Formal DSA in C++

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

Preface

1.1 Universe

In the context of data structures and algorithms, the term universe refers to the complete set of all possible elements or values that could be involved in a particular problem or data structure. It defines the range or domain from which data can be selected. We denote the universe with a capital U

1.2 Dynamic and static sets

A dynamic set is a data structure that supports not only membership queries but also allows the insertion, deletion, and sometimes modification of elements over time. Unlike static sets, which are fixed once defined, dynamic sets can change as elements are added or removed

Elementary complexity theory

- **Idea**: The same problem can frequently be solved with algorithms that differ in efficiency. The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called computational complexity was developed by Juris Hartmanis and Richard E. Stearns

  Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is. This cost can be measured in a variety of ways, and the particular context determines its meaning. This book concerns itself with the two efficiency criteria: time and space. The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. However, the most inefficient algorithm run on a Cray computer can execute much faster than the most efficient algorithm run on a PC, so run time is always system-dependent. For example, to compare 100 algorithms, all of them would have to be run on the same machine. Furthermore, the results of run-time tests depend on the language in which a given algorithm is written, even if the tests are performed on the same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in C or Ada may be 20 times faster than the same program encoded in BASIC or LISP.

- **Units**: To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data should be used

  If there is a linear relationship between the size n and time t, that is, $t_1 = cn_1$, then an increase of data by a factor of 5 results in the increase of the execution time by the same factor. If $n_2 = 5n_1$, then $t_2 = 5t_1$

  Similarly, if $t_1 = \log_2 n$, then doubling n increases t by only one unit of time. Therefore, if $t_2 = \log_2(2n)$, then $t_2 = t_1 + 1$.

- **Eliminating insignificant terms**: A function expressing the relationship between n and t is usually much more complex, and calculating such a function is important only in regard to large bodies of data; any terms that do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data.

- **Asymptotic complexity**: This measure of efficiency is called asymptotic complexity and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found

- **Big-O Notation**: The most commonly used notation for specifying asymptotic complexity—that is, for estimating the rate of function growth—is the big-O notation introduced in 1894 by Paul Bachmann.

  Given two positive-valued functions f and g, consider the following definition:

$f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant N$.

$$f(n) \text{ is } O(g(n)) \iff \exists\ c, N \in \mathbb{Z}^+ \mid f(n) \leq cg(n)\ \forall\ n \geq N.$$

Big-O notation says that for large enough n, the function $f(n)$ does not grow faster than a constant multiple of $g(n)$. So, $g(n)$ provides an upper bound on how fast $f(n)$ can grow as n increases.

In other words, f is big-O of g if there is a positive number c such that f is not larger than $c \cdot g$ for sufficiently large ns; that is, for all ns larger than some number N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that, in the long run, f grows at most as fast as g.

The problem with this definition is that, first, it states only that there must exist certain c and N, but it does not give any hint of how to calculate these constants. Second, it does not put any restrictions on these values and gives little guidance in situations when there are many candidates. In fact, there are usually infinitely many pairs of c's and N's that can be given for the same pair of functions f and g.

For example, suppose

$$f(n) = 2n^2 + 3n + 1 = O(n^2).$$

Where $g(n) = n^2$. Candidate values for c and N are

c	$\geqslant 6$	$\geqslant 3\frac{3}{4}$	$\geqslant 3\frac{1}{9}$	$\geqslant 2\frac{13}{16}$	$\geqslant 2\frac{16}{25}$	$\cdots$	$\rightarrow$	2
N	1	2	3	4	5	$\cdots$	$\rightarrow$	∞

We obtain these values by solving the inequality:

$$2n^2 + 3n + 1 \leqslant cn^2.$$

Or equivalently

$$2 + \frac{3}{n} + \frac{1}{n^2} \leqslant c.$$

For different n's

For large n, the terms $\frac{3}{n}$ and $\frac{1}{n^2}$ get smaller. Let's find N such that for all $n \geqslant N$, the right-hand side stays bounded.

As n gets larger, $\frac{3}{n}$ and $\frac{1}{n^2}$ approach zero. To simplify the analysis, choose $N = 1$ initially and check how small $\frac{3}{n}$ and $\frac{1}{n^2}$ are:

$$2 + \frac{3}{1} + \frac{1}{1^2} = 2 + 3 + 1 = 6.$$

From the inequality, at $N = 1$, we have $6 \leqslant c$. Therefore, we can choose $c = 6$. This ensures that for all $n \geqslant 1$, the inequality holds:

$$2 + \frac{3}{n} + \frac{1}{n^2} \leqslant 6.$$

Thus, you can choose $c = 6$ and $N = 1$.

different pairs of constants c and N for the same function $g(= n^2)$ can be determined.

- **Choosing the best c, N**: To choose the best c and N, it should be determined for which N a certain term in f becomes the largest and stays the largest.

  In the example above, The only candidates for the largest term are $2n^2$ and $3n$; these terms can be compared using the inequality $2n^2 > 3n$ that holds for $n > 1.5$. Thus, $N = 2$ and $c \geqslant \frac{15}{4} = 3.75$.

- **Significance**: What is the practical significance of the pairs of constants just listed? All of them are related to the same function $g(n) = n^2$ and to the same $f(n)$. For a fixed g, an infinite number of pairs of c's and N's can be identified. The point is that f and g grow at the same rate. The definition states, however, that g is almost always greater than or equal to f if it is multiplied by a constant c. "Almost always" means for all n's not less than a constant N. The crux of the matter is that the value of c depends on which N is chosen, and vice versa.

- **Inherent imprecision: Choosing best $g(n)$**: The inherent imprecision of the big-O notation goes even further, because there can be infinitely many functions g for a given function f. For example, the f from Equation 2.2 is big-O not only of n^2, but also of n^3, n^4, ..., n^k, ...for any $k \geqslant 2$. To avoid this embarrassment of riches, the smallest function g is chosen, n^2 in this case.

- **Big-o as approximating terms**: The approximation of function f can be refined using big-O notation only for the part of the equation suppressing irrelevant information. For example, in the equation below, the contribution of the third and last terms to the value of the function can be omitted

$$\begin{aligned} & f(n) = n^2 + 100n + \log(n) + 1000 \\ \implies & f(n) = n^2 + 100n + O(\log(n)). \end{aligned}$$

  Similarly,

$$\begin{aligned} & f(n) = 2n^2 + 3n + 1 \\ \implies & f(n) = 2n^2 + O(n). \end{aligned}$$

  This equation says that for large values of n, the expression $2n^2 + 3n + 1$ behaves like $2n^2$ plus some terms that grow linearly or slower (captured by $O(n)$). The exact contributions of $3n$ and 1 are not important for asymptotic analysis; what matters is that their growth is slower compared to $2n^2$.

- **Algorithm analysis: Most common time complexities**: Ranked slowest to fastest growth
  - $O(1)$**:** Constant time
  - $O(\log(\log(n)))$: Logarithmic time
  - $O(\log(n))$: Logarthmic time
  - $O(n)$: Linear time
  - $O(n\log(n))$: Log-linear time
  - $O(n^k)$, $k > 1$: Polynomial time
  - $O(a^n)$, $a > 1$: Exponential time
  - $O(n!)$: Factorial time

- **Ranking complexities from slowest to fastest: Process**: Given

  (a) $O(25)$
  (b) $O(n^{\frac{1}{2}} + \log^2(n))$
  (c) $O(\log^{200}(n))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(n^{200} + 3^n)$
  (f) $O(n \log^{40}(n))$
  (g) $O(4^n \log(n))$
  (h) $O(n^3 \log(\log(n)))$

  How can we go about sorting these slowest to fastest. Well, to start, in the expressions with plus or minus, we can throw out the slower terms. Thus,

  (a) $O(n^{\frac{1}{2}})$
  (b) $O(25)$
  (c) $O(\log^{200}(n))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(3^n)$
  (f) $O(n \log^{40}(n))$
  (g) $O(4^n \log(n))$
  (h) $O(n^3 \log(\log(n)))$

  In product terms, we disregard the slower term unless there are complexites with the same dominant term. For example, $O(n^3 \log(\log(n)))$ grows slower than $O(n^3 \log^4(n))$ because although they have the same dominant term n^3, $\log(\log(n))$ grows slower than $\log^4(n)$. Thus, the correct sequence is

  (b) $O(25)$
  (c) $O(\log^{200}(n))$
  (a) $O(n^{\frac{1}{2}} + \log^2(n))$
  (f) $O(n \log^{40}(n))$
  (h) $O(n^3 \log(\log(n)))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(n^{200} + 3^n)$
  (g) $O(4^n \log(n))$

- **Properties of Big-O notation**

  1. **Transitivity**: If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

     Proof: According to the definition, $f(n)$ is $O(g(n))$ if there exist positive numbers c_1 and N_1 such that $f(n) \leqslant c_1 g(n)$ for all $n \geqslant N_1$, and $g(n)$ is $O(h(n))$ if there exist positive numbers c_2 and N_2 such that $g(n) \leqslant c_2 h(n)$ for all $n \geqslant N_2$. Hence, $c_1 g(n) \leqslant c_1 c_2 h(n)$ for $n \geqslant N$ where N is the larger of N_1 and N_2. If we take $c = c_1 c_2$, then $f(n) \leqslant ch(n)$ for $n \geqslant N$, which means that f is $O(h(n))$.

  2. **Addition**: If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

Proof: If $f(n) \leqslant c_1 h(n)$, and $g(n) \leqslant c_2 h(n)$, then $f(n) + g(n) \leqslant c_1 h(n) + c_2 h(n) \leqslant (c_1 + c_2) h(n)$. ℓet $c = c_1 + c_2$, then $f(n) + g(n) \leqslant ch(n)$ and $f(n) + g(n)$ is $O(h(n))$

3. **Polynomial bounds**: The function an^k is $O(n^k)$

   Proof: $an^k \leqslant cn^k$ for $c \geqslant a$. Since we can always find some constant $c \geqslant a$, an^k is $O(n^k)$

   Observation: For $an^k \leqslant cn^k$ to hold, $c \geqslant a$ is necessary

4. **Domination of higher-degree polynomials**: n^k is $O(n^{k+j})\ \forall\ j > 0$

   This statement holds if $c = N = 1$

   It follows from all these facts that every polynomial is big-O of n raised to the largest power, or

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \text{ is } O(n^k).$$

5. **Logs**: The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \neq 1$.

   This correspondence holds between logarithmic functions. The fact above states that regardless of their bases, logarithmic functions are big-O of each other; that is, all these functions have the same rate of growth.

   Proof: Let $\log_a(n) = x$, and $\log_b(n) = y$, then $a^x = n,\ b^y = n.$ Take the natural log of both sides

$$\begin{aligned} \ln(a^x) = \ln(n) \quad \ln(b^y) = \ln(n) \\ \implies x\ln(a) = \ln(n) \quad y\ln(b) = \ln(n) \\ \implies x\ln(a) = y\ln(b). \end{aligned}$$

   Since $x = \log_a(n)$, and $y = \log_b(n)$, then we have

$$\begin{aligned} \ln(a)\log_a(n) &= \ln(b)\log_b n \\ \log_a(n) &= \frac{\ln(b)}{\ln(a)}\log_b(n). \end{aligned}$$

   ℓet $c = \frac{\ln(b)}{\ln(a)}$, then $\log_a(n) = c\log_b(n)$, which proves that $\log_a(n)$ and $\log_b(n)$ are multiples of each other. Thus, $\log_a(n)$ is $O(\log_b n)$

   Note: Because the base of the logarithm is irrelevant in the context of big-O notation, we can always use just one base.

$$\therefore \log_a(n) \text{ is } O(\mathrm{lg} n).$$

   For any positive $a \neq 1$, where $\lg(n)$ is $\log_2(n)$

- **Big-Ω**. The function $f(n)$ is $\Omega(g(n))$ iff $\exists\ c, N \in \mathbb{R}^+ \ \mid\ f(n) \geqslant cg(n)\ \forall\ n \geqslant N$.

In other words, $cg(n)$ is a lower bound on the size of $f(n)$, or, in the long run, f grows at least at the rate of g

There is an interconnection between these two notations expressed by the equivalence

$$f(n) \text{ is } \Omega(g(n)) \text{ iff } g(n) is O(f(n)).$$

There are an infinite number of possible lower bounds for the function f; that is, there is an infinite set of gs such that $f(n)$ is $\Omega(g(n))$ as well as an unbounded number of possible upper bounds of f. This may be somewhat disquieting, so we restrict our attention to the smallest upper bounds and the largest lower bounds. Note that there is a common ground for big-O and Ω notations indicated by the equalities in the definitions of these notations: Big-O is defined in terms of "$\leqslant$" and Ω in terms of "$\geqslant$"; "$=$" is included in both inequalities. This suggests a way of restricting the sets of possible lower and upper bounds.

- **Big-Θ**: $f(n)$ is $\Theta(g(n))$ iff $\exists\ c_1, c_2, N \in \mathbb{R}^+ \ \mid \ c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\ \forall\ n \geqslant N$

  We see that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

  When applying any of these notations, do not forget that they are approximations that hide some detail that in many cases may be considered important.

- **Double O notation**: f is $OO(g(n))$ if it is $O(g(n))$ and the constant c is too large to have practical significance. Thus, $10^8 n$ is $OO(n)$. However, the definition of "too large" depends on the particular application.

- **Using asymptotic complexity to estimate time**: If an algorithm is $O(n^2)$, the time to process n elements is proportional to n^2.

  Let $T(n)$ represent the time, so $T(n) = k \cdot n^2$ where k is a constant.

  To find the time for 1 million elements ($n = 10^6$):

  $$T(10^6) = k \cdot (10^6)^2 = k \cdot 10^{12}$$

  For example, if processing 1000 elements takes 1 second, then:

  $$T(1000) = k \cdot 1000^2 = k \cdot 10^6 \implies k = \frac{1}{10^6}$$

  Now, for $n = 10^6$:

  $$T(10^6) = \frac{1}{10^6} \cdot (10^6)^2 = 10^6 \text{ seconds} = 1,000,000 \text{ seconds} \approx 11.57 \text{ days}.$$

- **Finding asymptotic complexites**: Asymptotic bounds are used to estimate the efficiency of algorithms by assessing the amount of time and memory needed to accomplish the task for which the algorithms were designed. This section illustrates how this complexity can be determined. In most cases, we are interested in time complexity, which usually measures the number of assignments and comparisons performed during the execution of a program. For now let's focus on assignments

  Consider a simple loop to calculate the sum of numbers in an array

```
for (i = sum = 0; i < n; i++)
    sum += a[i];
```

First, two variables are initialized, then the for loop iterates n times, and during each iteration, it executes two assignments, one of which updates sum and the other of which updates i. Thus, there are $2 + 2n$ assignments for the complete run of this for loop; its asymptotic complexity is $O(n)$.

Complexity usually grows if nested loops are used, as in the following code, which outputs the sums of all the subarrays that begin with position 0:

```
for (i = 0; i < n; i++) {
    for (j = 1, sum = a[0]; j <= i; j++)
        sum += a[j];
    cout<<"sum for subarray 0 through "<< i <<" is
↪   "<<sum<<endl;
}
```

Before the loops start, i is initialized. The outer loop is performed n times, executing in each iteration an inner `for` loop, print statement, and assignment statements for i, j, and `sum`. The inner loop is executed i times for each $i \in \{1, \ldots, n-1\}$ with two assignments in each iteration: one for `sum` and one for j. Therefore, there are

$$1 + 3n + \sum_{i=1}^{n-1} 2i = 1 + 3n + 2(1 + 2 + \cdots + n - 1) = 1 + 3n + n(n-1)$$

$= O(n) + O(n^2) = O(n^2)$ assignments executed before the program is completed.

- **Amortized complexity**: amortized analysis can be used to find the average complexity of a worst case sequence of operations

Linked lists

3.1 Singly-linked lists

If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a linked list, which is a data structure composed of nodes, each node holding some information and a pointer to another node in the list. If a node has a link only to its successor in this sequence, the list is called a singly linked list

Each node resides on the heap

Linked lists can easily grow and shrink in size without reallocating memory or moving elements. Adding or removing nodes (especially at the beginning or middle) is more efficient compared to arrays, as no shifting of elements is required. Memory is allocated as needed, avoiding wasted space typical in arrays with fixed sizes.

However, each node requires extra memory for the pointer to the next node. Accessing elements requires traversal from the head, making lookups slower (O(n)) compared to arrays, which offer O(1) access via indexing. Nodes are scattered in memory, leading to poor cache performance compared to arrays, which have contiguous memory locations.

3.1.1 Structure of the node

The node structure is typically implemented in the following way

```
struct node {
    node* next = nullptr;
    T data = 0;

    node() = default;
    node(data) : data(data) {}
    node(next, data) : next(next), data(data) {}
}
```

A node includes two data members: info and next. The info member is used to store information, and this member is important to the user. The next member is used to link nodes to form a linked list. It is an auxiliary data member used to maintain the linked list. It is indispensable for implementation of the linked list, but less important (if at all) from the user's perspective. Note that node is defined in terms of itself because one data member, next, is a pointer to a node of the same type that is just being defined. Objects that include such a data member are called self-referential objects.

