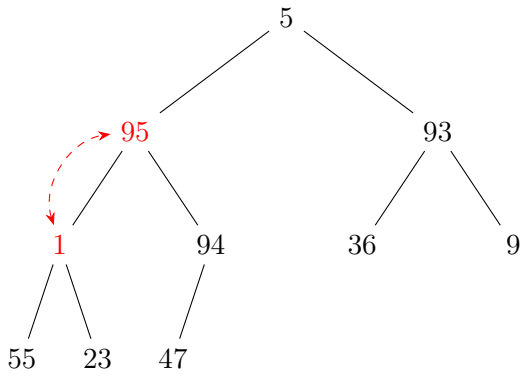


Figure 53: Heapify BST

Now the subtree rooted at 1 needs to be heapified as well, as shown in figure 54.

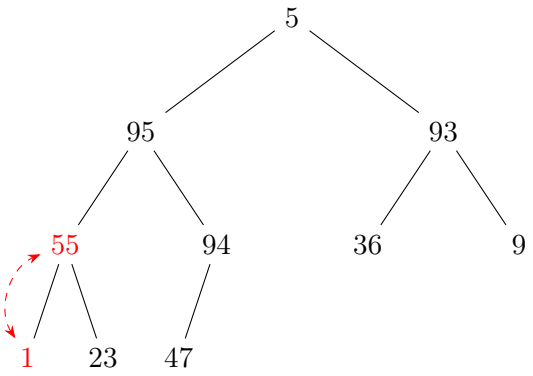


Figure 54: Heapify BST

This level is now complete and we move on to the next level containing the subtree rooted at 5. We heapify it as shown in figure 55.

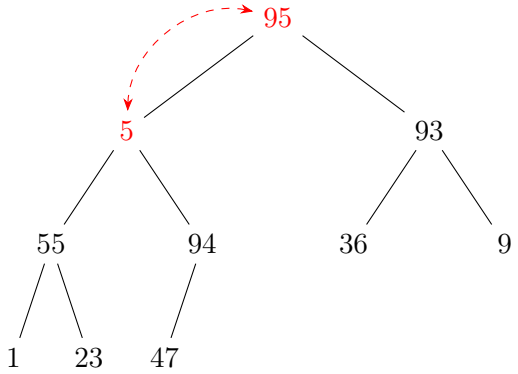


Figure 55: Heapify BST

Continuing the process, the subtree rooted at 5 needs to be heapified as shown in figure 56.

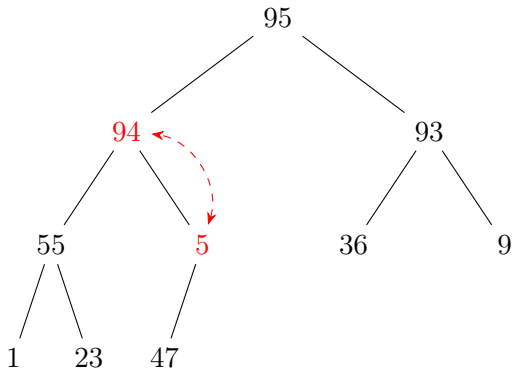


Figure 56: Heapify BST

To continue, the subtree rooted at 5 must be heapified, as shown in figure 57.

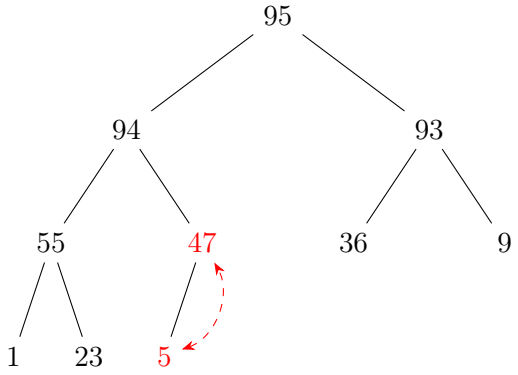


Figure 57: Heapify BST

With this final pass of heapify, we have converted the original BST into a Max Heap resulting in the Max Heap shown in figure 58.

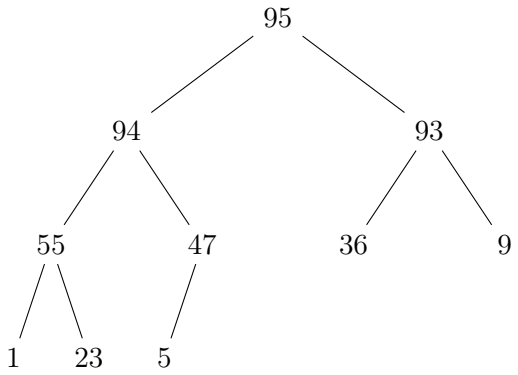


Figure 58: Heapify BST

The pseudocode for the *Build Max Heap* operation is shown in listing 57 below:

Algorithm 57 Build Max Heap

```
1: function BuildMaxHeap(A)
2:     heapSize ← A.length
3:     for ⌊heapSize/2⌋ > j ≥ 0 do
4:         MaxHeapify(A, j)
5:     end for
6: end function
```

9.103 Heapsort

Given an unsorted array, HeapSort produces a sorted array. The first step for doing so is to use the operation *Build Heap In Place*. Once we have a heap, we swap the max element with the last element in the array. With that, we consider the maximum element to be in its right place and don't have to look at it anymore.

We keep following this process until the heap is empty. At the end we will have an array sorted in ascending order.

The pseudocode for Heap Sort is shown in listing 58 below:

Algorithm 58 Heap Sort

```
1: function HeapSort(A)
2:     BuildMaxHeap(A)
3:     while heapSize > 0 do
4:         i ← heapSize − 1
5:         A[i] ← ExtractMax(A)
6:     end while
7:     return A
8: end function
```

Because extracting the maximum element of a heap is such a cheap operation, heaps are also used to implement Priority Queues.

9.105 Heapsort's complexity

To calculate the time complexity of the Heapsort algorithm, let's annotate the pseudocode:

To analyze the time complexity of *Build Max Heap*, let's annotate its pseudocode:

With *Extract Max* taking time proportional to $\log n$, we can conclude that *Build Max Heap* carries the time complexity, which we consider to be $\Theta n \log n$.

