Алгоритмы Структуры данных

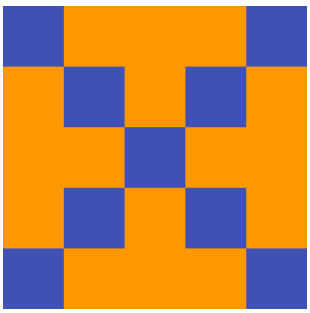
Структуры нужны чтобы:

- Управлять сложностью своих программ, делая их доступней для понимания.
- Создавать высокопроизводительные программы, эффективно работающие с памятью.

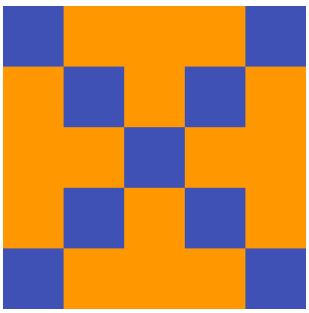
Структуры могут быть организованы с помощью алгортмов, а алгоритмы в свою очередь используют структуры в качестве входных и выходных данных. Использование определнных структур данных может существенно повысить эффективность алгоритмов.

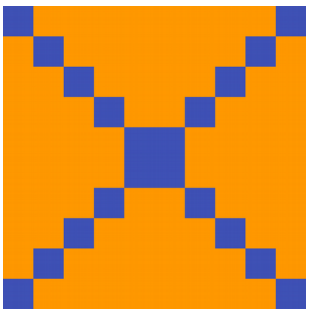
Алгоритм это последовательнось действий которая решает класс/семейство задач.

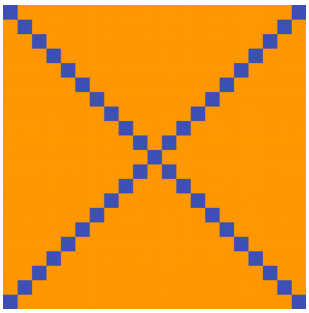
Пример задачи:
Дано квадратное изображение 5x5 пикселей нарисовать на нем крестик по диагоналям.



Пример семейства задач
Дано квадратное изображение NxN пикселей нарисовать на нем крестик по диагоналям.







Решение одной и той же задачи может быть не единственным. Для задачи описанной выше можно однозначно расчитать положение пикселей которые необходимо закрасить:
(x, y) = (i, i),
(i, N-1-i)

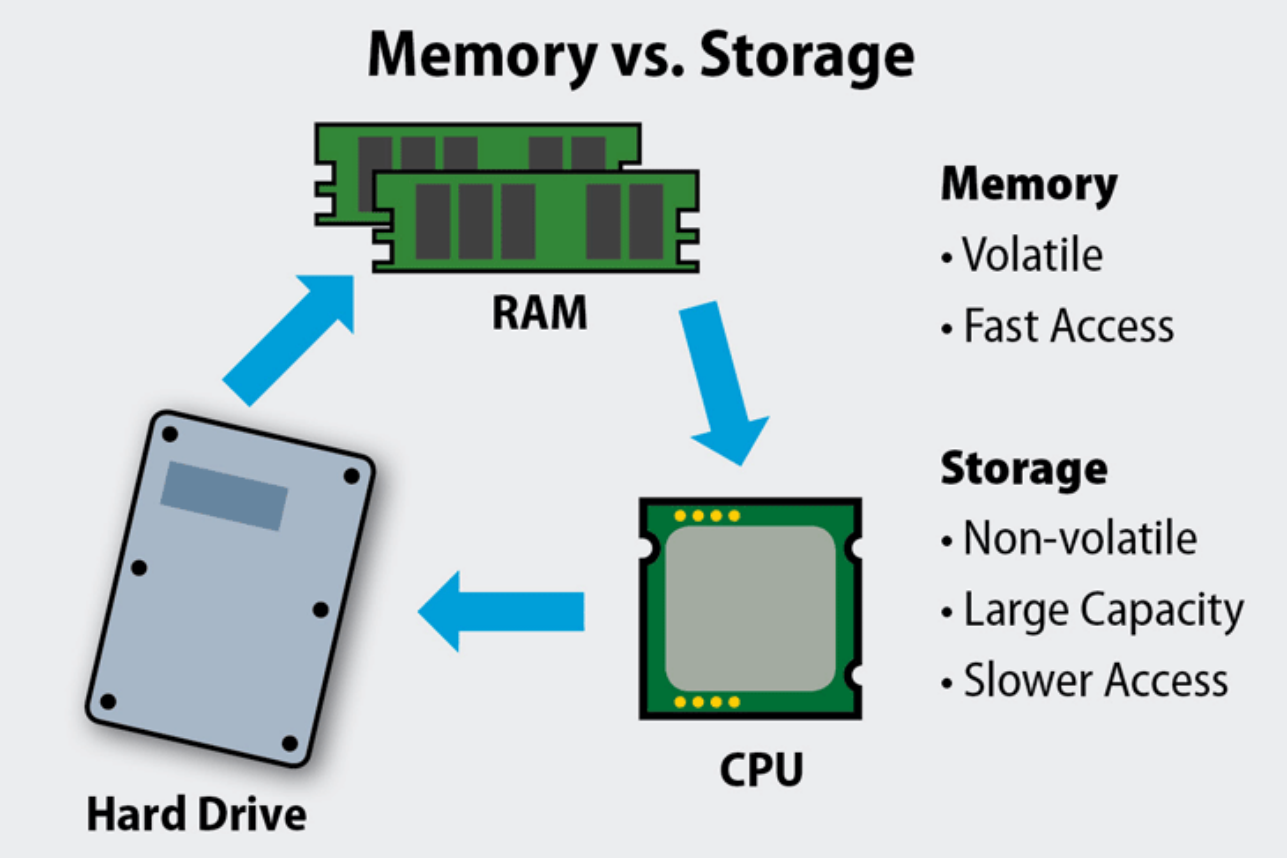
Для каждой строки i = 0 …. N-1.
Однако способы переходов между пикселями могут быть ранообразны. Если представить что у нас есть агент – робот который перемещается от пикселя к пикселю и зарисовывает его, то количество движений бота будет равно “сложности” алгоритма.

Действия робота:
1. Поворот на 90 градусов
2. Шаг на один пиксель.

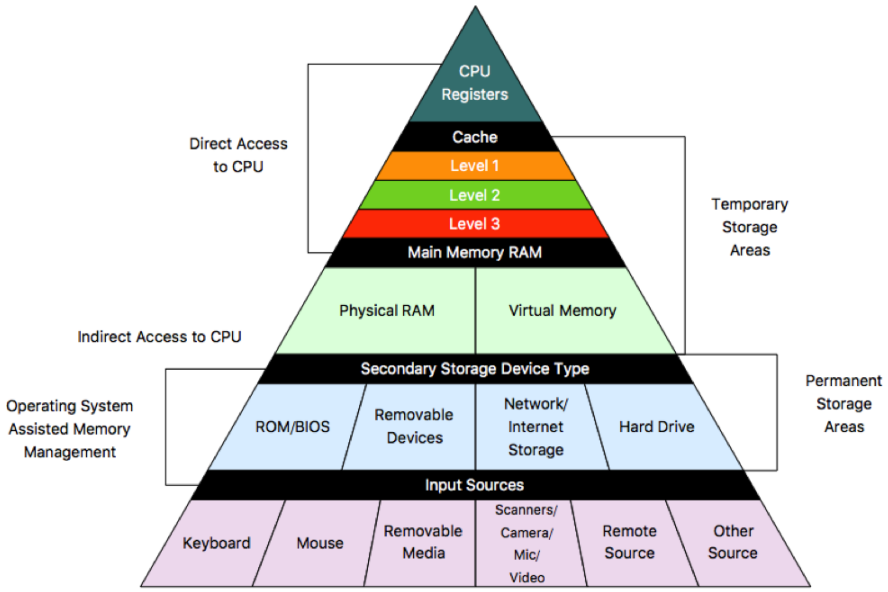
В этом курсе под сложностью алгоритма мы будем понимать **время** выполнения и занятую при этом **память**.

