Une Revue Critique et Praticien sur: ‘Sorting Algorithms as Special Cases of a Priority Queue Sort’

ÖZEL, F.
UYU, A.
KARAKAYA, B.
KARADAĞ, İ.

Programme de Mathématiques de l’Université de Galatasaray
MAT 232 - Algorithme et Programmation Avancée
4e semestre April 27, 2023

Préface

Cette documentation ; Il contient l'explication et les applications de l'article 'Algorithmes de tri comme cas particuliers d'un tri par file d'attente prioritaire', c'est aussi un journal d'un parcours d'apprentissage individuel. De nombreuses explications sur des concepts et des applications informatiques inconnus sont données.

Parlons de quelques notions qu'il convient de connaître au préalable. Comme le nom du cours l'indique, nous devons savoir ce que signifie algorithme : Un algorithme est une suite finie et non ambiguë d'instructions et d'opérations permettant de résoudre une classe de problèmes. [1] Informallement de plus, un algorithme est une procédure de calcul bien définie qui prend une valeur ou un ensemble de valeurs en entrée et produit une valeur ou un ensemble de valeurs en sortie. Un algorithme est donc une séquence d'étapes de calcul qui transforme l'entrée en sortie. [2]

Nous pouvons parler de bien d'autres concepts et définitions, mais nous ne devons pas aller au-delà de notre sujet, alors parlons brièvement des algorithmes de tri. Un algorithme de tri est, en informatique ou en mathématiques, un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée. Les objets à trier sont des éléments d'un ensemble muni d'un ordre total. Il est par exemple fréquent de trier des entiers selon la relation d'ordre usuelle 'est inférieur ou égal à'. Les algorithmes de tri sont utilisés dans de très nombreuses situations. Ils sont en particulier utiles à de nombreux algorithmes plus complexes dont certains algorithmes de recherche, comme la recherche dichotomique. Ils peuvent également servir pour mettre des données sous forme canonique ou les rendre plus lisibles pour l'utilisateur.

Une priority queue est une structure de données qui permet de stocker des éléments avec des priorités assignées, et d'accéder à l'élément de plus haute priorité en premier. Cette structure de données est souvent utilisée dans les algorithmes de tri et de recherche. Il existe plusieurs implémentations de priority queue telles que les heaps binomiaux, les heaps de Fibonacci et les heaps binaires. [3] [4]

Un D-ary heap est une structure de données qui est similaire à un binary heap, mais au lieu de stocker deux enfants pour chaque nœud, chaque nœud stocke D'enfants. Les D-ary heaps sont couramment utilisés dans les algorithmes de tri et de recherche, et sont souvent plus efficaces que les binary heaps pour des valeurs de D supérieures à 2. [5]

On pense qu'on se souvient de tous les mots-clés nécessaires, on dit qu'on va examiner plus en détail les concepts qui s'y rapportent, et on passe à la documentation, en supposant que vous connaissiez les mathématiques.

[1] La notion de problème peut être vue dans un sens large, il peut s'agir d'une tâche à effectuer, comme trier des objets, assigner des ressources, transmettre des informations, traduire un texte, etc. Il reçoit des données (les entrées), par exemple les objets à trier, la description des ressources à assigner, des besoins à couvrir, un texte à traduire, les informations à transmettre et l'adresse du destinataire, etc., et fournit éventuellement des données (la sortie), par exemple les objets triés, les associations ressource-besoin, un compte-rendu de transmission, la traduction du texte, etc.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). Introduction to algorithms. MIT press.

[3] Knuth, D. E. (1997). The art of computer programming: sorting and searching (Vol. 3). Addison-Wesley Professional.

[4] Williams, J. W. J. (1964). Algorithm 232: Heapsort. Communications of the ACM, 7(6), 347.

[5] Knuth, D. E. (1997). The art of computer programming: sorting and searching (Vol. 3). Addison-Wesley Professional.