Algorithm 59 Heap Sort

```
1: function HeapSort(A)
2:     BuildMaxHeap(A)                          ▷ A
3:     while heapSize > 0 do                    ▷ n
4:         i ← heapSize − 1                     ▷ c1
5:         A[i] ← ExtractMax(A)                 ▷ B
6:     end while
7:     return A                                 ▷ c2
8: end function
```

Algorithm 60 Build Max Heap

```
1: function BuildMaxHeap(A)
2:     heapSize ← A.length                      ▷ c1
3:     for ⌊heapSize/2⌋ < j ≤ 0 do              ▷ n
4:         MaxHeapify(A, j)                     ▷ log n
5:     end for
6: end function
```

9.107 Heaps, heapsort and priority queues

Please read Chapter 6, pp.151–69, from the book:

- Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

Accessible from here.

Week 19

Key Concepts

- Explain and apply the basic concepts of computer networking
- Describe TCP/IP model and layers in the model
- Identify network protocols in each layer.

10.001 Graphs: Introduction

A graph is a visual representation of an interconnected system using circles (known as nodes, or vertices) and lines (known as links or edges). Figure 59 below shows a graph representation of the Knogisberg Bridges:

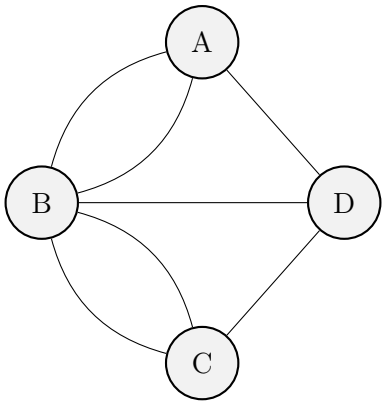


Figure 59: Konigsberg Bridges

Graphs are powerful tools used to analyze and solve problems in a wide range of interconnected systems. For example, we can use graphs to model interactions between book or movie characters, or to model computer networks.

Graphs can classified as Directed or Undirected Graphs. Directed graphs have arrows connnecting vertices. These arrows signal the direction of the relationship between two vertices. Undirected graphs, however, have symmetrical relationships. The symmetrical relationship is represented with a line.

There is another way of classifying graphs. Graphs can be either weighted or unweighted. Weighted graphs have a *cost* associated with the relationship between vertices. This could be used, for example, to model the length of a road from one junction to another. Unweighted edges is the same as having every road with the same length.

There are, then, 4 main types of graphs to study: Weighted Directed Graphs, Weighted Undirected Graphs, Unweighted Undirected Graphs, and Unweighted Directed Graphs.

Graphs can have different topologies as well, the most common are:

Bus Nodes form a line

Ring The last node is connected to the first node

Tree Follows the shape of a Tree Data Structure

Manhattan Regular grid

Mesh Arbitrary grid

10.003 Graphs: Representations

Using the graph depicted in figure 60 below, we study various ways of representing the graph data structure.

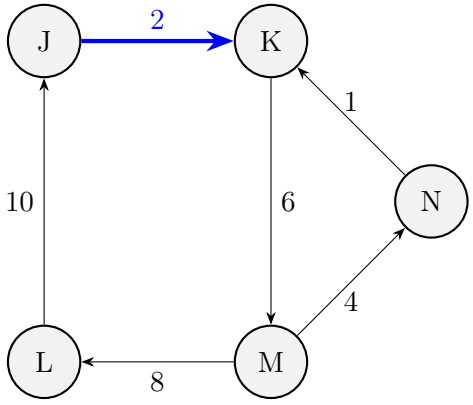


Figure 60: Graph

The first representation is known as *Edge List*. In this representation, each edge is stored as triplet containing the starting node, ending node and the weight of that edge. The edge list for the graph shown in figure 60 must contain 6 triplets because the graph contains 6 different edges. The edge list can be represented as shown below:

$$E = \{(J, K, 2), (K, M, 6), (L, J, 10), (M, L, 8), (M, N, 4), (N, K, 1)\};$$

The Edge List representation has a space complexity $\Theta(E)$ where E is the number of edges.

Our next possible representation is called the *Adjacency Matrix*.

$$\begin{array}{c} \begin{matrix} J & K & L & M & N \end{matrix} \\ \begin{pmatrix} \infty & 2 & \infty & \infty & \infty \\ \infty & \infty & \infty & 6 & \infty \\ 10 & \infty & \infty & \infty & \infty \\ \infty & \infty & 8 & \infty & 4 \\ \infty & 1 & \infty & \infty & \infty \end{pmatrix} \begin{matrix} J \\ K \\ L \\ M \\ N \end{matrix} \end{array}$$

An adjacency matrix is always a square matrix, i.e. has the same number of rows and columns, this number is equal to the number of nodes in the graph. Inside the matrix itself, there are as many number as there are edges in the graph. Each of the numbers corresponds to the weight of the edge going from the *row* to the *column*. For example, in the matrix above, the weight of the edge going from J to K is 2.

In unweighted graphs, normally the number 1 is used. We also employ the symbol ∞ to signify that there is no edge going from that *row* to that *column*. The number -1 is another common choice for this scenario.

The space complexity of the adjacency matrix is $\Theta(V^2)$ where V is the number of vertices (or nodes) in the graph. In situations where the number of vertices is significantly higher than the number of edges, the adjancency matrix becomes inefficient in terms of memory usage because most positions will be used to signal an absent link.

The third way of representing graphs is known as *Adjancency List*. This solves the problem of excessive memory usage in sparse graphs.

There are two ways of creating an Adjacency List, both of which require the creation of an Array of Lists. We initialize the array so that every position points to *NULL*, representing an empty list. Then, for each vertex in the graph and for each of its edges, we add the information (ending node and weight of the edge) to the correct array position.

The second method is almost the same, but it stores the full information of an edge (starting node, ending node and weight).

The space complexity of an adjacency list is $\Theta(V + E)$.

In the table below, we summarize the most common Graph operations:

Method	**Description**
`Graph(V,E)`	Create a new graph with the given set of vertices and edges
`addVertex(v)`	Adds vertex `v` to the graph
`addEdge(e)`	Adds edge `e` to the graph
`removeVertex(v)`	Removes vertex `v` from the graph
`removeEdge(e)`	Removes edge `e` from the graph
`vertices(G)`	Returns a list of vertices from graph `G`
`from(e)`	Returns the source vertex of edge `e`
`to(e)`	Returns the destination vertex of edge `e`
`neighbours(v)`	Returns a list of vertices directly connected to `v`
`edges(G)`	Returns a list of edges from graph `G`
`weight(e)`	Returns the weight of edge `e`

10.005 Minimum spanning tree

A minimum spanning tree is a tree calculated from a graph that:

1. Includes **all vertices** of an undirected graph G
2. Uses the subset of edges with minimum total weight

In figure 61 we can see an example of minimum spanning tree from a graph G.