3.1.2 The list class/struct

We also implement the list structure as a class or struct.

```cpp
class single_list {
    node* head = nullptr;
public:
    ...
};
```

3.1.3 Interface of a singly linked list stack

The interface typically includes the following operations:

1. **Insert:** Add a node at the beginning, end, or a specific position in the list.
2. **Delete:** Remove a node from the beginning, end, or a specific position.
3. **Search:** Find a node with a given value.
4. **Traverse:** Iterate through the list to access or print each node's data.
5. **IsEmpty:** Check if the list is empty.
6. **Size:** Return the number of nodes in the list. The first node is called the head, and the last node points to nullptr (indicating the end of the list).

3.1.4 Traversing

Traversing a list is simple.

```c
node* curr = head;

while (curr) {
    curr = curr->next;
    ...
}
```

3.1.5 Printing

Now that we can traverse, we can print each node

```
node* curr = head;
while (curr) {
    cout << curr->data;
    curr=curr->next;
}
```

3.1.6 Printing in reverse

Printing in reverse requires creating a stack.

```
if (!head) return; // noop, dont even bother creating a vector.

vector<node*> stack;
node* curr = head;

while (curr) {
    stack.push_back(curr);
    curr=curr->next;
}

for (int i=(int)stack.size()-1; i>=0; --i) {
    cout << stack[i]->data << " ";
}
cout << endl;
```

3.1.7 Getting the length

While we traverse, just increment a counter.

```
size_t len() {
    size_t len = 0;
    for (node* curr = head; curr; curr=curr->next, ++len);
    return len;
}
```

3.1.8 Clearing

```
void clear() {
    node* curr=head, *prev=nullptr;

    while (curr) {
        prev=curr;
        curr=curr->next;
        delete prev;
    }
    head = nullptr;
}
```

3.1.9 Reversing

Reversing is pretty straight forward

```
void reverse() {
    node* prev=nullptr, *curr=head, *next=nullptr;

    while(curr) {
        next=curr->next;
        curr->next = prev;
        prev = curr;
        curr=next;
    }

    head = prev;
}
```

In each iteration, next temporarily holds the next node so you don't lose track of it when reversing the link.

The curr->next pointer is set to prev, effectively reversing the link.

Prev is then updated to curr, and curr is updated to next to continue the process.

3.1.10 Pushing

```
void push(int element) {
    if (!head) {
        head = new node(element);
        return;
    }

    node* curr = head;
    while (curr->next) {
        curr=curr->next;
    }
    curr->next = new node(element);
}
```

3.1.11 Inserting

```
void insert(int pos, int element) {
    if (!head || pos == 0) {
        node* new_node = new node(element);
        new_node->next = head;
        head = new_node;
        return;
    }
    node* curr = head;

    int count=0;
    while (count != pos-1 && curr->next) {
        curr=curr->next;
        ++count;
    }
    node* new_node = new node(element);

    new_node->next = curr->next;
    curr->next = new_node;
}
```

1. **Check if the list is empty or inserting at the head (position 0):**
   - If head is nullptr (meaning the list is empty) or pos == 0 (you want to insert at the beginning), a new node is created with the given element.
   - The new node's next pointer is set to the current head (which could be nullptr if the list is empty), and then head is updated to point to this new node.
   - This handles the case where the new node becomes the first node in the list.
2. **Traverse to the correct position:**
   - If you are inserting somewhere other than the head, the function uses a loop to find the node just before the desired position (pos - 1).
   - It starts at the head and moves along the list until it reaches the node right before where the new node will be inserted.
3. **Insert the new node:**
   - Once the loop finds the right place (curr points to the node before the insertion position), a new node is created.
   - The new node's next pointer is set to curr->next (the node currently in the target position).
   - Then, curr->next is updated to point to the new node, effectively inserting the new node into the list.

3.1.12 Popping

```
void pop() {
    if (!head) return;
    if (!head->next) {
        delete head;
        head=nullptr;
        return;
    }

    node* prev=nullptr, *curr = head;
    while (curr->next) {
        prev=curr;
        curr=curr->next;
    }
    delete curr;
    prev->next=nullptr;
}
```

1. **Empty List Check:** If the list is empty (head == nullptr), it does nothing.
2. **Single Node Case:** If the list has only one node, it deletes the head and sets head to nullptr.
3. **Multiple Nodes:** It traverses to the last node using two pointers (prev and curr), deletes the last node (curr), and sets the second-to-last node's next pointer (prev->next) to nullptr to mark the new end of the list.

3.1.13 Erasing

```
void erase(int element) {
    if (!head) return;

    while (head->data == element) {
        if (head->next && head->data == element) {
            node* tmp = head;
            head = head->next;
            delete tmp;
        }
    }

    node* prev=nullptr, *curr=head;

    while (curr) {
        if (curr->data == element) {
            node* tmp = curr;
            prev->next = curr->next;
            curr=curr->next;
            delete tmp;
        } else {
            prev=curr;
            curr=curr->next;
        }
    }
}
```

This erase function removes all nodes with a specific value (element) from the list:

- **Empty List Check:** If the list is empty (head == nullptr), it returns immediately.
- **Head Node Deletion:** If the head contains the target value, it deletes the head and updates it to the next node. We keep doing this until the head node no longer contains the data we want to remove
- **Traverse and Delete:** It iterates through the list, and for each node with the target value, it removes the node by adjusting the next pointer of the previous node and deleting the current node.

3.1.14 Searching

```
node* search(int element) {
    node* curr = head;
    while (curr) {
        if (curr->data == element) {
            return curr;
        }
    }
    return nullptr;
}
```

Recursion

4.1 Recursion vs iteration

In theory, any problem that can be solved recursively can be solved iteratively. This also means that any problem that can be solved iteratively can also be solved recursively.

The question is, for any problem that can be solved, which method can be used such that the problem is easier to solve.

4.2 Elementary recursion

A recursive definition consists of two parts. In the first part, called the anchor or the ground case, the basic elements that are the building blocks of all other elements of the set are listed. In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects. For example, to construct the set of natural numbers, one basic element, 0, is singled out, and the operation of incrementing by 1 is given as:

1. $0 \in \mathbb{N}$
2. If $n \in \mathbb{N},\ then(n+1) \in \mathbb{N}$
3. There are no other objects in the set $\mathbb{N}$

It is more convenient to use the following definition, which encompasses the whole range of Arabic numeric heritage:

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \in \mathbb{N}$
2. If $n \in \mathbb{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbb{N}$
3. These are the only natural numbers

Recursive definitions serve two purposes: generating new elements, as already indicated, and testing whether an element belongs to a set. In the case of testing, the problem is solved by reducing it to a simpler problem, and if the simpler problem is still too complex it is reduced to an even simpler problem, and so on, until it is reduced to a problem indicated in the anchor

4.3 Base cases

In recursion, a base case is a condition that stops further recursive calls and provides a direct answer without further recursion

If there were no base case, there would be nothing to stop the recursion. Thus, it would go on until the program crashes. For this reason, all recursive functions must have at least one base case.

If a base case in a recursive function returns a value, then every recursive call leading up to that base case should also return a value. This is necessary to ensure that the result of the recursion is propagated back up the call stack.

In a recursive function, the base case stops the recursion, and if the base case returns something (e.g., a node pointer, integer, etc.), the recursive calls that occur before reaching the base case need to return that result so it can propagate back to the original caller.

4.3.1 Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n \neq 0 \end{cases}.$$

```cpp
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n-1);

    // Expands to
    // n * n-1 * n-2 * ... * 1
}
```

4.3.2 Powers

Consider the recursive definition for a power of x

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}.$$

```cpp
constexpr int power(int x, int n) {
    if (n == 0) return 1;
    return x * power(x,n-1);
}
```

The function power() can be implemented differently, without using any recursion, as in the following loop:

```cpp
int power2(int x, int n) {
    int res = 1;

    for (res = x; n > 1; --n) {
        res*=x;
    }
    return res;
}
```

Do we gain anything by using recursion instead of a loop? The recursive version seems to be more intuitive because it is similar to the original definition of the power function. The definition is simply expressed in C++ without losing the original structure of the definition. The recursive version increases program readability, improves self-documentation, and simplifies coding. In our example, the code of the nonrecursive version is not substantially larger than in the recursive version, but for most recursive implementations, the code is shorter than it is in the nonrecursive implementations

4.4 Tail recursion

Tail recursion is a type of recursion where the recursive call is the last thing the function does before returning a result. This means there are no more computations or operations to perform after the recursive call.

Because of this, tail recursion can be optimized by some compilers or interpreters to avoid adding new frames to the call stack, making it more memory-efficient than regular recursion.

In simple terms, if a recursive function calls itself, and after that call there's nothing left to do, it's tail recursion. This allows the function to reuse the same memory space, preventing stack overflow in cases with deep recursion.

the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. For example, the function tail() defined as

```
void tail(int i) {
    if (i > 0) {
        cout << i << '';
        tail(i-1);
    }
}
```

Is an example of a function with tail recursion, whereas the function nonTail() defined as

```
void nonTail(int i) {
    if (i > 0) {
        nonTail(i-1);
        cout << i << '';
        nonTail(i-1);
    }
}
```

Is not. Tail recursion is simply a glorified loop and can be easily replaced by one. In this example, it is replaced by substituting a loop for the if statement and decrementing the variable i in accordance with the level of recursive call. In this way, tail() can be expressed by an iterative function:

```
void iterativeEquivalentOfTail(int i) {
    for ( ; i > 0; i--)
    cout << i << '';
}
```

Is there any advantage in using tail recursion over iteration? For languages such as C++, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight. In languages endowed with a loop or its equivalents, such as an if statement combined with a goto statement, tail recursion should not be used.

Another problem that can be implemented in recursion is printing an input line in reverse order. Here is a simple recursive implementation:

```
void reverse() {
    char ch;
    cin.get(ch);
    if (ch != '\n') {
        reverse();
        cout.put(ch);
    }
}
```

Compare the recursive implementation with a nonrecursive version of the same function:

```
void simpleIterativeReverse() {
    char stack[80];
    int top = 0;
    cin.getline(stack,80);
    for (top = strlen(stack) - 1; top >= 0; cout.put(stack[top--]));
}
```

functions like strlen() and getline() from the standard C++ library can be used. If we are not supplied with such functions, then our iterative function has to be implemented differently:

```
void iterativeReverse() {
    char stack[80];

    register int top = 0;
    cin.get(stack[top]);

    while(stack[top]!='\n') {
        cin.get(stack[++top]);
    }
    for (top -= 2; top >= 0; cout.put(stack[top--]));
}
```

4.5 Indirect Recursion

The preceding sections discussed only direct recursion, where a function $f()$ called itself. However, $f()$ can call itself indirectly via a chain of other calls. For example, $f()$ can call $g()$, and $g()$ can call $f()$. This is the simplest case of indirect recursion. The chain of intermediate calls can be of an arbitrary length, as in:

$$f() \to f_1() \to f_2() \to ... \to f_n() \to f().$$

There is also the situation when $f()$ can call itself indirectly through different chains. Thus, in addition to the chain just given, another chain might also be possible. For instance

$$f() \to g_1() \to g_2() \to ... \to g_m() \to f().$$

This situation can be exemplified by three functions used for decoding information. receive() stores the incoming information in a buffer, decode() converts it into legible form, and store() stores it in a file. receive() fills the buffer and calls decode(), which in turn, after finishing its job, submits the buffer with decoded information to store(). After store() accomplishes its tasks, it calls receive() to intercept more encoded information using the same buffer. Therefore, we have the chain of calls

$$\text{recieve()} \to \text{decode()} \to \text{store()} \to \text{recieve()} \to \text{decode()} \to ....$$

4.6 Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(n)) & \text{if } n \leqslant 4 \end{cases}.$$

4.7 Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a nonrecursive approach. If recursion is too deep (for example, computing $5.6^{100{,}000}$), then we can run out of space on the stack and our program crashes. But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent

However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

$$\mathrm{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1) & \text{otherwise} \end{cases}.$$

The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence.

How can this definition be implemented in C++? It takes almost term-by-term translation to have a recursive version, which is

```
constexpr unsigned long fib(int n) {
    if (n < 2) return n;
    return fib(n-2) + fib(n-1);
}
```

The function is simple and easy to understand but extremely inefficient. To see it, compute Fib(6), the seventh number of the sequence, which is 8. Based on the definition, the computation runs as follows:

$$\begin{aligned} Fib(6) &= Fib(4) + Fib(5) \\ &= Fib(2) + Fib(3) + Fib(5) \\ &= Fib(0) + Fib(1) + Fib(3) + Fib(5) \\ &= 0 + 1 + Fib(3) + Fib(5) \\ &= 1 + Fib(1) + Fib(2) + Fib(5) \\ &= 1 + Fib(1) + Fib(0) + Fib(1) + Fib(5). \end{aligned}$$

Etc... The source of this inefficiency is the repetition of the same calculations because the system forgets what has already been calculated. For example, Fib() is called eight times with parameter n = 1 to decide that 1 can be returned. For each number of the sequence, the function computes all its predecessors without taking into account that it suffices to do this only once.

It takes almost a quarter of a million calls to find the twenty-sixth Fibonacci number, and nearly 3 million calls to determine the thirty-first! This is too heavy a price for the simplicity of the recursive algorithm. As the number of calls and the run time grow exponentially with n, the algorithm has to be abandoned except for very small numbers

An iterative algorithm may be produced rather easily as follows:

```
unsigned long iterativeFib(unsigned long n) {
    if (n < 2)
    return n;
    else {
        register long i = 2, tmp, current = 1, last = 0;
        for ( ; i <= n; ++i) {
            tmp = current;
            current += last;
            last = tmp;
        }
        return current;
    }
}
```

However, there is another, numerical method for computing Fib(n), using a formula discovered by Abraham de Moivre:

$$\mathrm{Fib}(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}.$$

Where $\phi = \frac{1}{2}(1+\sqrt{5})$, and $\hat{\phi} = 1 - \phi = \frac{1}{2}(1-\sqrt{5})$. $\hat{\phi}$ becomes very small when n grows, thus it can be omitted.

$$\mathrm{Fib}(n) = \frac{\phi^n}{\sqrt{5}}.$$

Approximated to the nearest integer

```
unsigned long deMoivreFib(unsigned long n) {
    return ceil(exp(n*log(1.6180339897) - log(2.2360679775)) -
↪   .5);
}
```

4.8 Backtracking

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called backtracking, and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other To solve this problem, we try to put the first queen on the board, then the second so that it cannot take the first, then the third so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion, which is a natural implementation of backtracking

```
putQueen(row)
    for every position col on the same row
        if position col is available
            place the next queen in position col;
            if (row < 8)
                putQueen(row+1);
            else success;
            remove the queen from position col;
```

This algorithm finds all possible solutions without regard to the fact that some of them are symmetrical.

4.9 Recursion in singly linked lists

4.9.1 Traversing

To traverse a linked list using recursion, you need to define a recursive function that processes the current node and then calls itself with the next node until the list is fully traversed (i.e., until the current node is nullptr).

```
void TraverseList(node* head) {
    if (!head) {
        return;
    }
    TraverseList(head->next);

    // ...
}
```

4.9.2 Printing

We can use this, for example, to print each nodes data member

```
void PrintList(node* head) {
    if (!head) return;

    cout << head->data << " ";
    PrintList(head->next);
}
```

4.9.3 Printing in reverse

We a slight alter in the print example, we can reverse print the list.

```
void PrintListReverse(node* head) {
    if (!head) return;

    PrintListReverse(head->next);
    cout << head->data << " ";
}
```

4.9.4 Getting the length

4.9.5 Clearing

We can also use this to clear the list

```
void clear() {
    std::function<void(node*)> r_clear = [&] (node* p) = {
        if (!head) return;

        r_clear(head->next);
        delete head;
    }
    r_clear(head);
    head=nullptr;
    size=0;
}
```

4.9.6 Reversing

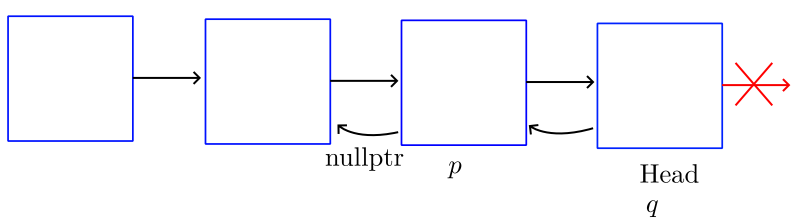
Let's first take a look at the reverse code

```
void reverse() {
    std::function<void(node*)> r_reverse = [&] (node* p) -> void {
        if (!p->next) {
            head = p;
            return;
        }

        r_reverse(p->next);
        node* q = p->next;
        q->next = p;
        p->next = nullptr;

    };
    r_reverse(head);
}
```

The base case is that we are at the end, in this case we set head to this position. Head is now at the end of the list.

Once the base case is triggered and the head is set to the last node in the list, we will be sent back to the n-1 node call.

To get the intuition for linked list logic, we must examine a diagram of the list.



This figure shows the three operations done after each recursive call. In the figure above, we are at the node after the call that set the end to head. We

1. Get a pointer to the node ahead of the current (q).
2. This allows us to severe its old next pointer and reverse its direction.
3. Then, set p next to nullptr (set up for next return).

