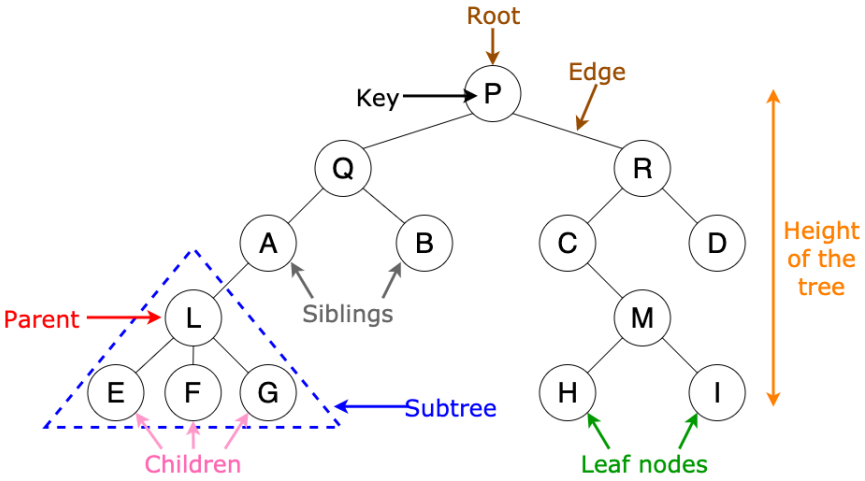
Algorithms and Data Structures (COE 428): Notes

Adam Szava

Winter 2022

Introduction

This document is a compilation of my notes from Algorithms and Data Structures (COE 428) from Ryerson University. All information comes from my professor's lectures, the course textbook *Introduction to Algorithms by Thomas H. Cormen*, and online resources.



Contents

1 Chapter 1: The Role of Algorithms in Computing

Basic Definitions

An **algorithm** is any well-defined computational procedure that takes some set of values as **input** and produces some set of values as **output**. An algorithm can thus be thought of as a transformation between the input set to the output set.

A **computational problem** can be solved by an algorithm in a finite amount of time. There are many computational problems which have many applications in real life such as:

- Sorting.
- Searching.
- Finding the shortest path.
- Minimization.
- Maximization.

An **instance of a problem** consists of the input needed to compute a solution to the problem (like an array of numbers to be sorted).

An algorithm is said to be **correct** is, for every input instance, it halts (meaning stops trying to compute any further) with the correct output, we then say that the algorithm **solves** the given computational problem. An **incorrect** algorithm may not halt at some input instances, or it might halt with an incorrect answer.

There are two characteristics what are common to many interesting algorithmic problems:

1. They have many candidate solutions, the overwhelming majority of which do not solve the problem.
2. They have many important practical applications.

Because of the massive number of possible solutions, efficiency is important. Computing time is a bounded resource, as well as memory space.

Some of the things we use in this course to evaluate efficiency include:

- Runtime.
- Memory space required.
- Bandwidth requirements.
- Power requirements.
- Logic gate requirements.

In this course we also look at **data structures** which are ways to store and organize data in order to facilitate data access and modifications. Difference data structures are used for different purposes. Data structures have associated algorithms to perform certain operations which maintain the properties of the data structure.

Analysing Algorithms

Generally in this course we care about **asymptotic performance** which means the performance as the size of the input (n) tends to infinity.

We denote the time it takes for algorithm k to complete with an input of size n as:

$$T_k(n)$$

Empirical Analysis

Empirical analysis requires the tester to run different algorithms on some hardware in order to measure the time it takes for it to complete the algorithm with varying sizes of input.

The limitations of this method are:

- We have to fully execute an algorithm to study its runtime experimentally.
- In order to compare two algorithms, the same hardware and software environments must be used for the comparison to be fair.

Theoretical Analysis

Theoretical analysis uses a high-level description of the algorithm instead of a specific implementation of it. Upper and lower bounds on $T_k(n)$ are developed and used to evaluate efficiency. This allows us to evaluate the speed of an algorithm independent of the hardware/software environment.

We make a list of **assumptions** about the environment while doing theoretical analysis, this is called the **random-access machine** (RAM) model of computation:

- One processor.
- One instruction at a time.
- Only basic and commonly supported instructions (primitive operations)
  - Assignment.
  - Calling a method.
  - Arithmetic.
  - Comparison.
  - Indexing an array (accessing an element of an array at some index).
  - Return from a method.
- Each of the primitive instructions take "unit time". $T_k(n)$ is measured in unit time.
- Fixed-size operands.
- Fixed-size memory storage.

In order to determine the specific time it takes for an algorithm to execute, we count the number of times the primitive operations are used. We find a best-case scenario (for example if the array is already sorted in a sorting algorithm) and a worst-case scenario (for example if the array is exactly in the wrong order for a sorting array).

We denote $t(n)$ as the number of primitive operations executed by an algorithm, and we bound it as follows:

$$\#\text{ of operations in best case} \leq t(n) \leq \#\text{ of operations in worst case}$$

The best and worst case scenarios are functions and can have many different kinds of growth patterns. Recall that we are interested in asymptotic performance and so the growth rate is important to us.

The following table summarizes common order-of-growth classifications:

order of growth	name	typical code framework	description	example	$T(2N)$ / $T(N)$
1	constant	`a = b + c;`	statement	add two numbers	1
$\log N$	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Ceiling and Floor Functions

$$\lceil x \rceil = \text{the smallest integer greater than or equal to } x$$

$$\lfloor x \rfloor = \text{the largest integer less than or equal to } x$$

2 Chapter 2: Getting Started

In this section we formally define the sorting problem, and a particular algorithm that solves it. We also look at an example of complexity analysis.

The working problem can be stated as:

Input: sequence $(a_1, a_2, \ldots, a_n)$ of numbers.

Output: permutation $(a'_1, a'_2, \ldots, a'_n)$ such that $a'_1 < a'_2 < \cdots < a'_n$.

Sorting is a crucial operation needed in many applications, and many other algorithms rely on a sorted array.

Insertion Sort

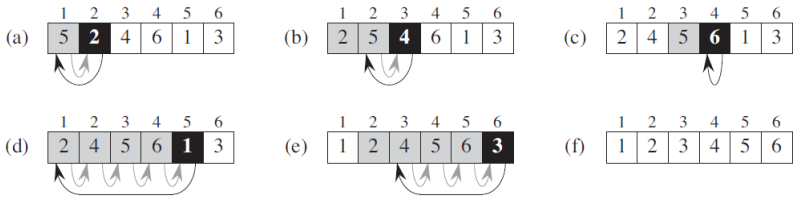
Insertion sort is an algorithm with solves the sorting problem. It builds the sorted list one item at a time.

To get a feeling for how this algorithm works, imagine we are sorting the array:

$$5, 2, 4, 6, 1, 3$$

To sort this array using insertion sort, you start on the left side with 5. Consider 5 as an array and it is already sorted (trivially). Now we really begin by moving to 2 (first diagram), we then *insert* 2 into the array to the left of it in the correct position by moving 5 up a position and placing 2 where 5 was. We then continue this process moving right through the array.

At all times the array to the left of the position we are considering is **sorted**. This is called a loop invariant. We continue to insert the next number into the correct position in the loop.



The following is the pseudocode which describes this algorithm (not that in the pseudocode, indexing starts at 1 **not** 0):

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Let's take it line by line (I highly recommend comparing to the image):

1. The for loop controls the "moving right" process through the array. This loop starts at position $j = 2$ since position 1 is trivially sorted (its just one element). The loop goes up to the length of the array (which is the array t be sorted).

2. Now inside the loop the first thing we do here is we "hold" the number currently held in this position in a variable called key.

3. Comment.

4. Here we declare a new variable i which is always the array index of the number we are comparing the key with. It begins as the index directly before the one we are considering. So if $j = 4$ for example, then $i = 3$.

5. Here we begin the process of checking each of the elements to the left of the position we are considering. The while loop will continuously check elements moving to the left while:

- We haven't reached the end of the loop. If we've reached the end of the loop then the key must be the smallest element so far.
- The element at position i is still bigger than the key. If the element at position i is smaller, then we don't need to consider lower elements as, remember, the array to the left of the index we are considering is always already sorted.

6. This line moves elements **up** the array in the lefthand array, making room for where we may place the key.
7. This line just decrements i so we compare with elements moving left in the array.
8. For whatever reason we leave the while loop, we should then go back a step on i, and place the key there.

Analyzing Insertion Sort

We usually concentrate on finding the worst-case running time. This is because it gives us an upper bound on the time the algorithm will take, additionally the worst case scenario often happens in algorithms (like a search algorithm searching an array for an element that is not there).

The following image shows the number of times the lines in insertion sort will run in the worst case scenario. Each line has some arbitrary constant cost called c which does not depend on n.

INSERTION-SORT(A)		*cost*	*times*
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i = j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] = key$	c_8	$n-1$

Let's understand where these numbers come from:

1. Generally a for loop through all the elements of an array of size n will run $n+1$ times. This is because it has to do one additional check at the end to realize it made it to the end of the array. This +1 is gone from this array since we are missing one element by starting at 2 so the number of times it will run is $n+1-1=n$.
2. Generally, everything inside the for loop (which is not a loop itself) will run one less time than the actual header of the for loop (since once again the header has to run one additional time to check when it reached the end of the array). For this reason this line runs $n-1$ times.
3. Similar reasoning as previous line.
4. Similar reasoning as previous line.
5. In this line we define a number t_j as the number of times the while loop will run for element j. This t_j is thus bounded by 1 (best case the element is already in the correct position) and j (worst case the element is i the opposite end of the array). Since we are doing worst-case analysis you will see later we replace t_j with j. For now the number of times this line runs is equal to the sum of all the t_j values, as in:
$$\sum_{j=2}^{n} t_j$$
6. Generally, everything inside a while loop will run one less time than the header. For each j, this is true, and so this line will run:
$$\sum_{j=2}^{n}(t_j-1)$$
7. Similar reasoning as the previous line.
8. Similar reasoning as lines 2-4.

Determining the worst case scenario by setting $t_j = j$, and adding all the times together multiplied by their arbitrary cost, in addition to the formulas we know for finite sums, we get:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

Which is a quadratic in the form $an^2 + bn + c$.

Recall that we are interested in the *Asymptotic performance* of an algorithm, and so the particular constants do not matter as $n \to \infty$. In fact the n and constant terms don't matter at all. Asymptotically it is enough to say the algorithm grows on the order of n^2 or in other notation $O(n^2)$.

2.3: Designing Algorithms

In this subsection we take a look at an approach to algorithm design called "divide-and-conquer" which allows us to develop Merge Sort which performs much higher asymptotically than Insertion Sort.

Divide and Conquer

The **divide and conquer paradigm** involves three steps:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough they can be trivially solved.
3. **Combine** the solutions to the subproblems into the solution to the original problem.

This methodology requires recursion, which is the idea that a function will call itself until it reaches some trivial base case, at which time the functions begin to return to the original function call.

Merge Sort Introduction

Merge sort is an algorithm to solve the sorting problem. It follows this methodology very well:

1. **Divide** the $n-$element sequence into two subsequences of $n/2$ elements each.
2. **Conquer** (sort) each subsequence recursively (dividing them again if needed). A subsequence is **conquered** when it has a length of 1 in which case it is trivially sorted.
3. **Combine** (merge) the sorted subsequences in order to produce the sorted array.

The function signature for merge sort is:

```
MergeSort(A,p,r)
```

Where:

- A is the array to be sorted.
- p is the index of the first element to be sorted.
- r is the index of the last element to be sorted.

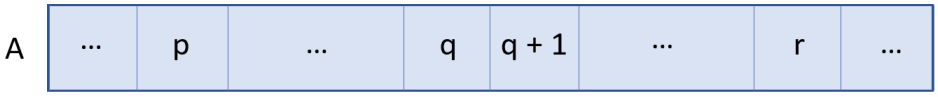
The pseudo-code for Merge Sort is:

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

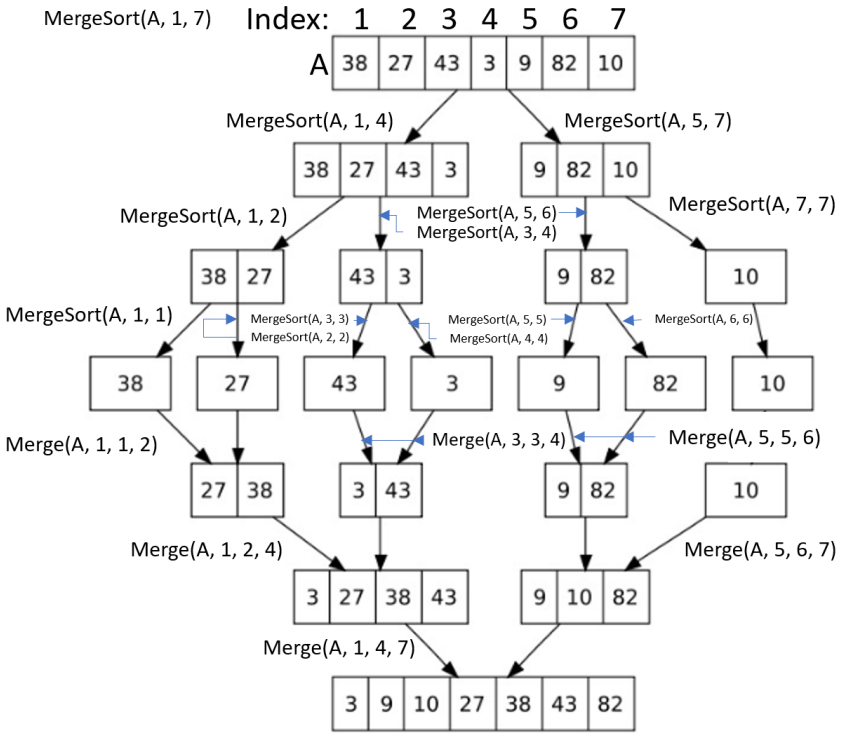
Here are some notes:

- The function is recursive since it calls itself.
- The if statements requires that the index of the first element is greater than the index of the last element. This is because the only other case is that they are the same, in which case the array has only one element and it trivially sorted (note that $p > r$ will never happen). This is the **base case**.
- q is the element in the middle of the array. The floor function is used since an array of 5 elements for example cannot be split in half directly.
- The function calls itself twice, once on the first half of the array ($A[p...q]$), and then once again on the second half of the array ($A[q+1...r]$). Once these function calls return, the two halves will be sorted and we must just merge it.
- The **merge** command takes in the array which has two sorted halves, the index of the first element (p), the index of the last element (r), and then index of the dividing element (q). This function modifies the given array to be in complete sorted order. We will describe how this function works shortly.

