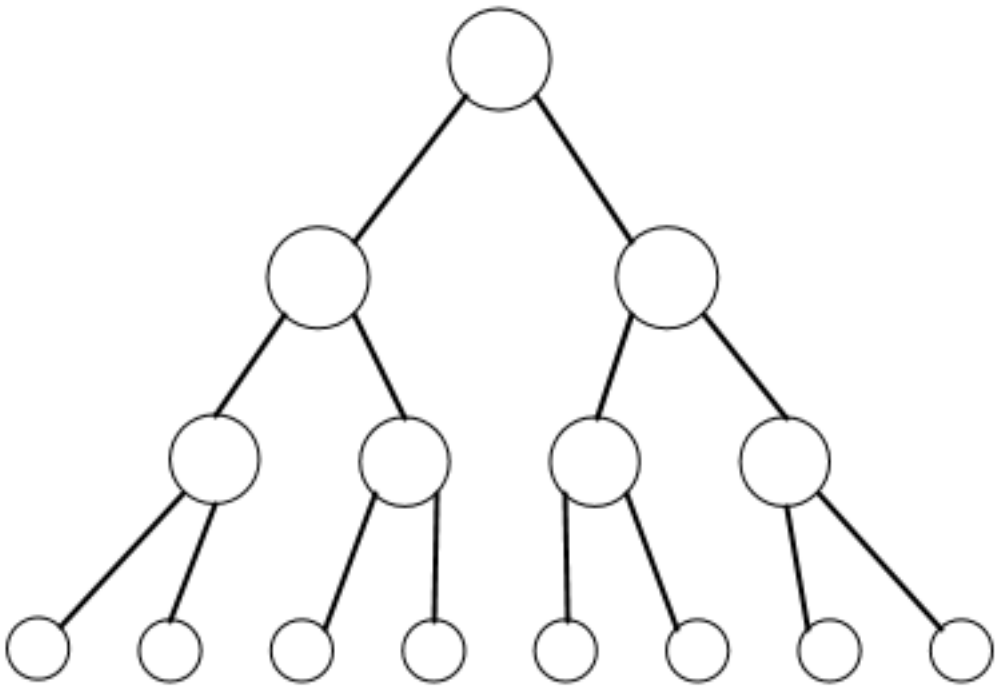
Деревья и кучи

Структуры данных «деревья»

Деревья данных очень полезны во многих случаях. В разработке видеоигр структуры деревьев используются для подразделения пространства, позволяющего разработчику быстро находить находящиеся рядом объекты без необходимости проверки каждого объекта в игровом мире. Даже несмотря на то, что структуры деревьев являются фундаментальными в информатике, на практике в большинстве стандартных библиотек нет непосредственной реализации контейнеров на основе деревьев. Я подробно расскажу о причинах этого.

Простое дерево

Структура данных дерева начинается с корневого узла. Каждый узел может разветвляться на дочерние узлы. Если у узла нет дочерних элементов, то он называется узлом листа. Когда деревьев несколько, это называется лесом. Вот пример дерева. В отличие от настоящих деревьев они растут сверху вниз: корневой узел обычно рисуется сверху, а листья — внизу.



Одним из первых возникает вопрос: сколько каждый узел может иметь дочерних элементов?

Многие деревья имеют не больше двух дочерних узлов. Они называются двоичными деревьями. На примере выше показано двоичное дерево. Обычно дочерние элементы называются левым и правым дочерними узлами. Ещё одним распространённым в играх типом деревьев является дерево с четырьмя дочерними узлами. В дереве квадрантов (quadtree), которое можно использовать для покрытия сетки, дочерние узлы обычно называются по закрываемому ими направлению: NorthWest (северо-запад) или NW, NorthEast (северо-восток) или NE, SouthWest (юго-запад) или SW и SouthEast (юго-восток) или SE.

Деревья используются во многих алгоритмах (я уже упоминал о двоичных деревьях). Существуют сбалансированные и несбалансированные деревья. Бывают красно-чёрные деревья, АВЛ-деревья, и многие другие.

Хотя теория деревьев и удобна, она страдает от серьёзных недостатков: места для хранения и скорости доступа. Каким способом лучше всего хранить дерево? Наиболее простым способом является построение связанного списка, он же оказывается самым худшим. Предположим, что нам нужно построить сбалансированное двоичное дерево. Мы начинаем со следующей структуры данных:

```
// Узел дерева
Node* left;
Node* right;
sometype data;
```

Достаточно просто. Теперь представим, что в нём нужно хранить 1024 элемента. Тогда для 1024 узлов придётся хранить 2048 указателей.