Un Trés Important Rappel: Complexité (de Quicksort)

```python
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

Worst-Case

La taille du tableau est prise comme $n-1$ lorsque le pivot est le plus grand ou le plus petit, ou lorsque tous les éléments sont égaux. C'est la pire situation attendue. Si cela se produit, chaque appel récursif invoque un tableau avec un élément manquant dans la liste précédente. En conséquence, nous effectuons n appels imbriqués d'une liste de longueur n vers une liste à un élément. Cela crée une chaîne de $n-1$ boules de long. $O(n-1)$ commence à s'exécuter. Par consequent

$$\sum_{i=0}^{n}(n-i) = O(n^2)$$

Best-Case

Nous obtenons le meilleur des cas si nous pouvons diviser la liste en deux parties égales à chaque fois que nous effectuons l'opération de division. Cela signifie qu'une liste de demi-longueur est traitée à chaque récursivité. Cela signifie que la profondeur de l'arbre des appels est log2 n. Mais deux appels au même niveau de l'arbre d'appels ne traitent pas la même partie de la liste d'origine ; ainsi, chaque niveau d'appels n'a besoin que de temps $O(n)$ tous ensemble (chaque appel a une surcharge constante, mais comme il n'y a que $O(n)$ appels à chaque niveau, cela est inclus dans le facteur $O(n)$). Le résultat est que l'algorithme n'utilise que le temps $O(n\log n)$.

Average-Case

Pour trier un tableau de n éléments distincts, le tri rapide prend $O(n\log n)$ en attente, moyenné sur tous les $n!$ permutations de n éléments avec probabilité égale. Alternativement, si l'algorithme sélectionne le pivot uniformément au hasard à partir du tableau d'entrée, la même analyse peut être utilisée pour limiter le temps d'exécution attendu pour toute séquence d'entrée; l'attente prend alors le pas sur les choix aléatoires effectués par l'algorithme.

Recurrence

Une approche alternative consiste à établir une relation de récurrence pour le facteur $T(n)$, le temps nécessaire pour trier une liste de taille n. Dans le cas le plus déséquilibré, un seul appel de tri rapide implique $O(n)$ travail plus deux appels récursifs sur des listes de taille 0 et $n-1$, donc la relation de récurrence est

$$T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1).$$

C'est la même relation que pour le tri par insertion et le tri par sélection, et elle résout le pire des cas $T(n) = O(n^2)$. Dans le cas le plus équilibré, un seul appel de tri rapide implique $O(n)$ travail plus deux appels récursifs sur des listes de taille $\frac{n}{2}$, donc la relation de récurrence est

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right).$$

Le théorème principal des récurrences diviser pour mieux régner nous dit que T(n) = O(n log n). L'esquisse d'une preuve formelle de la complexité en temps espérée $O(n\log n)$ suit. Supposons qu'il n'y ait pas de doublons, car les doublons pourraient être traités avec un prétraitement et un post-traitement en temps linéaire, ou considérés comme des cas plus faciles que ceux analysés. Lorsque l'entrée est une permutation aléatoire, le rang du pivot est aléatoire uniforme de 0 à $n-1$. Alors les parties résultantes de la partition ont des tailles i et $n-i-1$, et i est aléatoire uniforme de 0 à $n-1$. Ainsi, en faisant la moyenne de toutes les divisions possibles et en notant que le nombre de comparaisons pour la partition est $n-1$, le nombre

moyen de comparaisons sur toutes les permutations de la séquence d'entrée peut être estimé avec précision en résolvant la relation de récurrence :

$$
\begin{aligned}
C(n) &= n-1+\frac{1}{n}\sum_{i=0}^{n-1}(C(i)+C(n-i-1)) = n-1+\frac{2}{n}\sum_{i=0}n-1C(i)\\
nC(n) &= n(n-1)+2\sum_{i=0}^{n-1}C(i)\\
nC(n)-(n-1)C(n-1) &= n(n-1)-(n-1)(n-2)+2C(n-1)\\
nC(n) &= (n+1)C(n-1)+2n-2\\
\frac{C(n)}{n+1} &= \frac{C(n-1)}{n}+\frac{2}{n+1}-\frac{2}{n(n+1)} \leq \frac{C(n-1)}{n}+\frac{2}{n+1}\\
\frac{C(n)}{n+1} &= \frac{C(n-1)}{n}+\frac{2}{n+1}-\frac{2}{n(n+1)} \leq \frac{C(n-1)}{n}+\frac{2}{n+1}\\
&= \frac{C(n-2)}{n-1}+\frac{2}{n}-\frac{2}{(n-1)n}\frac{2}{n+1} \leq \frac{C(n-2)}{n-1}+\frac{2}{n}+\frac{2}{n+1}\\
&\vdots\\
&= \frac{C(1)}{2}+\sum_{i=2}^{n}\frac{2}{i+1} \leq 2\sum_{i=1}^{n-1}\frac{1}{i} \approx 2\int_1^n \frac{1}{x}\mathrm{d}x = 2\ln n
\end{aligned}
$$

On peut donc conclure que le nombre moyen de comparaisons est $C(n) = 2n\ln n \approx 1.39n\log_2 n$

Théorème (Master Theorem)**.** *En informatique, et plus particulièrement en analyse de la complexité des algorithmes, le master theorem ou théorème sur les récurrences de partition permet d'obtenir une solution en termes asymptotiques (en utilisant les notations en O) pour des relations de récurrence d'un certain type rencontrées dans l'analyse de complexité d'algorithmes qui sont régis par le paradigme diviser pour régner.*

$$T(N) = aT(\frac{n}{b}) + f(n) \text{ avec } a \geq 1 \text{ et } b > 1.$$

Oú $f(n)$ est une fonction à valeurs entières positives.:

- *Si $f(n) = O(n^{\log_b a-\varepsilon})$ pour un certain $\varepsilon > 0$, alors $T(n) = O(n^{\log_b a})$.*
- *Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a}\log n)$.*
- *Si $f(n) = \Omega(n^{\log_b a+\varepsilon})$ pour un certain $\varepsilon > 0$, alors $T(n) = \Omega(n^{\log_b a}\log n)$.*

Proof. Si $f(n) = O(n^{\log_b a-\varepsilon})$ pour un certain $\varepsilon > 0$, alors $T(n) = O(n^{\log_b a})$.

$$
\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(\frac{n}{b^j}) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right)\\
&= O\left(n^{\log_b a-\varepsilon}\frac{a^j}{\left(\sum_{j=0}^{\log_b n-1} b^{\log_b a}-\varepsilon\right)^j}\right) = O\left(n^{\log_b a-\varepsilon}\frac{a^j}{\left(a^j\left(b^{-\varepsilon}\right)\right)^j}\right)\sum_{j=0}^{log_b n-1}\\
&= O\left(n^{\log_b a-\varepsilon}\left(b\sum_{j=0}^{\log_b n-1}\right)\right) = O(n^{\log_b a-\varepsilon}(((b^\varepsilon)^{\log_b n})-1)/(b^\varepsilon-1))\\
&= O(n^{\log_b a-\varepsilon}(((b^\varepsilon)^{\log_b n})-1)/(b^\varepsilon-1)) = O(n^{\log_b an^{-\varepsilon}(n^\varepsilon-1)/(b^\varepsilon-1)})\\
&= O\left(n^{\log_b a}\right)
\end{aligned}
$$

Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a}\log n)$

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(\frac{n}{b^j}) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right)$$
$$= \Theta\left(n^{\log_b a} \frac{a^j}{\left(\sum_{j=0}^{\log_b n-1} b^{\log_b a}\right)^j}\right) = \Theta\left(n^{\log_b a} \frac{a^j}{(a^j)^j}\right) \sum_{j=0}^{log_b n-1}$$
$$= \Theta\left(n^{\log_b a} \log n\right)$$

Si $f(n) = \Omega(n^{\log_b a+\varepsilon})$ pour un certain $\varepsilon > 0$, alors $T(n) = \Omega(n^{\log_b a} \log n)$.

- Tandis que $g(n)$ contain $f(n)$, $g(n) = \Omega(f(n))$.
- Tandis que $af(n/b) \leq cf(n)$, $a^j f(\frac{n}{b^j}) \leq c\left(\frac{n}{b^j}\right)^{\log_b a}$.

$g(n) = \sum_{j=0} \log_b n - 1 a^j f(\frac{n}{b^j}) \leq \sum_{j=0} \log_b n - 1 c\left(\frac{n}{b^j}\right)^{\log_b a} = c\sum_{j=0} \log_b n - 1 \left(\frac{n}{b^j}\right)^{\log_b a}$. Par consequent,

$$g(n) = \Omega(n^{\log_b a+\varepsilon}) = \Omega(f(n)).$$

□

Grand O Notation

En informatique, la notation de petit-o est utilisée pour décrire la vitesse de croissance d'une fonction. C'est une mesure de la rapidité à laquelle la fonction croît lorsque l'entrée augmente. Plus formalement, si $f(x)$ est une fonction et $g(x)$ est une fonction, alors on dit que $f(x)$ est $O(g(x))$ si et seulement si il existe une constante positive c telle que la valeur absolue de $f(x)$ soit toujours inférieure ou égale à $c * g(x)$ pour toutes les valeurs suffisamment grandes de x. Par exemple, si $f(x) = x^2$ et $g(x) = x$, alors on peut dire que $f(x)$ est $O(g(x))$. Cela est dû au fait que, pour toutes les valeurs suffisamment grandes de x, la valeur de x^2 est toujours inférieure ou égale à $c * x$ pour une certaine constante c. La notation de petit-o est souvent utilisée dans l'analyse d'algorithmes pour décrire leur complexité temporelle ou spatiale. Par exemple, si un algorithme prend un temps $O(n^2)$ pour s'exécuter, cela signifie que le temps d'exécution de l'algorithme croît à un taux de n^2 lorsque la taille de l'entrée n augmente. Soient $f(x)$ et $g(x)$ des fonctions telles que $f(x)$ est une fonction bornée supérieurement par $g(x)$, c'est-à-dire que pour tous les x suffisamment grands, $f(x) \leq cg(x)$ pour une constante positive c. Dans ce cas, nous pouvons écrire $f(x) = O(g(x))$. Voici un exemple de cette notation en action : Soit $f(x) = x^2$ et $g(x) = x$. Nous voulons vérifier si $f(x) = O(g(x))$. Pour cela, nous devons trouver une constante c telle que, pour tous les x suffisamment grands, $x^2 \leq cx$. Si nous fixons $c = 2$, nous pouvons voir que cette condition est satisfaite pour tous les $x \geq 2$, donc nous pouvons écrire $f(x) = O(g(x))$. Il est important de noter que la notation de petit-o ne décrit pas la fonction $f(x)$ de manière précise, mais plutôt son taux de croissance relativement à la fonction $g(x)$. Par exemple, si $f(x) = x^2$ et $g(x) = x$, alors nous pouvons dire que $f(x)$ croît plus rapidement que $g(x)$ (puisque $f(x) = x^2$ et $g(x) = x$), mais nous ne pouvons pas dire de manière précise combien de fois $f(x)$ est plus grand que $g(x)$ pour une valeur donnée de x.

Petite O Notation