This diagram may help understanding the indexing:



This diagram may help in understanding how this algorithm sorts an example array:



Merge Function

So far we have taken the merge function as a given, but here we will look at how the merge function works. Essentially it considers both subarrays like cards stacked on top of each other with the smallest card on top. The algorithm then compares the top card of each pile and picks out the smallest one, this repeats until one subarray has run out of cards at which point the algorithm places the entirety of the remaining subarray on the other.

The pseudocode for this algorithm is:

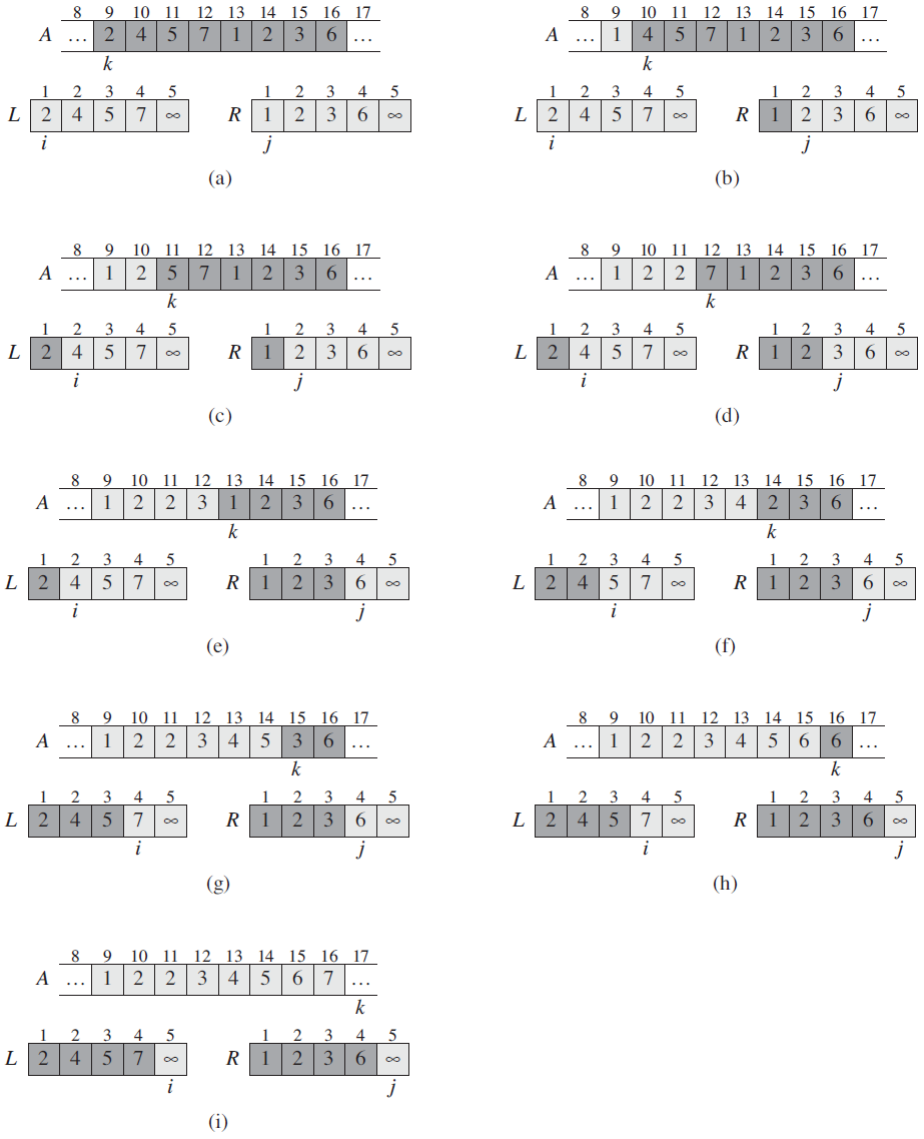
```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Before we begin understanding this line by line, remember the function takes in an array A made up of two subarrays with first index p, final index r, and index q dividing A into the two subarrays, as in:

$$p \leq q < r$$

1. n_1 is the length of the first subarray.
2. n_2 is the length of the second subarray.
3. Initialize new arrays L and R which represent the left and right subarrays. These are literally new arrays in the implementation of the code. Note that they have one additional spot allocated compared to their intended length, more on this in line 8 and 9.
4. Lines 4 and 5 assign the first subarray to the L array.
5. See above.
6. Lines 6 and 7 assign the second subarray to the R array.
7. See above.
8. This line assigns an arbitrarily big number to the final element in the L array. This is so that once this array runs out of actual elements, the final "card" will be larger than all other cards in the other array.
9. Similar to above but with the R array.
10. Index declaration. This index will go through the L array.
11. Index declaration. This index will do through the R array.
12. We now want a for loop to go through the entire array A, from p to r.
13. (Inside for loop) If the ith element of L is smaller than or equal to the jth element of R.
14. (Inside for loop) (Inside if statement) Assign the kth element of A to be the ith element of L since it was the smallest.
15. (Inside for loop) (Inside if statement) Increment i meaning we want to now consider the next element in L for comparison.
16. (Inside for loop) If the condition failed, that means the element in R was smaller, and so assign that to the kth element of A.
17. (Inside for loop) (Inside else statement) Increment j meaning we want to now consider the next element in R for comparison.

The following diagrams may help in understanding the process of the merge function:



Analysis of the Merge Algorithm

The worst case scenario of the merge algorithm is of linear order. This is trivial to show as every operation inside the loops are comparisons or assignments, and do not depend on the number of elements in the array.

Analysis of the Merge Sort Algorithm

The merge sort algorithm is a special case of the general recurrence divide-and-conquer paradigm. This means it follows this general pattern for all divide-and-conquer algorithms:

$$T(n) = \begin{cases} O(1), & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$$

Where:

- $T(n)$ is the running time of a problem of size n.
- c is the size of the subproblem which can be trivially solved in constant time.
- a and b represent that we are dividing our problem into a subproblems each $\frac{1}{b}$ the size of the original problem.
- $D(n)$ is the time it takes to divide the problem into subproblems.
- $C(n)$ is the time it takes to combine solutions.

Given this structure, we can think of merge sort as:

- **Divide** is just computing the middle of the array which is constant time so:

$$D(n) = O(n)$$

- **Conquer** is where we recursively solve the two subproblems, each of size $\frac{n}{2}$ which contributes $2T(\frac{n}{2})$ to the total time. Meaning $a = 2$ and $b = 2$.

- **Combine** is where we use the merge algorithm which we already showed is linear, so:

$$C(n) = O(n)$$

Therefore the recurrence relation for the time for merge sort to run is:

$$T(n) = \begin{cases} O(1), & \text{if } n = 1 \\ 2T(n/2) + O(n), & \text{if } n > 1 \end{cases}$$

We will see in chapter 4 ways to solve this, for just know that $T(n)$ is of order $O(n \log_2(n))$, which is better than insertion sort.

Everything in this course up to this point can be considered "introductory". We have not formally defined a lot of things, instead we just got a feeling of what working with algorithms is like. In the next sections we begin to formalize our thinking with algorithms.

Determining the Complexity of Code Structures

When you look at code structures, there is a simple set of rules to help deal with understanding the complexity of that code. In almost all of these cases we want to know on what order the number of iterations grows. The difference between a loop running n and $n + 1$ is irrelevant.

- The complexity of a loop is the number of iterations in the loop, times the complexity of the body. If the complexity of the body is constant, then the complexity of the loop is just the number of times it runs. For example the following code has complexity $O(n)$.

```
for (int i = 0; i < n; i++){
        O(1);
}
```

- If the indexing variable is not being incremented, but rather it is multiplying by some constant m then the complexity will be $O(lg_m(n))$. For example the following code has complexity $O(lg_n)$:

```
int i = 1;
while (i < n){
        O(1);
        i *= 2;
}
```

- The complexity of nested loops is just the product of their individual complexities. For example the following code has complexity $O(nlgn)$:

```
int i, j = 1;
while (i <= n){
        j = 1;
        while (j <= n){
                O(1);
                j *= 2;
        }
        i += 1;
}
```

- If the inner loop is dependent on the value of the indexing variable of the outside loop (i), then you want to find how many times the inner loop runs manually by trying a few incrementing values of i and forming a finite sum.

- If the problem size is a constant, then the whole code structure has complexity $O(1)$.

- If you have independent code structures within one larger code structure, then the complexity of the whole code is the highest complexity of all the individual pieces. For example the following code has complexity $O(n^2)$:

```
for (int j = 0; j < n*n; j++){
        O(1);
}

for (int i = 0; i < n; i++){
        O(1);
}
```

- To determine the complexity of an if statement, we just look at the highest complexity of all the conditions, or branches.

- Sometimes the above rule does not apply. Consider the following code structure:

```
for (int i = 1; i <= n; i++){
        if (i == 1){
                block1;
        } else {
                block2;
        }
}
```

  If:

$$O(block1) > O((n-1)O(block2))$$

  ... then the whole thing is $O(block1)$, otherwise it is $O(n * block2)$.

- To determine the complexity of a switch statement, just consider the highest complexity of all the cases, including default.

3 Chapter 3: Growth of Functions

In this section we formally understand notations like $O(), \Omega()$ and, $\Theta()$, and how they relate to the growth of functions which evaluate the running time of an algorithm.

The **order of growth** of the runtime of an algorithm as a function of the input sizes gives a way for us to characterize an algorithm's efficiency and compare the performance to different algorithms. While we sometimes can find the exact runtime of an algorithm depending on hardware-specific coefficients, it is typically too much work and irrelevant to asymptotic performance as just the highest order terms really matter.

This chapter gives several standard methods for simplifying the asymptotic analysis of algorithms.

3.1: Asymptotic Notation

Asymptotic notations apply to functions, and are a way to characterize asymptotic runtime. In this course, asymptotic notation applies to the runtime of algorithms, however in general it can apply to other characteristics of algorithms, or functions which don't relate to algorithms at all.

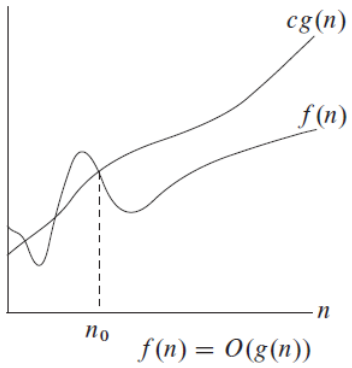
O-notation

Formally, the definition of this notation is:

Definition 3.1.1 O-notation

For a given function $g(n)$, we denote $O(g(n))$ as the following set of functions:

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$$



You can informally think of a function in this set (usually denoted $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$) as a function which is bounded above by some multiple of the function $g(n)$.

The following example shows that (as we intuitively know) that all quadratic functions are $O(n^2)$.

Example 3.1.1 Quadratics are $O(n^2)$

Show that $f(n) = an^2 + bn + c$ is $O(n^2)$ by definition.

To solve this question, we must use the definition. This means that if its true, there exists some constants c and n_0 such that:

$$0 \leq an^2 + bn + c \leq cn^2, \forall n \geq n_0$$

One way to go about doing this is to say:

$$an^2 + bn + c \leq (|a| + |b| + |c|)n^2 + (|a| + |b| + |c|)n + (|a| + |b| + |c|), n \geq 1$$

... which you should convince yourself is true. Further:

$$(|a|+|b|+|c|)n^2+(|a|+|b|+|c|)n+(|a|+|b|+|c|) \leq (|a|+|b|+|c|)n^2+(|a|+|b|+|c|)n^2+(|a|+|b|+|c|)n^2 = 3(|a|+|b|+|c|)n^2, n \geq 1$$

Which you should also convince yourself is true. Therefore:

$$an^2 + bn + c \leq 3(|a| + |b| + |c|)n^2, n \geq 1$$

Picking:

$$c = 3(|a| + |b| + |c|), n_0 = 1$$

We have proven that $f(n)$ is $O(n^2)$.

Typically you want to pick the tightest upper bound for the function. For example for the function:

$$f(n) = 3n + 1$$

we would say that $f(n)$ is $O(n)$ and not that $f(n)$ is $O(n^2)$ even though both are true.

The following is a list of useful rules for $O()$ complexity:

- If:
$$g(n) = O(G(n)), f(n) = O(F(n))$$
... then:
$$f(n) + g(n) = O(F(n)) + O(G(n)) = O(max(F(n), G(n)))$$
- If:
$$g(n) = O(G(n)), f(n) = O(F(n))$$
... then:
$$f(n) \cdot g(n) = O(F(n)) \cdot O(G(n)) = O(F(n) \cdot G(n))$$
- If (for some constant k):
$$g(n) = O(kG(n))$$
... then:
$$g(n) = O(G(n))$$
- If $f(n)$ is a polynomial of degree d, then:
$$f(n) = O(n^d)$$

Using these laws we can find the order of growth of a function without using the definition, such as in the following example:

$$\begin{aligned} T(n) &= 2n^3 + 5n^2 + 10 \\ &= O(2n^3 + 5n^2 + 10) \\ &= O(2n^3) + O(5n^2) + O(10) \\ &= O(n^3) + O(n^2) + O(1) \\ &= O(n^3) \end{aligned}$$

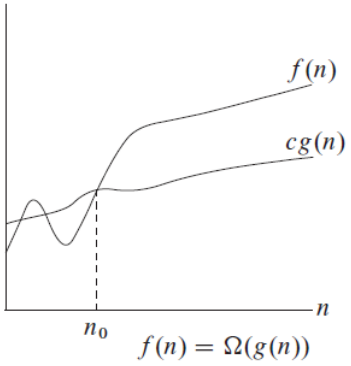
Ω-notation

Formally, the definition of this notation is:

Definition 3.1.2 Ω-notation

For a given function $g(n)$, we denote $\Omega(g(n))$ as the following set of functions:

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$$



$f(n) = \Omega(g(n))$

Informally you can think of a function in this set (usually denoted $f(n) = \Omega(g(n))$ instead of $f(n) \in \Omega(g(n))$) as a function which is bounded below by some multiple of the function $g(n)$.

The list of rules which worked for $O()$ notation also works for $\Omega()$ notation.

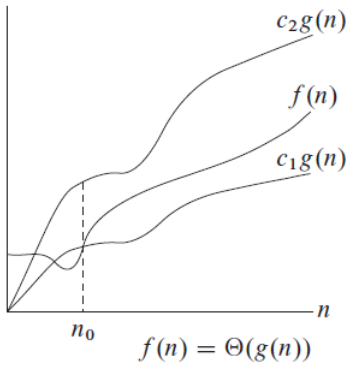
Θ-notation

Formally, the definition of this notation is:

Definition 3.1.3 Θ-notation

For a given function $g(n)$, we denote $\Theta(g(n))$ as the following set of functions:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Informally we can think of a function in this set (usually denoted $f(n) = \Theta(g(n))$ instead of $f(n) \in \Theta(g(n))$) as a function which is sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for all input values past some point n_0.

Another word for this notation is **asymptotically tight bound** for $f(n)$.

The following theorem is helpful to think about the $\Theta()$ notation:

Theorem 3.1.1 Θ-notation

A function $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $\Omega(g(n))$.

The idea of $O(g(n))$ notation is to provide a loose upper bound on the runtime complexity of an algorithm. Similarly, $\Omega(g(n))$ provides a loose lower bound, while $\Theta(g(n))$ provides a **tight** bound.

We can also consider two new notations being $o(g(n))$ (pronounced "little-oh"), and $\omega(g(n))$ (pronounces "little-omega"). These notations guarantee a less specific runtime complexity, rather they guarantee the algorithm will always run better than some function (o), or worse than some function (ω).

Definition 3.1.4 o-notation

For a given function $g(n)$, we denote $o(g(n))$ as the following set of functions:

$$o(g(n)) = \{f(n) \mid c, n_0 > 0 \text{ s.t. } \forall c > 0, \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}$$

Notice that this is the same definition as $O(g(n))$ notation, but the inequality now has to hold for **all** $c > 0$. What this means for a function $f(n)$ to be $o(g(n))$ is that the function $f(n)$ should become insignificant compared to $g(n)$ as $n \to \infty$. This can be represented as the limit:

$$f(n) = o(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

For example, the function $f(n) = 2n + 1$ is $O(n)$ but $o(n^2)$. This is because $f(n)$ does not grow slower than **all** function in the form $g(n) = cn$ (take $c = 1$ for example). $f(n)$ **does** grow slower than all $g(n) = cn^2$ asymptotically, and so it is $o(n^2)$.

Definition 3.1.5 ω-notation

For a given function $g(n)$, we denote $\omega(g(n))$ as the following set of functions:

$$\omega(g(n)) = \{f(n) \mid c, n_0 > 0 \text{ s.t. } \forall c > 0, \forall n \geq n_0,\ 0 \leq cg(n) \leq f(n)\}$$

In this case, the function $g(n)$ should become insignificant compared to $f(n)$ asymptotically ($f(n)$ becomes arbitrarily large), which can be represented by this limit:

$$f(n) = \omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Comparing the Growth of Functions

In fact, we can use this limit idea to compare functions even if we are not considering labeling them with asymptotic notations, especially two functions which have the same order of growth. When comparing two functions $f(n)$ and $g(n)$:

- $f(n)$ grows slower than $g(n)$ ($f < g$)
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < 1$$
- $g(n)$ grows slower than $f(n)$ ($f > g$)
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 1$$
- $f(n)$ grows at the same rate as $g(n)$ ($f = g$)
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$
- $f(n)$ is asymptotically smaller than $g(n)$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
- $g(n)$ is asymptotically smaller than $f(n)$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

You can use these limits to find the values of the constants when determining a functions asymptotic notation.

Finally, asymptotic notation of all types has a list of applicable rules, shown below:

Basic properties of asymptotic notations

- Transitivity:

$$\begin{aligned}
&f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)),\\
&f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)),\\
&f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n))\\
&f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \text{ imply } f(n) = o(h(n)),\\
&f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \text{ imply } f(n) = \omega(h(n)).
\end{aligned}$$

- Reflexivity:

$$\begin{aligned}
f(n) &= \Theta(f(n)),\\
f(n) &= O(f(n)),\\
f(n) &= \Omega(f(n)).
\end{aligned}$$

- Symmetry

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)).$$

- Transpose symmetry:

$$\begin{aligned}
&f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)),\\
&f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).
\end{aligned}$$

4 Chapter 4: Divide-and-Conquer

Divide and conquer is a class of problems in which we solve a problem recursively, by applying three steps at each level of recursion:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. Keep solving recursively until there is some base case.
3. **Combine** the solutions to the problems into the solution to the original problem.

Recursive algorithms call themselves to solve smaller problems. The code for the recursive functions are always the same, but the passed parameters change with each invocation. General some parameter must be getting closer to a base case with each invocation.

In general, recursive solutions to problems perform better than their non-recursive equivalents.

We need an explicit form of recurrence time complexity to compare performance. We can create implicit formulas for the time complexity for many algorithms:

- **Merge Sort**:
$$T(n) = 2T(n/2) + \Theta(n)$$
The $2T(n/2)$ represents that we are cutting the problem into two pieces, each with half of the elements. The $\Theta(n)$ represents the non-recursive code running (like a loop or an if check).
- **Factorial**:
$$T(n) = T(n-1) + \Theta(1)$$

Box Method

Box method is a way to visualize the flow of recursive calls. You make a box diagram by:

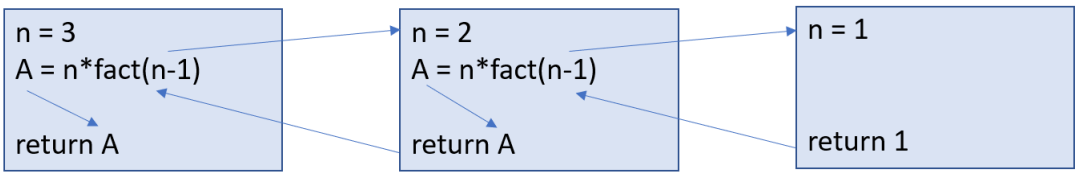
- Label each recursive call.
- Represent each call to a function by a new box.
- Draw arrows from the box that makes the call to newly created box.
- Create new box executing the body of the function.
- On an existing function, cross of current box and follow its arrow back.

Example

Example 4.1.1 Box Method for Factorial

Make a box diagram for the following code if you call *fact(3);*