When the callstack returns to the first call, and does its operations, the list will be reversed

It is also a good idea to examine the iterative method.

```cpp
void Itreverse() {
    node* prev=nullptr, *curr=head, *next=nullptr;

    while (curr) {
        next=curr->next; // Move next to the next node
        curr->next=prev; // Change the direction of current
        ↪ nodes next pointer

        prev=curr; // Advance prev
        curr=next; // Advance curr
    }
    head=prev; // Prev is last node, set head to end
}
```

4.9.7 Pushing

```cpp
void push(int data) {
    if (!head) {
        head = new node(nullptr, data);
        return;
    }
    std::function<void(node*, int)> r_push = [&] (node* curr, int data) -> void {

        if (!curr->next) {
            curr->next = new node(nullptr, data);
            ++size;
            return;
        }
        r_push(curr->next, data);
    };
    r_push(head, data);
}
```

Base case:

1. **Empty list**: No recursion, make head the new node

Otherwise, recurse from the head until we get to the last node, simply set last nodes next pointer to new node and return

4.9.8 Inserting

```cpp
void insert(unsigned pos, int element) {
    std::function<void(node*&, unsigned)> r_insert = [&] (node*& p, unsigned curr_pos) {
        if (curr_pos == 0) {
            node* new_node = new node(nullptr, element);
            new_node->next = p;
            p = new_node;
            return;
        }
        r_insert(p->next,  curr_pos-1);
    };
    r_insert(head, pos);
}
```

Base case

1. **Recursed the same number of times as the `pos` arg**: In this case, make a new node, set next to current node in the recursive traversal, set current node to new node.

Otherwise, keep recursing, subtracting one from the curr_pos.

4.9.9 Popping

```
void pop() {
    if (!head) return;

    if (!head->next) {
        delete head;
        head=nullptr;
        return;
    }

    std::function<void(node*)> r_pop = [&] (node* p) -> void {
        if (!p->next->next) {
            delete p->next;
            p->next = nullptr;
            --size;
            return;
        }
        r_pop(p->next);
    };
    r_pop(head);
}
```

Base cases:

1. **Empty list**: Noop
2. **One node (head)**: Delete then reset head

Otherwise, recurse until we are at the second to last node. Then, delete the second to last nodes next node, which is the last node. Set second to last nodes next pointer to nullptr.

4.9.10 Erasing

```cpp
void erase(int element) {
    std::function<void(node*&)> r_erase = [&] (node*& p) -> void {
        if (p == nullptr) {
            return;
        }

        r_erase(p->next);

        if (p->data == element) {
            node* tmp = p;
            p = p->next;
            delete tmp;
        }
    };
    r_erase(head);
}
```

Base case:

1. **Reached the end**: Return, start unwinding

We traverse to the end of the list recursively, once we reach the end the recursion stops and we start unwinding the call stack, going backwards in the list.

For each node, we check if its data is equal to the element, if it is we set this node equal to its next node, then delete.

4.9.11 Searching

```cpp
node* search(int element) {
    std::function<node*(node*)> r_search = [&] (node* p) -> node* {
        if (p == nullptr)  {
            return nullptr;
        }
        if (p->data == element) {
            return p;
        }
        return r_search(p->next);
    };
    return r_search(head);
}
```

Base cases:

1. **Reached the end of the list**: Element is not in list, return nullptr
2. **Found the first node with the element**: Return the node

Otherwise, recurse through the nodes until we hit one of the base cases.

Binary trees

5.1 Terminology

- **Node:** The basic unit of a binary tree, containing data and references to left and right children.
- **Root:** The topmost node in a tree.
- **Child:** A node directly connected to another node when moving away from the root.
- **Descendants**: The descendants of a node are all nodes that come after a given node.
- **Parent:** The node directly above a child node.
- **Grandparents**: The grandparents of a node is all nodes above the parent up to the root.
- **Ancestors**: The ancestors of a node are all the nodes above a node up to the root
- **Leaf:** A node with no children.
- **Branch node**: A non-leaf node is called a branch node
- **Internal Node:** A branch node, a node with at least one child.
- **Subtree:** A tree consisting of a node and its descendants.
- **Height of a node:** The number of edges on the longest path from a node to a leaf.
- **Height of a tree:** The height of the tree is the height of the root
- **Depth:** The number of edges from the root to a node.
- **Depth of a tree**: The depth of a tree is the depth of the deepest node
- **Degree of a node**: The number of subtrees of a node is called the degree of the node. In a binary tree, all nodes have degree 0, 1, or 2.
- **Degree of a binary tree**: The degree of a tree is the maximum degree of a node in the tree. A binary tree is degree 2.

5.2 Type of binary trees

- **Full Binary Tree:** Every internal node has two children, all leaf nodes have zero children. Thus, all nodes are either zero or two, never one.
- **Complete Binary Tree:** All levels, except possibly the last, are fully filled, and all nodes are as far left as possible.
- **Perfect Binary Tree:** A binary tree where all internal nodes have exactly 2 children, and all leaf nodes are at the same level.
- **Balanced Binary Tree:** A binary tree where the height of the left and right subtrees of every node differs by at most one.
- **Degenerate (or pathological) Tree:** A tree where each parent node has only one child, essentially forming a linked list.
- **Skewed Tree:** A special case of a degenerate tree, where all nodes are skewed to the left or right, forming a linear structure.

5.3 Maximum height of a binary tree

The maximum height of a binary tree with n nodes can be as large as $n1$ (in the case of a degenerate or skewed tree where each node has only one child). This is true for any binary tree:

$$h_{\max} = n - 1.$$

Which occurs for degenerate trees.

5.3.1 Minimum height of a binary tree

The minimum height (best case) for a binary tree with n nodes is achieved when the tree is perfectly balanced:

$$h_{\min} = \lfloor \log_2(n) \rfloor.$$

This is because the tree would need to spread nodes evenly across levels

5.3.2 Number of Leaves in a Binary Tree

For any binary tree with n nodes, the number of leaves l satisfies the following relationship:

$$l \leqslant \frac{n+1}{2}.$$

This formula gives the maximum number of leaves, assuming that the tree is full (every internal node has 2 children).

5.3.3 Relationship Between Internal Nodes and Leaves:

In any binary tree, the number of internal nodes i (nodes with at least one child) and the number of leaves l are related as follows:

$$i \leqslant l - 1.$$

5.3.4 Maximum Number of Nodes at Height h

The maximum number of nodes possible at a given height h (where the height is counted from the root as level 0) in a binary tree is:

$$\text{Max nodes at height } h = 2^h.$$

5.3.5 Number of Edges in a Binary Tree:

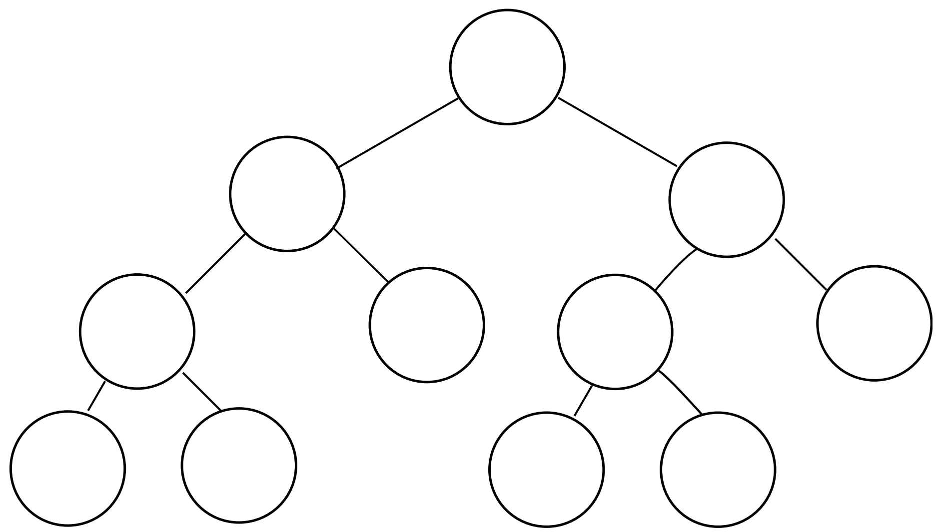
For any binary tree with n nodes, the number of edges e is always

$$e = n - 1.$$

This holds because every node (except the root) is connected to exactly one parent, so there are $n1$ edges in the tree.

5.4 Full trees

A full tree is a tree where all internal nodes are degree two, and all leaf nodes are degree zero. Observe



The next three subsections refer to the *full binary tree theorem*, which states for a nonempty, full tree T

5.4.1 Number of leaves

If T has I internal nodes, the number of leaves is given by

$$L = I + 1.$$

If T has a total of N nodes, the number of leaves is

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

5.4.2 Number of nodes

If T has I internal nodes, the total number of nodes is

$$N = 2I + 1.$$

If T has L leaves, the total number of nodes is

$$N = 2L - 1.$$

5.4.3 Number of internal nodes

If T has a total of N nodes, the number of internal nodes is

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

If T has L leaves, the number of internal nodes is

$$I = L - 1.$$

5.5 Complete Binary Tree

A complete binary tree has a specific structure defined by how the nodes are filled level by level.

1. **All levels, except possibly the last, are fully filled:**
   In a complete binary tree, every level up to the second-to-last (penultimate) level must be completely filled with nodes. This means that if the tree has height h, levels 0 through $h - 1$ (from the root to the second-to-last level) will have the maximum possible number of nodes for that level.
2. **All nodes are as far left as possible:**
   - On the last level, the nodes don't need to completely fill the level, but the nodes must be positioned as far to the left as possible.
   - For example, if some nodes are missing from the last level, they will always be missing from the right side, not from the left.

Notes: The tree is balanced in terms of node distribution, with all the levels except possibly the last fully filled.

Nodes on the last level are always added from the leftmost position first.

5.5.1 Number of nodes

The height h of a complete binary tree is defined as the number of edges on the longest path from the root to a leaf node.

The total number of nodes in a complete binary tree is given by

$$n = 2^{h+1} - 1.$$

5.5.2 Height

The height h of a complete binary tree with n nodes can be derived as:

$$h = \lfloor \log_2(n) \rfloor.$$

5.5.3 Number of Leaf Nodes (L) in a Complete Binary Tree

The number of leaf nodes in a complete binary tree can be calculated based on the number of internal nodes or the height of the tree

$$L = \lceil \frac{n}{2} \rceil.$$

5.5.4 Number of internal nodes

The number of internal nodes (non-leaf nodes) in a complete binary tree can be calculated as:

$$I = N - L$$
$$I = \lfloor \frac{n}{2} \rfloor.$$

5.5.5 Parent and Child Relationships in a Complete Binary Tree

Parent of node at index i (1-based index):

$$\text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

Left child of node at index i:

$$\text{Left child}(i) = 2i$$

Right child of node at index i:

$$\text{Right child}(i) = 2i + 1$$

These relationships assume a 1-based indexing system for the nodes in the tree (common in heaps or array-based representations).

5.6 Perfect binary tree

5.6.1 Number of Nodes

$$N = 2^{h+1} - 1.$$

5.6.2 Number of Leaf Nodes

$$L = 2^h.$$

5.6.3 Height of the Tree

$$h = \log_2(N + 1) - 1.$$

5.6.4 Number of Internal Nodes

$$I = N - L = 2^h - 1.$$

5.6.5 Depth

$$d = h.$$

Applications of binary trees

6.1 Binary search trees

A binary search tree (BST) is a binary tree in which each node has at most two children and follows these properties:

- **Left Subtree Property:** The value of each node in the left subtree is less than the value of the node itself.
- **Right Subtree Property:** The value of each node in the right subtree is greater than the value of the node itself.
- Both left and right subtrees must also be binary search trees.

6.1.1 Interface

The interface of a Binary Search Tree (BST) typically includes a set of operations for managing and accessing the tree's nodes.

- **Insert(value):** Inserts a new value into the BST while maintaining its properties.
- **Remove(value):** Removes a value from the BST, adjusting the structure to maintain its properties.
- **Predecessor(node)**: Finds the predecessor of a node
- **Succesor(node)**: Finds the successor of a node
- **Find(value):** Searches for a value in the BST and returns the node containing it or null if not found.
- **FindMin():** Returns the node with the smallest value in the BST.
- **FindMax():** Returns the node with the largest value in the BST.
- **IsEmpty():** Checks if the BST is empty.
- **Traverse(order):** Traverses the tree in a specific order (e.g., in-order, pre-order, post-order).
- **Height():** Returns the height of the BST.
- **Clear():** Removes all nodes from the tree, making it empty

6.1.2 Traversals

We can traverse BST's in one of four ways

- Level order
- Preorder
- Inorder
- Postorder

6.1.2.1 Level order

Level-order traversal is a way of visiting all the nodes in a binary tree by levels, from top to bottom. It starts at the root and visits nodes level by level, left to right, for each level.

- Start with the root node (the topmost node).
- Visit all the nodes on the next level (children of the root) from left to right.
- Then, visit all nodes on the level below that (grandchildren of the root) from left to right, and so on.

A queue is often used to implement level-order traversal, as it helps keep track of nodes to visit in the correct order.

```
void levelorderPrint() {
    if (!root) return; // noop for empty tree

    queue<node*> q;
    q.push(root);

    while (!q.empty()) {
        node* curr = q.front();
        q.pop();

        cout << curr->data << endl;
        if (curr->left) {
            q.push(curr->left);
        }
        if (curr->right) {
            q.push(curr->right);
        }
    }
}
```

1. If the list is nonempty, construct a queue and push the root node.
2. While the queue is nonempty, grab the front, process the front, pop the front.
3. Push left and right nodes to queue, if they exist.

6.1.2.2 Preorder

Pre-order traversal is a way of visiting nodes in a binary tree where you:

1. Visit the root node first.
2. Recursively visit the left subtree.
3. Recursively visit the right subtree.

To explain simply:

1. Start with the root node.
2. Go as far left as possible, visiting each node along the way.
3. Once you've reached the end of the left subtree, backtrack and visit the right subtree.

```cpp
void preorderPrint() {
    std::function<void(node*)> r_preorderPrint = [&] (node* p) {
        if (p == nullptr) return;

        cout << p->data << endl;
        r_preorderPrint(p->left);
        r_preorderPrint(p->right);
    };
    r_preorderPrint(root);
}
```

6.1.2.3 Inorder

in-order traversal is a way of visiting nodes in a binary tree where you:

1. Recursively visit the left subtree first.
2. Visit the root node.
3. Recursively visit the right subtree.

To explain simply:

1. Start by going all the way to the left, visiting nodes along the way.
2. Once you reach the leftmost node, visit it, then move up to its parent (the root).
3. After visiting the root, visit the right subtree.

For a BST, printing the tree with an inorder traversal yields a sorted sequence.

```
void inorderPrint() {
    std::function<void(node*)> r_inorderPrint = [&] (node* p) ->
    ↪ void {
        if (!p) return;

        r_inorderPrint(p->left);
        cout << p->data << endl;
        r_inorderPrint(p->right);
    };
    r_inorderPrint(root);
}
```

6.1.2.4 Postorder

Post-order traversal is a way of visiting nodes in a binary tree where you:

1. Recursively visit the left subtree first.
2. Recursively visit the right subtree.
3. Finally, visit the root node.

To explain simply:

1. Start by going to the leftmost node, but don't visit it yet.
2. Then, go to the right subtree and process it.
3. After both subtrees have been visited, visit the root.

```
void postorderPrint() {
    std::function<void(node*)> r_postorderPrint = [&] (node* p)
    -> void {
        if (!p) return;

        r_postorderPrint(p->left);
        r_postorderPrint(p->right);
        cout << p->data << endl;
    };
    r_postorderPrint(root);
}
```

6.1.3 Successor of a node

The successor of a node is defined mathematically as

$$\mathrm{succ}(X) = \min\{A : \; A > X\}.$$

thus, we find the set of all nodes that have values greater than that of X, then find the minimum in that set.

By properties of binary search trees we find the successor of a node X by

1. **If X has a right child:** The successor is the leftmost node in the right subtree of X (the smallest node in the right subtree).
2. **If X has no right child**:
   - If the node is the left child of its parent, then the parent is its successor.
   - If the node is the right child of its parent, you move upward until you find a node that is the left child of its parent, and that parent is the successor.

6.1.4 Predecessor

The predecessor of a node X is defined as

$$\text{pred}(X) = \max\{A : \ A < X\}.$$

In other words it is the largest node that is less than X. To find the predecessor:

1. **If X has a left child**: The predecssor is the rightmost node in the left subtree
2. **If X has no left child**: The predecessor is the nearest ancestor for which the node is in the right subtree.

6.1.5 The node

The node is similar to a linked list node, but instead of a single next pointer, it has two. A left pointer and a right pointer.

```
struct node{
    node* left = nullptr;
    node* right = nullptr;
    int data = 0;

    node() = default;
    node(int data) : data(data) {}
    node(node* left, node* right, int data) : left(left),
    ↪  right(right), data(data) {}
};
```

6.1.6 The class

For simplicity, we often define the Binary Search Tree (BST) as a class. This allows each instance of the class to hold its own root node, along with other data members such as the size of the tree, that we may need.

If it were not a class, then each function would need to take the root node as an argument and return the (potentially modified) root node to maintain the structure.”