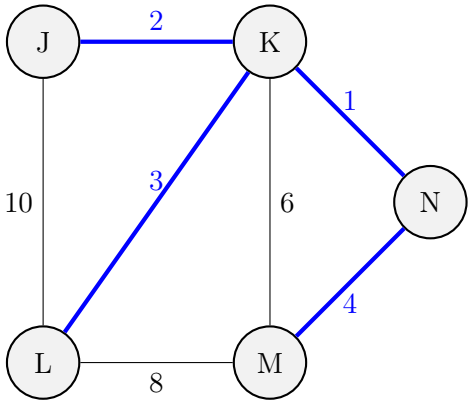


Figure 61: Minimum Spanning Tree

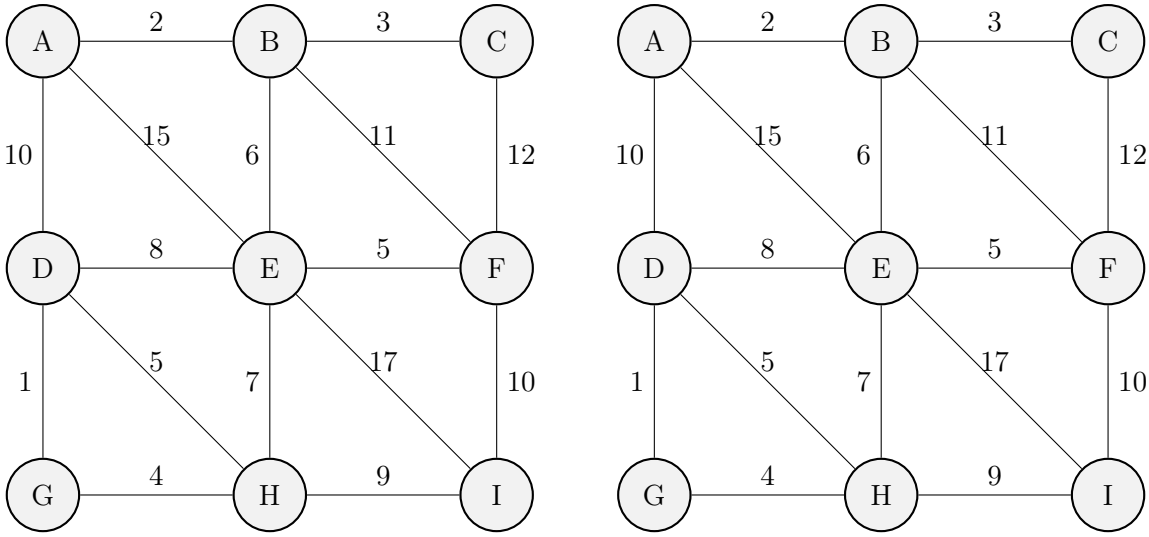
If a graph G has V vertices, the Minimum Spanning Tree has V vertices (i.e, all vertices from the graph are included) and $V-1$ edges (i.e., there can't be cycles or parallel edges).

To find minimum spanning trees, two well-known algorithms are employed:

Prim's Algorithm works by growing a single tree

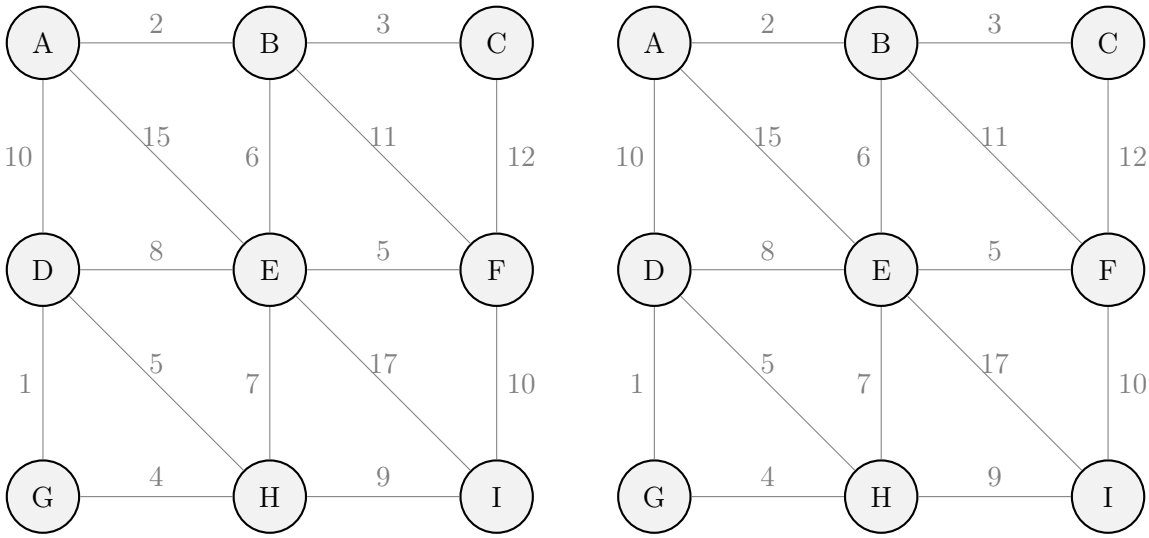
Kruskal's Algorithm works by connecting two trees together

In the following figures, 62 through 71, we show a comparison of how Prim's and Kruskal's algorithms behave over the same graph.