```
int fact(int n){
        if (n==1){
                return 1;
        } else {
                return n * fact(n-1);
        }
}
```



The recurrence formula for the factorial function is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(n-1) + \Theta(1) & n > 1 \end{cases}$$

For each step of recursion there is a non-recursive and recursive part, as well a base case. The non-recursive part must be there to check if the base case is reached, and it cannot be ignored until you have an explicit formula.

In order to find the explicit formula, you want to make some kind of a guess using the first few values of n and try to generalize. For factorial that is:

$$T(1) = c_0$$

$$T(2) = T(1) + c_1 = c_0 + c_1$$

$$T(3) = T(2) + c_1 = c_0 + c_1 + c_1$$

$$\vdots$$

$$T(n) = c_0 + (n-1)c_1 = \Theta(n)$$

This is more of a guess of the complexity than an actual proof. We will now discuss how to actually prove a formula is true.

Mathematical Induction

Mathematical induction is a method to prove a statement is true. Generally it works like this:

- Given some fact $f(n) = g(n)$ is true for $n0$ (base case).
- We now assume that it is true for $n = k$, if this implies it is true for $n = k + 1$ then we can conclude...
- We can conclude that it must be true for all $n \geq 0$.

Substitution Method

There are three basic parts to the substitution method:

- Guess the form of the solution.
- Verify by induction.
- Solve for constants.

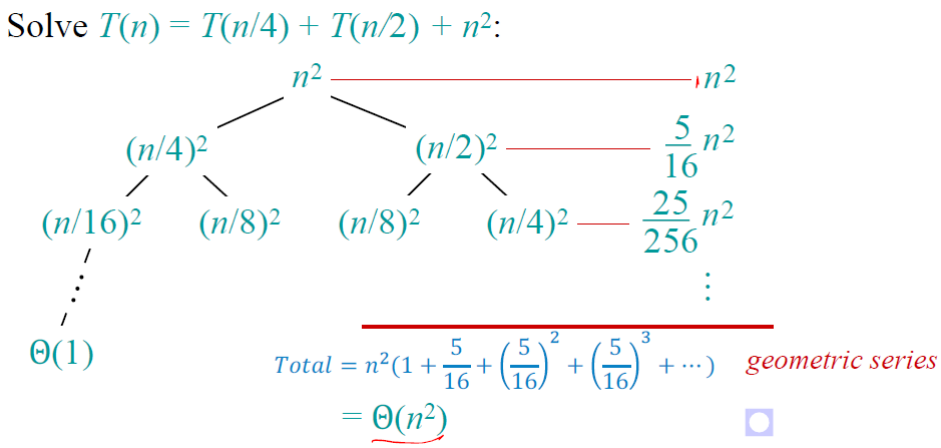
We substitute the guessed solution for the function when applying the inductive hypothesis. The following five steps go through the method:

1. Try a few substitutions to find a pattern.
2. Guess the recurrence formula after trying k substitutions.
3. Set k so that we get the base case.
4. Put k back into the formula to find a potential closed form.
5. Prove the potential closed form using induction.

Recursion Tree

A recursion tree models the costs of a recursive algorithm. This lets you make a good guess for substitution method. In a recursion tree, each node represents the cost of a single subproblem, we grow the tree and substitute the parent with the non-recursive part of the cost. We then sum all the costs within each level to obtain a set of costs, and then we sum all those costs together to determine the total cost.

For example:



5 Chapter 6: Heapsort

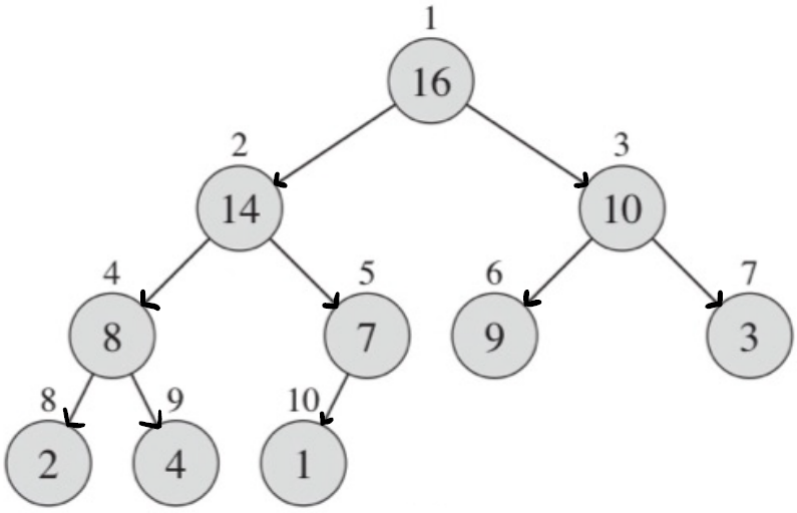
This section is about **Heapsort** which is the best sorting algorithm we will learn about in this class. Before we do this we have a long section introducing what a tree is to begin with.

Trees

A tree is a data structure meaning it is a dynamic set of nodes which store elements. Trees store elements in a parent-child relationship. A tree is a directed graph with some special properties:

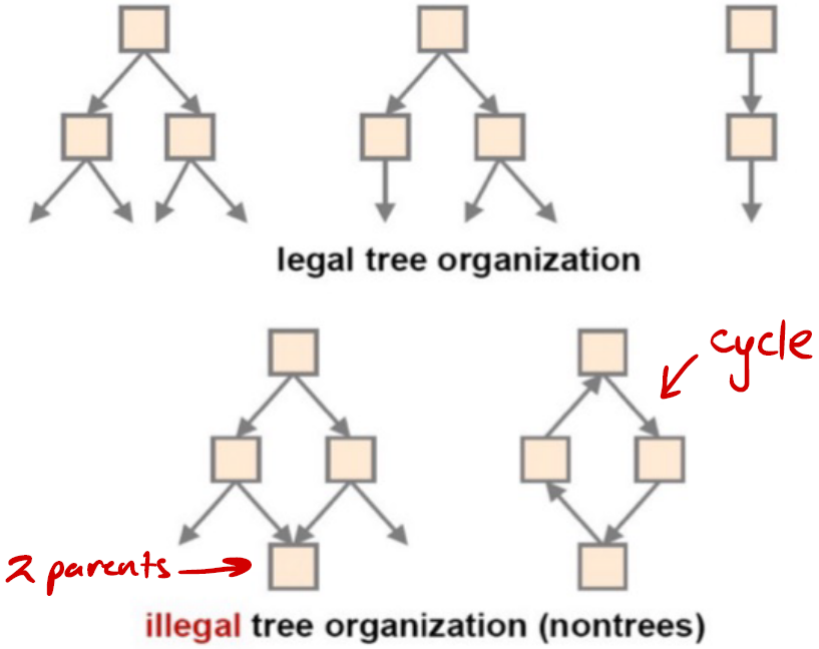
- Each node contains an element and branches leading to other nodes.
- A tree has a unique node called the *root* which is at the top of the tree.
- Each node except the root has exactly one parent.
- A parent can have several children.
- There are no cycles in the tree.

We like to think of nodes as being in discrete depths (like energy levels of an electron around an atom). A parent is one level above its children. The following is an example of a graph (numbers inside the node represents the elements, number outside the node represents the index):



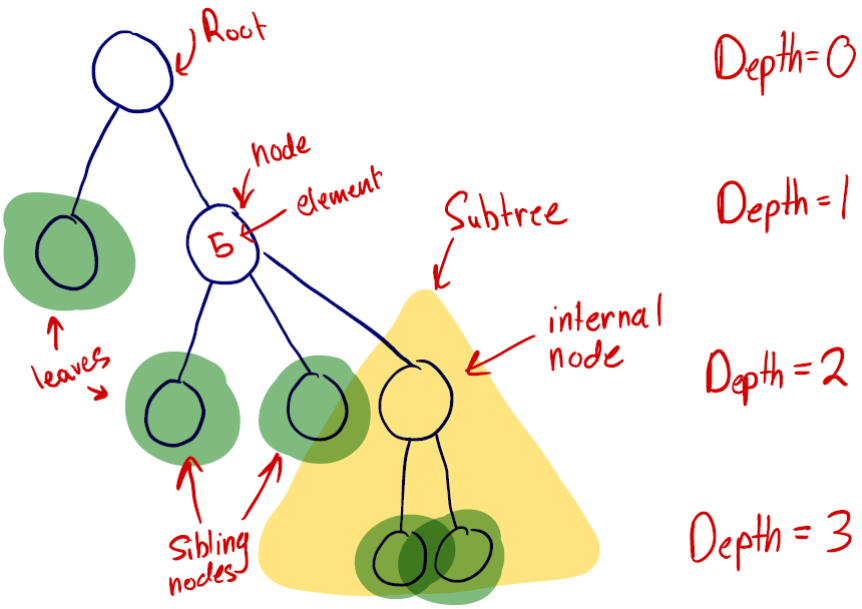
If arrows are not drawn on a tree, it is assumed to be downward.

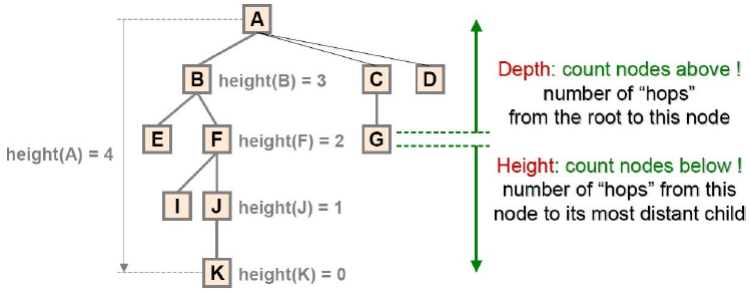
Here are more examples of legal and illegal trees:



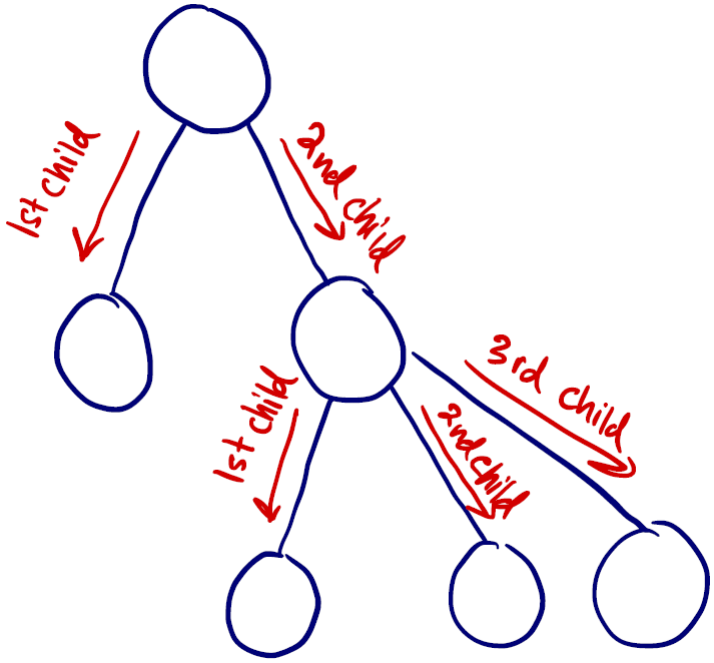
The following is a list of terminology used for trees:

- **Node** is an element of the tree.
- **Root** is the topmost node with no parent.
- **Siblings** are nodes which have the same parent.
- **External Nodes/Leaves** are nodes with no children.
- **Internal Nodes** are nodes with one or more children.
- **Ancestors** refers to the parents, grandparents, greatgrandparents, and so on of a node.
- **Descendants** refers to the children, grandchildren, greatgrandchildren, and so on of a node.
- **Depth** refers to the number of ancestors of a node (distance to root).
- **Height of a tree** refers to the max node depth of the tree.
- **Height of a node** refers to the distance between the node and its most distance child (opposite of depth).
- **Subtree** refers to a tree consisting of a node (which becomes the root of the subtree) and all its descendants. Its like you take one branch of the tree and call that its own tree.





An **ordered tree** is just a tree where we define a linear ordering of the children (1st, 2nd, 3rd, and so on) usually from left to right.



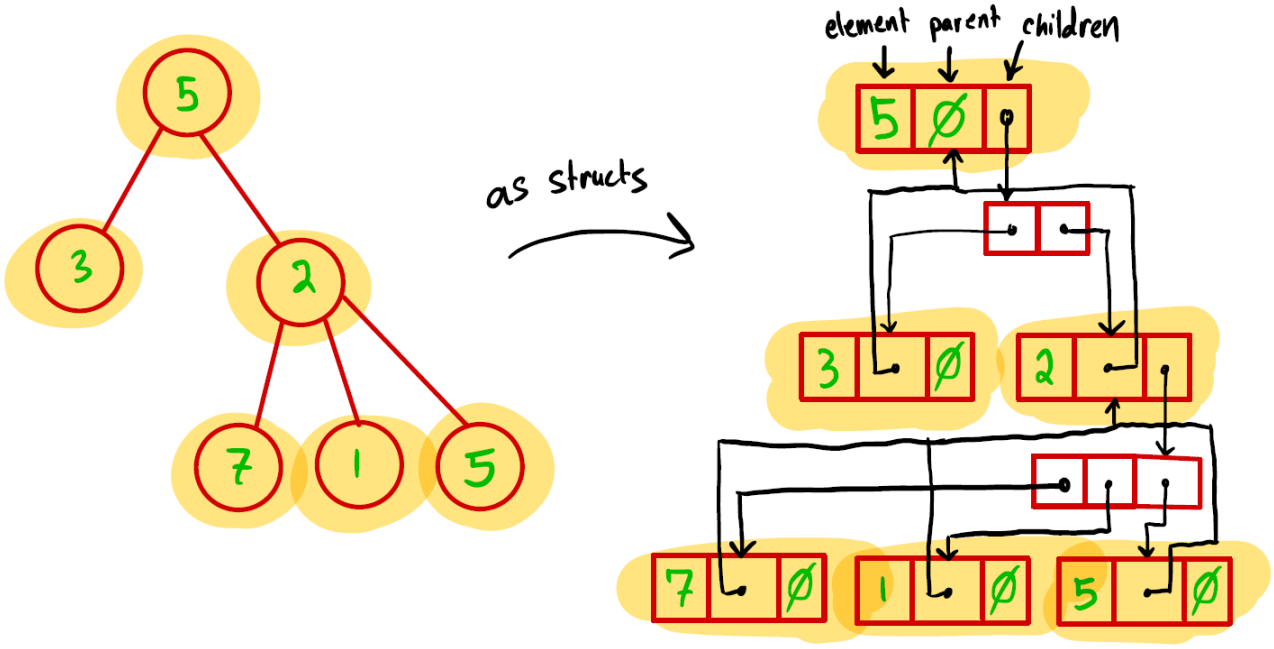
Supported Methods for Trees and Implementation

Tree structures have supported methods like any data structure:

- *size(): int;* returns the number of nodes in the tree.
- *isEmpty(): boolean;* returns whether the tree has any nodes or not.

- *root(): Node;* returns the root of the tree if it exists.
- *parent(v:Node): Node;* returns the parent node to the node v.
- *children(v:Node): Node[];* returns a collection containing the children of node v.
- *isInternal(v:Node): boolean;* returns if the node is internal.
- *isExternal(v:Node):boolean;* returns if the node is external.
- *isRoot(v:Node):boolean;* returns if the node is the root.

We can implement a tree by creating Node structs in C which have an element, pointer to parent, and an array of pointers to its children, the following summarizes this:



Common Tree Algorithms

Given some tree T and a node v, we may want to determine the depth of that node. We do this by finding the the depth of the parent recursively and adding 1 (as the depth of the child is one greater than the depth of the parent). The base case is reaching the root.

```
public int depth(Tree T, Node v){
        if (isRoot(v)){
                return 0;
        } else {
                return (1+depth(T, parent(v)));
        }
}
```

This algorithm is worst case $O(n)$.

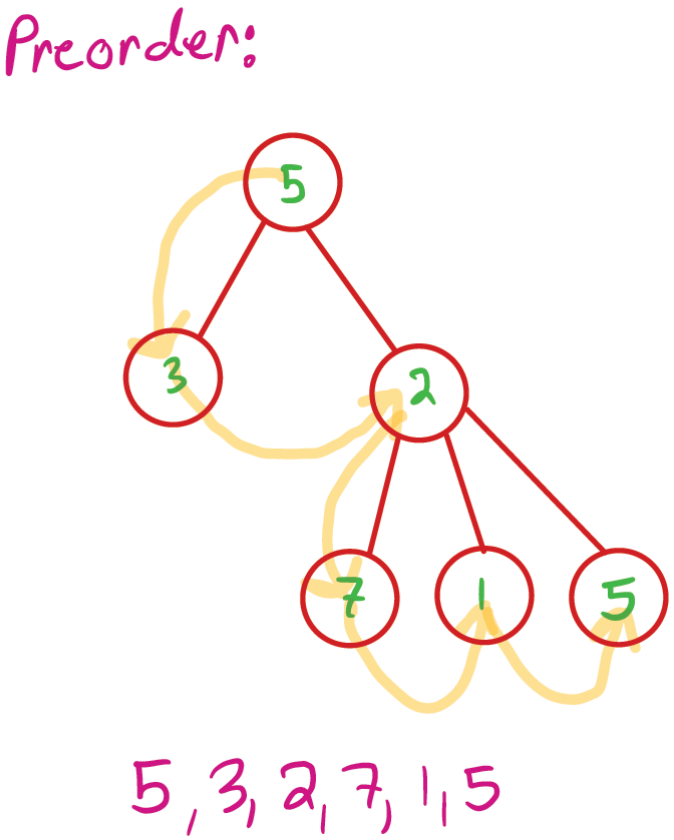
Tree Traversal

Traversing a tree is a systematic way of accessing all the nodes of a tree where each node is visited exactly once, and produces a linear ordering of the nodes of the tree. This can be useful for counting the number of nodes in a tree, or for searching for a particular node.

There are three types of traversal:

- **Preorder Traversal**
  - The root is visited first and then the each sub tree which begins as the root's children from left to right.

– A node is visited before its descendants.



– There are two ways to implement preorder traversal. In both cases the word "visit" refers to some action you want to take on each node. One is recursively:

Algorithm *preorder*(T : ordered rooted tree)

1: r = root of T
2: visit r
3: **for** each child c of r from left to right **do**
4: $T(c)$ = subtree with c as its root
5: *preorder*($T(c)$)
6: **end for**

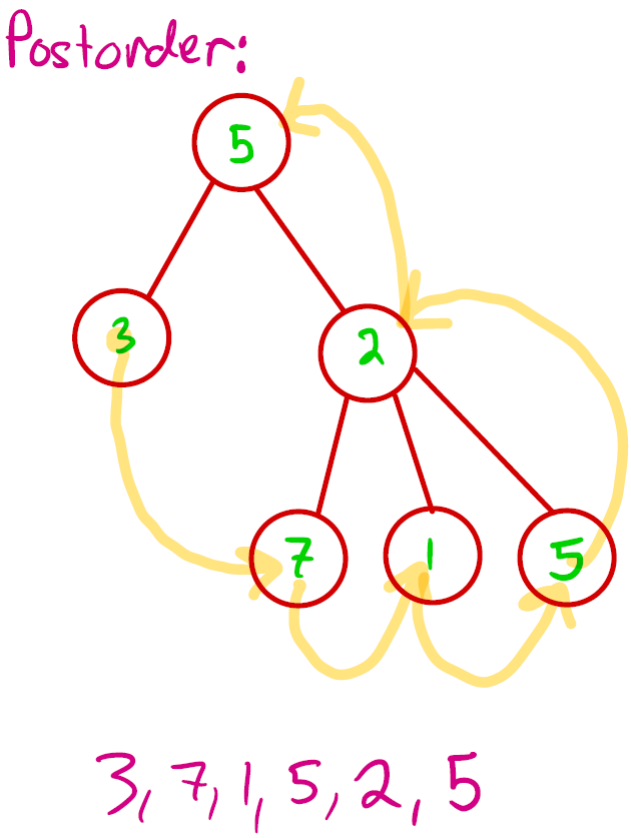
... and the other is as a stack:

1. Push root onto stack.
2. While stack is not empty, pop the stack and "visit" that node, then push all the children of that node onto the stack.

- **Postorder Traversal**

– In this type of traversal you recursively traverse the subtrees rooted at the children and then visit the root itself.

– You visit all the children before the parent.



– There are once again two ways to implement postorder traversal. One is recursively:

Algorithm *postorder*(T : ordered rooted tree)