If it were not a class, than each function would have to take as an argument a root node, and return the root node to maintain the structure

```
class BST {
private:
    node* root;
    ...
public:
    ...
};
```

6.1.7 Recursive Insertion

Because of the nature of BSt's, we often use recursion to define the needed operations.

```
void insert(int element)  {
    // If the tree is empty, insert new element as root
    if (!root) {
        root = new node(element);
        return;
    }

    std::function<void(node*)> r_insert = [&](node* p) -> void {

        // If the element is less than current node, and p->left exists, go left
        if (element < p->data && p->left) {
            r_insert(p->left);

            // If the element is greater than current node, and p->right exists, go right
        } else if (element > p->data && p->right) {
            r_insert(p->right);
        }

        // If the element is less than current node, and p->left doesn't exist, insert node as current nodes left child
        if (element < p->data && !p->left) {
            p->left = new node(element);
            return;

            // If the element is greater than current node, and p->right doesn't exist, insert node as current nodes right child
        } else if (element > p->data && !p->right) {
            p->right = new node(element);
            return;
        }
    };
    // Start recursion from the root
    r_insert(root);
}
```

If the tree is empty, it creates a new root node with the given element.

Otherwise, it uses a recursive lambda function (r_insert) to:

- Traverse the tree: going left if the element is smaller, or right if the element is larger.
- Once it finds an appropriate spot (where a left or right child doesn't exist), it inserts the new node as a left or right child accordingly.

The process starts from the root and recursively finds the right place to insert the new element.

6.1.8 A better recursive insert

```
node* r_insertC(node* p, int element) {
    if (!p) return new node(element);

    if (element < p->data) {
        p->left =  r_insertC(p->left,element);
    } else if (element > p->data) {
        p->right =  r_insertC(p->right, element);
    }
    return p;
}

void insertC(int element) {
    if (!root) root = new node(element);
    r_insertC(root,element);
}
```

The main insertion function, insertC, initiates the process. If the tree's root is null, meaning the tree is empty, it creates a new root node with the given element. Otherwise, it calls the recursive helper function r_insertC on the root to handle the insertion process.

The r_insertC function operates recursively to find the correct location for the new element within the tree. Starting from the given node p, it checks whether p is null; if so, it creates and returns a new node with the specified element, making this node the new leaf of the tree at this position. If p is not null, the function compares the element to p->data. If the element is smaller, the function recursively calls r_insertC on p->left to continue searching in the left subtree, and if the element is larger, it calls r_insertC on p->right to search in the right subtree. After setting the appropriate child link, it returns the current node p, maintaining the correct structure of the tree at each level of recursion. This ensures the BST properties are preserved, with each node's left subtree containing values less than the node's data and the right subtree containing values greater.

Left as an exercise to the reader to see why we must return *p* to maintain the tree pointer chain.

6.1.9 Iterative insert

```cpp
void insertB(int element) {
    if (!root) {
        root = new node(element);
        return;
    }

    node* p = root, *trail = nullptr;
    bool left;

    while (p) {
        trail = p;
        if (element < p->data) {
            p=p->left;
            left=true;
        } else if (element > p->data) {
            p=p->right;
            left=false;
        } else {
            return; // noop if already exists
        }
    }
    if (left) {
        trail->left = new node(element);
    } else {
        trail->right = new node(element);
    }
}
```

If the tree is empty, it creates a new root node with the element.

It then iteratively traverses the tree starting from the root:

- Moves left if the element is smaller than the current node's data.
- Moves right if the element is larger.
- If the element already exists, it does nothing and returns.

Once it finds an empty spot (either left or right child is nullptr), it inserts the new node as the left or right child of the parent node (trail), depending on the comparison.

6.1.10 Recursive removing

To remove a node with a given value from a BST, there are three cases

1. Node has no children
2. Node has one child
3. Noe has two children

For case I, we can simply set the nodes parent to nullptr, and then delete the node.

For case II, we must divert the connection from the nodes parent to the nodes child, and then free the node.

Case III is more involved, we first must find the successor of the node. Once we find the successor, we replace the nodes data value with its successor. Then, instead of deleting the node, we delete its successor. Since to be in this case the node must have exactly two children, the successor is found in the simple way.

1. Go right once
2. Go as far left as possible.

Once we have the successor node, it will either have no children, or exactly one child (a right child), if it were to have a left child, it would not be the true successor because we would have not gone as far left as possible.

```cpp
void remove(int element) {
    if (!root) return; // Noop for empty tree

    std::function<void(node*&, node*&)> r_remove = [&] (node*&
    p, node*& last) -> void {
        if (!p) return; // Not found in tree

        if (element < p->data) {
            r_remove(p->left, p);
        } else if (element > p->data) {
            r_remove(p->right, p);
        } else { // Found
            // Case I: Node has zero children
            if (!p->left && !p->right) {
                node* tmp = p;
                p=nullptr;
                delete tmp;
                // Case II: Node has one child (note the use of
    xor)
            } else if (!p->left ^ !p->right) {
                node* tmp = p;
                p = (p->left ? p->left : p->right);
                delete tmp;
                // Case III: Two children
            } else {
                node* successor = p->right;
                node* successorParent = p;

                // Find the in-order successor
                while (successor->left) {
                    successorParent = successor;
                    successor = successor->left;
                }

                // Replace nodes value with successor value
                p->data = successor->data;

                // Now we need to delete the successor node
                // The successor is a leaf or has a right child
                if (successorParent->left == successor) {
                    successorParent->left = successor->right;
                } else {
                    successorParent->right = successor->right;
                }
                delete successor;
            }
        }
    };
    r_remove(root,root);
}
```

6.1.11 Clearing

```
void clear() {
    if (!root) return;

    std::function<void(node*)> r_clear = [&](node* p) -> void {
        if (!p) return;

        r_clear(p->left);
        r_clear(p->right);

        delete p;
    };
    r_clear(root);
    root = nullptr;
}
```

This function deletes all nodes in a binary search tree. It recursively traverses the tree, deleting each node after its children have been deleted, and finally sets the root to nullptr, effectively clearing the entire tree

6.1.12 Counting the height of the tree (root)

```cpp
size_t height() {
    std::function<size_t(node*)> r_height = [&](node* p) -> size_t {
        // Base case height of a nullptr is zero
        if (!p) return 0;
        return 1+std::max(r_height(p->left), r_height(p->right));
    };
    // Height is counting edges, so its number nodes in longest path from root to leaf - 1
    return r_height(root) -1;
}
```

This code defines a height() function that calculates the height of a binary tree by counting the edges. It uses a recursive lambda function r_height to traverse the tree. For each node, it returns 1 + max(left subtree height, right subtree height) to find the longest path from the root to any leaf. Since r_height counts nodes, the function subtracts 1 at the end to convert the node count to edge count, which is the definition of height.

6.1.13 Counting the height of a node

```cpp
int getHeight(node* p) {
    if (!p) return -1;
    return 1 + std::max(getHeight(p->left), getHeight(p->right));
}
```

If height counts edges, then a nullptr nodes must return -1. If height counts vertices, then nullptr nodes must return 0.

6.1.14 Getting the depth of the node

```
int nodeDepth(node* p) {
    if (p == root) return 1;
    return r_nodeDepth(root, p, 1);
}

int r_nodeDepth(node* curr, node* p, int depth) {
    if (curr == nullptr) {
        return -1;  // Node not found
    }
    if (p == curr) {
        return depth;  // Node found, return the depth
    }

    // Recursively search in the left subtree if p's data is smaller
    if (p->data < curr->data) {
        return r_nodeDepth(curr->left, p, depth + 1);
    }
    // Recursively search in the right subtree if p's data is greater
    if (p->data > curr->data) {
        return r_nodeDepth(curr->right, p, depth + 1);
    }

    return -1;  // This should never be reached if the tree is valid
}
```

This code defines two functions that work together to calculate the depth of a given node p in a binary search tree (BST):

nodeDepth: This function is the entry point to calculate the depth of the node p.

It first checks if p is the root node. If so, it returns 1 since the root node is considered to have a depth of 1.

If p is not the root, it calls the helper function r_nodeDepth to recursively search for the node, starting from the root with an initial depth of 1.

r_nodeDepth: This is a recursive helper function that searches for the node p in the tree, while tracking the current depth.

It first checks if curr (the current node in the search) is nullptr, which indicates that the node p is not in the tree. In that case, it returns -1.

If curr matches p, it returns the current depth.

Otherwise, it recursively searches in the left or right subtree, depending on whether p's data is less than or greater than the current node's data, and increments the depth by 1 at each recursive step.

6.1.15 Counting the number of nodes

```
template <typename NODE>
int count(NODE * root) {
    if (!root) return 0;

    return 1 + count(root->left) + count(root->right);
}

```

6.1.16 Comparison traversals

In this section we show a typical recursive algorithm to compare two binary search trees via an inorder traversal.

```
// Node compare
bool nc(node* p, node* q) {
    return p->data == q->data;
}

bool r_inorderComp(node* p, node* q) {
    if (!p && !q) return true;
    if (!p || !q) return false;

    if (!r_inorderComp(p->left, q->left)) return false;
    if (!nc(p,q)) return false;
    if (!r_inorderComp(p->right, q->right)) return false;

    return true;

}

bool inorderComp(tree& t1, tree& t2) {

    if (!t1.root && !t2.root) return true;
    if (!t1.root || !t2.root) return false;

    return r_inorderComp(t1.root, t2.root);
}
```

The core of the comparison is performed by two functions, nc and r_inorderComp, which are used to check if each corresponding node in the two trees holds the same data value and is structured identically.

The nc function is a helper that takes two nodes as parameters and simply compares their data values, returning true if the values are equal and false otherwise.

The function r_inorderComp performs a recursive, in-order traversal of two nodes, p and q. If both nodes are nullptr, it means that this position in both trees is empty, so they match and the function returns true. If one node is nullptr and the other is not, it indicates a structural mismatch, so the function returns false. The recursion first compares the left child nodes, then the current nodes themselves using nc, and finally the right child nodes. If any comparison fails, the function returns false; otherwise, it completes successfully, confirming that the structures and values match in this part of the trees.

The inorderComp function is the main entry point for the comparison. It takes two tree objects, t1 and t2, and checks if both trees are empty, returning true if they are. If one tree is empty while the other is not, it returns false due to a structural difference. If both trees have a root node, inorderComp then calls r_inorderComp on the root nodes of t1 and t2, performing the recursive in-order comparison on the entire structures.

6.1.17 Finding the smallest and largest values

By properties of BST's, to find the smallest value in a tree, we start from the root and go as far left as possible. To find the largest, we go as far right as possible

```
node* r_min(node* p) {
    if (!p->left) return p;
    return r_min(p->left);
}
node* min() {
    if (!root) return nullptr;
    return r_min(root);
}

node* r_max(node* p) {
    if (!p->right) return p;
    return r_max(p->right);
}
node* max() {
    if (!root) return nullptr;
    return r_max(root);
}
```

6.1.18 Getting the widths of a bst

```
std::vector<int> getWidths() {
    if (!root) return;

    std::vector<int> w(height());
    std::queue<node*> q;
    w[0] =1;
    q.push(root);
    q.push(nullptr);

    int level = 1, width=0;
    while (!q.empty()) {
        node* curr = q.front();
        q.pop();

        if (curr == nullptr) {
            w[level++] = width;
            width = 0;
            q.push(nullptr);
            continue;
        }
        if (level == height()) {
            break;
        }

        if (curr->left) {
            q.push(curr->left);
            ++width;
        }
        if (curr->right) {
            q.push(curr->right);
            ++width;
        }
    }
    return w;
}
```

This C++ code defines a getWidths function that calculates and returns a vector of integers representing the width of each level in a binary tree. The width of a level is defined as the number of nodes at that level.

The function first creates a vector w with a size equal to the height of the tree (obtained from a hypothetical height() function) and initializes the first element, w[0], to 1, assuming the root level has one node. It also sets up a queue q to facilitate level-order traversal (breadth-first traversal) of the tree, starting by pushing the root node and a nullptr marker, which denotes the end of each level.

The main loop continues as long as the queue is not empty. It dequeues the front node in each iteration, checking if it's nullptr. When a nullptr is encountered, it means the current level has ended, so the function records the width for that level in the w vector, resets the width counter, and pushes another nullptr if there are more nodes to process at deeper levels. The loop then proceeds to the next level.

If the dequeued node is not nullptr, it examines its left and right children. If they exist, they are added to the queue, and the width counter is incremented for each child node. This process continues until all levels are processed or the traversal reaches the specified tree height, returning the vector w with each level's width at the end.

6.1.19 Degenerate Binary Search trees

A degenerate binary search tree is a tree where each parent node has only one child, causing the tree to resemble a linked list

Consider building a binary search tree, taking values from a sorted array from left to right. Since all subsequent entries will be greater than the previous, the insertions will only go in one direction, right. Thus, the final tree will resemble a linked list and thus we will not be able to use the $\lg(n)$ property of binary trees.

6.1.20 Verifying a binary search tree

To verify a binary search tree, we need to check that for a given node, all nodes to the left have values less than the node, and all nodes to the right have values greater than the node. his can be achieved using recursion and passing down minimum and maximum value constraints for each subtree.

```cpp
bool r_is_bst(node* p, int min, int max) {
    if (!p) return true;

    if (p->data < min || p->data > max) return false;

    return r_is_bst(p->left, min, p->data) && r_is_bst(p->right, p->data, max);
}

bool is_bst() {
    if (!root) return true;
    return r_is_bst(root, std::numeric_limits<int>::min(), std::numeric_limits<int>::max());
}
```

- We start with the root node, which has the range of INT_MIN to INT_MAX.
- For the left child, we update the maximum allowed value to the parent's value.
- For the right child, we update the minimum allowed value to the parent's value.

6.1.21 Complexities

Because BST have no guarantee of being well formed, (ie degenerate trees), the complexity of many operations in the worst case is $O(n)$.

Operation	Best Case	Average Case	Worst Case
Insertion	$O(\log n)$	$O(\log n)$	$O(n)$
Search	$O(1)$	$O(\log n)$	$O(n)$
Removal	$O(\log n)$	$O(\log n)$	$O(n)$
Height	$O(\log n)$	—	$O(n)$
Traversal	$O(n)$	$O(n)$	$O(n)$

Note: $\Omega(\lg(n)$ for BST operations like search, insert, and delete (best case).

$\Omega(\lg(n))$ for operations that require visiting all nodes, like traversals.

6.2 Adelson-Velsky and Landis Trees (AVL trees)

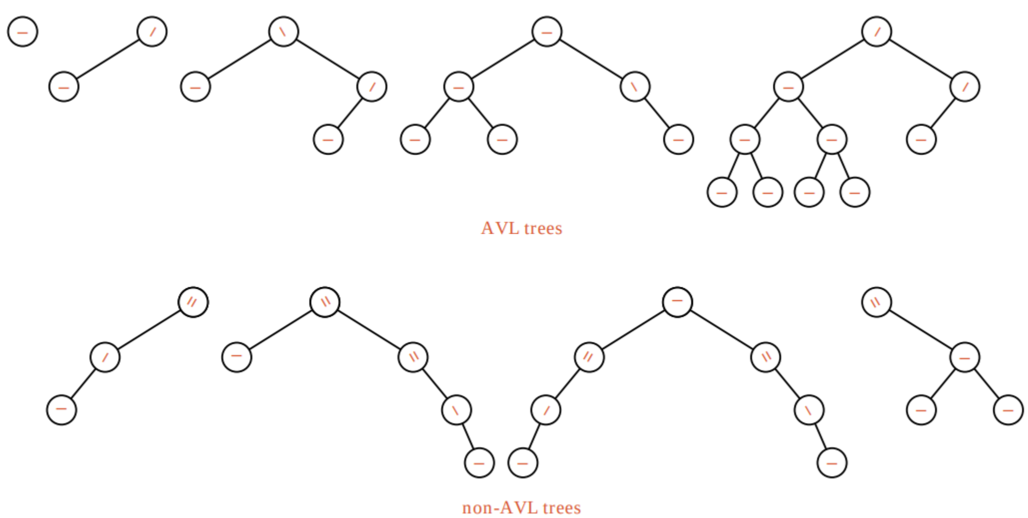
When dealing with binary search trees, insertions and removals occur continually, with no predictable order. In some of these applications, it is important to optimize search times by keeping the tree very nearly balanced at all times. The method in this section for achieving this goal was described in 1962 by two Russian mathematicians, G. M. ADEL'SON-VEL'SKI˘I and E. M. LANDIS, and the resulting binary search trees are called AVL trees in their honor.

AVL trees achieve the goal that searches, insertions, and removals in a tree with n nodes can all be achieved in time that is $O(\log n)$, even in the worst case. The height of an AVL tree with n nodes, as we shall establish, can never exceed lg n, and thus even in the worst case, the behavior of an AVL tree could not be much below that of a random binary search tree. In almost all cases, however, the actual length of a search is very nearly lg n, and thus the behavior of AVL trees closely approximates that of the ideal, completely balanced binary search tree.