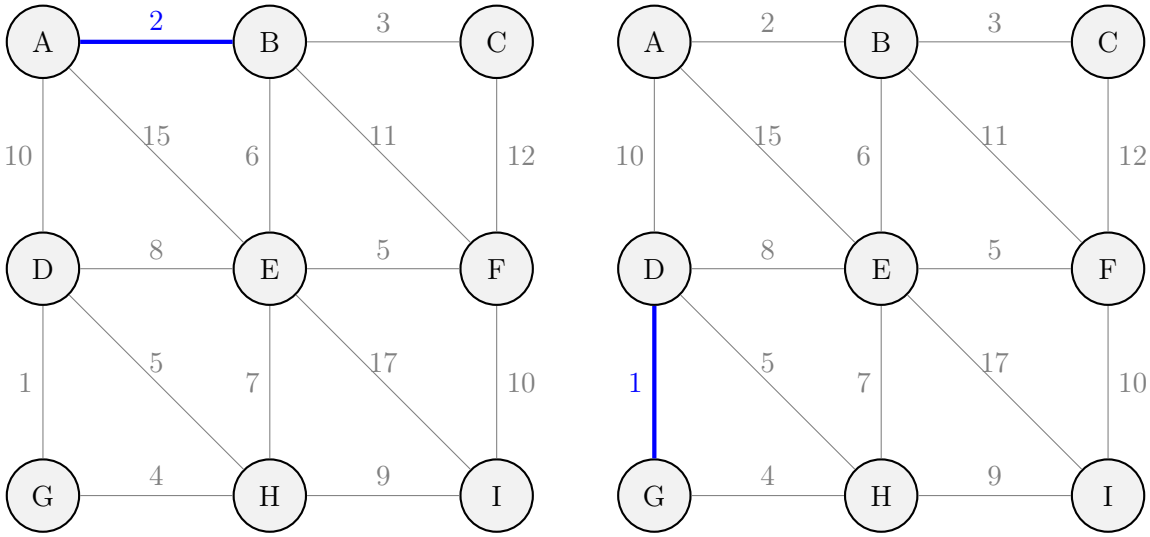
(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 62: Minimum Spanning Tree Algorithms



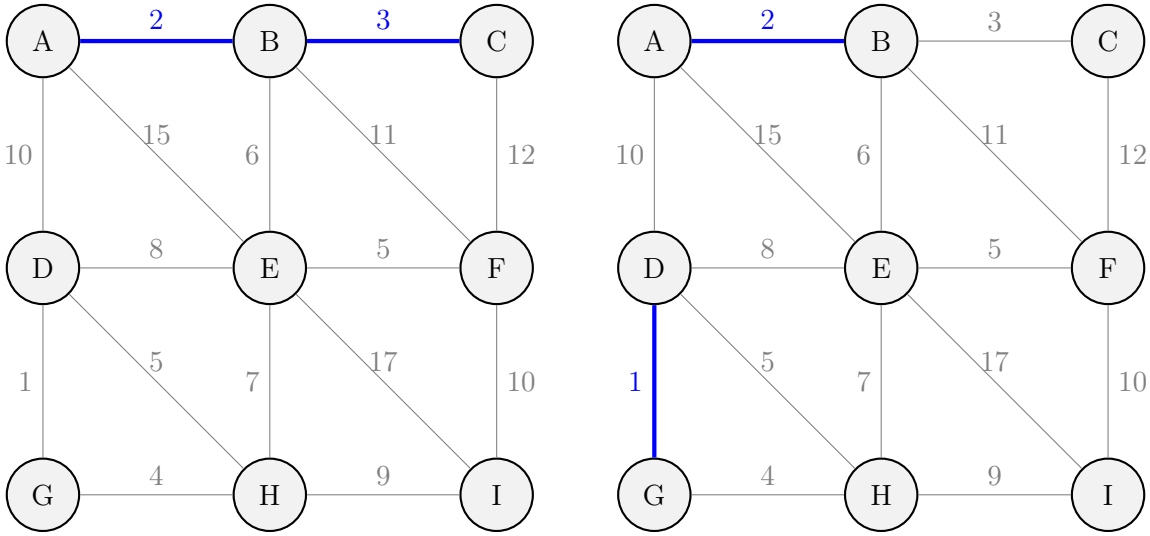
(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 63: MST: Step 1



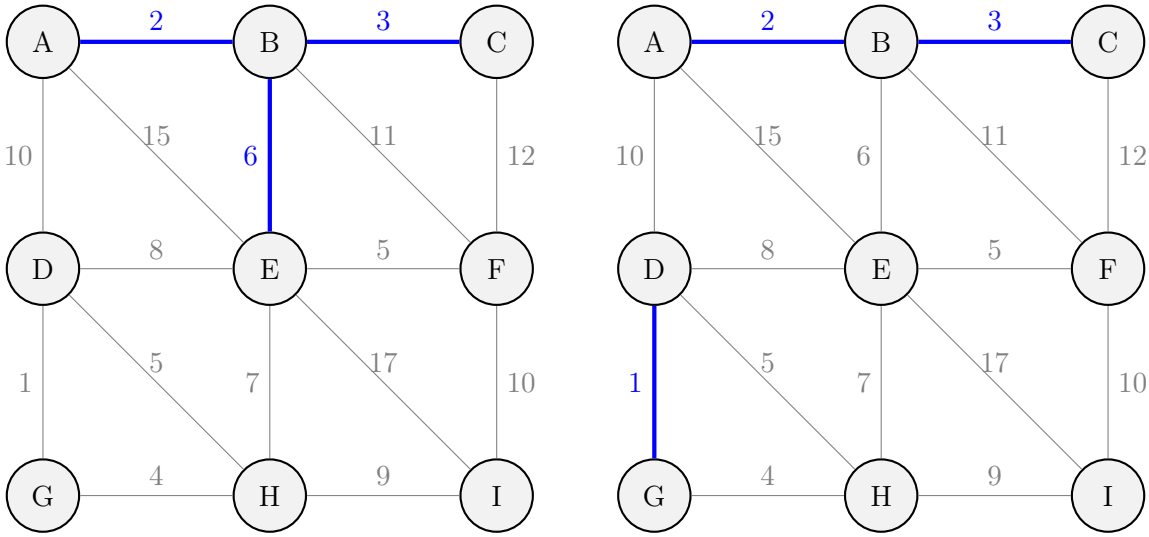
(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 64: MST: Step 2



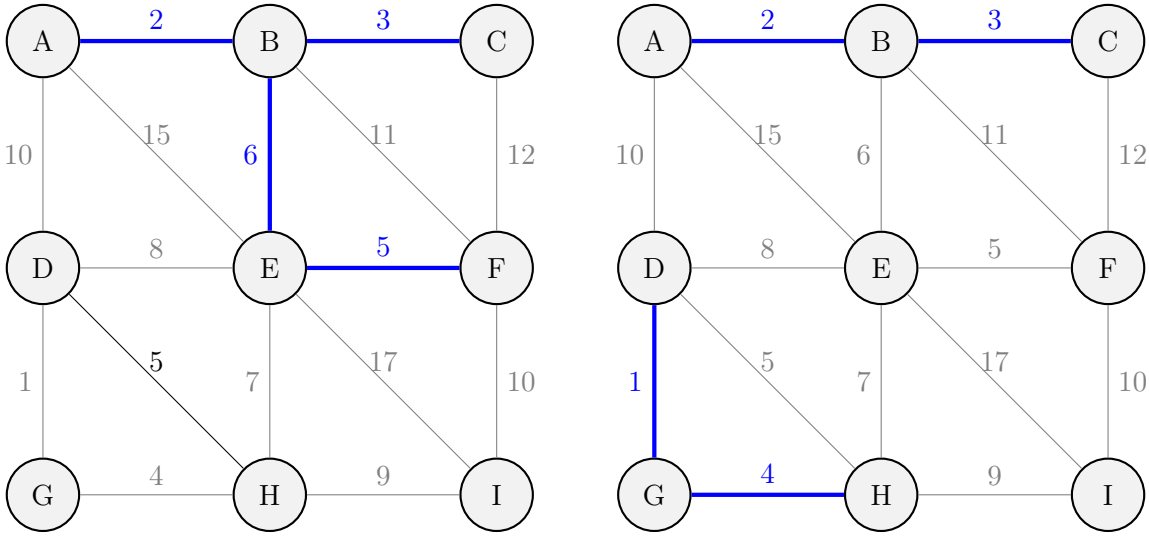
(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 65: MST: Step 3



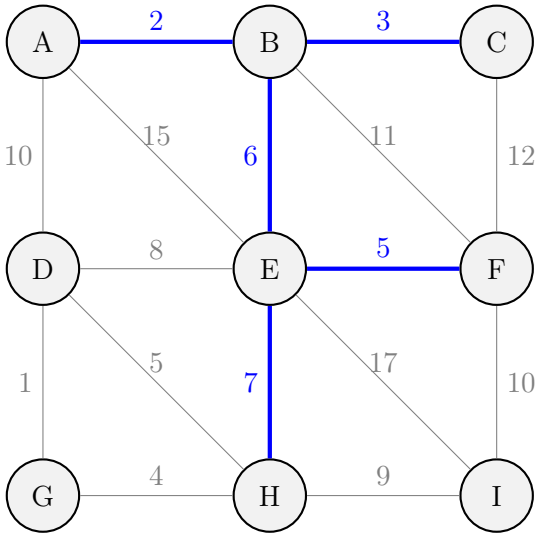
(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 66: MST: Step 4