```
1: r = root of T
2: for each child c of r from left to right do
3:     T(c) = subtree with c as its root
4:     postorder(T(c))
5: end for
6: visit r
```

... and the other is as a stack:

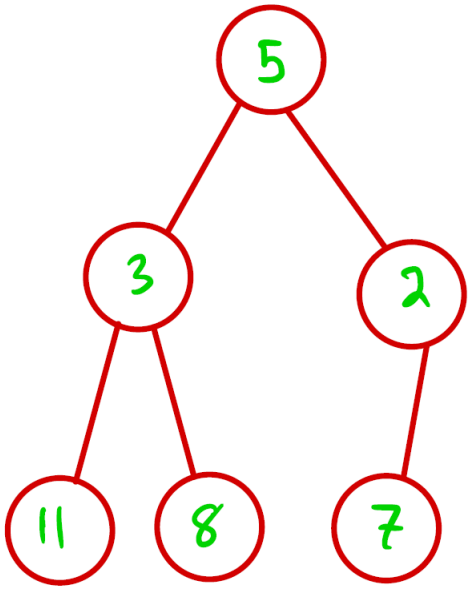
1. Push root onto stack A.
2. While stack A is not empty: pop stack A and push onto stack B, then push all children of popped node from right to left onto stack A.
3. Print contents of stack B.

- **Inorder Traversal** only applies to binary trees which we will discuss next.

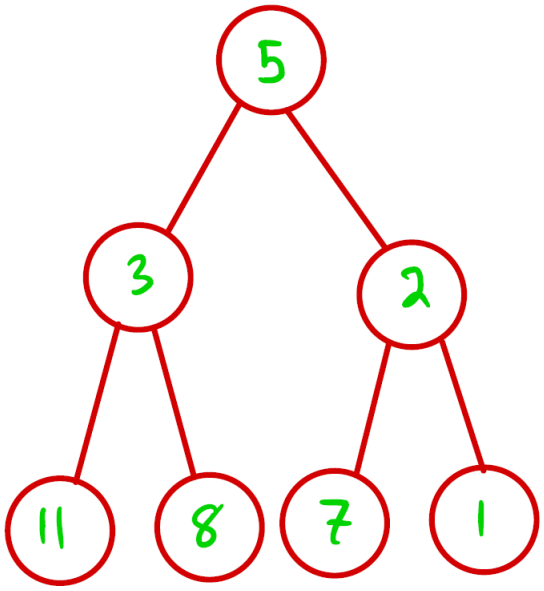
Binary Trees

Binary trees are trees with the additional property that every internal node can have at most two children.

If every internal node has exactly two children we call the binary tree **full**, and we can think about the children of a node as an ordered pair.

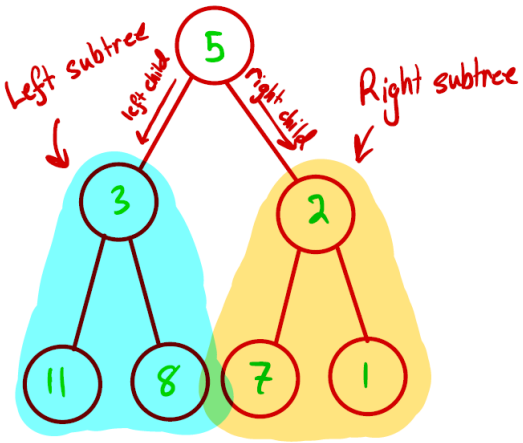


Binary Tree



Full Binary Tree

Additionally, every node in a binary tree has a left and right subtree, as illustrated below:



Properties of Binary Trees

The following is a list of notations used for binary trees:

- n denotes the number of nodes.
- e denotes the number of external nodes.
- i denotes the number of internal nodes.
- d denotes the depth.
- h denotes the height.
- *Level* d of a binary tree denotes all the nodes with a depth of d.

The following is a list of equations and inequalities which hold for binary trees. No motivation or derivation is given in the lecture so its just good to know:

- Level d of a binary tree has at most 2^d nodes.

- $$h + 1 \leq n \leq 2^{h+1} - 1$$
- $$e = i + 1$$
- $$n = 2e - 1$$
- $$h \leq i$$
- $$2h + 1 \leq n \leq 2^{h+1} - 1$$
- $$lg(n + 1) - 1 \leq h \leq (n - 1)/2$$
- $$h + 1 \leq e \leq 2^h$$
- $$h \geq lg(e)$$

Binary Tree Methods

Other than the tree methods that all binary trees inherit, we have some ones specifically for binary trees:

- *left(v:Node):Node;* returns the left child of node v if it exists.
- *right(v:Node):Node;* returns the right child of node v if it exists.
- *hasLeft(v:Node):boolean* returns if the node v has a left child.
- *hasRight(v:Node):boolean* returns if the node v has a right child.

Traversing binary trees can be done in three ways:

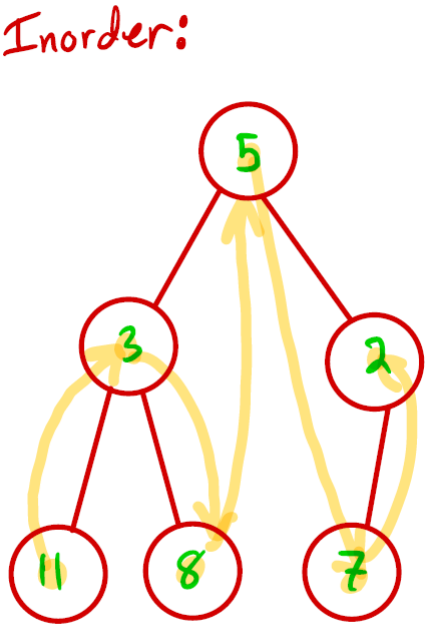
- **Preorder: node, left, right** (as discussed before):

```
Algorithm preorder(v)
   visit(v)
if hasLeft(v)
   preorder(left(v))
if hasRight(v)
   preorder(right(v))
```

- **Postorder: left, right, node** (as discussed before):

```
Algorithm postorder(v)
if hasLeft(v)
   postorder(left(v))
if hasRight(v)
   postorder(left(v))
 visit(v)
```

- **Inorder: left, node, right** which is a new traversal method. Here, we traverse the left subtree of the node first, then we visit the node itself, and then the right subtree.



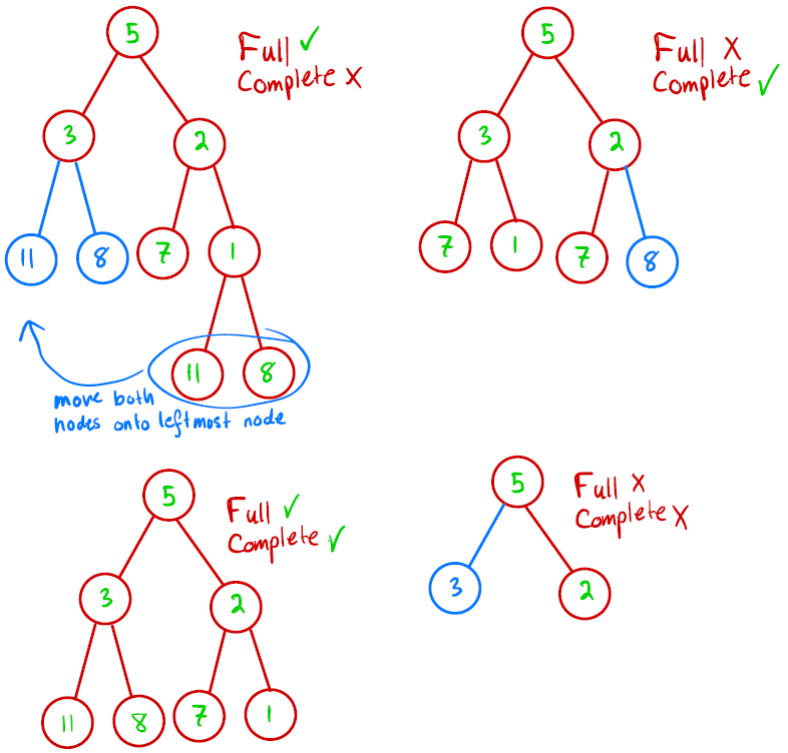

```
Algorithm inOrder(v)
    if hasLeft (v)
        inOrder (left (v))
    visit(v)
    if hasRight (v)
        inOrder (right (v))
```

Complete and Full Binary Trees

A binary tree is **full** if each node is either a leaf or possesses exactly two child nodes.

A binary tree with n levels is **complete** if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.

The following diagram shows four binary trees (in red) with different combinations of full and complete. In blue I have included what must be done to fix the graph to become full and complete. Remember the elements of each node is completely arbitrary and I have only included it to remind you that we are using this data structure to organize elements of a set.



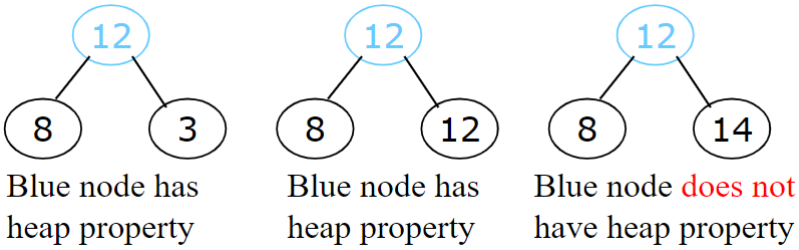
Heaps

Binary heaps (here on referred to as heaps) are array objects that can be viewed as a *nearly complete* binary tree, meaning it just has its last level to fill before it is complete. On the lowest level the tree may be filled from the left up to a point (always fill left to right).

A heap is just a binary tree with some added conditions, mainly: A binary tree is a *heap* if all nodes have the heap property, defined next.

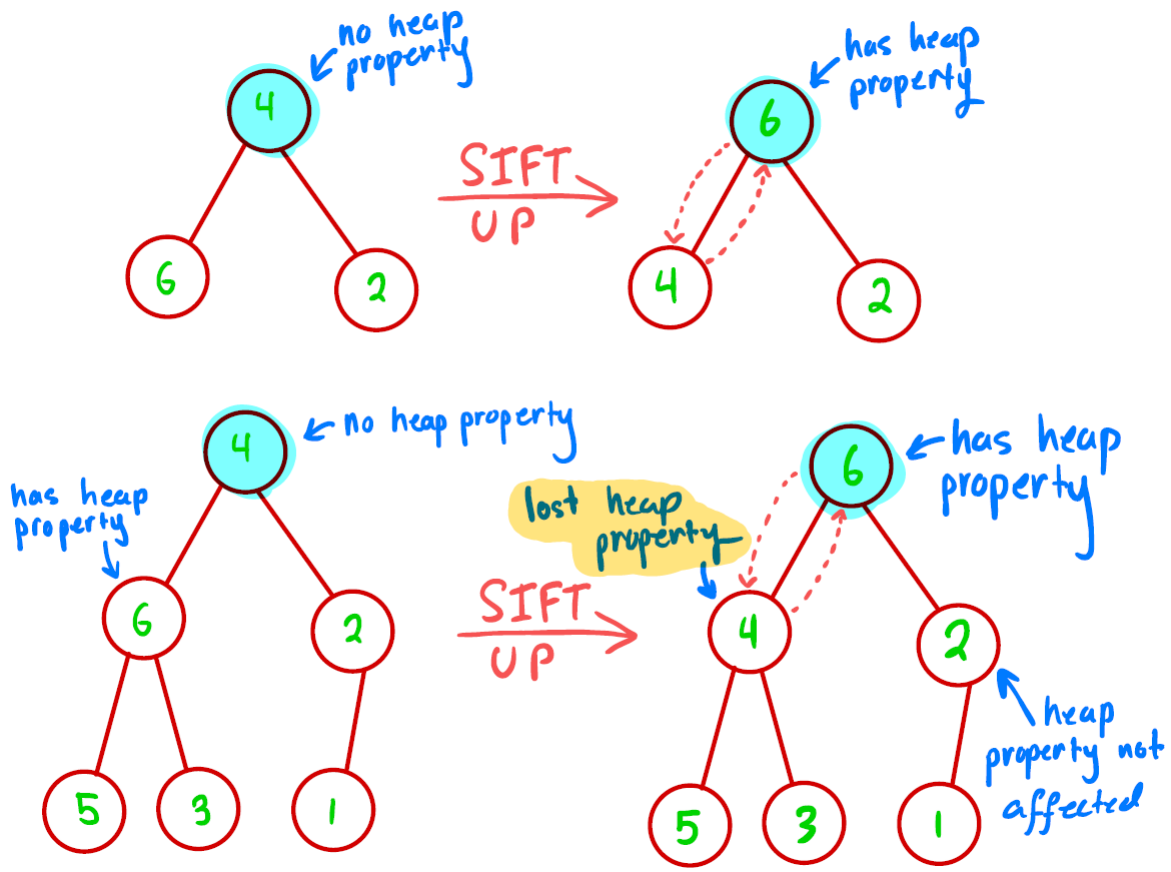
The Heap Property

A node has the (max) heap property if the value in the node is as large as or larger than the values of its children.



All leaf nodes trivially have the heap property.

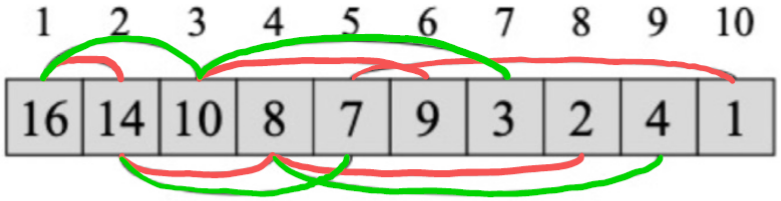
We can give a node (which doesn't currently have the heap property) the heap property by *sifting up* which means exchanging its value with the largest child. By sifting up, the child node may have lost the heap property (since its parent node decreased).

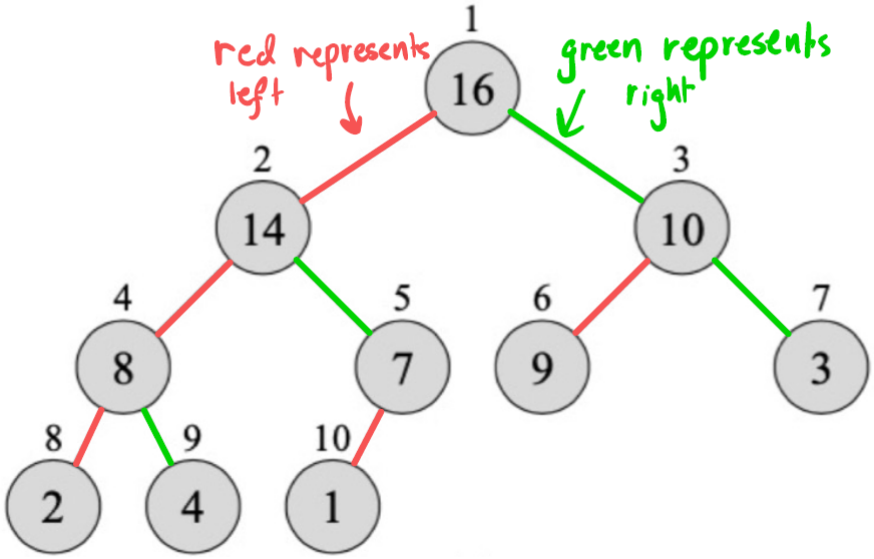


Given all nodes in a binary tree have the heap property (meaning the array is a heap), we have that the root is the greatest element of the array.

Binary Tree as an Array

We can represent a binary tree easily as an array (or vice-versa) using the following pattern as an example:





Note that:

- $A[1]$ is the root of the tree.
- The left child of $A[i]$ is $A[2i]$.
- The right child of $A[i]$ is $A[2i+1]$.
- The parent of $A[i]$ is $A[\lfloor i/2 \rfloor]$.
- For a tree with n nodes, the elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are all leaves. In the example this is index 6 onward.

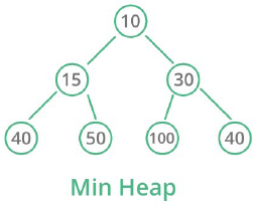
The array has two properties:

- *length* which is just the standard set length of the array.
- *heapLength* which is the length of the array actually being used to represent a heap. This second property will be helpful in heapsort as we decrease *heapLength* but keep *length* the same.

Max/Min Heaps

There are two types of heaps:

1. **Max-Heaps**
   - The heap property ensures that any node is greater than its two children.
   - The root is the largest element.
   - This is the type of heap we have been discussing so far.
2. **Min-Heaps**
   - The heap property ensures that any node is smaller than its two children.
   - The root is the smallest element.
   - For example:



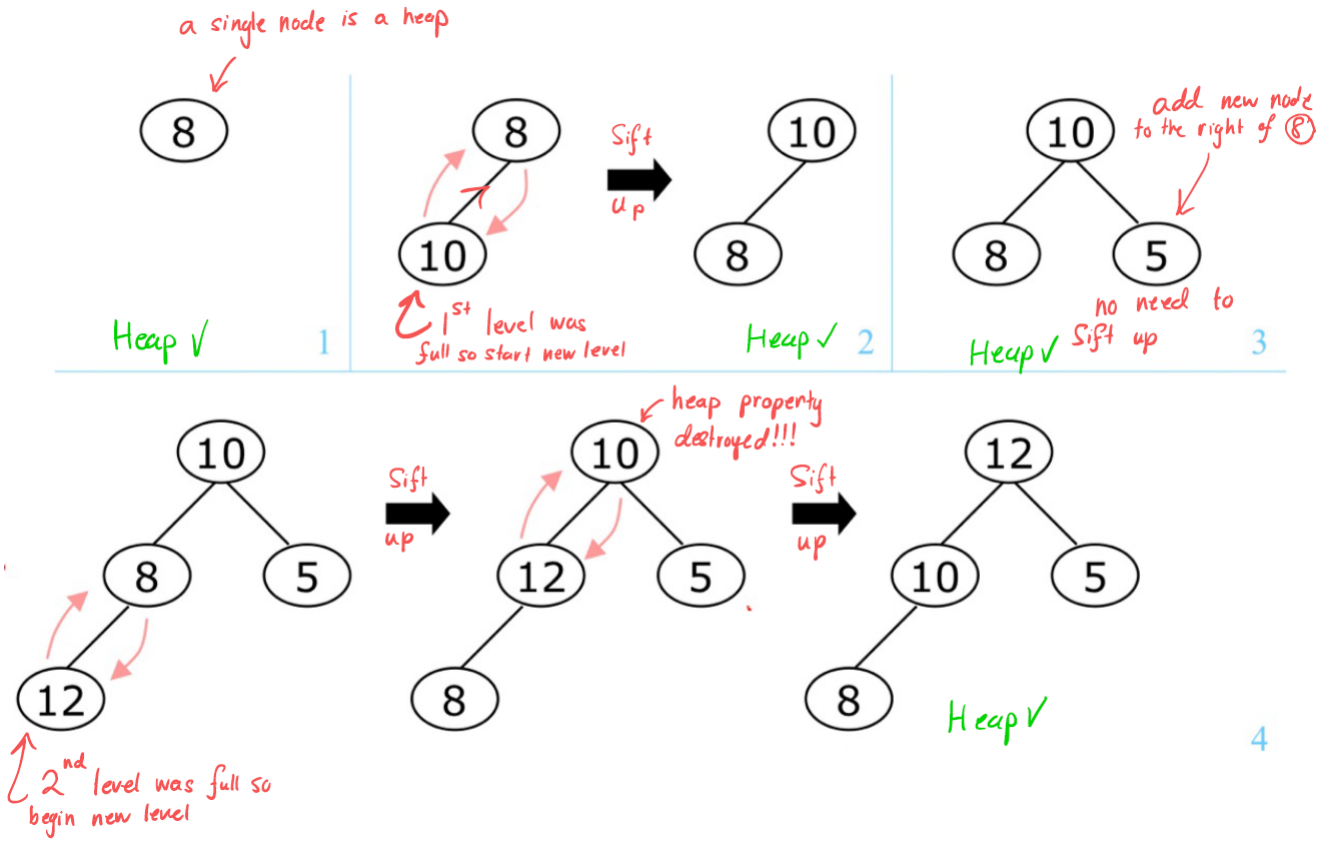
Min Heap

Constructing a Heap: Algorithm

To construct a heap we follow these steps:

1. Begin with a single node (automatically a heap).
2. Add a node directly to the right of the rightmost node in the deepest level. This this level is full, then begin the next level on the leftmost side.
3. Sift up on the parent of the newly added node. Repeat until you sift up to the root or we reach nodes which don't need to be swapped.
4. Repeat the process until you have no more nodes to add to the heap.

For example, let's say we want to make the array $[8, 10, 5, 12,]$ into a heap by this construction algorithm:



During the sift up process, you may expect the heap property to be destroyed for more nodes, but this is not the case. For example we did not have to even check if the heap property was still intact for node 5 in step 4 because it's parent went *up*. This reduces computation complexity.

Constructing a Heap: Code

There are three steps to implementing this:

1. Represent an arbitrary array as a binary tree.
2. Devise a *MaxHeapify(A,i)* algorithm which maintains the heap property of any given node i in a tree A with left and right subtrees *already as given to be heaps*.
3. Devise a *BuildMaxHeap(A)* algorithm that uses *MaxHeapify(A,i)* to construct a heap from an array A.

The following is that code:

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

takes in: 1) array A 2) index i

index of left subtree's root (2i)

index of right subtree's root (2i+1)

if left node > input node

if i isn't the largest

largest is used to tag the index of the largest of i, l, and r

recursively call down the tree

code highlighted in yellow is not relevant until heapsort as at the moment A.heap-size = A.size

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

takes in an array A being represented as a tree

call on all internal nodes from the bottom up

we can skip all the leaves as they all already have the heap property

The complexity of each is:

- **MaxHeapify()**:
  - Time complexity: $O(lg(n))$
- **BuildMaxHeap()**:
  - Time complexity: $O(n)$

Therefore the time complexity of both together is $O(n\ lg(n))$

Heapsort

We have learned about two sorting algorithms so far, and in this section we introduce a new one:

1. **Insertion Sort**:
   - Worst case time complexity $O(n^2)$.
   - Space complexity $O(1)$.
2. **Merge Sort**:
   - Worst case time complexity $O(n\ ln(n))$.
   - Space complexity $O(n)$.
3. **Heapsort**:

- Worst case time complexity $O(n\ ln(n))$.
- Space complexity $O(1)$.

A space complexity of 1 tells us that the array to be sorted is sorted *in place* and no other arrays need to be allocated to allow for the algorithm to be run. Heapsort is the best case of both other sorting algorithms we have learned.

We are now ready to learn Heapsort which comes in two phases:

1. **Phase 1:** Call *BuildMaxHeap(A)* on the array A you want to sort.
2. **Phase 2:**
   - Swap the root (biggest element) with the last element in the array.
   - Decrement the heapSize by 1 (size of the array stays constant.) This causes the biggest element to be put at the end of the array and then forgotten about from the heap.
   - Call MaxHeapify() on the new root (which was the last element).
   - Repeat until there is only one element left (which will be the smallest element)
   - The array is now in ascending order.

The following is the pseudo-code of Heapsort:

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)                    ← Phase 1
2  for i = A.length downto 2            ← Skip the 1st element as it is the smallest
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)                ← this is where heapSize comes in
```

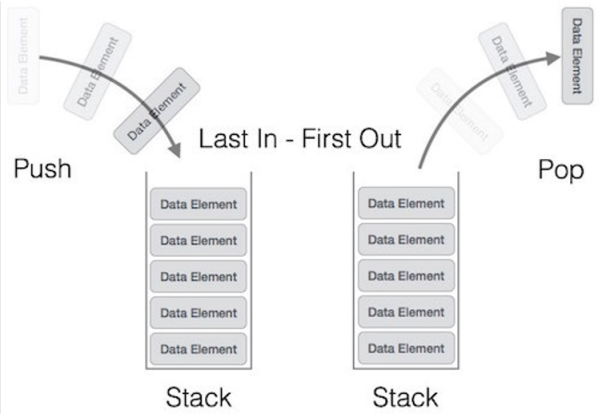
This is where *heapSize* differs from the actual array length. We put the largest element at the end of the heap, and then reduce what section of the array we call the heap by one, essentially discarding the largest element from the heap (but not from the array). The new heap has a new largest element which will repeat the process.

6 Chapter 10: Elementary Data Structures

When studying algorithms (which is a significant part of programming and computer science), we consider dynamic arrays which are sets that change size and have their elements changed often. For this we need data structures, which are ways to represent finite dynamic sets.

Data Structure: Stack

A stack is a data structure which acts like a stack of books. You can stack objects on top of each other, however you can only remove the top element from the stack. This is the *last-in-first-out* principle (LIFO).



There are some commonly supported methods relating to stacks:

- *push(o)*: returns void, inserts object o on top of the stack.
- *pop()*: returns object, removes object from top of the stack.
- *top()*: returns object, returns op object but does not remove the object.
- *size()*: returns in, returns size of Stack (number of elements)
- *isEmpty()*: returns boolean, returns if the Stack has at least one element.

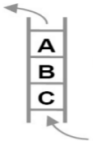
The following is some pseudocode for these methods:

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

The complexity of each operation is $O(1)$. The space used to store this information in memory is $O(n)$.

Data Structure: Queue

A queue is a data structure which is similar to a stack, however it follow the *first in first out* (FIFO) principle. This means that there is a notion of the start and end of a queue, and when you remove an object it comes from the start, and when you add an object it goes to the end.

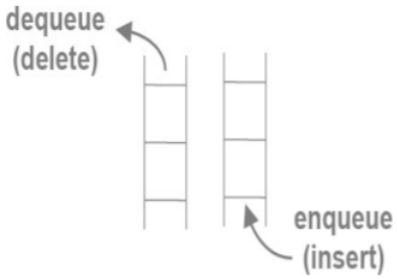


Elements are inserted as the rear (**enqueued**) and removed from the front (**dequeued**).

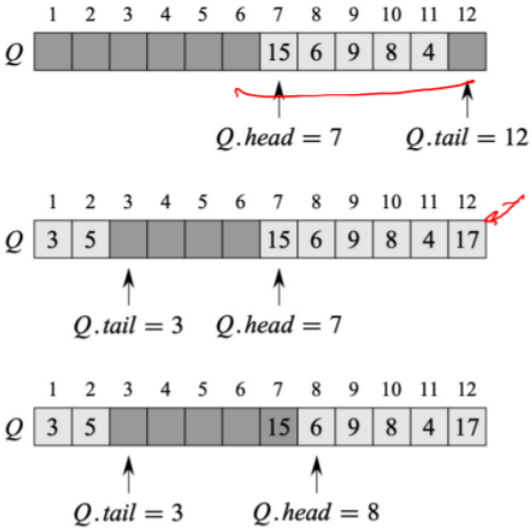
There are some commonly supported methods for queues:

- *enqueue(Q,x)*: inserts element x at the end of the queue.
- *dequeue(Q)*: removes the element at the start of the queue.
- *front()*: returns the element at the front of the queue without removing it.

- *size*: returns integer equal to the size of the queue.
- *isEmpty*: returns boolean related to if the queue is empty.