6.2.1 Definition

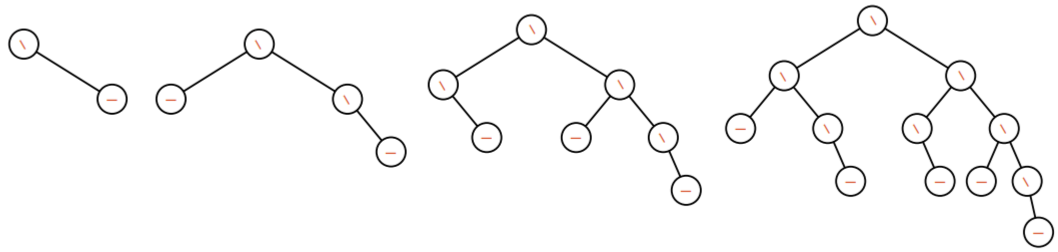
An AVL tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

With each node of an AVL tree is associated a *balance factor* that is `lefthigher`, `equal-height`, or `right-higher` according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.



In drawing diagrams, we shall show a left-higher node by '/,' a node whose balance 358

factor is equal by '−,' and a right-higher node by ''. The figure above shows several small AVL trees, as well as some binary trees that fail to satisfy the definition. Note that the definition does not require that all leaves be on the same or adjacent levels.



The figure above shows several AVL trees that are quite skewed, with right subtrees having greater height than left subtrees.

6.2.2 AVL Nodes

A typical node for an AVL tree is as follows

```
struct node {
    node* left{nullptr}, *right{nullptr};
    int data{0};
    int height{0};
    balance b{0};

    // Constructors
        ...
};
```

In AVL trees, it is common for the node structure to include a height member. This height field is essential for efficiently maintaining the balance of the tree, as the balance factor of a node (the difference in height between its left and right subtrees) is used to determine whether the tree needs rebalancing after insertions or deletions.

Without the height member, recalculating the height of each node during every operation would require traversing the entire subtree, significantly increasing the time complexity. By storing the height in each node, you can retrieve it in constant time, allowing rotations and rebalancing operations to remain efficient.

6.2.3 Storing the height

In AVL trees, we should store the height of a node as a data member, updating on insertions or removals into the tree.

6.2.4 Defining balance factors in C++ with enums

We employ an enumerated data type to record balance factors

```
enum Balance_factor { left_higher, equal_height, right_higher };
```

Balance factors must be included in all the nodes of an AVL tree, and we must adapt our former node specification accordingly.

6.2.5 Defining balance factors with a height calculation

Instead, we can define the balance factor of a node as height($left$) − height($right$). A negative balance implies a subtree is heavy on the right, a positive balance implies a subtree is heavy on the right. A balance of zero implies equal height subtrees. A balance factor ranges from -2 to 2. If |balance| = 2, we must rotate the tree to restore balance.

```
int getBalance(node* p) {
    if (!p) return -1;
    return (p->left ? p->left->height : -1) - (p->right ? p->right->height : -1);
}
```

6.2.6 Interface

The interface of a Binary Search Tree (BST) and an AVL Tree is generally the same. Both are types of binary trees, and they share similar operations such as:

- **Insert:** Insert a new element into the tree.
- **Remove/Delete:** Remove an element from the tree.
- **Search:** Find whether a particular element exists in the tree.
- **Traversal:** In-order, pre-order, post-order, and level-order traversals.

However, behind the scenes, the AVL tree performs additional work to maintain its balance property, but this doesn't typically change the public interface

6.2.7 Balancing an AVL tree

As we insert or remove nodes from the tree, it may happen that the resulting tree fails to satisfy the conditions imposed by AVL trees. To *rebalance* the tree, we have a set of operations, called rotations. We have

1. **Right Rotation (RR):** Applied when a left subtree is too deep. The subtree is rotated to the right, reducing the height of the left side.
2. **Left Rotation (LL):** Applied when a right subtree is too deep. The subtree is rotated to the left, reducing the height of the right side.
3. **Left-Right Rotation (LR):** Occurs when a left subtree has a deep right subtree. First, a left rotation is performed on the left subtree, followed by a right rotation.
4. **Right-Left Rotation (RL):** Occurs when a right subtree has a deep left subtree. First, a right rotation is performed on the right subtree, followed by a left rotation.

These rotations are applied based on the balance factor, ensuring the tree remains balanced with a height difference of at most 1 between subtrees.

Note: Left-right and Right-left rotations are also called double right and double left respectively.

When writing our rotation algorithms, we only need to define two. A left rotation algorithm and a right rotation algorithm

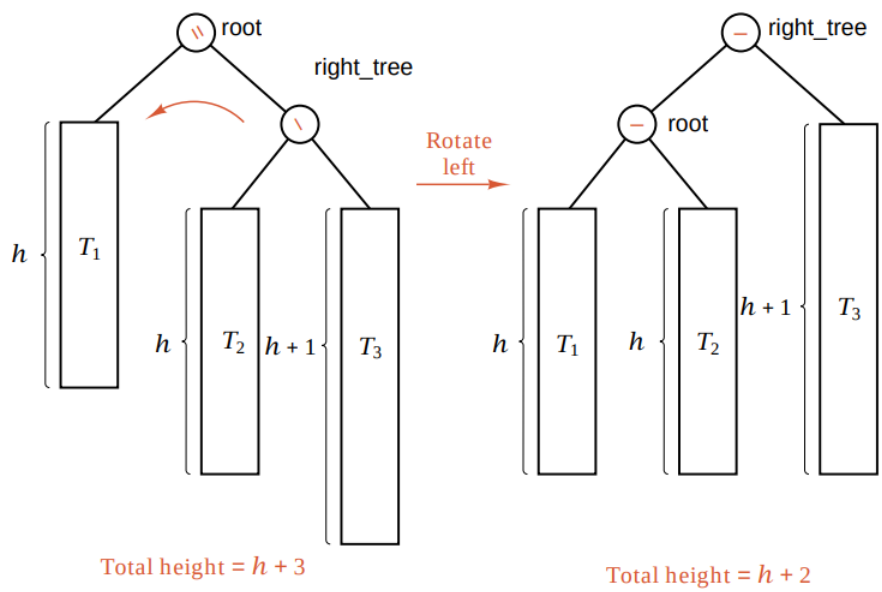
Note: Performing a rotation when the height difference is only 1 would be unnecessary and could actually disrupt the balancing of the tree. The tree is still balanced in this case because a height difference of 1 between subtrees is allowed in AVL trees.

6.2.8 Rotations: Right tree

Let us now consider the case when a new node has been inserted into the taller subtree of a root node and its height has increased, so that now one subtree has height 2 more than the other, and the tree no longer satisfies the AVL requirements. We must now rebuild part of the tree to restore its balance. To be definite, let us assume that we have inserted the new node into the right subtree, its height has increased, and the original tree was right higher. That is, we wish to consider the case covered by the function right_balance. Let root denote the root of the tree and right_tree the root of its right subtree.

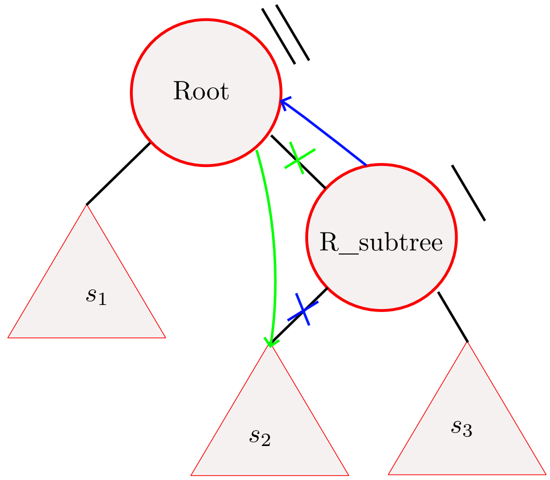
There are three cases to consider, depending on the balance factor of right_tree.

6.2.8.1 Case 1: Right higher The first case, when right_tree is right higher. The action needed in this case is called a left rotation. We have rotated the node right_tree upward to the root, dropping root down into the left subtree of right_tree; the subtree T2 of nodes with keys between those of root and right_tree now becomes the right subtree of root rather than the left subtree of right_tree. A left rotation is succinctly described in the following C++ function. Note especially that, when done in the appropriate order, the steps constitute a rotation of the values in three pointer variables. Note also that, after the rotation, the height of the rotated tree has decreased by 1; it had previously increased because of the insertion; hence the height finishes where it began.



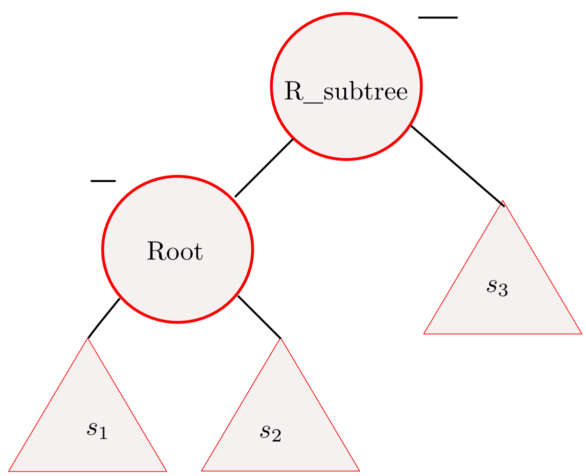
Thus, we come to the following implementation of a left rotation.

```
void left_rotate(node* root) {
    if (!root || !root->right) return;

    right_subtree = root->right;
    root->right = right_subtree->left;
    right_subtree->left = root;
    right_subtree=root;
}
```



1. **Step 1 (green)**: Attach right_subtrees left subtree to the right of root
2. **Step 2 (blue)**: Attach root to the left of right_subtree
3. **Step 3**: Make r_subtree the new root.

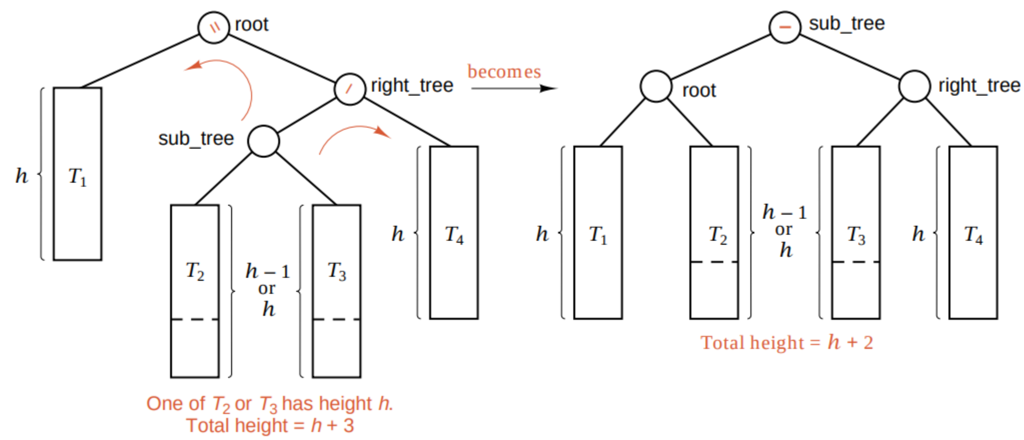
Yields the rotated tree



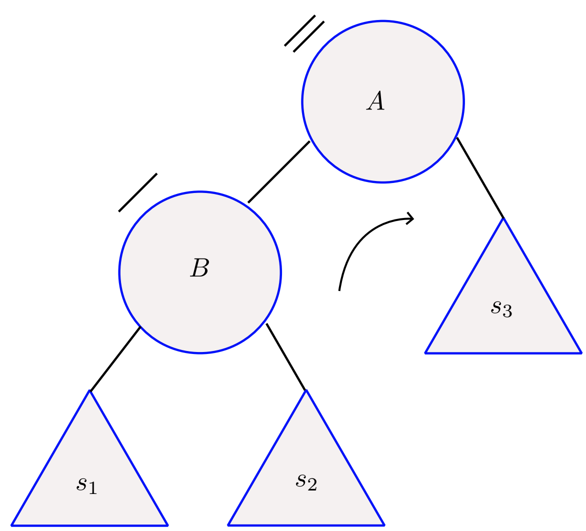
Note that we perform a left rotation when the roots right subtree has an insertion into its right subtree. Notice that the balance symbols point in the same direction. The only balance factors that change are the balance factors of root and right_subtree, they become even (balanced).

6.2.8.2 Case 2: Left higher

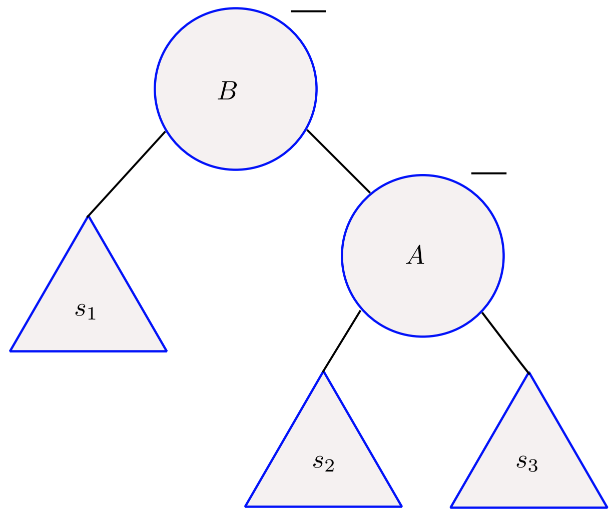
The second case, when the balance factor of right_tree is left higher, is slightly more complicated. It is necessary to move two levels, to the node sub_tree that roots the left subtree of right_tree, to find the new root. This process is shown in Figure 10.20 and is called a double rotation, because the transformation can be obtained in two steps by first rotating the subtree with root right_tree to the right (so that sub_tree becomes its root), and then rotating the tree pointed to by root to the left (moving sub_tree up to become the new root).



We see from the figure that we first perform a right rotation on right_tree, and right_trees left subtree, this shifts the subtree unbalance in right_subtree such that it becomes unbalanced on the right instead of on the left. This enables us to perform a left rotation on root and right_tree. Note that after the right rotation, the balance symbols point in the same direction. A right rotation is of the form



Which becomes



Thus, we see that like the left rotation, the only subtree that moves is s_2. In the left rotation, the left subtree of B becomes the right subtree of A. In the right rotation, the right subtree of B becomes the left subtree of A. A right rotation occurs when a tree becomes unbalanced on the left. Whereas a left rotation occurs when a tree becomes unbalanced on the right.

Thus, we come to the following implementation of a left rotation.

```
void right_rotate(node* root) {
    if (!root || !root->right) return;

    left_subtree = root->left;
    root->left = left_subtree->right;
    left_subtree->right = root;
    left_subtree=root;
}
```

6.2.9 C++ Rotations

```cpp
node* leftRotate(node* p) {
    node* subp = p->right;
    p->right = subp->left;
    subp->left = p;

    p->height = 1 +
        std::max(p->left ? p->left->height : -1,
                p->right ? p->right->height : -1);

    subp->height = 1 +
        std::max(subp->left ? subp->left->height : -1,
                subp->right ? subp->right->height : -1);

    p->balance = getBalance(p);
    subp->balance = getBalance(subp);
    return subp;
}

node* rightRotate(node* p) {
    node* subp = p->left;
    p->left = subp->right;
    subp->right = p;

    p->height = 1 +
        std::max(p->left ? p->left->height : -1,
                p->right ? p->right->height : -1);

    subp->height = 1 +
        std::max(subp->left ? subp->left->height : -1,
                subp->right ? subp->right->height : -1);

    p->balance = getBalance(p);
    subp->balance = getBalance(subp);
    return subp;
}
```

In leftRotate, the function takes a node pointer p (the root of a subtree that is right-heavy) and performs a left rotation to reduce the height of its right subtree. First, it saves the right child of p (denoted as subp). Then, it adjusts the pointers so that the left child of subp becomes the new right child of p, and subp becomes the new root of this subtree with p as its left child. After adjusting the structure, it recalculates the height and balance of both p and subp, where the height of each node is determined by adding 1 to the maximum height of its left and right children. The function then returns subp as the new root of the subtree.

Similarly, rightRotate takes a node p (the root of a left-heavy subtree) and performs a right rotation to rebalance it. It stores the left child of p as subp, then updates pointers so that the right child of subp becomes the new left child of p, and subp becomes the new root with p as its right child. Heights and balances for p and subp are recalculated using the heights of their children, and subp is returned as the new root of this subtree. These rotations are essential for keeping the AVL tree balanced after insertions and deletions.

Note: To understand the return values of the rotateMethods, it is necessary to understand the insertion method below.

6.2.10 Balancing

```
node* balance(node* p) {
    if (p->balance > 1) {  // Left heavy
        if (p->left && p->left->balance < 0) {
            leftRotate(p->left);  // Left-Right case
        }
        return rightRotate(p);  // Left-Left case
    }
    if (p->balance < -1) {  // Right heavy
        if (p->right && p->right->balance > 0) {
            rightRotate(p->right);  // Right-Left case
        }
        return leftRotate(p);  // Right-Right case
    }
    return p;
}
```

This function takes a pointer p to a node and checks its balance factor (the difference in height between its left and right subtrees) to determine if the node is unbalanced. If the balance factor is greater than 1, the tree is left-heavy, meaning the left subtree is taller than the right subtree. In this case, the function first checks if a "Left-Right" rotation is needed—this would happen if the left child of p is itself right-heavy, indicated by a negative balance factor. If so, a left rotation is applied to the left child of p to balance it before proceeding. Then, a right rotation is applied to p to correct the "Left-Left" imbalance.