La notation "petit-o" (ou o notation en anglais) est une notation mathématique utilisée pour exprimer la croissance d'une fonction. Elle décrit spécifiquement le comportement asymptotique d'une fonction lorsque la valeur d'entrée tend vers l'infini. En d'autres termes, elle décrit comment la valeur de sortie d'une fonction change par rapport à la valeur d'entrée lorsque celle-ci devient arbitrairement grande. La notation petit-o est généralement écrite comme suit : $f(x) = o(g(x))$ Cette notation est lue comme "$f(x)$ est petit-o de $g(x)$". Cela signifie que la fonction $f(x)$ croît plus lentement que $g(x)$ lorsque x tend vers l'infini. En termes mathématiques plus précis, cela signifie que la limite de $\frac{f(x)}{g(x)}$ lorsque x tend vers l'infini est 0. La notation petit-o est généralement utilisée pour comparer les taux de croissance de différentes fonctions, et peut être utile pour analyser les algorithmes et prédire la complexité temporelle d'un algorithme donné. Par exemple, si un algorithme a une complexité temporelle de $O(n^2)$, cela signifie que le temps d'exécution de l'algorithme augmente comme le carré de la taille d'entrée. Si un autre algorithme a une complexité temporelle de $O(nlog(n))$, cela signifie que le temps d'exécution de l'algorithme augmente comme la taille d'entrée multipliée par le logarithme de la taille d'entrée. Il est important de noter que la notation O

fournit une borne supérieure pour la fonction, elle nous indique que la fonction ne va jamais croitre plus vite que la fonction représentée par la notation O. Pendant ce temps notation petit-o est une borne plus serrée que la notation O, cela nous indique que la fonction va croitre arbitrairement proche de la fonction représentée par la notation petit-o mais ne dexédera jamais cette fonction.

Petite Omega Notation

La notation "petite omega" (ou ω notation en anglais) est une notation mathématique utilisée pour exprimer la croissance d'une fonction. Elle est similaire à la notation "petit-o", mais elle décrit le comportement asymptotique inférieur d'une fonction. En d'autres termes, elle décrit comment la valeur de sortie d'une fonction change par rapport à la valeur d'entrée lorsque celle-ci devient arbitrairement petite. La notation petite omega est généralement écrite comme suit : $f(x) = \omega(g(x))$ Cette notation est lue comme "$f(x)$ est petite omega de $g(x)$". Cela signifie que la fonction $f(x)$ croît plus vite que $g(x)$ lorsque x tend vers 0. En termes mathématiques plus précis, cela signifie que la limite de $\frac{g(x)}{f(x)}$ lorsque x tend vers 0 est 0. La notation petite omega est utilisé pour montrer que l'algorithme croit plus vite qu'une fonction donnée, et est souvent utilisé en conjonction avec la notation O pour donner un intervalle sur lequel la fonction peut croitre. Il est important de noter que la notation petite omega est une borne inférieure pour la fonction, il nous dit que la fonction ne croitra jamais plus lentement que la fonction représentée par la notation omega.

Petite Theta Notation

La notation "petite theta" (ou θ notation en anglais) est une notation mathématique utilisée pour exprimer la croissance d'une fonction. Elle est similaire à la notation "petit-o", mais elle décrit le comportement asymptotique supérieur et inférieur d'une fonction. En d'autres termes, elle décrit comment la valeur de sortie d'une fonction change par rapport à la valeur d'entrée lorsque celle-ci devient arbitrairement grande ou petite. La notation petite theta est généralement écrite comme suit : $f(x) = \theta(g(x))$ Cette notation est lue comme "$f(x)$ est petite theta de $g(x)$". Cela signifie que la fonction $f(x)$ croît à la même vitesse que $g(x)$ lorsque x tend vers l'infini ou 0. En termes mathématiques plus précis, cela signifie que la limite de $\frac{f(x)}{g(x)}$ lorsque x tend vers l'infini est 1 et la limite de $\frac{g(x)}{f(x)}$ lorsque x tend vers 0 est 1. La notation petite theta est utilisé pour montrer que l'algorithme croit à la même vitesse qu'une fonction donnée, et est souvent utilisé en conjonction avec la notation O pour donner un intervalle sur lequel la fonction peut croitre. Il est important de noter que la notation petite theta est une borne inférieure et supérieure pour la fonction, il nous dit que la fonction ne croitra jamais plus lentement ou plus vite que la fonction représentée par la notation theta.

Grand Omega Notation

La notation "grand omega" (ou Ω notation en anglais) est une notation mathématique utilisée pour exprimer la croissance d'une fonction. Elle est similaire à la notation "grand-o", mais elle décrit le comportement asymptotique supérieur d'une fonction. En d'autres termes, elle décrit comment la valeur de sortie d'une fonction change par rapport à la valeur d'entrée lorsque celle-ci devient arbitrairement grande. La notation grand omega est généralement écrite comme suit : $f(x) = \Omega(g(x))$ Cette notation est lue comme "$f(x)$ est grand omega de $g(x)$". Cela signifie que la fonction $f(x)$ croît plus lentement que $g(x)$ lorsque x tend vers l'infini. En termes mathématiques plus précis, cela signifie que la limite de $\frac{f(x)}{g(x)}$ lorsque x tend vers l'infini est 0. La notation grand omega est utilisé pour montrer que l'algorithme croit plus lentement qu'une fonction donnée, et est souvent utilisé en conjonction avec la notation O pour donner un intervalle sur lequel la fonction peut croitre. Il est important de noter que la notation grand omega est une borne supérieure pour la fonction, il nous dit que la fonction ne croitra jamais plus vite que la fonction représentée par la notation omega.

Contents

1 Structures des Données

Il est utile de rafraîchir nos connaissances en mentionnant quelques types et structures de données. Ici, nous n'inclurons des informations qu'à des fins de rappel. Mais nous devons expliquer en détail les concepts contenus dans 'X'.

Une structure de données est un stockage utilisé pour stocker et organiser des données. C'est une façon d'organiser les données sur un ordinateur afin qu'elles puissent être consultées et mises à jour efficacement.

Une structure de données n'est pas seulement utilisée pour organiser les données. Il est également utilisé pour le traitement, la récupération et le stockage des données. Il existe différents types de structures de données de base et avancées qui sont utilisées dans presque tous les programmes ou systèmes logiciels qui ont été développés. Nous devons donc avoir une bonne connaissance des structures de données.

1.1 Array

Un tableau est une collection d'éléments stockés à des emplacements de mémoire contigus. L'idée est de stocker plusieurs éléments du même type ensemble. Cela facilite le calcul de la position de chaque élément en ajoutant simplement un décalage à une valeur de base, c'est-à-dire l'emplacement mémoire du premier élément du tableau (généralement désigné par le nom du tableau).

1.2 Liste Chaînée

Une liste chaînée est une structure de données linéaire qui est composée de plusieurs nœuds. Chaque nœud contient deux champs, un champ de données et un champ de pointeur. Le champ de données contient des informations sur le nœud et le champ de pointeur contient l'adresse du nœud suivant. Le dernier nœud de la liste chaînée contient un pointeur NULL pour indiquer la fin de la liste.

Dans une liste chaînée, chaque élément est appelé un nœud ou un élément de la liste. Chaque nœud contient deux parties principales : les données et un pointeur vers le nœud suivant. Le pointeur est une référence à l'emplacement de la mémoire où se trouve le nœud suivant de la liste. La dernière valeur de la liste pointe généralement vers une valeur nulle pour indiquer la fin de la liste.

La raison pour laquelle les listes chaînées sont utilisées est qu'elles sont très efficaces pour les opérations d'insertion et de suppression. Dans une liste chaînée, l'insertion ou la suppression d'un élément ne nécessite pas de déplacer les éléments suivants de la liste. Au lieu de cela, il suffit de mettre à jour les pointeurs des nœuds adjacents pour que la liste reste cohérente. Cette opération est très rapide, même pour les grandes listes.

Les opérations principales sur une liste chaînée sont l'insertion, la suppression et la recherche. L'insertion consiste à ajouter un nouvel élément à la liste. Pour insérer un nouvel élément, il suffit de créer un nouveau nœud, de le connecter au nœud suivant de la liste et de mettre à jour les pointeurs des nœuds adjacents. La suppression consiste à retirer un élément de la liste. Pour supprimer un élément, il suffit de mettre à jour les pointeurs des nœuds adjacents pour que le nœud à supprimer soit omis de la liste. Enfin, la recherche consiste à trouver un élément spécifique dans la liste. Pour rechercher un élément, il faut parcourir la liste en suivant les pointeurs de chaque nœud jusqu'à ce que l'élément soit trouvé.

Les listes chaînées sont largement utilisées dans de nombreux domaines de la programmation, notamment pour la mise en œuvre de structures de données plus complexes telles que les piles, les files d'attente et les arbres binaires. Elles sont également utilisées dans les applications de traitement du langage naturel, les bases de données et les systèmes de gestion de fichiers.

En conclusion, une liste chaînée est une structure de données dynamique qui peut être utilisée pour stocker une collection d'éléments. Elle est efficace pour les opérations d'insertion et de suppression et peut être utilisée pour implémenter des structures de données plus complexes. Les opérations principales sur une liste chaînée sont l'insertion, la suppression et la recherche. Les pointeurs des nœuds de la liste sont utilisés pour connecter les nœuds et maintenir la cohérence de la liste.

List Chaînée