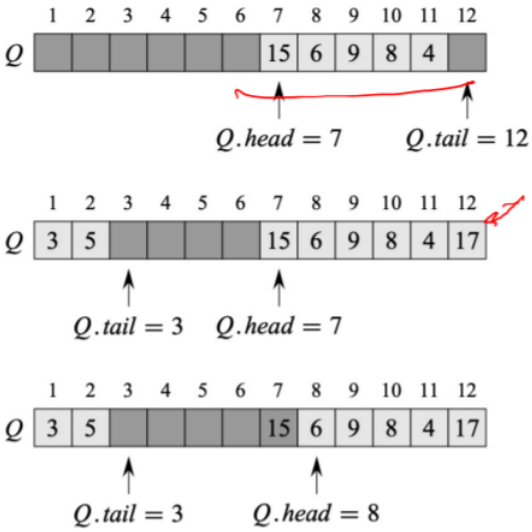
A queue has an attribute *head* which points to the first element, and a *tail* index which points to where a new element will go. Given a queue of size n, locations in the memory are allocated spaces $1...n$. We use the *wrap around* idea to let the queue *head* have a larger index than the *tail* index by wrapping the queue back to the start of the allocated space:



For a queue Q:

- The queue is empty if $Q.head = Q.tail$.
- The queue is full if $(Q.head = Q.tail + 1)$ or $((Q.head = 1)\text{and}(Q.tail = Q.length))$.

The following pseudocode for some of the commonly supported methods can help in understanding how queues work:



Priority Queues

A priority queue is a variation on a normal queues which assigns a priority to each element in the queue. When an element is popped from the queue we take the one with the highest priority, not necessarily the element which has been in the list for the longest. You can imagine however that new element would be generally given a lower priority than older element, however this is not necessarily true (like an emergency waiting room).

You can implement a priority queue using a MaxHeap, where each node stores a priority (called *key*) and also the actual element (maybe an instance of a person object) which is assigned that priority.

Priority Queues have the following supported methods for an array A:

- *HeapMaximum(A)* returns the index of the maximum (highest priority) element in the Heap ($O(1)$).
- *HeapExtractMax(A)* returns the actual object which is the max element and removes it from the queue ($O(lg(n))$).
- *HeapIncreaseKey(A,i,key)* increase the priority of a node i to *key* and make sure the queue is still a max heap ($O(lg(n))$).
- *MaxHeapInsert(A, key)* insert a new element into the queue with a priority of *key* ($O(lg(n))$).

The following is the pseudocode for all the methods (note that they all assume that the only thing at the node is the priority key, however you can add an object there easily in actual code):

```
HEAP-MAXIMUM(A)
1  return A[1]
```
← the 1st element of the heap has the largest priority.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```
← make sure there is at least 1 element in the queue

very similar to heapsort where we switch the max and last elements and decrease the heapSize by 1

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```
← make sure new key is larger

← assign the new key

maintain the heap property in the queue

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```
increase the heapSize

← add a new element with lowest possible priority

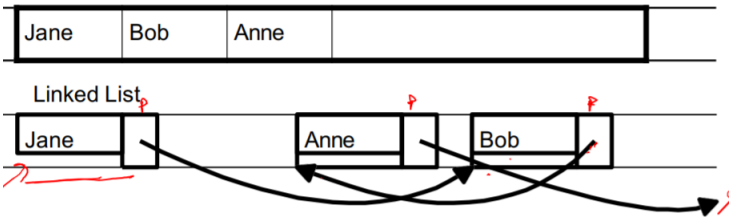
↑ increase the priority of that new element to key

Data Structure: Linked List

A linked list is a fundamental data structure which is used to construct things like *Stacks* and *Queues*.

Linked lists consist of structures called nodes which contain a *key* (come particular value relevant to the context), and then one or more references to the address(es) of other node(s). A linked list is an alternate approach to an array of object, where instead of indexing them we link them through variables which contains references.

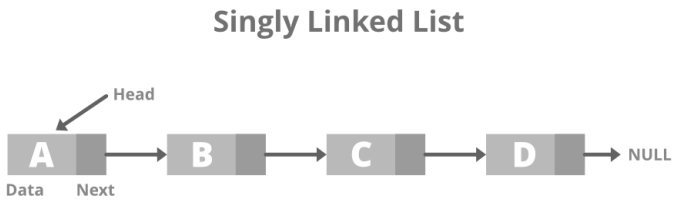
Rather than having to allocate a large section of memory for an array, we dynamically link objects together as part of the linked list.



A node's predecessor is the previous node in the sequence, and a node's successor is the next node in the sequence. The length of a list is the number of elements in it.

Singly Linked Lists

In a singly linked list, each node has a key, and a reference *next* to the next node in the list (the successor).



The last node in the linked list contains null as its next pointer, and the pointer variable *head* points to the address of the first element in the linked list.

Its helpful to look at some pseudocode relating to linked lists to understand their functionality:

- The following code searches the linked list L for a key k:

```
listSearch(L,k){
        x = L.head
        while ((x != NULL) && (x.key != k)){
                x = x.next
        }
        return x
}
```

- The following code inserts a node at the beginning of a list L with a key x:

```
insert(L,x){
        Node u = new Node()
        u.key = x
        u.next = head
        head = u
}
```

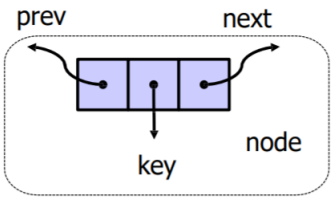
- The following code searches a linked list L for a node with key x and then removes it from the list:

```
delete(L,x){
        if (L.head == NULL){
                return
        }
        u = L.head
        if (u.key = x){
                head = u.next
                return
        }
        while (u.next != NULL) {
                prev = u
                u = u.next
                if (u.key == x){
                        prev.next = u.next
```

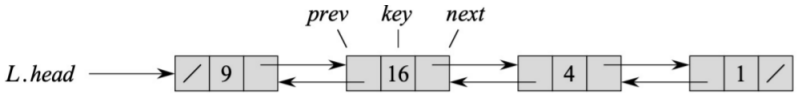
```
                        return
                }
        }
}
```

Doubly Linked Lists

A doubly linked list is a linked list where each node had a key, and **two** pointers, one to the predecessor, and one to the successor in the list. The structure of each node looks like:



The structure of the whole list looks like (with some example keys):



The first node in the list has null in its previous pointer, and the final node in the list has null in its next pointer.

- The following code searches a doubly linked list L for a key k:

```
listSearch(L,k){
        x = L.head
        while ((x != NULL) && (x.key != k)){
                x = x.next
        }
        return x
}
```

- The following code inserts a node x at the beginning of a doubly linked list L:

```
listInsert(L,x){
        x.next = L.head
        if (L.head != NULL){
                L.head.prev = x
        }
        L.head = x
        x.prev = NULL
}
```

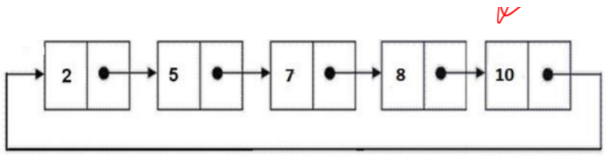
- The following code deletes a node x from a linked list L:

```
listDelete(L,x){
        if (x.prev != NULL){
                x.prev.next = x.next
        } else {
                L.head = x.next
        }
        if (x.next != NULL){
                x.next.prev = x.prev
        }
}
```

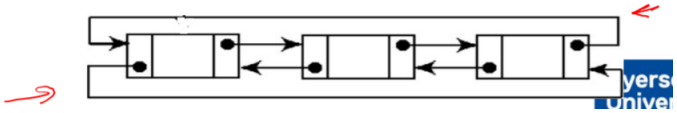
Circular Linked Lists

A circular linked list is a linked list where you set the pointer to the last node to be the address of the first node. This makes all nodes accessible from any starting node, and makes it so that you can loop through all the nodes by going from node to node. You can have singly linked or doubly linked circular linked lists:

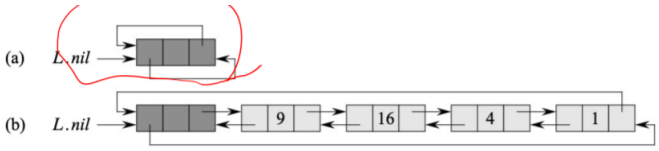
Circular Singly linked list



Circular Doubly linked list



We often use a sentinel node to denote where the beginning of the circular linked list begins. If the sentinel node wasn't there, then the would be no notion of *start* and *end* of the list. We call this object *nill* and we set its next pointer to the head of list and its prev pointer to the tail of the list:



- The following code searches a circularly linked list L for a key k:

```
listSearch(L,k){
        x = L.nil.next
        while ((x != NULL) && (x.key != k)){
                x = x.next
        }
        return x
}
```

- The following code inserts a node x at the start of a circularly linked list L:

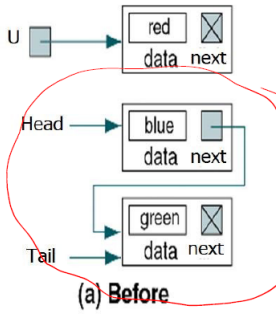
```
listInsert(L,x){
        x.next = L.nil.next
        L.nil.next.prev = x
        L.nil.next = x
        x.prev = L.nil
}
```

- The following code deletes a node x from a circularly linked list L:

```
x.prev.next = x.next
x.next.prev = x.prev
```

Representing Stacks by Linked Lists

We can use a linked list structure to represent a stack:

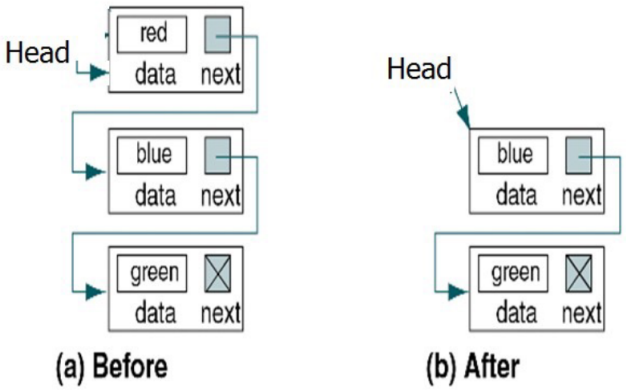


In general, we use a pointer variable to point to the address of the first node in the list, meaning the top of the stack. The pseudocode for the functions reflects this:

```
push(x)
    u←newnode(x)
    u.next←head
    head←u
    n←n+1
```

Here, x represents a new key being pushed. The newnode(x) function takes in a key and creates a node object with that key which we assign to u. The pop method would be done as follows:

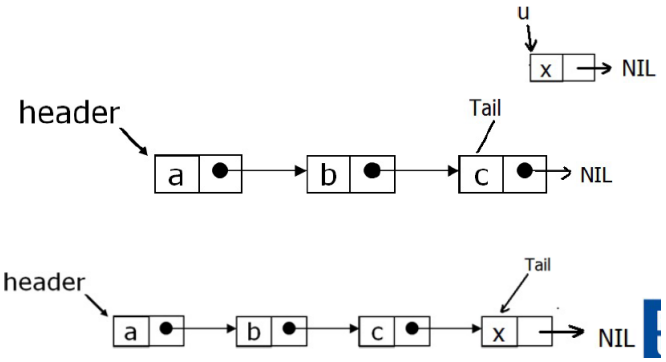
```
pop()
    if n = 0 then return nil
    x ← head.x
    head ← head.next
    n ← n − 1
    return x
```



Representing Queues by Linked Lists

A singly linked list can also be used to implement a Queue with its supported methods. The dequeue method is exactly the same like popping from a stack, and the enqueue method is described below (called add(x)):

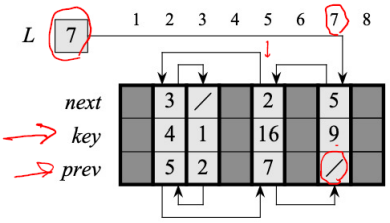
```
add(x)
    u ← new_node(x)
    if n = 0 then
        head ← u
    else
        tail.next ← u
    tail ← u
    n ← n + 1
    return true
```



Representing Objects by Multiple Arrays

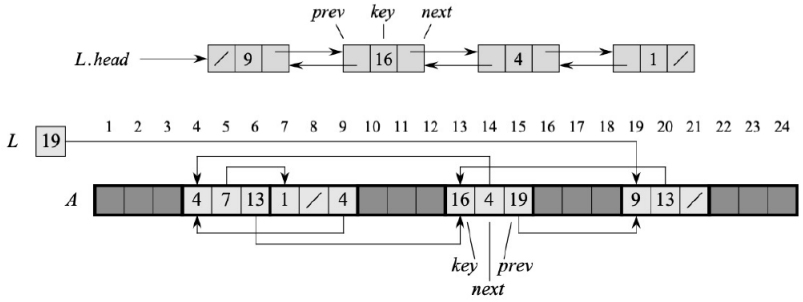
Given a set of objects with some characteristics, we can model this object by a collection of arrays where the index represents which object we are talking about, and there is one array for each characteristic.

Take the following for example:



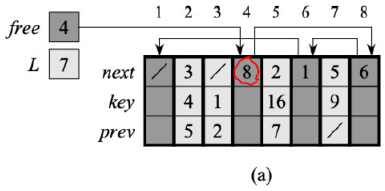
Here we are implementing a doubly linked list using 3 arrays. The head variable points to the index of the first element of the doubly linked list. Every vertical slice of the 3 arrays represents one instance of the object.

You can also use a single array to represent an object as in the following:



Here, you contextually know that the first element, second element, and so on each have a unique meaning.