Conversely, if the balance factor is less than -1, the node is right-heavy, indicating the right subtree is taller. For a "Right-Left" case, where the right child of p is left-heavy, a right rotation is applied to the right child of p first. This corrects the imbalance locally and prepares it for the final step: a left rotation applied to p to balance the "Right-Right" case. If p is already balanced, the function simply returns p without modification.

6.2.11 Insertions

```
node* r_insert(node* p, int data) {
    if (!p) return new node(data);

    if (data < p->data) {
        p->left = r_insert(p->left, data);
    } else if (data > p->data) {
        p->right = r_insert(p->right, data);
    }

    p->height = 1 + std::max(p->left ? p->left->height : -1, p->right ? p->right->height : -1);
    p->balance = getBalance(p);

    return balance(p);
}

void insert(int data) {
    root = r_insert(root, data);
}
```

The r_insert function takes two parameters: p, a pointer to the current node in the tree, and data, the integer value to be inserted. If p is nullptr (indicating an empty spot in the tree), a new node with the given data is created and returned, establishing the base case for recursion.

If the node already exists, r_insert decides where to place the new value by comparing data with p->data. If data is less than p->data, it recursively calls r_insert on the left child (p->left). Otherwise, if data is greater, it calls r_insert on the right child (p->right). This ensures that the AVL tree retains its binary search property.

After the recursive insertion, r_insert updates the height of the current node p by setting it to 1 plus the maximum height of its left and right children. It then calculates the node's balance factor using getBalance(p), which is essential for determining if the subtree rooted at p has become unbalanced. Finally, balance(p) is called to apply any necessary rotations to maintain the AVL tree's balance, and the function returns the (possibly new) root of the subtree.

The insert function is a wrapper that starts the insertion process at the root node of the tree, calling r_insert with root and the specified data value. This encapsulates the recursive insertion logic and initiates it from the top of the tree.

Note: Worry about ancestory heights (and therefore balance factors) after rotations is not necessary, height and balance calculations occurs only when the insert method returns to that node in the recursion, and thus only after a rotation will an ancestory height and balance calculation be made.

6.2.12 Removing nodes

To implement removal from an AVL tree, you need to perform a few key steps: find and remove the node, balance the tree afterward, and adjust heights and balance factors as necessary

Start by locating the node with the target value. If found, delete it using standard BST deletion rules

After removing a node, retrace your steps back to the root, updating the height and balance factor of each ancestor. If a node is unbalanced, apply rotations to restore balance.

As with insertion, use recursive balancing and height adjustments after deletion.

```
node* succ(node* p) {
    while (p->left) p = p->left;
    return p;
}
node* r_remove(node* p, int data) {
    if (!p) return nullptr;  // Node not found
    // Traverse the tree to find the node to delete
    if (data < p->data) {
        p->left = r_remove(p->left, data);
    } else if (data > p->data) {
        p->right = r_remove(p->right, data);
    } else {  // Node to delete is found
        if (!p->left) {
            node* temp = p->right;
            delete p;
            return temp;
        } else if (!p->right) {
            node* temp = p->left;
            delete p;
            return temp;
        } else {  // Node has two children
            node* temp = succ(p->right);
            p->data = temp->data;  // Replace data
            p->right = r_remove(p->right, temp->data);
        }
    }
    // Update height and balance
    p->height = 1 + std::max(p->left ? p->left->height : -1, p->right ? p->right->height : -1);
    p->balance = getBalance(p);

    // Balance the node if necessary
    return balance(p);
}
void remove(int data) {
    root = r_remove(root, data);
}
```

r_remove, performs a recursive search for the node to delete based on the value data. If data is less than p->data, it recurses into the left subtree; if data is greater, it recurses into the right subtree. When the node to delete is found, three cases are handled based on the structure of p:

1. If p has no left child, p is replaced by its right child, effectively bypassing it in the tree.

2. If p has no right child, p is replaced by its left child.

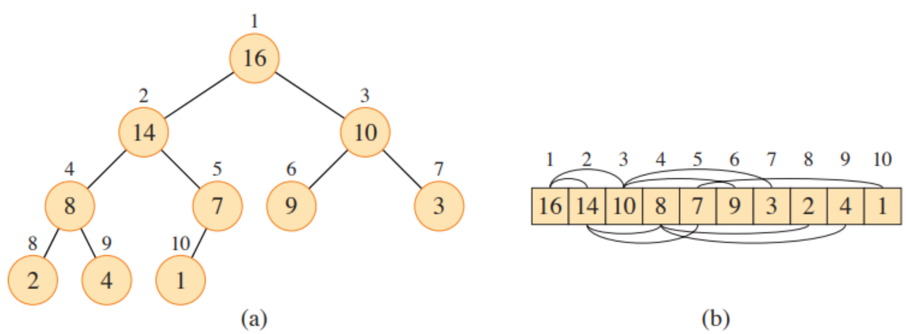
3. If p has two children, the in-order successor (smallest node in p->right) is found using succ. The data in p is replaced with this successor's data, and the successor node is then removed from p->right to avoid duplication.

After deletion, the function updates the height and balance of p, recalculating the height as one plus the maximum height of its children. The balance factor, which measures the difference in height between the left and right subtrees, is also recalculated. Finally, the function calls balance(p) to ensure that any imbalance introduced by the deletion is corrected. The remove function wraps this entire process, starting the deletion at the root. This ensures the AVL tree remains balanced after a node is removed.

Heaps and Priority Queues (Zero based)

The *(binary) heap* data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array $A[0 : n-1]$ that represents a heap is an object with an attribute A.heap-size, which represents how many elements in the heap are stored within array A. That is, although $A[0 : n-1]$ may contain numbers, only the elements in $A[0 : A.\text{heap-size} - 1]$, where $0 \leqslant A.\text{heap-size} \leqslant n$, are valid elements of the heap. If $A.\text{heap-size} = 0$, then the heap is empty. The root of the tree is $A[0]$, and given the index i of a node,

The *(binary) heap* data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array $A[1 : n]$ that represents a heap is an object with an attribute A.heap-size, which represents how many elements in the heap are stored within array A. That is, although $A[1 : n]$ may contain numbers, only the elements in $A[1 : A.\text{heap-size}]$, where $0 \leqslant A.\text{heap-size} \leqslant n$, are valid elements of the heap. If $A.\text{heap-size} = 0$, then the heap is empty. The root of the tree is $A[1]$, and given the index i of a node,



there's a simple way to compute the indices of its parent, left child, and right child with the one-line procedures PARENT, LEFT, and RIGHT.

```
int parent(i) {
    return (i+1)/2 // Floor division
}

int left(i) {
    return 2i + 1
}

int right(i) {
    return 2i + 2:
}
```

For one based indexing, we would have $\frac{i}{2}$, $2i$, and $2i+1$

On most computers, the `LEFT` procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left by one bit position. Similarly, the `RIGHT` procedure can quickly compute $2i+1$ by shifting the binary representation of i left by one bit position and then adding 1. The `PARENT` procedure can compute $\left\lfloor \frac{i}{2} \right\rfloor$ by shifting i right one bit position. Good implementations of heapsort often implement these procedures as macros or inline procedures.

7.1 Max and Min heaps

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node i other than the root,

$$A[\text{parent}(i)] \geqslant A[i].$$

That is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A is organized in the opposite way: the min-heap property is that for every node i other than the root,

$$A[\text{parent}(i)] \leqslant A[i].$$

The smallest element in a min-heap is at the root.

7.2 Heapify an array

Heapify is a process used to maintain the heap property in a binary heap, either a max-heap or a min-heap. The heap property ensures that for every node in the heap:

- In a max-heap, the value of each node is greater than or equal to the values of its children.
- In a min-heap, the value of each node is less than or equal to the values of its children.

Heapify is typically used when you have an unsorted array and you want to turn it into a valid heap, or after inserting or deleting an element in a heap to restore the heap property.

The bottom-up approach to turning an array into a heap is also called the heapify process. This method is efficient for building a heap from an unordered array and runs in $O(n)$ time, which is faster than building the heap using successive insertions ($O(nlg\ n)$)

We start from the first non-leaf node, which is at position $\frac{n-1}{2}$, where n is the size of the array.

Traverse all non-leaf nodes from the last one to the root (from right to left in the array

For each non-leaf node, apply the sift down operation (also called percolate down), which ensures that the subtree rooted at this node satisfies the heap property.

Percolate Down Operation:

1. Compare the node with its children.
2. If the heap property is violated (for example, in a max-heap, the node is smaller than one of its children), swap the node with the largest child (for max-heap) or smallest child (for min-heap).
3. Repeat this process down the subtree until the heap property is restored or the node becomes a leaf.

7.3 Min-heap in c++

```cpp
void min_heapify(int arr[], int n, int i) {
    int smallest = i;    // Initialize smallest as root
    int left = 2 * i + 1;   // Left child index
    int right = 2 * i + 2;  // Right child index

    // If left child is smaller than the root
    if (left < n && arr[left] < arr[smallest])
        smallest = left;

    // If right child is smaller than the smallest so far
    if (right < n && arr[right] < arr[smallest])
        smallest = right;

    // If the smallest is not the root
    if (smallest != i) {
        std::swap(arr[i], arr[smallest]);

        // Recursively heapify the affected subtree
        min_heapify(arr, n, smallest);
    }
}

void build_heap(int arr[], int n) {
    // Start from the last non-leaf node and heapify each node
    for (int i = (n - 1) / 2; i >= 0; --i) {
        min_heapify(arr, n, i);
    }
}
```

The min_heapify function is designed to maintain the min-heap property for a subtree rooted at a given index i. The process begins by identifying the left and right children of the node at index i. The function then compares the current node with its children to find the smallest value. If one of the children is smaller than the current node, a swap occurs between the node and the smallest child. After the swap, the function is recursively called on the affected child to ensure that the min-heap property is maintained further down the subtree.

7.4 Max-heap in c++

```cpp
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}


void build_heap(int arr[], int n) {
    for (int i=n-1/2; i>=0; --i) {
        heapify(arr, n, i);
    }
}
```

7.5 Percolating

Percolating in heaps refers to adjusting a node's position to maintain the heap property, which can occur in two directions: percolate up or percolate down.

Note that the percolate direction refers to the direction in which we move a node to get its correct locating in the heap.

If we are comparing a given node to its children, its percolate down. If we compare with its parent, its percolate up.

7.5.1 Percolate up

Used when adding a new element to the heap (usually at the end). The new element is moved up the heap by comparing it with its parent node. If it violates the heap property (e.g., it's smaller than its parent in a min-heap), it swaps with the parent. This continues until the node is correctly positioned.

```
void percUp(vector<int>& v, int i) {
    while (i>0) {
        int parent = (i-1)/2;
        if (v[i] > v[parent]) {
            swap(v[i], v[parent]);
            i = parent;
        } else break; // In the correct place
    }
}
```

7.5.2 Percolate down

Used when removing the root element (like in a heap deletion). The last element is moved to the root and then ”percolates down” by swapping with the smaller child (in a min-heap) if it violates the heap property. This process repeats until the node is correctly placed.

```
void percDown(vector<int>& v, int n, int i) {
    // Assume the largest is the current node
    int largest = i;
    // Get index of both children
    int left = 2*i+1;
    int right = 2*i+2;

    // If the left child exists, and is larger, update largest
    if (left < n && v[left] > v[largest]) {
        largest = left;
    }
    // If the right child exists, and is larger, update largest
    if (right < n && v[right] > v[largest]) {
        largest = right;
    }

    // If the largest was changed, swap. Then call percUp on the node that had the largest value.
    if (largest != i) {
        swap(v[i], v[largest]);
        percUp(v,n,largest);
    }
}
```

Because this function takes a node and compares it with its children, the value in the node moves down. Hence, percolate down.

7.6 Inserting into a heap

In a top-down approach to inserting into a min-heap, you maintain the heap property while inserting a new element. Specifically, you start by placing the new element at the bottom of the heap (in the next available position), and then you ”bubble up” (also known as ”heapify up” or ”percolate up”) to restore the heap property.

- **Insert the new element at the bottom:** Add the new element in the first available position (the next available spot in the array representation, which maintains a complete binary tree structure)
- **Bubble up (heapify up):** Compare the newly inserted element with its parent.

  If the new element is smaller than its parent (which violates the min-heap property), swap the two.

  Continue this process (i.e., compare with the next parent) until:

  - The new element is larger than or equal to its parent, or
  - The element reaches the root of the heap.
- The process terminates when the new element is in a position where it is larger than its parent or becomes the root.

The code for a min-heap insert is as follows

```
void percUp(vector<int>& v, int i) {
    int parent = (i-1)/2;

    while (i>0 && v[i] < v[parent]) {
        std::swap(v[i], v[parent]);

        i = parent;
    }
}

void heapInsert(vector<int>& v, int element) {
    v.push_back(element);
    int index = v.size() - 1;
    percUp(v, index);
}
```

7.7 Removing the root

Removing an element from a min-heap (typically the root, i.e., the minimum element) involves maintaining the min-heap property after removal. The most common removal operation is to delete the root (the smallest element in a min-heap). The process of removal involves the following steps:

1. **Replace the root with the last element:** The root (at index 0) is the element to be removed, so we replace it with the last element in the heap (this ensures the complete binary tree property is maintained).

2. **Remove the last element:** After swapping, the last element is removed from the heap (usually by reducing the heap's size).

3. **Heapify down (percolate down):** Starting from the root, compare the new root element with its children.

   If the root is larger than any of its children, swap it with the smaller child (to maintain the min-heap property).

   Continue this process until the heap property is restored (i.e., the element is smaller than both children or it reaches a leaf node).

```
void removeRoot(vector<int>& v) {
    std::swap(v[0], v[v.size()-1]);
    v.pop_back();
    min_heapify(v,v.size(), 0);
}
```

7.8 Removing an arbitary node

To remove an arbitrary node from a min-heap (not just the root), the process is slightly more complex than simply removing the root

1. **Replace the node to be deleted with the last element:** Swap the node to be deleted with the last element in the heap. This ensures that the complete binary tree property is maintained.
2. **Remove the last element:** After the swap, the last element (now at the position of the deleted node) is removed from the heap.
3. **Restore the heap property:** After the swap, the heap property might be violated both upwards and downwards. So, depending on the value of the swapped element, either heapify up or heapify down from the position where the node was deleted.
   - Heapify up if the swapped element is smaller than its parent.
   - Heapify down if the swapped element is larger than its children

We continue this process until there are no more nodes that match the passed element. The c++ code for a min-heap erase function is as follows.

```
void erase(vector<int>& v, int element) {
    bool found = false;
    while (!found) {
        found = false;
        int i=0;
        for (;i<(int)v.size(); ++i) {
            if (v[i] == element) {
                found = true;
                std::swap(v[i], v[v.size()-1]);
                v.pop_back();

                if (v[i] > v[(i-1)/2]) {
                    min_heapify(v, v.size(), i);
                } else {
                    percUp(v, i);
                }
            }
        }
        if (i == v.size()) break;
    }
}
```

7.9 Priority queues

Heaps are used to implement priority queues because they efficiently maintain the highest (or lowest) priority element at the root. In a max-heap, the largest element is always at the top, while in a min-heap, the smallest element is. This allows for quick access to the highest priority element in $O(1)$ time. Insertion and removal (reordering) of elements take $O(lg\ n)$ time, making heaps an optimal choice for priority queues where these operations need to be fast and frequent.

7.9.1 Interface

- **Insert (or Enqueue):** Adds an element to the priority queue based on its priority.
- **Pop**: Retrieve and remove the item with the highest priority (root of the heap)
- **Top**: Retrieve the item with the highest priority
- **Size:** Get the number of items in the queue
- **Empty:** Checks if the priority queue has any elements.

```
class priority_queue {
    vector<int> heap; // or any random access container

    void percDown(int n, int index);
    void PercUp(int index)

public:
    void insert(int element);
    int pop();
    int top();
    size_t size();
    bool empty();
};
```

Percolate down, up, insert, and pop can be found in the heap examples above.

7.9.2 Insert, pop, and top

These are simple methods to implement.

```
// Insert into the heap
void insert(int element) {
    heap.push_back(element);
    int index = heap.size() - 1;
    percUp(index);
}

// Retrieve and remove the root
int pop() {
    if (heap.empty()) return -1;

    int ret = heap[0];
    std::swap(heap[0], heap[heap.size()-1]);
    heap.pop_back();
    percDown((int)heap.size(), 0);
    return ret;
}

// Retrieve the root
int top() {
    return heap[0];
}
```

7.9.3 Size and Empty

```
size_t size() {
    return (size_t)heap.size();
}

bool empty() {
    return heap.empty();
}
```

Sorting

Sorting is a fundamental concept in data structures and algorithms where elements in a collection are arranged in a specific order, typically ascending or descending. Sorting enables efficient data access, searching, and data organization in applications. There are various sorting algorithms, each with different characteristics and efficiency, such as time and space complexity.

8.1 Bubble, selection, insertion

The three simplest sorting algorithms are bubble sort, selection sort, and insertion sort. They are all $O(n^2)$ comparison based sorting algorithms.