```python
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

        def __init__(self):
            self.head = None

        def printList(self):
            temp = self.head
            while (temp):
                print(temp.data)
                temp = temp.next
```

L'Operation des Listes Chaînées

```python
class LinkedList:

            def __init__(self):
                self.head = None

            def printList(self):
                temp = self.head
                while (temp):
                    print(temp.data)
                    temp = temp.next

            def push(self, new_data):
                new_node = Node(new_data)
                new_node.next = self.head
                self.head = new_node

            def insertAfter(self, prev_node, new_data):
                if prev_node is None:
                    print("The given previous node must inLinkedList.")
                    return
                new_node = Node(new_data)
                new_node.next = prev_node.next
                prev_node.next = new_node

            def append(self, new_data):
                new_node = Node(new_data)
                if self.head is None:
                    self.head = new_node
                    return
                last = self.head
                while (last.next):
                    last = last.next
                last.next = new_node
```

1.3 Queue

En informatique, une file d'attente (en anglais, "queue") est une structure de données linéaire qui suit une stratégie dite FIFO (First In First Out), ce qui signifie que le premier élément ajouté à la file d'attente est le premier élément à être retiré. Les éléments sont ajoutés à la fin de la file d'attente et retirés du début. Cette structure est souvent utilisée dans les algorithmes de traitement de données où les éléments doivent être traités dans l'ordre où ils sont reçus.

La file d'attente est généralement implémentée à l'aide d'une liste chaînée, bien qu'elle puisse également être implémentée avec un tableau circulaire. Chaque élément dans la file d'attente est représenté par un nœud contenant des données et un pointeur vers le nœud suivant. La file d'attente est également caractérisée par deux pointeurs, un pointeur "front" qui pointe sur le premier élément de la file d'attente et un pointeur "rear" qui pointe sur le dernier élément.

La file d'attente est largement utilisée dans de nombreuses applications, notamment pour gérer les tâches à effectuer, les demandes de services, les événements, etc. Les algorithmes de routage de paquets dans les réseaux informatiques sont également basés sur des files d'attente. La file d'attente est également utilisée dans la simulation informatique, où elle peut être utilisée pour simuler des événements en attente de traitement. [6]

Queue

```python
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)

    def printQueue(self):
        print(self.items)
```

Priority Queue (Case Special)

Une file d'attente prioritaire est un type de file d'attente qui organise les éléments en fonction de leurs valeurs de priorité. Les éléments avec des valeurs de priorité plus élevées sont généralement récupérés avant les éléments avec des valeurs de priorité inférieures.

Dans une file d'attente prioritaire, chaque élément a une valeur de priorité qui lui est associée. Lorsque vous ajoutez un élément à la file d'attente, il est inséré dans une position basée sur sa valeur de priorité. Par exemple, si vous ajoutez un élément avec une valeur de priorité élevée à une file d'attente prioritaire, il peut être inséré près du début de la file d'attente, tandis qu'un élément avec une valeur de priorité faible peut être inséré près du fond.

Il existe plusieurs façons d'implémenter une file d'attente prioritaire, y compris l'utilisation d'un tableau, d'une liste chaînée, d'un tas ou d'un arbre de recherche binaire. Chaque méthode a ses propres avantages et inconvénients, et le meilleur choix dépendra des besoins spécifiques de votre application.

Les files d'attente prioritaires sont souvent utilisées dans les systèmes en temps réel, où l'ordre dans lequel les éléments sont traités peut avoir des conséquences importantes. Ils sont également utilisés dans des algorithmes pour améliorer leur efficacité, tels que l'algorithme de Dijkstra pour trouver le chemin le plus court dans un graphe et l'algorithme de recherche A * pour la recherche de chemin.

[6] Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles, Fifth Edition by Narasimha Karumanchi, 2015, pp. 100-101

Priority Queue

```python
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    def isEmpty(self):
        return len(self.queue) == 0

    def insert(self, data):
        self.queue.append(data)

    def delete(self):
        try:
            max = 0
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[max]:
                    max = i
            item = self.queue[max]
            del self.queue[max]
            return item
        except IndexError:
            print()
            exit()
```

1.4 Arbres

1.4.1 Arbres Binaires

Un arbre binaire est une structure de données hiérarchique qui est composée d'un ensemble de nœuds connectés par des liens. Chaque nœud contient une valeur et au plus deux liens appelés fils gauche et fils droit. Les fils gauche et droit sont des sous-arbres. Un arbre binaire est un arbre particulier où chaque nœud a au plus deux fils. Les arbres binaires sont largement utilisés dans de nombreuses applications, notamment pour implémenter des structures de données plus complexes telles que les arbres binaires de recherche, les arbres binaires équilibrés, les arbres binaires de fusion, etc. Les arbres binaires sont également utilisés dans les algorithmes de recherche et de tri, ainsi que dans les algorithmes de compression de données.

Arbres Binaires

```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

class BinaryTree:
    def __init__(self):
        self.root = None

    def printTree(self, root):
        if root:
            self.printTree(root.left)
            print(root.val)
            self.printTree(root.right)
```

1.4.2 Arbres Binaires de Recherche

Un arbre binaire de recherche (abr) est une structure de données hiérarchique qui est composée d'un ensemble de nœuds connectés par des liens. Chaque nœud contient une valeur et au plus deux liens appelés fils gauche et fils droit. Les fils gauche et droit sont des sous-arbres. Un arbre binaire de recherche est un arbre particulier où chaque nœud a au plus deux fils. Les arbres binaires de recherche sont largement utilisés dans de nombreuses applications, notamment pour implémenter des structures de données plus complexes telles que les arbres binaires de recherche, les arbres binaires équilibrés, les arbres binaires de fusion, etc. Les arbres binaires de recherche sont également utilisés dans les algorithmes de recherche et de tri, ainsi que dans les algorithmes de compression de données.

Arbres Binaires de Recherche

```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, root, key):
        if root is None:
            return Node(key)
        else:
            if root.val == key:
                return root
            elif root.val < key:
                root.right = self.insert(root.right, key)
            else:
                root.left = self.insert(root.left, key)
        return root

    def printTree(self, root):
        if root:
            self.printTree(root.left)
            print(root.val)
            self.printTree(root.right)
```

1.5 Heap

En informatique, un tas (ou heap en anglais) est une structure de données spécialisée basée sur un arbre qui satisfait la propriété de tas, définie comme suit :

Pour un tas maximum (ou max-heap), la valeur de chaque nœud est supérieure ou égale aux valeurs de ses enfants. Pour un tas minimum (ou min-heap), la valeur de chaque nœud est inférieure ou égale aux valeurs de ses enfants.

L'application la plus courante des tas est dans la mise en œuvre de files de priorité, où la propriété de tas est utilisée pour garantir que l'élément ayant la plus haute (ou la plus basse) priorité se trouve toujours à la racine du tas, ce qui facilite sa récupération et sa suppression.

Il existe deux types principaux de tas : les tas binaires et les tas de Fibonacci. Les tas binaires sont implémentés sous forme d'arbres binaires, la propriété de tas étant maintenue en comparant les nœuds avec leurs enfants. Les tas de Fibonacci sont plus complexes, mais peuvent effectuer certaines opérations plus rapidement que les tas binaires, telles que la diminution de la valeur d'un nœud.

En termes de complexité temporelle, les tas binaires ont une complexité de temps de $O\log(n)$ pour l'insertion et la suppression d'éléments, tandis que les tas de Fibonacci ont une complexité de temps amortie de $O(1)$ pour certaines opérations.

Les tas sont largement utilisés en informatique, en particulier dans les algorithmes liés aux graphes, tels que l'algorithme de Dijkstra pour trouver les chemins les plus courts et l'algorithme de Prim pour trouver les arbres couvrants minimums.

On peut parler de 4 tas différents, mais nous traiterons du tas D-aire.

1.5.1 D-ary Heap

Un tas d-aire (ou d-ary heap en anglais) est un type de structure de données en tas où chaque nœud a au plus d enfants. En d'autres termes, c'est une généralisation du tas binaire, qui est un tas d-aire avec $d = 2$.

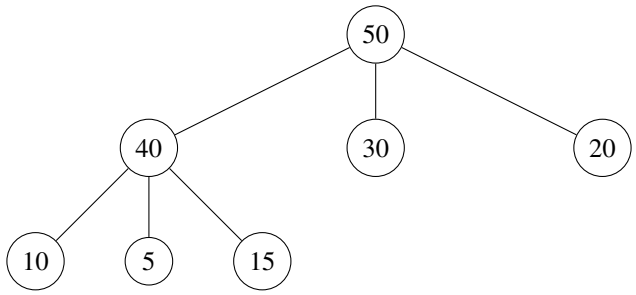
Dans un tas d-aire, les valeurs du nœud parent sont supérieures ou égales aux valeurs de ses enfants dans le cas d'un tas maximum (ou max-heap), ou inférieures ou égales aux valeurs de ses enfants dans le cas d'un tas minimum (ou min-heap). La propriété de tas garantit que le nœud racine contient la valeur maximale ou minimale du tas.

Les opérations d'insertion et de suppression d'éléments dans un tas d-aire sont similaires à celles dans un tas binaire. La principale différence est que dans un tas d-aire, chaque nœud a jusqu'à d enfants au lieu de deux.

La complexité temporelle de l'insertion d'un élément dans un tas d-aire est de $O(d\log_d n)$, où n est le nombre d'éléments dans le tas. La complexité temporelle de la suppression de l'élément maximal (ou minimal) dans un tas d-aire est de $O(d\log_d n)$.

Le choix de d dans un tas d-aire dépend de l'application spécifique et du compromis entre la complexité spatiale et temporelle. En général, des valeurs plus grandes de d peuvent réduire la hauteur du tas et améliorer la complexité temporelle de certaines opérations, mais peuvent également augmenter la complexité spatiale.

Les tas d-aire sont utilisés dans une variété d'algorithmes et d'applications, notamment le routage de réseau, les systèmes d'exploitation et les systèmes de base de données.



D-Ary Heap