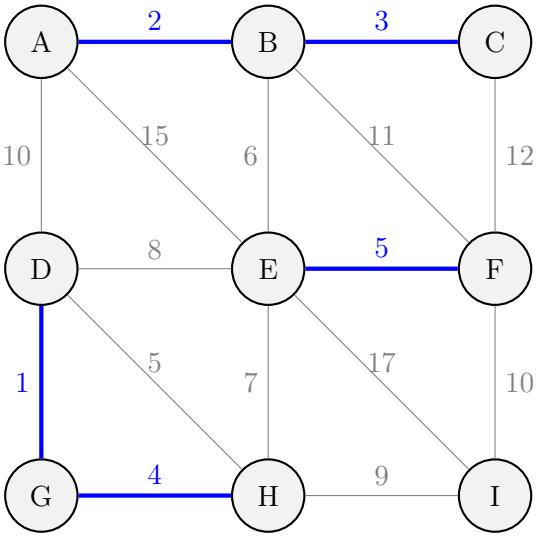


(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 67: MST: Step 5

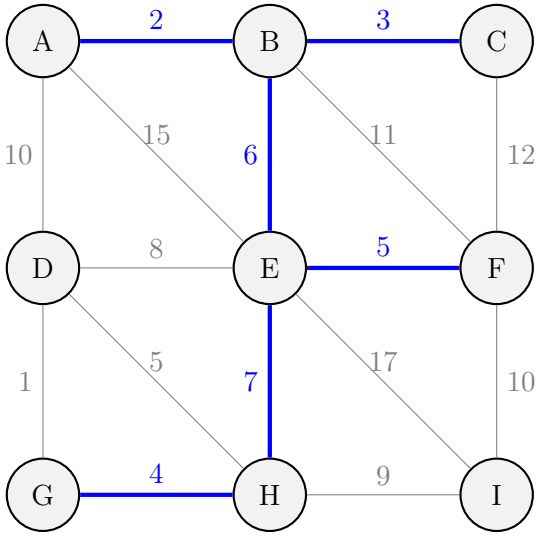


(a) Prim's Algorithm

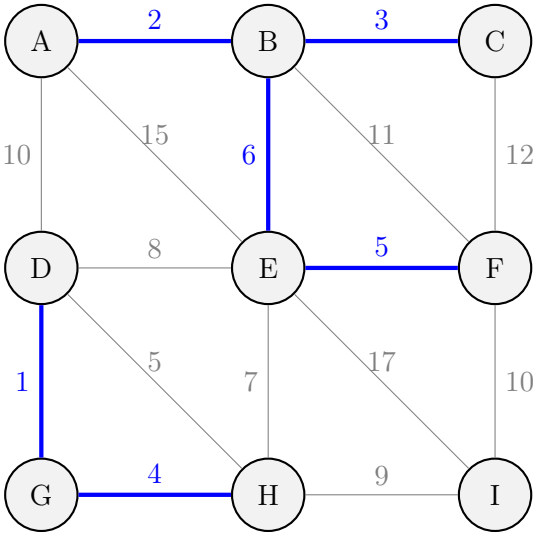


(b) Kruskal's Algorithm

Figure 68: MST: Step 6

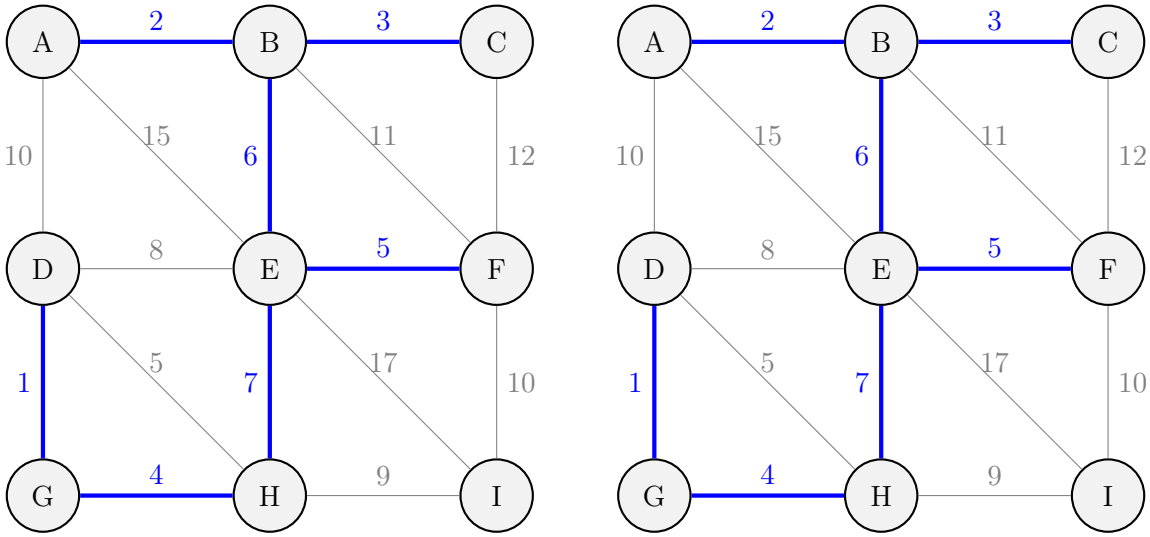


(a) Prim's Algorithm



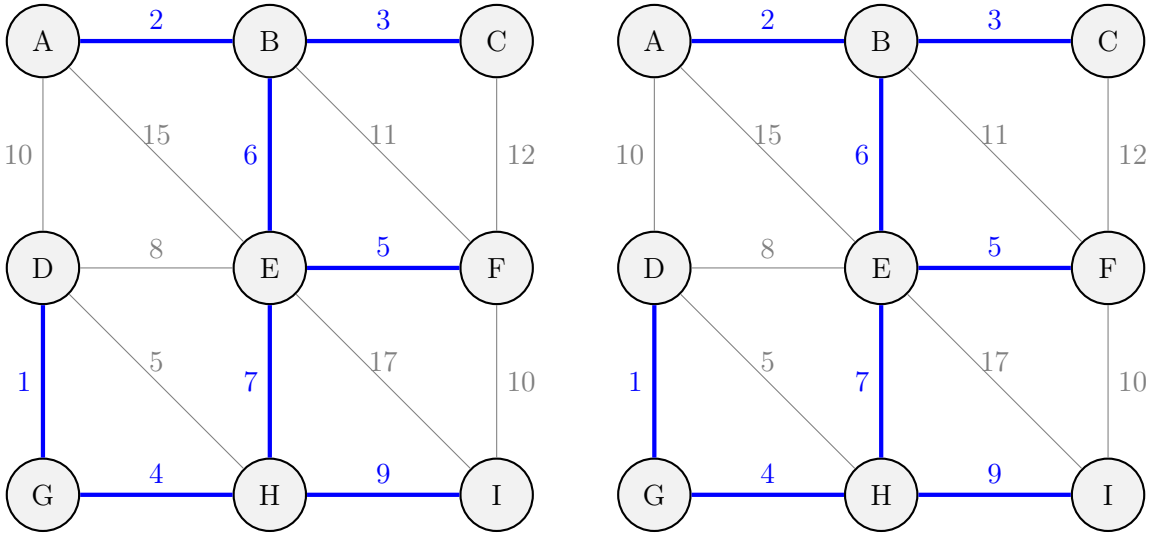
(b) Kruskal's Algorithm

Figure 69: MST: Step 7



(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 70: MST: Step 8



(a) Prim's Algorithm (b) Kruskal's Algorithm

Figure 71: MST: Step 9

As we can see from the step-by-step execution, both algorithms find the same minimum spanning tree. We can also see that that Prim's algorithm always adds a new node to an existing tree, thus growing it, while Kruskal's always finds the next minimal cost edge to be added, thus reducing the number of disconnected trees in the forest.

10.007 Prim's algorithm

Prim's Algorithm, as shown before, works by selecting a random node in the graph and expanding the tree by using the lowest weight edge that connects the partial tree to a node not yet in the tree.

The main steps of the algorithm are as follows:

1. Initialise the spanning tree with one vertex V from graph G
2. Enumerate all edges connecting vertices V of the partial spanning tree to another vertex W not yet in the tree
3. Choose the minimal weight edge and add W to the partial spanning tree
4. Repeat while there are vertices to be added

We can easily convert this step-wise algorithm into pseudocode by using the graph operations described before. The resulting pseudocode is shown in listing 61 below:

Algorithm 61 Prim's Algorithm

```
1:  function PrimMST(G)
2:      vs ← VERTICES(G)
3:      T ← newGraph(FIRST(vs), {})
4:      while |T| < |G| do                ▷ #nodes in T is less than #nodes in G
5:          L ← {e | e ∈ EDGES(G) ∧ FROM(e) ∈ T ∧ TO(e) ∈ G}
6:          newE ← min_{e∈L} WEIGHT(e)
7:          addVertex(T, TO(e))
8:          addEdge(T, newE)
9:      end while
10: end function
```

Professor added a note on how to implement the lines creating L and *newE*. It would look similar to the lines below:

Algorithm 62 Prim's Algorithm: Implementation node

```
1: w ← ∞
2: for e ∈ EDGES(G) ∧ FROM(e) ∈ T ∧ TO(e) ∈ G do
3:     if WEIGHT(e) < w then
4:         w ← WEIGHT(e)
5:         newE ← e
6:         newVertex ← TO(e)
7:     end if
8: end for
```

10.101 Kruskal's algorithm

We know that Kruskal's algorithm works by always choosing the next lowest-cost edge that connects two different trees without forming a cycle. We can think of the algorithm as if we started with a forest (several trees) each with a single node, then the goal is to merge trees together thus reducing the number of trees in the forest.

The algorithm finishes when all trees are merged into a single one. We can summarize Kruskal's algorithm with the list below:

1. Initialise a forest of V trees containing 1 node each
2. Sort **edges** in ascending order of their weight
3. Add edge to spanning tree if it joins two different trees
4. Join both trees into a single one
5. Repeat 3-4 for each edge in ordered set

To implement Kruskal's algorithm, we need to study some operations of a data structure that implements Disjoint Sets. A disjoint set is a collection of sets that do not have elements in common, i.e. it's a list of lists, like $F = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{G\}, \{H\}, \{I\}\}$. The three main operations of a disjoint set are:

`MAKE-SET(F,v)` Creates a new set in the disjoint-set F made of a single element v

`FIND(F,v)` Returns the value of the representative of set S containing element v

`UNION(F,v,u)` Merges sets v and u together producing a single set in the disjoint-set F

Given the disjoint-set operations, we can convert the step-wise instructinos of Kruskal's Algorithm into pseudocode, as shown in listing 63 below:

10.103 Dijkstra's algorithm

Dijkstra's Algorithm is a famous algorithm used to solve the problem of path finding. The aim is to find the lowest cost route from one node to another node in a graph.

We illustrate the algorithm behavior with the graph shown in figure 72 below:

Algorithm 63 Kruskal's Algorithm

```
1: function KruskalMST(G)
2:     vs ← vertices(G)
3:     T ← newGraph(vs, {})
4:     F ← newDisjointSet
5:     for v ∈ vs do
6:         MakeSet(F, v)
7:     end for
8:     L ← Sort(edges(G))
9:     for e ∈ L do
10:        if Find(F, from(e)) ≠ Find(F, to(e)) then
11:            addEdge(T, e)
12:            Union(F, from(e), to(e))
13:        end if
14:    end for
15: end function
```

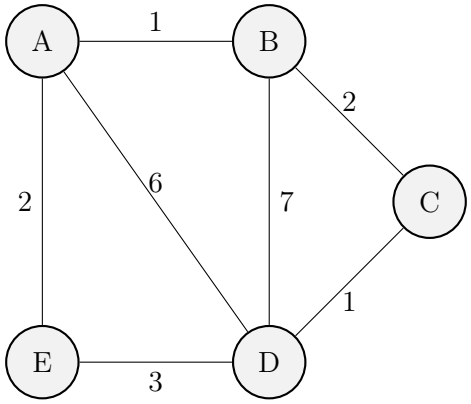


Figure 72: Dijkstra's Algorithm

Let's apply Dijkstra's Algorithm to find the lowest cost route from A to every other node. We start the algorithm by initializing a routing table that will record our route. The first column of the table is the destination node, the second column is the shortest path from A to the destination, the third and final column records the node we're coming from when reaching the destination.

To help our understanding, we're also going to mark the current node and start node. Start node is marked with a green fill, and current node is marked with a blue fill.

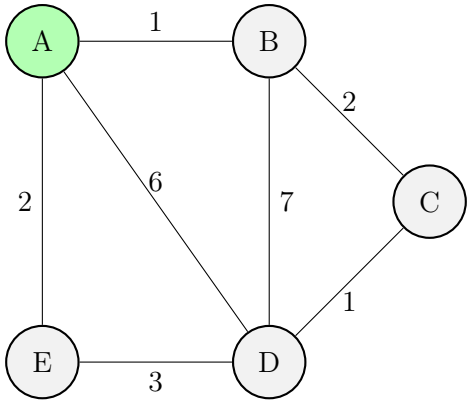


Figure 73: Dijkstra's Algorithm

Node	`dist(A,n)`	`prev(n)`
A	0	A
B	∞	–
C	∞	–
D	∞	–
E	∞	–

With our table initialized, we can move on to the second step in Dijkstra's Algorithm which is to initialize the set of unvisited nodes. At the moment, no nodes has been visited, therefore the set of unvisited nodes contains all nodes, i.e. $U = \{A, B, C, D, E, \}$.

The third and final step consists of the guts of the algorithm and can described like below:

Algorithm 64 Dijkstra's 3rd Step

```
1: while U has elements do
2:     for each neighbour n of u ∈ U do
3:         calculate new distance d = DIST(A, u) + WEIGHT(u, n)
4:         if d < DIST(A, u) in routing table then
5:             update table
6:         end if
7:     end for
8:     remove u from U
9: end while
```

Putting the steps together, we have the simplified pseudocode:

Continuing our simulation of the algorithm, the next step is to check if U is empty. It's not, therefore we look at the node with the minimum distance to the source node. That node, according to the routing table is A, with a distance of 0. Now we look at the every neighbour of A and calculate the total distance. The neighbours are B, D, and E with distances of 1, 6, and 2 respectively. Therefore, we update our routing table.

Algorithm 65 Dijkstra's Algorithm

```
1: Initialise routing table
2: Initialise set of unvisited nodes
3: while U has elements do
4:     for each neighbour n of u ∈ U do
5:         calculate new distance d = DIST(A, u) + WEIGHT(u, n)
6:         if d < DIST(A, u) in routing table then
7:             update table
8:         end if
9:     end for
10:    remove u from U
11: end while
```

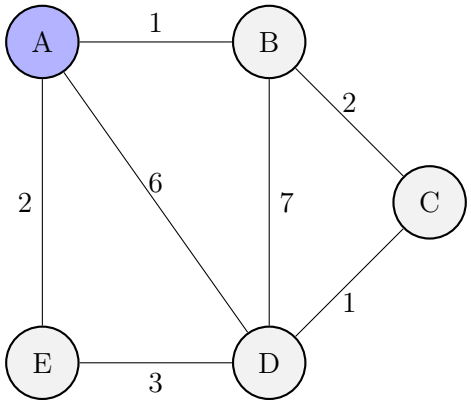


Figure 74: Dijkstra's Algorithm

Node	`dist(A,n)`	`prev(n)`
A	0	A
B	1	A
C	∞	–
D	6	A
E	2	A

Now that A is done, we remove it from U, resulting in $U = \{B, C, D, E\}$. Moving on, the next node in U is B, so we process it, look at its neighbours and update our routing table as shown below:

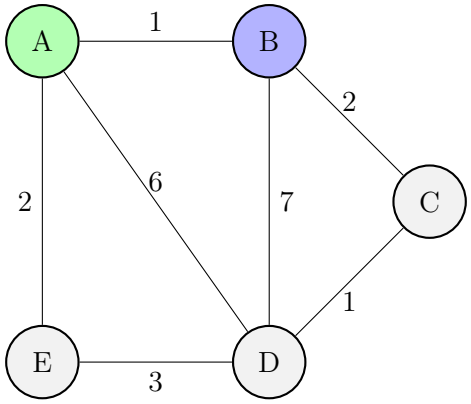


Figure 75: Dijkstra's Algorithm

Node	`dist(A,n)`	`prev(n)`
A	0	A
B	1	A
C	3	B
D	6	A
E	2	A

Because the distance from B to D is larger than the distance already recorded in the routing table, we don't change it. We're done with B and can update U by removing B, i.e. $U = \{C, D, E\}$.