Based off of the multiple array model of storing data objects we also want a way to keep track of where the next free spot is in the structure. We do this by making a linked list which contains only a *next* element which points to the next available spot. This free linked list acts like a stack. The head of this linked list points to the next index to be filled:



(a)

We have tow supported functions related to the *free* linked list:

```
ALLOCATE-OBJECT()
1  if free == NIL
2       error "out of space"
3  else x = free
4       free = x.next
5       return x

FREE-OBJECT(x)
1  x.next = free
2  free = x
```

- AllocateObject() returns the index of the location in the array which is to be filled by the next object, and makes the linked list point to the next address along the free linked list.
- FreeObject(x) takes in an index and adds it to the front of the free linked list by making its next variable point to the *current* first free index, and the making the first free index equal to the x index.

7 Chapter 11: Hash Tables

The idea of a hash table is similar to a dictionary, a big list of indexes (words) and definitions.

A dictionary is a data structure which supports three operations:

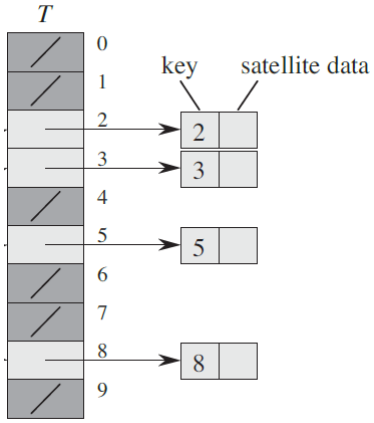
- *Search(S,t)*: which searches a dictionary S for an element who's key is k.
- *Insert(S,x)*: which adds the element pointed to by x to the dictionary S.
- *Delete(S,x)*: which deletes the element pointed to by x from S.

Every element in a dictionary is made up of a *key* and *satellite data*, just like a real dictionary which is made up of *words* and *definitions*. If you wanted to know the definition of a word, you would search for the key, and then relay the data.

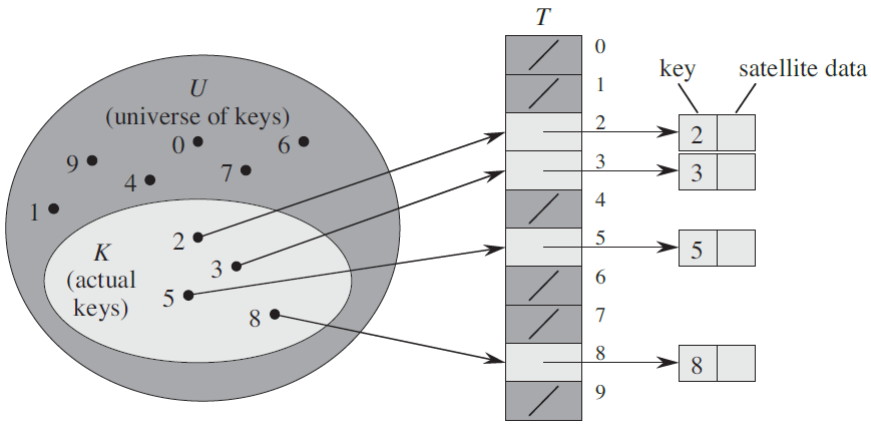
Before we talk about hash tables, we need to understand why just using basic arrays doesn't work for certain applications.

Direct Addressing

Let's say you are collecting age data in an elementary school and you make an int array T where the value (called *satellite data*) at index k is the amount of k year olds (let's say we only measure up to and including age 9). Then your array may look something like this.



We can think of this more abstractly as a way to map a subset of a universe of data into an array, where we use the actual key value as the index (like we used the key as the index).



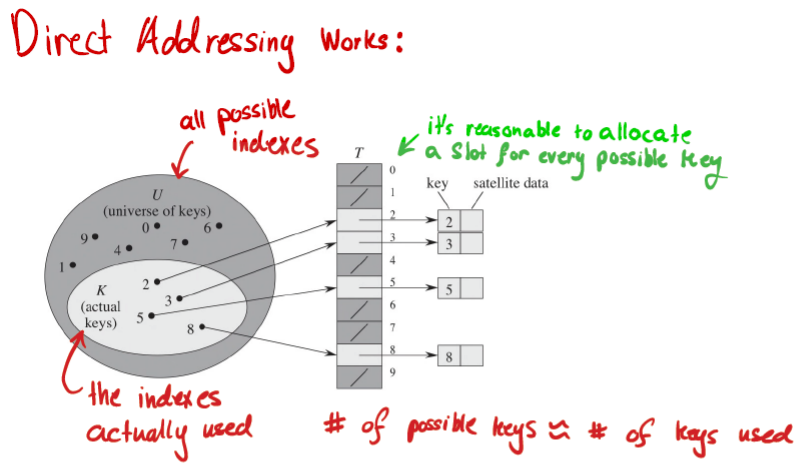
This is a *dictionary* since it is a list of keys (the ages) and values (the count of each age), and using *Direct Addressing* it is trivial to implement the dictionary operations:

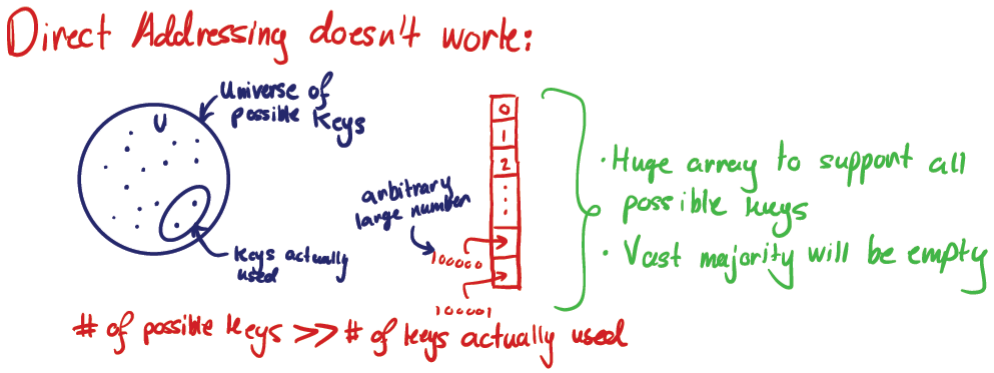
```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Direct addressing tables work because instead of storing the key and data in an external object, we store them directly in an array, and we use the key as the index. It is thus unnecessary to store the key of an object in a direct address table because it is just the index.

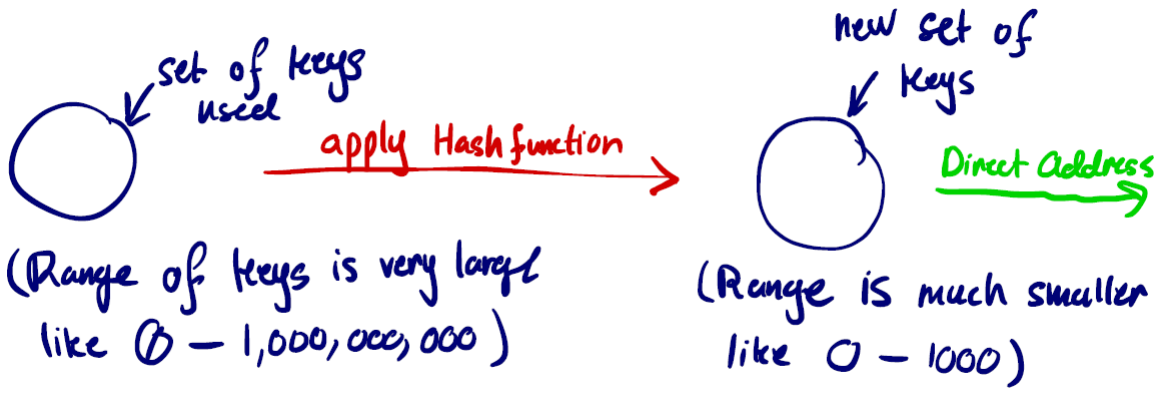
Intro to Hashing

What if we are now looking at the ages of stars in the universe. We then have a **huge** range of ages (from 0 to billions) and so we would need a **huge** array (one index for each age). This is a massive waste of space and computer power especially since you may just want to study a could hundred stars, most element of the array will be null. The following graphic summarizes these ideas:





There is a better way and its called a hash table. The big picture idea is this:



It works like this:

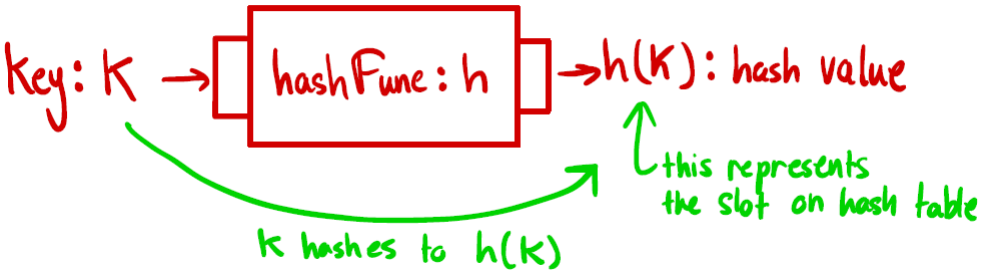
- We take our original set of keys which is only a small subset of the universe of possible keys (like all the words in a book compared to all the words that have ever existed).
- We apply a Hash function to each element of that set. The hash function is a function which takes in any kind of object as input, and then returns an int. Very importantly:
  - The hash function's output is bounded.
  - The hash function's output is evenly distributed throughout the range.

  The exact code of the hash function changes for every application, for example a hash function could take in a word and convert it to an int by adding together all the letters as numbers ($A = 1, B = 2, C = 3$, and so on), and then taking the modulus of that result to keep it bounded. We will discuss hash functions more later on.
- Now that we have a set which derives from the original, however now the keys are significantly smaller, we can direct address this into an array called a **Hash Table**.

There are a lot of details to go through in Hashing which we will do in the following sections. For example you may notice that we are using a function to turn a large set into a smaller set, and so **somewhere** there **must** be overlap (assuming a lot of the universe is used), we deal with this in the section called **Collisions**. Hashing is very important for modern day encryption.

With a direct addressing table we can do all the dictionary operations in $O(1)$ worse case time. With a hash table we can use significantly less storage to also do all the dictionary operations in $O(1)$ **average** case time.

Hash functions take in a key which may by very large and outputs an index in the hash table (which is smaller). The following diagram illustrates this and shows some terminology used:



You can informally think of the hash function as a way to break down the huge set of data into **categories** so that each category has minimal (ideally one) elements.

Collision Solution by Chaining

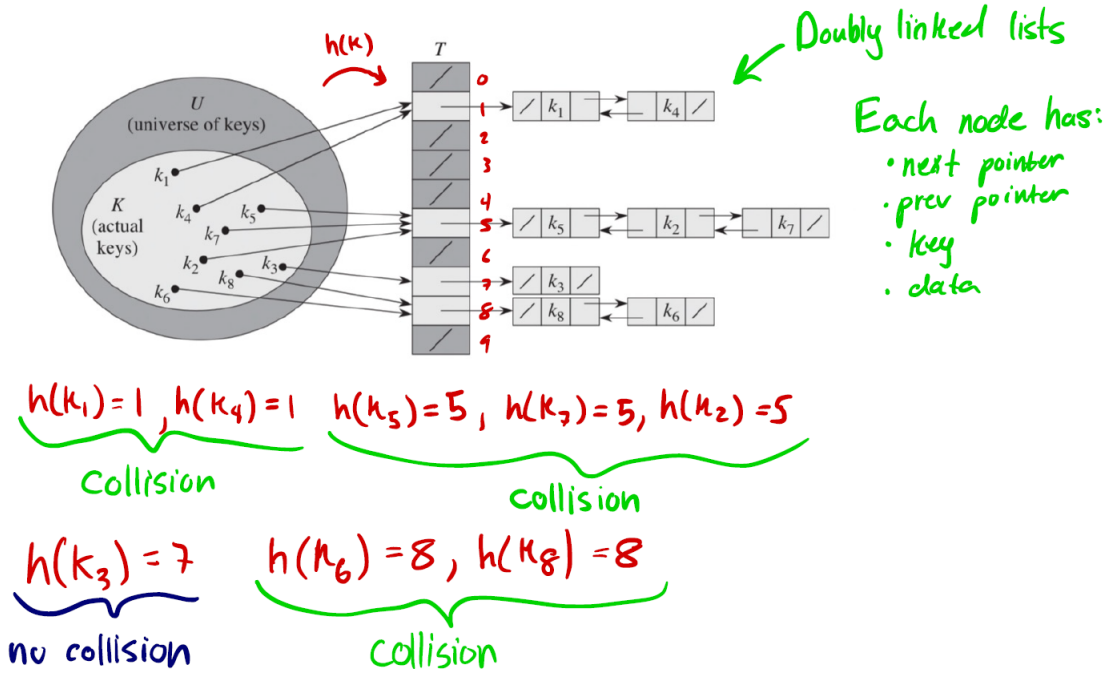
Collisions are inevitable, given that the set of all possible keys is greater than the length of the hash table (which is the whole point). A good hash function will minimize collisions by evenly distributing hash values into the different table indexes, but it is impossible to completely avoid collisions.

Using the hash function example I used earlier which turns words into numbers ($A = 1, B = 2, C = 3$, and so on) then:

$$h(\text{"}math\text{"}) = 13 + 1 + 20 + 8 = 42 = 2 + 21 + 19 = h(\text{"}bus\text{"})$$

You can see that "math" and "bus" would collide on the hash table, as they would go to the same address of 42.

One solution to this is called **Chaining**. This is where we place all the elements of the same slot into the same linked list. Every slot of the hash table contains a pointer to the head of the linked list. When we want to add an element to the slot, we add to the head of the linked list. The following diagram summarizes this:



Implementing dictionary operations using a hash table with collision handling done by chaining is done as follows:

Given a hash table T, a key-data pair x, and a key k:

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

Note that insertion assumes that the key-data pair is not already in the linked list, which may not always be true and ends up wasting memory by have duplicate nodes in the double linked list. You can get around this by adding a search call before you insert but this of course has additional cost.