```python
class DaryHeap:
    def __init__(self, d):
        self.d = d
        self.heap = []

    def parent(self, i):
        return (i - 1) // self.d

    def child(self, i, j):
        return self.d * i + j + 1

    def insert(self, key):
        self.heap.append(key)
        i = len(self.heap) - 1
        while i != 0 and self.heap[i] > self.heap[self.parent(i)]:
            self.heap[i], self.heap[self.parent(i)] = self.heap[self.parent(i)], self.heap[i]
            i = self.parent(i)

    def delete(self):
        if len(self.heap) == 0:
            raise IndexError("Heap is empty")
        max_val = self.heap[0]
        last_val = self.heap.pop()
        if len(self.heap) > 0:
            self.heap[0] = last_val
            self.heapify(0)
        return max_val

    def heapify(self, i):
        largest = i
        for j in range(self.d):
            c = self.child(i, j)
            if c < len(self.heap) and self.heap[c] > self.heap[largest]:
                largest = c
        if largest != i:
            self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]
            self.heapify(largest)
```

2 Sorting Algorithms as Special Cases of a Priority Queue Sort

Dans ”Sorting Algorithms as Special Cases of a Priority Queue Sort”, Tim Bell introduit le concept de d-heap, qui est une généralisation de la structure de données du tas binaire. Un tas d est une structure de données arborescente où chaque nœud a jusqu'à d enfants, et la priorité de chaque nœud est supérieure ou égale à la priorité de ses enfants.

Bell soutient que de nombreux algorithmes de tri peuvent être considérés comme des cas particuliers d'un tri de file d'attente prioritaire utilisant un d-heap. Plus précisément, il montre comment utiliser un d-heap pour implémenter le tri en tas, le tri en douceur et le tri par patience. Il présente également un nouvel algorithme de tri appelé d-heapsort, qui est une généralisation du tri en tas qui utilise un d-heap au lieu d'un tas binaire.

L'article comprend également une analyse détaillée des caractéristiques de performance des algorithmes de tri basés sur d-heap, y compris la complexité temporelle et spatiale. Bell montre que le tri en tas a la même complexité temporelle que le tri en tas ($O(n \log n)$ dans les cas moyens et les pires), mais peut avoir une meilleure complexité spatiale pour certaines valeurs de d .

Dans l'ensemble, ”Sorting Algorithms as Special Cases of a Priority Queue Sort” est une ressource utile pour ceux qui s'intéressent à la théorie et à la pratique des algorithmes de tri, et offre une nouvelle perspective sur la relation entre les files d'attente prioritaires et les algorithmes de tri.[7]

[7] AI generated.

Sorting Algorithms as Special Cases of a Priority Queue Sort

Tim Bell
University of Canterbury
Christchurch, New Zealand
+64 3 364-2987
tim.bell@canterbury.ac.nz

Bengt Aspvall
Blekinge Institute of Technology
Karlskrona, Sweden
+46 455-385003
bengt.aspvall@bth.se

ABSTRACT
This paper offers an exercise for revisiting the main sorting algorithms after they have been taught to students. This is done in a way that emphasizes the relationships between them, and shows how considering abstraction and extreme cases can lead to the generation of new algorithms. A number of authors (including textbook authors) have noted particular relationships between algorithms, such as an uneven split in merge sort being equivalent to insertion sort. In this paper we use a flexible priority queue, the d-heap, to derive three common sorting algorithms. We combine this with using a BST as a priority queue, plus prior observations in the literature, to show strong relationships between the main sorting algorithms that appear in textbooks. In the process students are able to revisit a number of algorithms and data structures and explore elegant relationships between them. This approach can also lead to exercises and exam questions that go beyond desk-checking to evaluate students' understanding of these algorithms.

Categories and Subject Descriptors
K.3.2 [**Computer and Education**]: Computer and Information Science Education – *computer science education.*

General Terms
Algorithms, Performance.

Keywords
Sorting algorithms, priority queue, d-heap.

1. INTRODUCTION
Sorting algorithms are a staple of algorithms courses, providing an excellent range of opportunities to illustrate analysis, best/average/worst case considerations, complexity bounds, and design techniques. Teaching each algorithm in isolation can lead to students being faced with a confusing catalogue of names and techniques to remember, and it can be helpful to use relationships between algorithms to help students distinguish them, and also explore which properties they have in common.

Textbooks often categorize sorting algorithms as either quadratic or $n\log n$ (for example, [1], [12]), and sometimes other features are used to relate algorithms, such as Merrit and Lau's "easy split" vs. "easy join" for mergesort and quicksort [11], used by Wood [16].

In this paper we introduce a way to relate sorting methods based on priority queue sorting, which can be combined with previously noted links to enable us to derive all the common introductory sorting algorithms from each other primarily by substituting related data structures. A priority queue (PQ) is an abstract data type that can return items in "priority" order based on their key value, so a PQ-based sorting method simply needs to insert all values in a PQ, and then have them returned in increasing (or decreasing) order. We will use a variety of priority queue structures to generate a number of algorithms; the structures include sorted and unsorted lists, heaps, the d-heap (a heap with a branch factor of d), and the binary search tree (which can be used as a priority queue). This data structure-driven approach uncovers all sorts of interesting relationships and shows how algorithms can be derived and not just plucked out of thin air. It also gives opportunities to revisit each algorithm rather than just teach it once, and it relates the analysis of various algorithms to each other, which reinforces the connections between different approaches.

The approach that we propose has been used successfully with algorithms classes, and is a joy to teach because each time a number of students are inspired by the beauty of these relationships. Only simple mathematical concepts are needed to understand the links, which makes it accessible to junior students. Because the ideas are relatively simple but lead to students uncovering powerful concepts for themselves, the approach is particularly suited to a constructivist approach to teaching — students can discover these elegant relationships for themselves with only a little guidance.

Being data-structure driven, it also illustrates the close relationship between data structures and algorithms, and it reinforces the value of abstraction, particularly using the priority queue abstract data type, rather than focusing on a particular implementation.

Several taxonomies have been used for sorting algorithms:

- Knuth's classic "bottom-up" approach, where the three categories are insertion (leading to Shellsort), exchange

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

(leading to quicksort) and selection (leading to heapsort), with mergesort as another case [8] (a similar taxonomy is used by Dromey [5], who derives algorithms by creating invariants from the specification of the output);

- Merritt's "inverted taxonomy" [10], where algorithms are divided into "split by value" (quick, selection and bubble sort) and "split by position" (merge and insertion sort); and
- Merritt and Lau's "logical inverted taxonomy" [11] which introduces "split by partial value" to include radix and distribution sort.

Each of these taxonomies partitions the algorithms into distinct families; this paper shows close relationships between the families in addition to the relationships noted by previous authors within families.

The main sorting algorithms that we consider here are selection sort, insertion sort, quicksort, merge sort and heap sort. Most other sorting algorithms have obvious relationships to these; for example, Shellsort is an optimization of insertion sort [11], as is binary insertion sort. Bubblesort can be regarded as combining the worst features of selection sort and insertion sort, and is perhaps best not mentioned anyway [2]! We will assume the common introductory form of quicksort, where the first (or last) item is used as the pivot; this is sufficient to characterize the algorithm, and keeps descriptions simple, although in practice improvements such as median-of-three would be used. For brevity we haven't included these obviously related methods in our discussion below, although we will introduce some other algorithms that are equivalent to the main five; while these other algorithms aren't of practical importance, they provide an important link between the well-known sorting algorithms. Also, we focus only on comparison-based sorting methods; Merrit and Lau [11] link key-based sorting (radix and distribution sorts) to comparison-based methods, and these connections can be used if key-based algorithms are included in the curriculum.

The five chosen algorithms are commonly presented in introductory computer science texts and courses in a reasonably independent manner, with the occasional relationships between pairs of algorithms noted, such as the worst case of quicksort being equivalent to selection sort [16], and quicksort's relationship with tree insertion sort [8]. This paper provides a more systematic view of the relationships between sorting algorithms, largely based on deriving them from a priority queue (PQ) sort. Rather than the hierarchical views described above, we combine the PQ relationships with those previously identified by Merritt and Lau [11] to show close relationships between all the commonly taught sorting algorithms. We advocate this not as a top-down teaching tool, but as a retrospective review of sorting algorithms that places them all in perspective, relates them to data structures, introduces ideas about algorithm design, and encourages students to think about extreme cases.

The main tool we use to link sorting algorithms is the d-heap, which is a heap priority queue with a branch factor of d. (The name "d-heap" appears to be the most common in current use, although the structure was originally referred to as a beta-heap [7] and has other names such as d-ary heap [3].) The d-heap is explained in section 2, and in section 3 we explore its use in PQ-based sorting algorithms. Section 4 extends this view of sorting to expose many relationships between what might appear to be quite distinct algorithms. We conclude in Section 5 with a summary of all these relationships.

Before continuing, we will note that there is a duality between min- and max-based algorithms, and a similar duality between min- and max- priority queues, and ascending and descending sorted lists. For example, selection sort is often presented as minimum-selection (for example, [1]), but sometimes as maximum-selection (for example, [9]), or both [16]. Heap sort is almost always presented using a max-heap (i.e. based on the delete-max operation) as this leads to a natural in-place sort, although Weiss [14] introduces it using a min-heap, and later points out that a max-heap can be used to avoid reverse-sorting the list for in-place sorting. An in-place min-heap sort is possible, but it is not particularly elegant! In this paper we regard the min- and the max- version of an algorithm as essentially the same since any algorithms that are based on a max-first approach can be converted to a min-first one, and vice versa, without changing the complexity or fundamental concept of the algorithm. Many texts use minimum selection sort but maximum heap sort. Teaching maximum selection sort leads more naturally to heap sort, although authors may have reasons for using minimum selection sort, such as relating it more closely to insertion sort. Because the analysis for min- and max- sorting are the same, we will use whichever is most convenient for our explanations at each stage.

2. THE d-HEAP

A heap [15] is a tree structure where (in the case of a max-heap) the value of every child node is not greater than that of its parent. It is commonly used as a Priority Queue (PQ) because the maximum value is easy to find (at the root), and updates on a heap are easily done in $O(\log n)$ time. The heap data structure is usually presented as a *binary* heap, which is a complete tree with a two-way branch factor (e.g. [1, 8, 12, 16]), such as the max-heap shown in Figure 1(a). By using a *complete* tree, a binary heap it is easily mapped onto an array, and can be navigated by multiplying and dividing the array index by 2, rather than using any explicit child/parent links. Some textbooks index the array from 0, in which case the children of node i are at $2i+1$ and $2i+2$; others index it from 1, in which case the children are at $2i$ and $2i+1$. The array storing the heap can be viewed as a partially ordered list, and its $O(\log n)$ time for both insert and delete-max is an excellent compromise between a PQ using a sorted array ($O(n)$ insert time) and an unsorted array ($O(n)$ delete-max time). In this paper we will refer to both max-heaps (as above), and min-heaps (where the ordering is reversed and the minimum value is easy to find).

A d-heap is a heap with a branch factor of d [7]. These are generally used as a priority queue for graph algorithms, but in this paper we show that they provide a useful teaching tool for enabling students to see the relationship between diverse sorting algorithms.

We would normally use a constructivist approach with students to introduce d-heaps, and will outline this approach to introduce the concept here. First, have the students consider what a 3-way (ternary) heap would be like (one is shown in Figure 1(b)). A good exercise is to work out where the children of a node are (multiply the index by 3 instead of 2) and its parents (divide by 3 instead of 2). Getting the exact formula is a good exercise to test their understanding of heaps. Then the students can analyze a delete-max operation on the 3-heap; again, it is a good exercise to

realize that the depth is now $\log_3 n$, but there are 3 comparisons at each level, giving a delete-max time of $3\log_3 n \approx 1.89\log_2 n$ (which is slightly better than the $2\log_2 n$ required for a binary heap).

This raises the question of whether a 4-heap would be even better; a quick analysis shows that $4\log_4 n = 2\log_2 n$ comparisons are required for a delete-max, so it is the same as a 2-heap, and it would appear that 3-heaps are optimal. (In fact, it is sometimes noted that 2- or 4-heaps might be faster than a 3-heap because the multiplications are just shift operations. Interestingly, the optimal value for the branch factor is $e \approx 2.71828$, which raises other interesting questions! Calculating the minimum of $d\log_d n$ is a simple and instructive math exercise.)

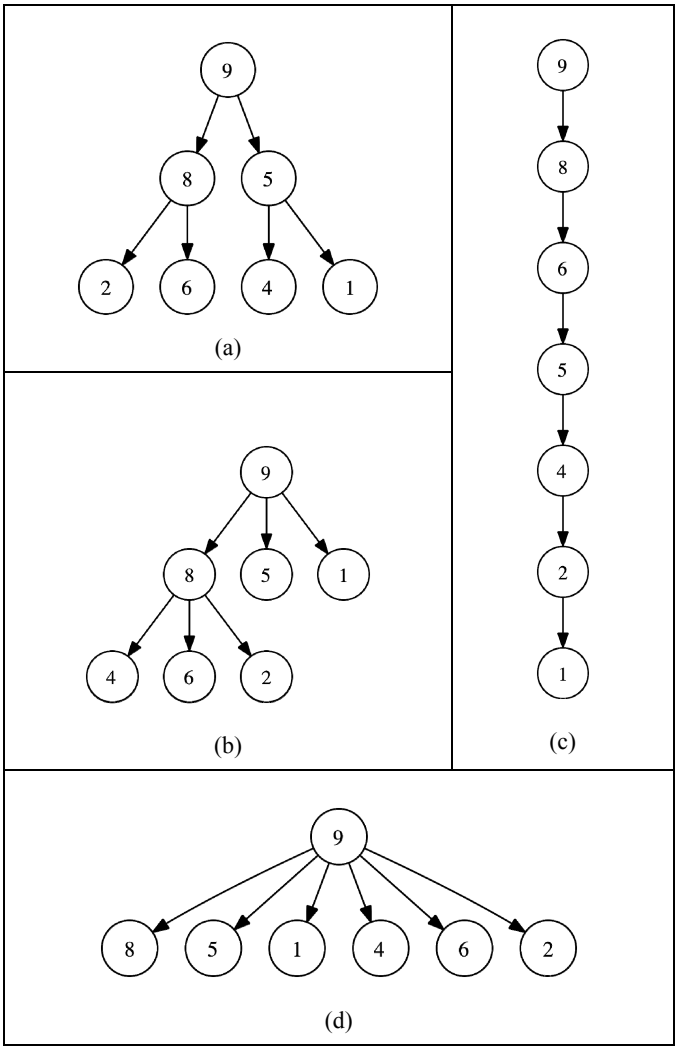


Figure 1: Heaps with branch factors of (a) d=2 (b) d=3 (c) d=1 (d) d=n

Because the value d can be tuned to change the performance of the heap, a d-heap with a higher value of d is useful for algorithms where there are significantly more insertions than delete-max operations (since insert takes $\log_d n$ time, and becomes faster as d increases). Tuning the value of d can be used where the expected ratio of inserts to deletes can be predicted based on the input data, such as in applications on dense vs. sparse graphs.

The next step for students is to get them to consider extreme values of d (exploring extreme cases is a good habit for computer scientists to have!) The lowest value usually considered for d is 2, and this yields a standard binary heap where both insert and delete-max operations take $O(\log n)$ time. However, considering d=1 leads to a structure that is essentially a sorted linked list (Figure 1(c)), with $O(n)$ average time for insert, but $O(1)$ time for delete-max. If d=n (more precisely, $d \geq n-1$), the structure is an unsorted list where the maximum has been pre-selected (Figure 1(d)), giving $O(1)$ time for insertion and $O(n)$ time for delete-max. Thus the d-heap is essentially a flexible data structure that can be tuned from an un-ordered list to an ordered list, with various types of partially-ordered heaps in between.

Getting computer science students to consider the d-heap and extreme cases such as d=1 and d=n is a good reminder of the value of looking for general cases and extreme cases. Of course these extreme values are primarily a thought experiment, and are not intended as a practical implementation. We note that Java has a "DHeap" class that provides this functionality, but requires that $d \geq 2$.

3. PQ BASED SORTING WITH A d-HEAP

The idea of using a priority queue (PQ) to introduce sorting algorithms is not new; for example, Knuth uses this approach, describing heap sort as a more efficient way to do selection sort [8]. Thorup [13] explores the relationship between priority queues and sorting, but focuses on general complexity bounds rather than using them for teaching the actual algorithms.

The idea of a PQ-based sort can be illustrated to students as follows. Suppose we are handed a sequence of n (comparable) items one at a time. The items should later be output in sorted (non-decreasing) order. How the sorting should be performed is not specified at this point; we can imagine that it is done in a "black box", which is effectively implementing a priority queue. Each data item is put into the black box, and then each is removed from it in ascending order using the delete-min operation repeatedly. We now investigate how the choice of data structure for the black box will lead to various well-known sorting algorithms.

The black box needs some way of organizing the items as they are handed to it. It could, of course, just record each new item, deferring all work until later. Alternatively, it could do some processing when handed a new item, using one of the following example approaches:

1. "put high effort into maintaining order": compare the new item to all previously received items,
2. "medium effort": compare it to some of the previously received items, or
3. "be lazy, postponing work": compare it to only one of the previously received items.

The d-heap is able to implement all of these possibilities. The three scenarios above correspond to d=1, 2, and n–1, respectively. Thus we may claim that the three approaches are all special cases of d-heap based sorting algorithms, and in turn, PQ based sorting.

In the first scenario, each new item is inserted into its correct place in the partial list obtained by appending it to the end of the 1-heap (linked list), and then sifting it up to its correct position. After inserting all items we will have a sorted list that is trivial to

output. The 1-heap PQ sort shares the best and worst cases of insertion sort: for a min-heap, if a new item is the largest so far then it is compared only with the bottom (last) item in the 1-heap; if it is the smallest so far then it will be compared with all the previous values. This gives a best case sorting complexity of $O(n)$, and corresponds to the input being a sorted list; the $O(n^2)$ worst case is caused by the input being a reverse sorted list, which for the 1-heap involves sifting each new value up through the entire heap. This is doing exactly the same processing as a standard insertion sort.

For d=2 we get the normal heap sort algorithm. After inserting all items we have a heap-ordered representation of the data. To get the sorted list we output the root item and rearrange so as to maintain heap order for the remaining items. The picture is the same if we choose a constant branching factor other than two; if the branch factor is d then we have n insertions each taking time no more than $O(\log_d n)$, followed by n delete-mins each requiring no more than $O(d\log_d n)$ time. Overall this gives us a bound of $O(dn\log_d n)$ independent of the order of the input, which means that it is an $O(n\log n)$ algorithm.

Finally, let's consider d=n–1 (or larger). The root of the min-heap will hold the smallest item seen so far. A new item is compared with the current root item. If smaller, it replaces the root item which will be stored in a new leaf node in the (single) level under the root; if larger, the new item itself will be stored in a new leaf. Only one comparison is made for each new item. This initial heap construction, which takes n–1 comparisons, is very closely related to the first pass of selection sort.

Once all n items have been inserted using the n–1 comparisons, the smallest item is now at the root ready to be output, after which we have to re-establish heap order. This will require finding the smallest of the items in the leaves and moving it to the root, taking n–2 comparisons, then n–3, and so on. Repeatedly outputting the items this way will essentially carry out a selection sort. The time complexity is $O(n^2)$ independent of the order of the input.

In the descriptions above we have seen that the use of a d-heap as the data structure for the "black box" leads to the well-known insertion sort, heap sort and selection sort algorithms, depending on the branching factor d. Let's continue exploring the interplay between the choice of data structure and the resulting algorithm.

In general, a d-heap maintains a partial order of the items in the form of a rooted tree. For d=1, the order is total (a degenerate tree of sorted values with no branching), and for d=n–1 it's not much of an order at all! Another tree data structure that maintains "order" is a binary search tree (BST). We note that although a BST would normally be used to implement a dictionary abstract data type, it can be used as a priority queue. A BST with the same values as those in the heaps in Figure 1 is shown in Figure 2. Working out how to perform a delete-min and delete-max on a BST is a good student exercise (for example, the minimum is found by taking left branches until there are no more, which is an $O(\log n)$ operation on average). Thus we instantly have a new PQ sorting algorithm: insert all items in the BST, and repeatedly delete the left-most node.

This algorithm is essentially the same as what is often called "tree sort" (originally called "tree insertion sort" by Knuth [8], and attributed to D.J. Wheeler and C.M. Berners-Lee [4]), which inserts all values in a tree and then performs an inorder traversal. It is good for students to simulate what happens in the insertion phase: the first item is put at the root, and all subsequent items are compared with it. With some guidance, students should recognize this "partitioning" as being equivalent to quicksort, but with comparisons being made in a different order. Knuth made this observation [8], but it is valuable for students to discover this for themselves.

Once the relationship between a BST and quicksort is established, other connections fall out easily; for example, if the input is a sorted list it is a worst case for both the BST and quicksort unless some effort is made to do some balancing. The sorted input causes the BST to degenerate to a linked list and for quicksort it causes the unbalanced partitioning that leads to behavior equivalent to selection sort. We also note that the degenerate linked list is the only non-trivial case where a tree can be both a heap and a BST, and provides a data structure-based link between the worst case of quicksort and the quadratic sorting algorithms. Finally, strictly sorted lists aren't the only worst case for the BST and quicksort; any ordering where the partitioning value is either the maximum or minimum will cause this behavior, so a choice of pivots such as 1, 10, 2, 9, 3, 8 etc. leads to worst-case partitioning. Such inputs also correspond to the worst case for a BST, where the branches are a zigzag rather than a linear list and the depth is $O(n)$. These duals provide yet another opportunity to explore the behavior of algorithms and data structures.

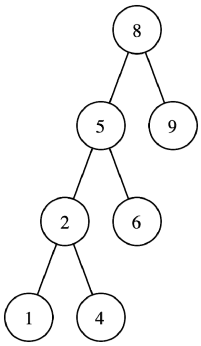


Figure 2: A BST being used to sort the example values

Another relationship that comes from this view is that tree sort can be viewed as an improvement on insertion sort, as follows. The invariant of insertion sort has a "sorted" component that grows with each step in the outer loop, and from a data structure point of view, we are searching for where a value belongs in this component and inserting it. Insertion sort uses a sorted array for this with $O(n)$ time required for insertion, but a BST is a more dynamic structure that can provide $O(\log n)$ searching and insertion times. Finding the insertion point for insertion sort can be improved to $O(\log n)$ time by performing a binary search for the insertion point, but the insertion still dominates with $O(n)$ time because of the linear nature of the array. Substituting a BST for the sorted component transforms insertion sort to tree sort. Thus quicksort, via tree sort, can be viewed as a more efficient way of doing insertion sort by using a better data structure. The usual caveats about keeping things balanced apply to both the BST and quicksort! This pathway from insertion sort to "tree insertion sort" is essentially how Knuth [8] introduced what is now referred to as tree sort.

Returning to heap sort, we note that it will probably have already been taught with a shortcut for the initial n insert operations; the "heapify" operation can be performed in $O(n)$ time if the heap is built bottom-up [6] (Floyd called the improved algorithm "treesort", although it is now usually regarded as the standard version of heap sort, and the name "tree sort" now commonly refers to what was "tree insertion sort"). Despite the improved heapify phase, the heap sort algorithm is still dominated by the $O(n\log n)$ time for the delete-max operations, so asymptotically the shortcut has no effect, and our thought experiment based on the d-heap is still relevant. Similar reasoning applies to performing an inorder traversal for tree sort instead of repeated delete-min operations; the other phase of the sorting will still dominate despite this $O(n)$ optimized version. In fact, the two optimizations provide a nice dual; the first phase of heap sort can be reduced to $O(n)$, as can the second phase of tree sort, but the remaining phase in each case is $O(n\log n)$, so the $O(n)$ component is mere fine tuning and not a fundamental improvement in the sorting algorithm.

4. RELATING OTHER SORTING ALGORITHMS

The d-heap and related priority queues were used in Section 3 above to connect all the commonly-taught sorting algorithms except merge sort (i.e. insertion, selection, heap and quicksort). In this section we will discuss the connection between merge sort and insertion sort so that relationships between all five main algorithms are clear. The connections in Section 3 plus the relationship between merge sort and insertion sort are the main ones that we recommend discussing with students, and there is no need to labor the point by exploring other links. However, in this section we will also note some other connections that occur; these sometimes come up in open-ended discussions with students, and/or can be used as the basis of assessment questions to see if students have understood the algorithms being related.

A direct relationship between merge sort and insertion sort was noted by Merritt [10], who observes that instead of splitting the n items into two lists of size $n/2$, one could perform a "singleton split" into lists of size 1 and $n–1$. Using a constructivist approach, it is worth asking students to imagine such an algorithm; after trying an example they should soon notice that the resulting algorithm is insertion sort since merging a list of size 1 is the same as inserting a value in a list. This can be exercised by considering the worst case (the singleton is larger than everything in the n-1 list) and the best case (it is smaller than everything else). This exposes a correspondence between these cases for merge sort and insertion sort, and exercises the idea that a merge is fastest if one list is exhausted before anything is taken from the other list, and slowest if both lists must be processed to their end.

We can also reverse this reasoning; merge sort can be thought of as an improvement on insertion sort, achieved by batching a group of insertions and pre-sorting them so that the insertion doesn't need to go back to the start of the array to find each insertion point.

A singleton split can be forced in merge sort simply by changing the partitioning point, so to change a standard merge sort program to run as insertion sort only the one calculation needs to be changed. Merritt [10] also noted that a singleton split in quicksort is equivalent to selection sort (which corresponds to the worst case of quicksort), although this occurs by chance and can't be forced by a simple change to the algorithm.

The above is sufficient to relate all our algorithms; however, we now mention some other relationships that might be observed.

We have been considering algorithms as equivalent if they perform the same key comparisons, even if the order of the comparisons is different; for example, this happens for tree sort and quicksort. Another place that this can be observed is between the worst case of insertion sort (a reverse-sorted list) and maximum-selection sort (for which all cases are the same). If we consider the largest item, in both cases it will be compared with every other item in the array; the second largest item is compared with all but the largest, and so on. Therefore both will perform the same number of comparisons ($n(n–1)/2$), and will also be comparing exactly the same pairs of values (that is, all ${}^{n}C_2$ distinct pairs), but in a different order.

In a similar manner, the number of comparisons made in the worst case of merge sort is the same as the best case of quicksort (which also emphasises the merge/quick and insert/select duality). In the worst case of merge sort, a merge will make $n–1$ comparisons to create a list of size n; quicksort always makes $n–1$ comparisons to perform the partition. The best case of quicksort is when the partition makes two even halves, and hence the recursive cases for quicksort are the same size as they would be for merge sort.

Another connection is through the PQ structures: if quicksort is given a reverse sorted list, this corresponds to tree sort where the BST degenerates to a linked list (which is also a 1-heap) with each inserted item being compared with everything that is already in the BST/list/heap. This corresponds to a worst case insertion sort, and we have just established that this performs the same comparisons as a selection sort.

5. CONCLUSION

By linking most sorting algorithms as special cases of a PQ sort (Section 3) and then incorporating some connections already made in the literature (Section 4), we are able to show multiple relationships between sorting algorithms, and in principle, each could have been derived from another by exploring a generalization or extreme case. These relationships provide an opportunity for students to re-visit each algorithm after it has been taught, and explore its performance from a different perspective. This approach shows the strong relationship between algorithms and data structures, and how generalizing an algorithm or data structure can lead to new ones. Because a PQ sort is a simple concept, the derivations are easily taught in a constructivist manner.

The main relationships identified are summarized in Figure 3. The idea of a PQ sort is a simple "black box" approach to the problem; the d-heap PQ then provides us with a range of data structures under a unifying description, leading directly to insertion, heap and selection sort for different values of d. If we use a BST instead of a heap for the PQ, we end up with tree sort, which is essentially equivalent to quicksort (although tree sort can also be thought of as a insertion sort where the sorted component of the array is replaced with a BST). We use Merritt's [10] "singleton split" to link quicksort to selection sort, and merge sort to insertion sort. Batching a number of insertions into a pre-sorted list converts insertion sort to merge sort. Finally, Merritt's [10] view of the two divide-and-conquer algorithms based on whether

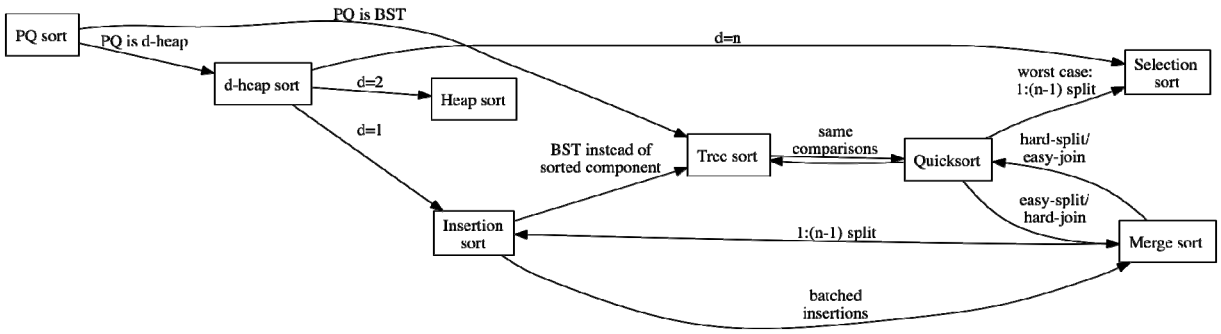


Figure 3. Relationships between common sorting algorithms

the split or join is hardest provides a relationship between quicksort and merge sort.

In the direct relationships we have described above, the various cases for complexity (best, average and worst) are naturally equivalent in terms of the number of comparisons. In addition, we have noted some other equivalences between specific cases:

- the worst case of insertion sort and all cases of selection sort;
- the worst case of merge sort and the best case of quicksort; and
- the worst case of quicksort, which can be seen in tree sort's degenerate BST, which is the same as a 1-heap in this case, which is in turn equivalent to the worst case of insertion sort and therefore also selection sort.

We reiterate that the view of sorting algorithms presented in this paper is not intended for deriving the sorting algorithms from the d-heap when first introducing them, but simply for using this as a thought experiment after the methods have been taught. It can be used to show that they aren't as unrelated as they might appear, to help students perform meaningful exercises relating to the algorithms, and to enable them to remember how they differ and how they are similar.

This approach to showing the relationship between sorting algorithms can inspire students with the elegance behind what can otherwise seen as a random catalogue of algorithms to be learned, and it is good training for computer scientists to always be considering special cases, abstractions, and extreme values.

6. REFERENCES

[1] Aho, A. V., Hopcroft J. E., and Ullman, J. D. 1983. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

[2] Astrachan, O. 2003. Bubble sort: an archaeological algorithmic analysis. *SIGCSE Bull.* 35, 1 (Jan. 2003), 1–5. DOI= http://doi.acm.org/10.1145/792548.611918.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms (3rd Ed.).* The MIT Press, Cambridge, MA.

[4] Douglas, A. S. 1959. Techniques for the recording of, and reference to data in a computer. *The Computer Journal* 2 (1959), 1-9

[5] Dromey, R. G. 1987. Derivation of Sorting Algorithms from a Specification. *The Computer Journal* 30, 6 (1987), 512–518.

[6] Floyd, R. W. 1964. Algorithm 245: Treesort. *Commun. ACM* 7, 12 (Dec. 1964), 701. DOI= http://doi.acm.org/10.1145/355588.365103

[7] Johnson, D. B. 1975. Priority queues with update and finding minimum spanning trees. *Information Processing Letters* 4, 3 (Dec. 1975) 53–57. DOI= http://dx.doi.org/10.1016/0020-0190(75)90001-0.

[8] Knuth, D. E. 1998. *The Art of Computer Programming, Volume 3 (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Reading, MA.

[9] Manber, U. 1989. *Introduction to algorithms: a creative approach*. Addison-Wesley Publishing Co., Inc., Reading, MA.

[10] Merritt, S. M. 1985. An inverted taxonomy of sorting algorithms. *Commun. ACM* 28, 1 (Jan. 1985), 96–99. DOI= http://doi.acm.org/10.1145/2465.214925.

[11] Merrit S. M., and Lau, K.-K. 1997. A logical inverted taxonomy of sorting algorithms. In *Proceedings of the Twelfth International Symposium on Computer and Information Sciences*, S. Kuru, M. Caglayan, and H. Akin, Eds. (Bogazici University 1997) 576–583.

[12] Sedgewick, R. 1983, 1988. *Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Reading, MA.

[13] Thorup, M. 2007. *Equivalence between priority queues and sorting*. J. ACM 54, 6 (Dec. 2007), 28. DOI= http://doi.acm.org/10.1145/1314690.1314692.

[14] Weiss, M. A. 1995. *Data Structures and Algorithm Analysis (2nd Ed.)*. Benjamin-Cummings Publishing Co., Inc., Readwood City, CA.

[15] Williams, J. W. J. 1964. Algorithm 232: Heapsort. *Commun. ACM* 7 (1964), 347–348.

[16] Wood, D. 1993. *Data Structures, Algorithms, and Performance*. Addison-Wesley Longman Publishing Co., Inc.

2.1 Un peu de Python...

Application d'Algortihme

```python
def parent(i, d):
    """
    Returns the index of the parent of node i in a d-heap.
    """
    return (i - 1) // d

def child(i, j, d):
    """
    Returns the jth child of node i in a d-heap.
    """
    return d * i + j + 1

def max_heapify(A, i, n, d):
    """
    Maintains the max-heap property of a d-heap.
    """
    largest = i
    for j in range(d):
        c = child(i, j, d)
        if c < n and A[c] > A[largest]:
            largest = c
    if largest != i:
        A[i], A[largest] = A[largest], A[i]
        max_heapify(A, largest, n, d)

def build_max_dheap(A, d):
    """
    Builds a max d-heap from an array A in O(n) time.
    """
    n = len(A)
    for i in range(n // d - 1, -1, -1):
        max_heapify(A, i, n, d)

def d_heapsort(A, d):
    """
    Sorts an array A using d-heapsort in O(n log n) time.
    """
    build_max_dheap(A, d)
    n = len(A)
    for i in range(n - 1, 0, -1):
        A[0], A[i] = A[i], A[0]
        max_heapify(A, 0, i, d)

```

```python
A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
d = 3
d_heapsort(A, d)
print(A)
```