Ресурсы

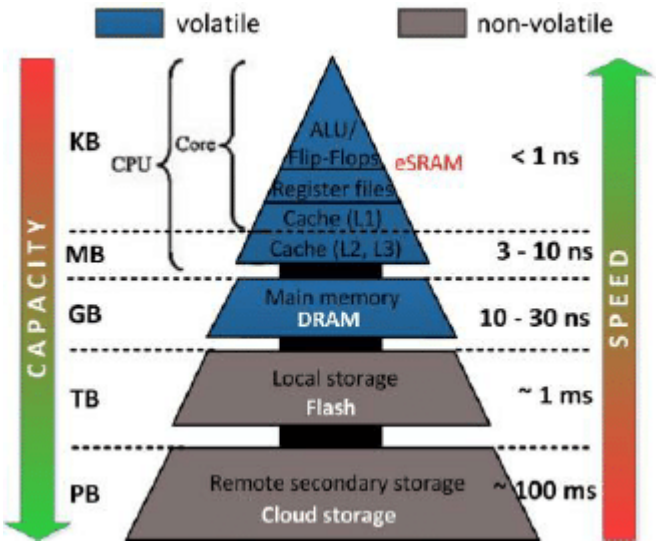
При рассмторении алгоритмов и структур данных мы обычно говорим про **время** (процессорное время/вычислительная способность) и **память** (объем оперативной памяти). На выполнение задаич мы можем потратить определенное время и занять некторый объем памяти. Так как алгоритмы обрабатывают данные, хранящиеся в структурах, а RAM является энергозависимой, появляется еще **диск** (объем дискового прострaства). При написании своих прогорамм следует позаюотится о том что бы не производить частых и объемных операций с диском.



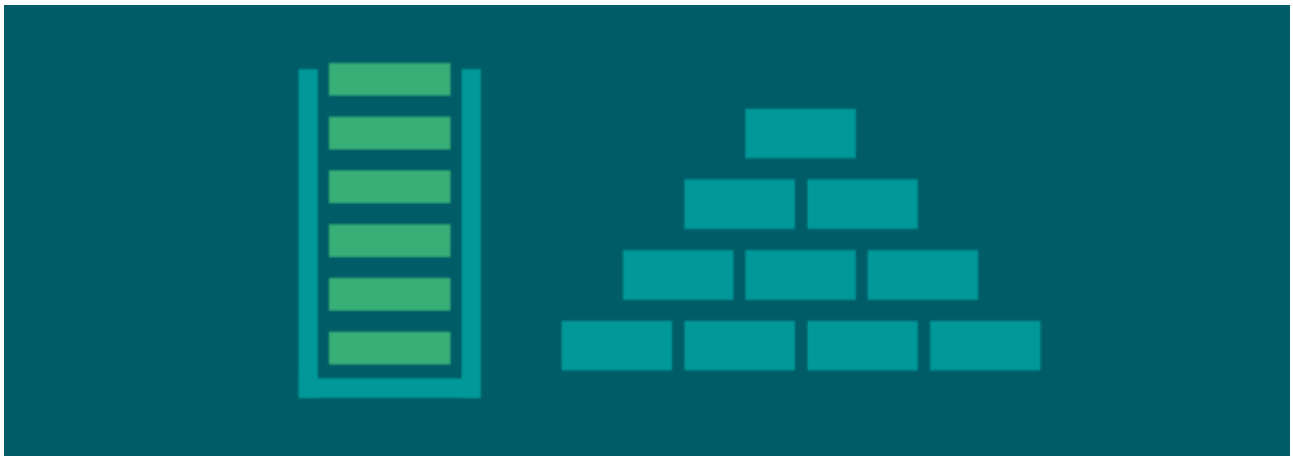
Более полная картина будет выглядеть следующим образом



Для сравнения ниже приведены скорости доступа к памяти:



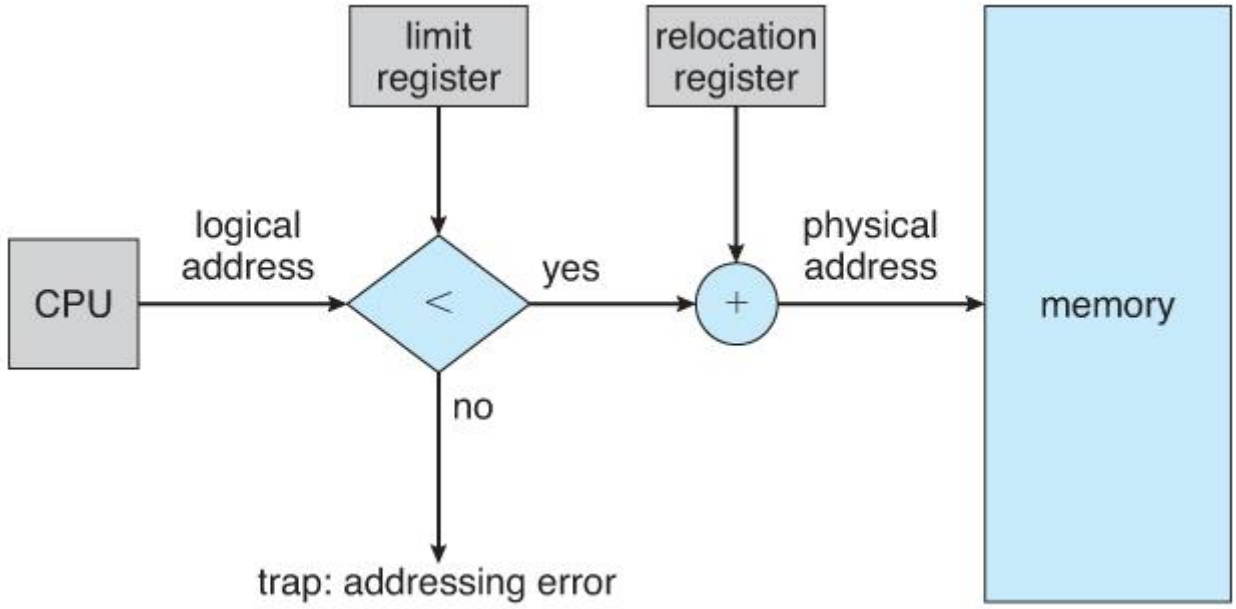
Стек и Куча



Стек — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

Куча — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физически, и это позволяет создавать динамические переменные.

allocation - time the program spends "allocating" and "deallocating" memory, including occasional *sbrk* (or similar) virtual address allocation as the heap usage grows

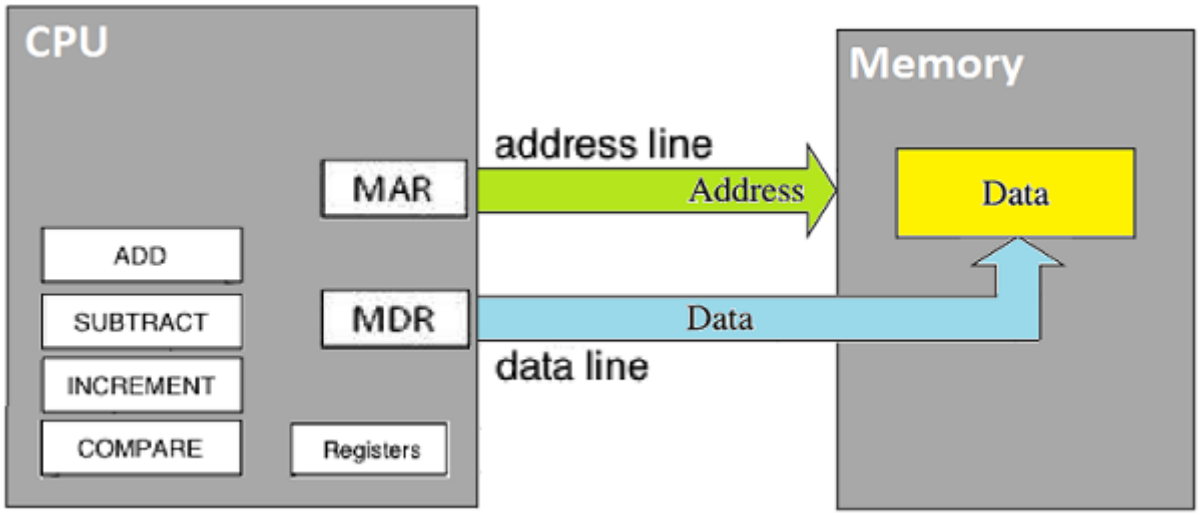


Для данных стека относительный адрес указателя стека (относительно регистра) может быть вычислен и записан в виде числа во время компиляции. Затем регистр указателя стека может быть скорректирован в соответсвии общего размера аргументов функции, локальных переменных, адресов возврата и пр. Добавление большего количества переменных которые попадут в стек просто изменит его общий размер.

Практически такой подход лишен накладных расходов на выделение/освобождение памяти во время выполнения, в то время как накладные расходы при использовании кучи возникаюит практически всегда и могут быть значительными для некоторых приложений.

Для кучи библиотека распределения кучи, данные в кототорой поддерживаются во время выполнения, должна отслеживать и обновлять инофрмацию что бы не потерять связь как орпделенные блоки кучи связаны с конкретными указателями, которые библиотека предоставила для приложение, пока оно не освободит или не удалит память.

access - differences in the CPU instructions used by the program to access globals vs stack vs heap, and extra indirection via a runtime pointer when using heap-based data,



Поскольку для стека абсолютный виртуальный адрес или адрес относительно регистра указателя стека может быть вычислен во время компиляции, доступ во время выполнения осуществляется очень быстро. С кучей программа должна обращаться к данным через определяемый во время выполнения указатель, содержащий адрес виртуальной памяти в куче, иногда со смещением от указателя на конкретный элемент данных, применяемый во время выполнения. На некоторых архитектурах это может занять немного больше времени. Для доступа к куче и указатель, и память кучи должны находиться в регистрах, чтобы данные были доступны (так что кэширование ЦП больше, а масштабирование - больше пропусков кеша / накладных расходов по сбоям). Примечание: эти затраты часто незначительны - даже не стоит смотреть или задумываться, если только вы не пишете что-то, где время ожидания или пропускная способность чрезвычайно важны.

Источники:

https://tproger.ru/translations/programming-concepts-stack-and-heap/
https://stackoverflow.com/questions/24057331/is-accessing-data-in-the-heap-faster-than-from-the-stack
https://habr.com/ru/post/310794/
https://habr.com/ru/post/312078/
https://www.researchgate.net/figure/Typical-structure-of-a-computer-memory-hierarchy_fig1_281805561

https://www.youtube.com/watch?v=_8-ht2AKyH4
https://www.youtube.com/watch?v=wJ1L2nSIV1s

Самостоятельно:
Компиляция и выполнение.
https://stackoverflow.com/questions/846103/runtime-vs-compile-time
Производительность stack-vs-heap.
https://github.com/spuhpointer/stack-vs-heap-benchmark