8.1.1 Bubble sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order, "bubbling" the largest unsorted element to the end of the list on each pass.

```
void bubbleSort(vector<int>& v, int n) {
    bool swapped = true;
    while (swapped) {
        swapped = false;
        for (int i=0; i<n-1; ++i) {
            if (v[i+1] < v[i]) {
                swap(v[i], v[i+1]);
                swapped=true;
            }
        }
        --n;
    }
}
```

By decrementing n after each complete pass, you avoid unnecessary comparisons in already sorted parts of the vector, making the algorithm slightly more efficient without affecting the $O(n^2)$ time complexity.

- **Outer Loop**: The `while` loop continues as long as swaps are happening, so it terminates early if the list becomes sorted before completing all n passes.
- **Inner Loop**: The `for` loop compares adjacent elements and swaps them if they're out of order. After each pass, the largest unsorted element "bubbles up" to the correct position, so the next pass can ignore the last element.

8.1.1.1 Complexity

Worst Case: $O(n^2)$

In the worst case (a reverse-sorted list), Bubble Sort requires n passes, and each pass involves up to $n-1$ comparisons.

Average Case: $O(n^2)$

On average, Bubble Sort still performs $O(n^2)$ comparisons due to repeated passes through the entire array.

Best Case: $O(n)$

If the list is already sorted, Bubble Sort can stop early if no swaps occur during a pass (often implemented with a flag to detect this). In this case, it only takes one pass to confirm that the list is sorted.

8.1.2 Selection sort

Selection Sort sorts an array by repeatedly finding the minimum element from the unsorted part and moving it to the beginning.

1. Start at the beginning of the array.
2. For each position i, find the smallest element in the unsorted portion (from i to the end).
3. Swap this minimum element with the element at position i
4. Move to the next position and repeat until the array is sorted.

```cpp
void selectionSort(vector<int>& v, int n) {
    for (int i=0; i<n-1; ++i) {
        int min = i;
        for (int j=i+1; j<n; ++j) {
            if (v[j] < v[min]) {
                min = j;
            }
        }
        std::swap(v[min], v[i]);
    }
}
```

8.1.2.1 Complexity

Worst Case: $O(n^2)$

Selection Sort always performs n passes, each requiring a search through the unsorted portion to find the minimum element, resulting in $O(n^2)$ comparisons.

Average Case: $O(n^2)$

Selection Sort performs the same number of comparisons regardless of the initial order of elements, as it always looks for the minimum in the unsorted portion.

Best Case: $O(n^2)$

Even if the array is already sorted, Selection Sort will still go through n passes and $O(n^2)$ comparisons, making it inefficient for nearly sorted data.

8.1.3 Insertion sort

Insertion Sort sorts an array by building a sorted portion one element at a time

1. Start with the second element, assuming the first element is trivially sorted.
2. For each element, compare it with the elements in the sorted portion to its left.
3. Shift larger elements one position right until you find the correct spot for the current element.
4. Insert the current element in its correct position.
5. Repeat until all elements are sorted.

```
    void insertionSort(vector<int>& v, int n) {
        for (int i=1; i<n; ++i) {
            int key=v[i];
            int j=i-1;

        while (j>=0 && v[j] > key) {
            v[j+1] = v[j];
            --j;
        }
        v[j+1] = key;
    }
}
```

Imagine sorting a hand of playing cards:

- You pick up cards one by one.
- For each new card, you compare it with the cards in your hand from right to left, sliding them over if they're larger than the new card, until you find the correct spot to insert it.
- Each card you add to your hand stays sorted with the previous ones, just like key gets inserted in the sorted sub-array.

This method is efficient when the list is nearly sorted since fewer shifts are needed, but it becomes less efficient for randomly ordered arrays due to repeated comparisons and shifts.

8.1.3.1 Complexity

Worst Case: $O(n^2)$

In the worst case (reverse-sorted list), each element is compared with all previous elements, resulting in $O(n^2)$ comparisons and shifts.

Average Case: $O(n^2)$

Insertion Sort generally performs $O(n^2)$ operations in the average case due to repeated comparisons and shifts.

Best Case: $O(n)$

If the array is already sorted, Insertion Sort only needs to confirm each element is in place, requiring just $O(n)$ comparisons and no shifts. This makes it very efficient for nearly sorted arrays.

8.2 Heap sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure, specifically a max-heap, to sort elements in ascending order (or a min-heap for descending order).

1. **Build a Max-Heap:** Convert the array into a max-heap, where each parent node is greater than its children. This ensures the largest element is at the root.
2. **Extract Maximum and Swap:** Swap the root (largest element) with the last element in the heap. This moves the largest element to its correct position in the sorted array.
3. **Heapify:** After the swap, the heap may no longer satisfy the heap property. Restore the max-heap by performing a ”heapify” operation on the root to maintain the max-heap structure.
4. **Repeat:** Continue extracting the maximum element, swapping, and heapifying, reducing the heap size each time. This process sorts the array in-place.

```
void heapSort(vector<int>& v, int n) {
    heapify(v, n);
    int last = n-1;
    while (last > 0) {
        swap(v[0], v[last--]);
        percDown(v, last, 0);
    }
}
```

The functions *heapify* and *percDown* can be found in the *max-heap in c++* section above, where they are called *build-heap* and *heapify* respectively.

8.2.1 Complexity

Heap Sort has a time complexity of $O(n\lg n)$ in the best, worst, and average cases, making it more efficient than $O(n^2)$ sorting algorithms for larger datasets

Bulding the max-heap from an arbitary container takes $O(n)$ time. Re-heapify the remaining elements (restore the max-heap property) by ”sifting down” the new root takes $O(\lg n)$ time. Since we do this sift operation n times, the complexity of *heapSort* is therefore $O(n\lg n)$, which makes the entire algorithm $O(n) + O(n\lg n) = O(n\lg n$.

8.3 BST Sort

BST Sort is a sorting algorithm that leverages a Binary Search Tree (BST) to sort elements. It works by inserting all elements into a BST and then performing an in-order traversal of the tree to retrieve the elements in sorted order.

8.3.1 Insert

The insert for bst follows the standard bst insert method described in a previous section.

```
node* insert(node* p, int data) {
    if (!p) return new node(data);

    if (data < p->data) {
        p->left = insert(p->left, data);
    } else {
        p->right = insert(p->right, data);
    }

    return p;
}
```

8.3.2 The inorder traversal

The inorder follows the same logic as before, but when processing each node, we insert the data member into a passed vector.

```
void inorder(node* p, vector<int>& v) {
    if (!p) return;
    inorder(p->left, v);
    v.push_back(p->data);
    inorder(p->right, v);
}
```

8.3.3 The BST sort function

```
void bstsort(vector<int>& v) {
    vector<int> sorted;
    node* root = nullptr;

    for (const auto& item : v) {
        root = insert(root, item);
    }
    inorder(root, sorted);
    clear(root);

    v = sorted;
}
```

The bstsort function sorts a vector of integers using a Binary Search Tree (BST) approach. It starts by creating an empty vector, sorted, which will eventually hold the sorted elements, and initializes root as nullptr, representing an initially empty BST.

For each item in the input vector v, the function calls insert to add the item to the BST, progressively building the tree in a way that respects the BST property (left child nodes are less than the parent node, and right child nodes are greater). After constructing the BST, the function performs an in-order traversal using inorder, which appends each node's data to the sorted vector in ascending order.

Finally, clear is called to delete all nodes from the BST, freeing up dynamically allocated memory. The function then assigns sorted back to the input vector v, so v now contains the elements in sorted order.

BST Sort is more of a conceptual exercise in data structures than a practical sorting method because balanced tree structures (e.g., AVL trees, red-black trees) are required to ensure $O(nlg\ n)$ performance consistently.

8.3.4 Inplace sorting

To perform the sort "in-place" in bstsort, we can eliminate the extra sorted vector by writing the sorted values directly back into the input vector v during the in-order traversal. This approach leverages an index to keep track of the position in v where each next sorted value should be placed

```
void inorder2(node* p, vector<int>& v, int& index) {
    if (!p) return;
    inorder2(p->left, v, index);
    v[index++] = p->data;
    inorder2(p->right, v, index);
}

void bstsort(vector<int>& v) {
    node* root = nullptr;

    for (const auto& item : v) {
        root = insert(root, item);
    }
    int index = 0;
    inorder2(root, v, index);
    clear(root);
}
```

8.3.5 Complexity

- **Average Case:** $O(n \lg n)$, where n is the number of elements. This is because inserting each element in a balanced BST takes $O(\lg n)$ time
- **Worst Case:** $O(n^2)$, if the tree becomes unbalanced, like when inserting elements in sorted order into a simple BST, resulting in a degenerate tree (like a linked list).
- **Space Complexity:** $O(n)$, for storing the BST nodes.

8.4 Getting characters from a string

We can index strings as if they were arrays

```
$s = "this string has alot of characters";
echo $s[0]; // t
```

Multi-way (m-way) search trees

A multiway search tree is a type of search tree where each node can have more than two children, unlike binary search trees, which are limited to two children per node. Multiway search trees generalize binary search trees by allowing nodes to hold multiple keys and have multiple pointers to child nodes, making them well-suited for managing large amounts of data in a balanced structure.

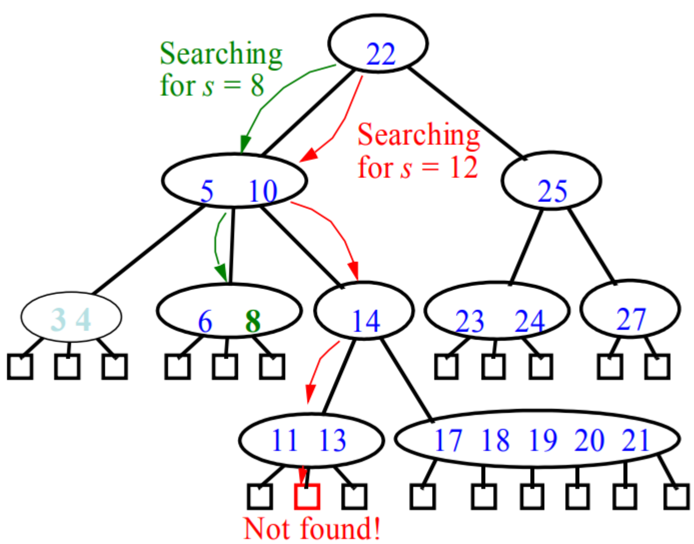
An m-Way tree of order m, each node contains a maximum of $m-1$ elements and m children.

The goal of an m-Way search tree of height h is to achieve $O(h)$ accesses for an insert, delete, or retrieval operation. Therefore, it ensures that the height h is close to $\log_m(n+1)$.

Each internal node of a multi-way search tree T:

1. Has at least two children
2. stores a collection of items of the form (k, x), where k is a key and x is an element
3. contains $d-1$ items, where d is the number of children

Children of each internal node are "between" items. All keys in the subtree rooted at the child fall between keys of those items



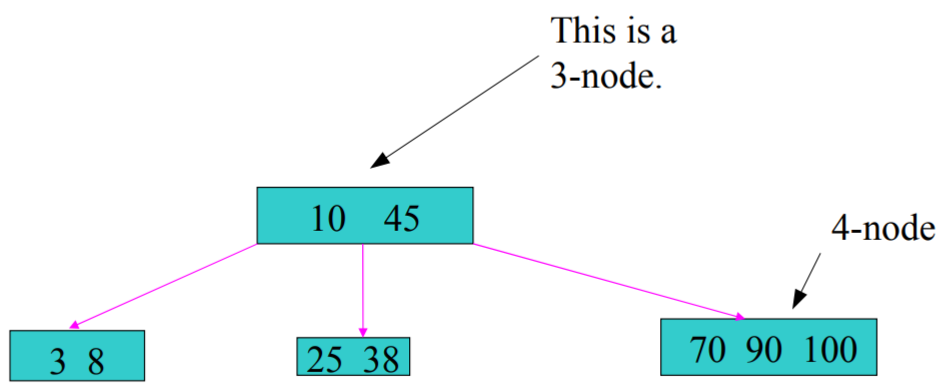
9.1 Multi-way Searching

Similar to binary searching, where if $s < k_1$, search the leftmost child. If $s > k_{d-1}$, search the rightmost child. But what if $d > 2$? Simply Find two keys k_{i-1} and k_i between which s falls, and search the child v_i .

9.2 2-4 (2-3-4) Trees

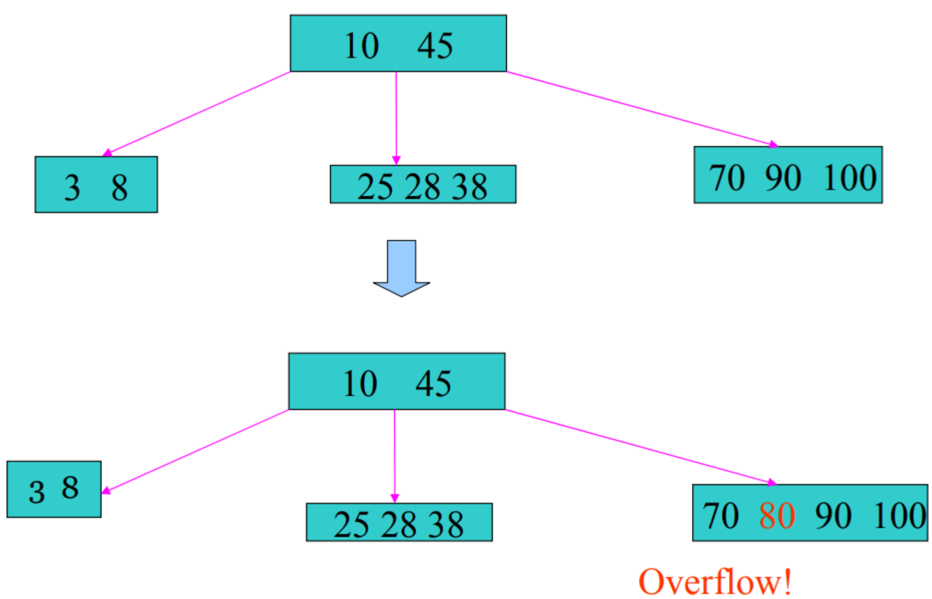
A 2-4 tree, also known as a 2-3-4 tree, is a multi-way search tree the following properties

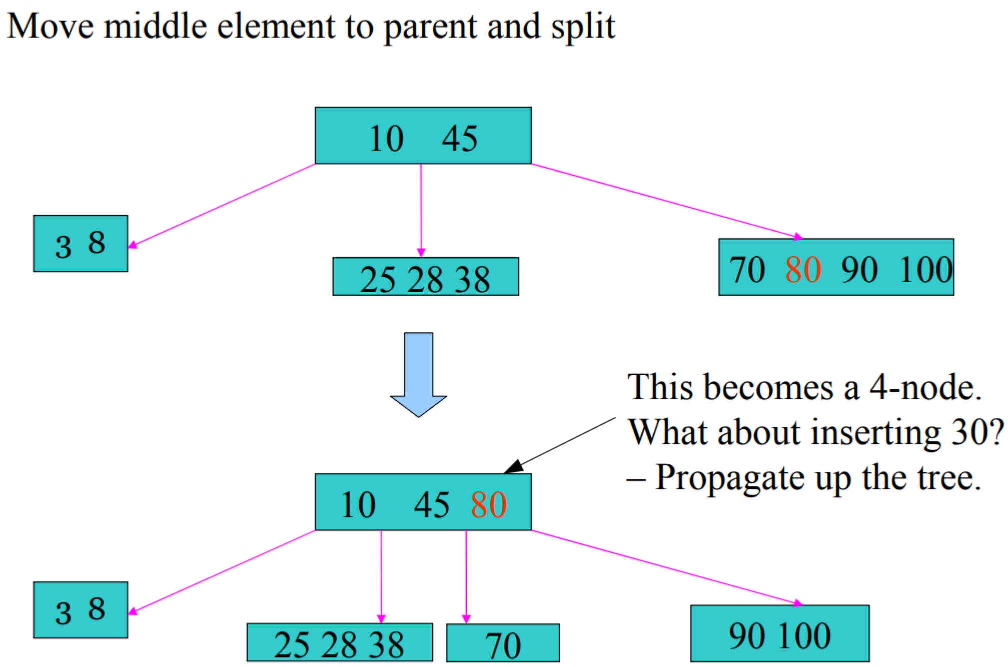
1. Nodes may contain 1, 2 or 3 items
2. A node with k items has $k+1$ children, except for leaf nodes. Such a node is called $(k+1)$-node
3. All leaves are on the same level



9.2.1 Insertion

To insert into a 2-4 tree, first, find the appropriate leaf. If there is room, just add the element to the leaf. If there is no room, move the middle item to parent and split remaining items among two children.



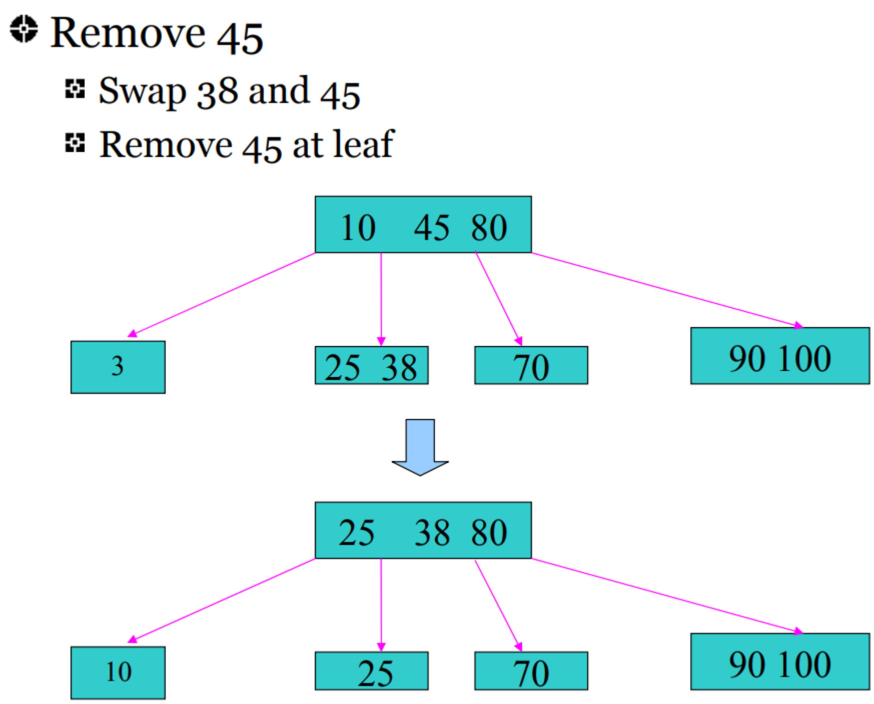


9.2.2 Removal

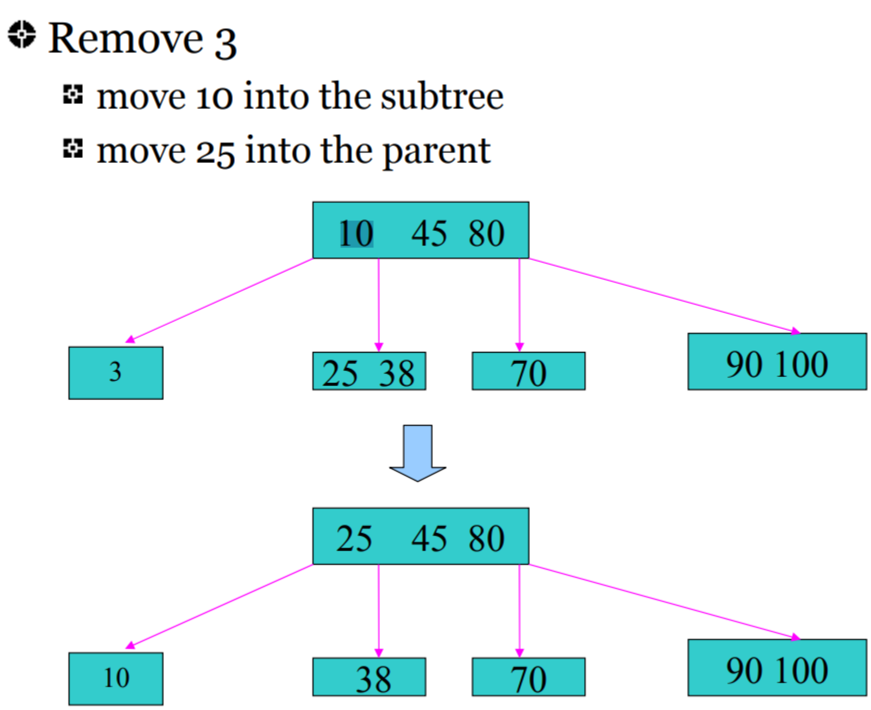
First, we find the key with a simple multi-way search. There are two cases

1. **Case 1:** It may be on the leave
2. **Case 2:** It may be in internal node

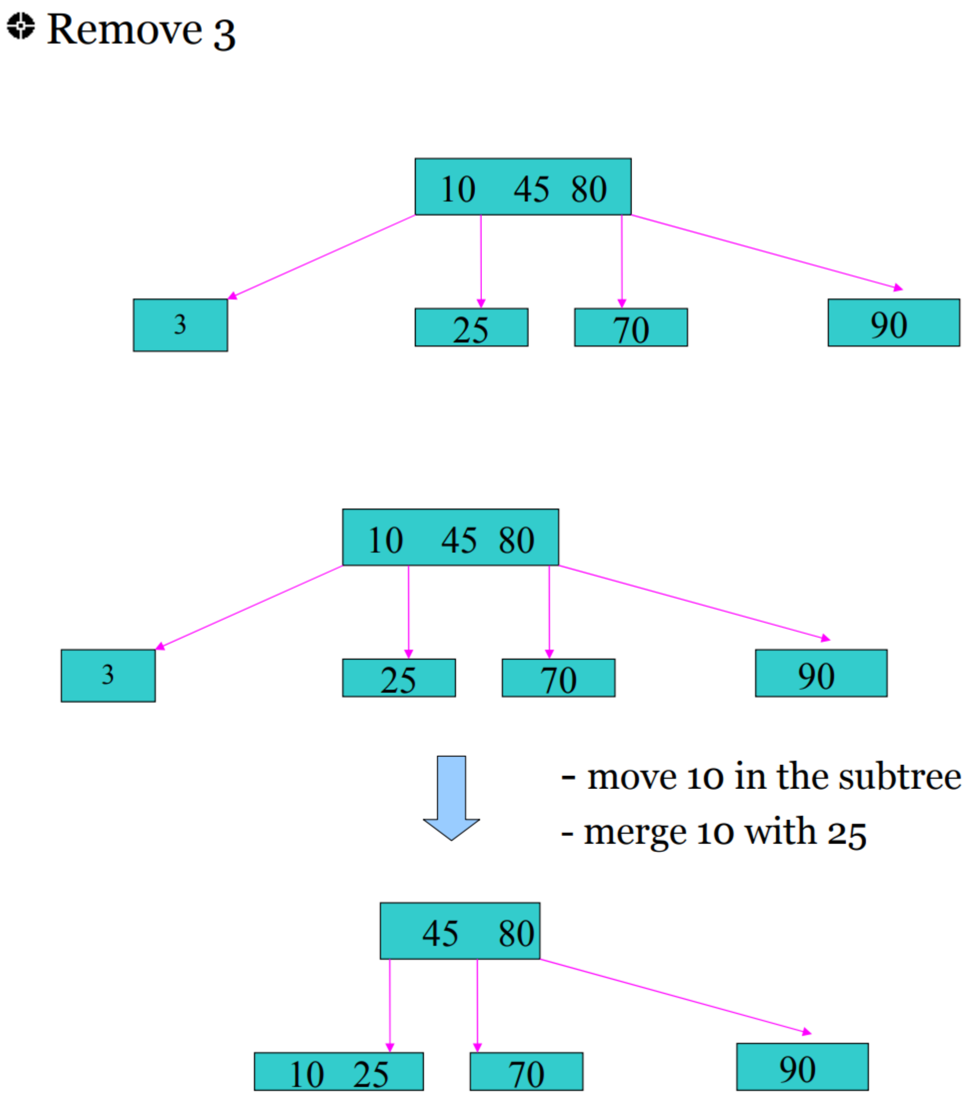
If the item to delete is in internal node, we can reduce to case 1 by first finding its immediate predecessor, swapping them, and then removing the item.



But what if there are not enough items in the node after removal? this is known as an *underflow*. In this case, pull an item from the parent, replace it with an item from a sibling - transfer



But what happens if the the siblings are 2-nodes (one element), we will not be able to steal from them. In this case, we preform *node merging*.



9.2.3 Properties

- 2-4 trees are easy to maintain
- Insertion and deletion take O(log n)
- Balanced trees

9.3 B-trees

Up to now, all data that has been stored in the tree has been in memory. If data gets too big for main memory, what do we do? If we keep a pointer to the tree in main memory, we could bring in just the nodes that we need.

For instance, to do an insert with a BST, if we need the left child, we do a disk access and retrieve the left child. If the left child is NIL, then we can do the insert, and store the child node on the disk

But storing the data requires disk accesses, which is expensive, compared to execution of machine instructions. If we can reduce the number of disk accesses, then the procedures run faster

The only way to reduce the number of disk accesses is to increase the number of keys in a node, wee see the problem in using this technique with binary search trees... The BST allows only one key per leaf/node.

If we increase the number of keys in the nodes, how will we do any tree operations effectively?

A B-tree is a self-balancing search tree data structure that maintains sorted data and allows searches, insertions, deletions, and sequential access in logarithmic time. B-trees are especially useful for managing large blocks of data in systems like databases and file systems, where data is stored on disk or other slow-access storage and needs to be accessed efficiently.

Unlike binary trees, where each node has at most two children, B-trees are multi-way trees where each node can have multiple children. The number of children a node can have is determined by the order of the tree.

B-trees maintain balance by ensuring that every path from the root to a leaf node has the same length, which guarantees logarithmic height and thus efficient operations.

Each node can contain a range of keys (from a minimum to a maximum number). This enables efficient storage and retrieval by storing more keys in fewer nodes, which minimizes the number of disk reads needed to access data.

The order m of a B-tree defines the maximum number of children each node can have. An order-m B-tree node can have up to $m - 1$ keys and m children.

The minimum degree t (often used instead of order) specifies the minimum number of children a non-root node must have, which is t or more.

- **Insertion:** When adding a key, nodes split if they reach their maximum capacity, ensuring the B-tree remains balanced.
- **Deletion:** When deleting a key, nodes may need to merge with their siblings if they go below their minimum capacity.

Because each node contains multiple keys, B-trees have a low height relative to the number of keys they store. This makes B-trees very efficient for disk-based storage, as it minimizes the number of disk accesses needed.

B-trees are widely used in database indexing, file systems, and other applications that involve managing large volumes of data that cannot fit into main memory.

Hashing (hash tables)

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\Theta(n)$ time in the worst case. In practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$

A hash table generalizes the simpler notion of an ordinary array. Directly addressing into an ordinary array takes advantage of the $O(1)$ access time for any array element.

To use direct addressing, you must be able to allocate an array that contains a position for every possible key

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, we compute the array index from the key.

10.1 Direct-address table

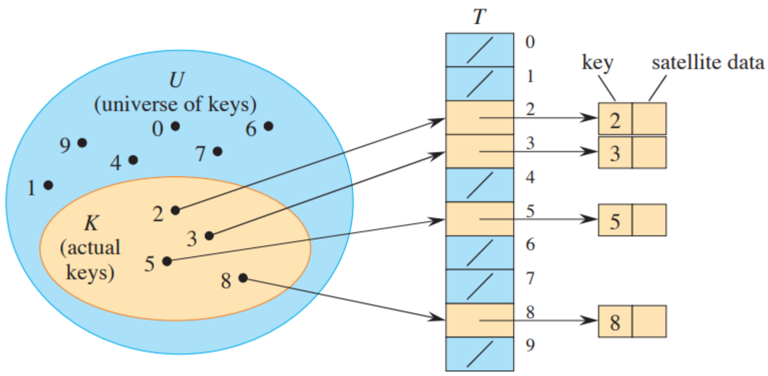
Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a distinct key drawn from the universe $U = \{0, 1, .., m-1\}$, where m is not too large.

To represent the dynamic set, you can use an array, or *direct-address table*, denoted by $T[0 : m-1]$, in which each position, or *slot*, corresponds to a key in the universe U. Figure 11.1 illustrates this approach. Slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

The dictionary operations DIRECT-ADDRESS-SEARCH, DIRECT-ADDRESS-INSERT, and DIRECT-ADDRESS-DELETE on the following page are trivial to implement. Each takes only $O(1)$ time.

The dictionary operations DIRECT-ADDRESS-SEARCH, DIRECT-ADDRESS INSERT, and DIRECT-ADDRESS-DELETE on the following page are trivial to implement. Each takes only $O(1)$ time

For some applications, the direct-address table itself can hold the elements in the dynamic set. That is, rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object, save space by storing the object directly in the slot. To indicate an empty slot, use a special key. Then again, why store the key of the object at all? The index of the object is its key! Of course, then you'd need some way to tell whether slots are empty




```
direct-address-search(T,k)
    return T[k]
direct-address-insert(T,x)
    T[x.key] = x
direct-address-delete(T,x)
    T[x.key] = nil
```

10.2 Hash tables

The downside of direct addressing is apparent: if the universe U is large or infinite, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirement reduces to $\Theta(|K|)$ while maintaining the benefit that searching for an element in the hash table still requires only $O(1)$ time. The catch is that this bound is for the *average-case time*[1], whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k, but with hashing, we use a *hash function* h to compute the slot number from the key k, so that the element goes into slot $h(k)$. The hash function h maps the universe U of keys into the slots of a *hash table* $T[0 : m-1]$:

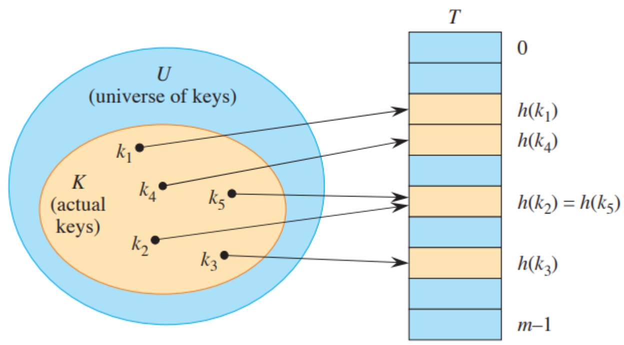
$$h : U \to \{0, 1, \ldots, m-1\}.$$

where the size m of the hash table is typically much less than $|U|$. We say that an element with key k *hashes to* slot $h(k)$, and we also say that $h(k)$ is the *hash value* of key k. The hash function reduces the range of array indices and hence the size of the array. Instead of a size of $|U|$, the array can have size m

An example of a simple, but not particularly good, hash function is $h(k) = k \bmod m$

There is one hitch, namely that two keys may hash to the same slot. We call this situation a collision. Fortunately, there are effective techniques for resolving the conflict created by collisions.

Of course, the ideal solution is to avoid collisions altogether. We might try to achieve this goal by choosing a suitable hash function h. One idea is to make h appear to be <random,= thus avoiding collisions or at least minimizing their number.



although a well-designed, random looking hash function can reduce the number of collisions, we still need a method for resolving the collisions that do occur

10.3 Independent uniform hashing

An "ideal" hashing function h would have, for each possible input k in the domain U, an output $h(k)$ that is an element randomly and independently chosen uniformly from the range $\{0, 1, \ldots, m-1\}$. Once a value $h(k)$ is randomly chosen, each subsequent call to h with the same input k yields the same output $h(k)$.

We call such an ideal hash function an independent uniform hash function. Such a function is also often called a random oracle

When hash tables are implemented with an independent uniform hash function, we say we are using independent uniform hashing

Independent uniform hashing is an ideal theoretical abstraction, but it is not something that can reasonably be implemented in practice.

10.4 Collision resolution by chaining

At a high level, you can think of hashing with chaining as a nonrecursive form of divide-and-conquer: the input set of n elements is divided randomly into m subsets, each of approximate size $\frac{n}{m}$. A hash function determines which subset an element belongs to. Each subset is managed independently as a list.

each nonempty slot points to a linked list, and all the elements that hash to the same slot go into that slot's linked list. Slot j contains a pointer to the head of the list of all stored elements with hash value j. If there are no such elements, then slot j contains NIL

The worst-case running time for insertion is $O(1)$ The insertion procedure is fast in part because it assumes that the element x being inserted is not already present in the table. To enforce this assumption, you can search (at additional cost) for an element whose key is x.key before inserting

For searching, the worst-case running time is proportional to the length of the list

Deletion takes $O(1)$ time if the lists are doubly linked. If the hash table supports deletion, then its linked lists should be doubly linked in order to delete an item quickly.

10.5 Analysis of hashing with chaining

Given a hash table T with m slots that stores n elements, we define the load factor α for T as $\alpha = \frac{n}{m}$, that is, the average number of elements stored in a chain. Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n. The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function—no better than using one linked list for all the elements. We clearly don't use hash tables for their worst-case performance.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average

for now we assume that any given element is equally likely to hash into any of the m slots. That is, the hash function is uniform. We further assume that where a given element hashes to is independent of where any other elements hash to. In other words, we assume that we are using independent uniform hashing

Because hashes of distinct keys are assumed to be independent, independent uniform hashing is universal: the chance that any two distinct keys k_1 and k_2 collide is at most $\frac{1}{m}$.

10.6 Hash functions

For hashing to work well, it needs a good hash function. Along with being efficiently computable, what properties does a good hash function have? How do you design good hash functions?

This section first attempts to answer these questions based on two ad hoc approaches for creating hash functions: hashing by division and hashing by multiplication.

Although these methods work well for some sets of input keys, they are limited because they try to provide a single fixed hash function that works well on any data, an approach called static hashing.

We then see that provably good average-case performance for any data can be obtained by designing a suitable family of hash functions and choosing a hash function at random from this family at runtime, independent of the data to be hashed. The approach we examine is called random hashing

Math algorithms

11.0.1 Euclidean GCD Algorithm

The GCD of two integers a and b (with $a \leqslant b$) is the largest integer that divides both a and b. The Euclidean algorithm is based on the principle that

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

This means that the GCD of two numbers doesn't change if the larger number is replaced by its remainder when divided by the smaller number. You keep repeating this until the remainder is 0, and the GCD will be the last non-zero remainder

```
int gcd(int a, int b)  {
    if (!b) return a;

    return gcd(b, a%b);
}
```