Moving to the next node, C, we get the following new state:

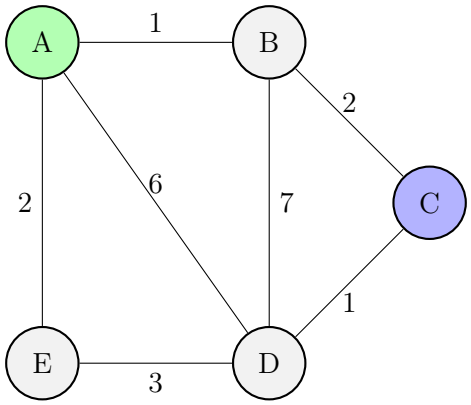


Figure 76: Dijkstra's Algorithm

Node	`dist(A,n)`	`prev(n)`
A	0	A
B	1	A
C	3	B
D	4	C
E	2	A

It's clear that the distance from C to D is shorter than the distance from A to D, so we update our table and remove C from U, therefore $U = \{D, E\}$. Choosing our next node, D, results in the following:

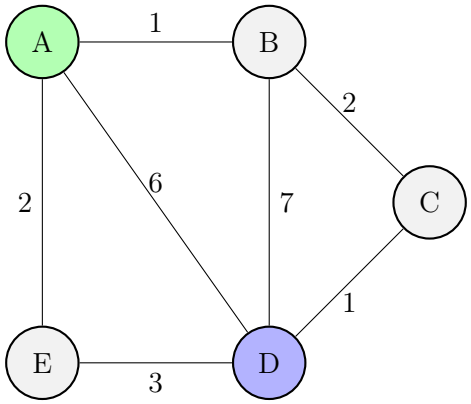


Figure 77: Dijkstra's Algorithm

Node	`dist(A,n)`	`prev(n)`
A	0	A
B	1	A
C	3	B
D	4	C
E	2	A

Because every distance from D from any other node is larger than what's already recorded in the routing table, we don't do anything. We can remove D from U resulting in $U = \{E\}$.

Now E has a single neighbour D, but the distance is shorter than what we can achieve through E, so we don't change the table. Here's the graph for completion:

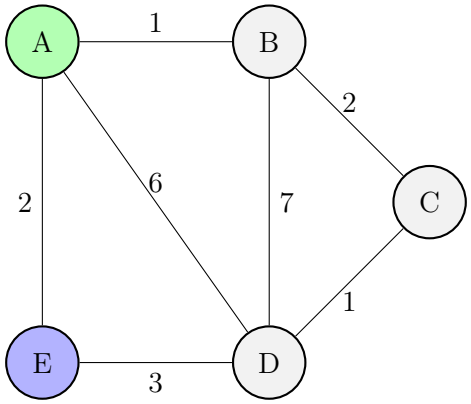


Figure 78: Dijkstra's Algorithm

Node	dist(A,n)	prev(n)
A	0	A
B	1	A
C	3	B
D	4	C
E	2	A

We can remove E from U and, because U is now an empty set, the algorithm terminates. The only missing piece is how do we get the route? Let's say we want to find the route from A to D. We look in the routing table and we know that the shortest path has a length of 4 and goes through C. Looking at C we know the shortest path passes through B, looking at B, the shortest path passes through A, therefore the shortest path from A to D is $A \rightarrow B \rightarrow C \rightarrow D$, highlighted below:

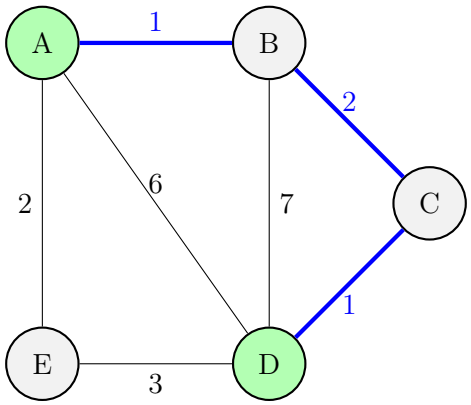


Figure 79: Shortest Path $A \rightarrow D$

10.104 Dijkstra's algorithm pseudocode

After discussing the behavior of the algorithm, we can convert it into pseudocode form shown in listing 66 below:

10.106 Graph representation, MST and Dijkstra

Cormen, T.H., C.E. Leiserson, R.L. Rivest and C. Stein Introduction to algorithms. (MIT Press, 2009) 3rd edition [ISBN 9780262533058].

- Chapter 22, pp.589–92, Section 22.1 Representations of graphs
- Chapter 23, pp.624–42
- Chapter 24, pp.658–66, Section 24.3 Dijkstra's algorithm

Accessible from here.

Algorithm 66 Dijkstra's Algorithm

```
1: function DIJKSTRA(G, start, end)
2:     dist ← newARRAY()                                   ▷ Distance table
3:     prev ← newARRAY()                                   ▷ Previous node table
4:     Q ← newMINHEAP(dist)                                ▷ Unvisited nodes
5:     for v ∈ G do          ▷ Initialise routing table and unvisited nodes minHeap
6:         if v = start then
7:             dist[v] ← 0
8:         else
9:             dist[v] ← −1                                ▷ −1 used in place of ∞
10:        end if
11:        INSERT(Q, v)                                    ▷ Insert current node in minHeap
12:    end for
13:    while ¬EMPTY(Q) do
14:        u ← EXTRACTMIN(Q)                               ▷ Select next unexplored node
15:        if u = end then                                 ▷ Build route from start to end
16:            s ← newSTACK()
17:            while u ≠ start do
18:                PUSH(s, u)
19:                u ← prev[u]
20:            end while
21:            return s
22:        end if
23:        for v ∈ NEIGHBOURS(G, u) do                     ▷ Process next unvisited node
24:            d = dist[i] + WEIGHT(G, u, v)
25:            if d < dist[v] then
26:                dist[v] ← d
27:                prev[v] ← u
28:                DECREASEKEY(Q, v, dist[v])
29:            end if
30:        end for
31:    end while
32: end function
```