Это нормально, указатели малы и можно обойтись небольшим пространством.

Вы можете помнить, что при каждом размещении объекта он занимает небольшую часть дополнительных ресурсов. Точное количество дополнительных ресурсов зависит от библиотеки используемого вами языка. Многие популярные компиляторы и инструменты могут использовать различные варианты — от всего лишь нескольких байтов для хранения данных до нескольких килобайтов, позволяющих упростить отладку. Я работал с системами, в которых размещение занимает не меньше 4 КБ памяти. В этом случае 1024 элементов потребуют около 4 МБ памяти. Обычно ситуация не настолько плоха, но дополнительные затраты на хранение множества мелких объектов нарастают очень быстро.

Вторая проблема — скорость. Процессорам «нравится», когда объекты находятся в памяти рядом друг с другом. У современных процессоров есть участок очень быстрой памяти — кэш — который очень хорошо справляется с большинством данных. Когда программе требуется один фрагмент данных, кэш загружает этот элемент, а также элементы рядом с ним. Когда данные не загружены в очень быструю память (это называется «промахом кэша»), программа приостанавливает свою работу, и ждёт загрузки данных. В самом очевидном формате, когда каждый элемент дерева хранится в собственном участке памяти, ни один из них не находится рядом с другим. Каждый раз при обходе дерева программа приостанавливается.

Если создание дерева напрямую связано с такими проблемами, то стоит выбрать структуру данных, работающую как дерево, но не обладающую его недостатками. И эта структура называется…

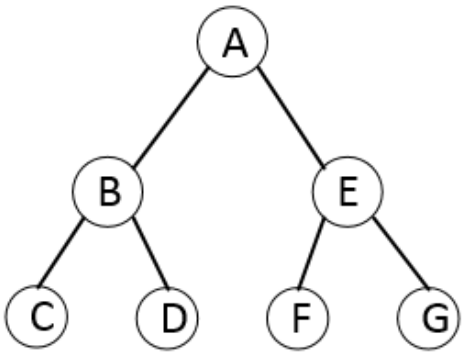
Куча

Структура данных «куча» — это, в сущности, то же самое, что и дерево. У неё есть корневой узел, у каждого узла есть дочерние узлы, и так далее. Куча добавляет ограничения, её сортировка всегда должна выполняться в определённом порядке. Необходима функция сортировки — обычно оператор «меньше чем».

При добавлении или удалении объектов из кучи структура сортирует себя, чтобы стать «полным» деревом, в котором каждый уровень дерева заполнен, за исключением, возможно, только последнего ряда, где всё должно быть смещено в одну сторону. Это позволяет очень эффективно обеспечить пространство для хранения и поиск по куче.

Кучи можно хранить в простом или динамическом массиве, то есть на её размещение тратится мало места. В C++ есть такие функции, как push_heap() и pop_heap(), позволяющие реализовать кучи в собственном контейнере разработчика. В стандартных

библиотеках Java и C# нет похожего функционала. Вот дерево и куча с одинаковой информацией:



Почему их нет в стандартных библиотеках

Это простые, фундаментальные и очень полезные структуры данных. Многие считают, что они должны присутствовать в стандартных библиотеках. За несколько секунд в поисковике вы можете найти тысячи реализаций деревьев.

Оказывается, что хотя деревья очень полезны и фундаментальны, существуют более хорошие контейнеры. Есть более сложные структуры данных, обладающие преимуществами дерева (стабильность и форма) и преимуществами кучи (пространство и скорость). Более совершенные структуры данных обычно являются сочетанием таблиц данных с таблицами поиска. Две таблицы в сочетании обеспечивают быстрый доступ, быстрое изменение, и хорошо проявляются себя и в плотных, и в неплотных ситуациях. Они не требуют переноса элементов при добавлении и удалении элементов, не потребляют излишней памяти и не фрагментируют память при расширенном использовании.

Заключение

Важно знать о структурах данных «дерево», потому что в работе вам часто придётся их использовать. Также важно знать, что эти структуры данных при прямой реализации имеют недостатки. Вы можете реализовывать собственные структуры деревьев, просто знайте, что существуют более компактные типы. Зачем же я рассказал о них, если они на самом деле не используются в стандартных библиотеках? Они применяются в качестве внутренних структур в нашей следующей теме: