**Tri par Tas :**

Description :

Le tri par tas est un tri à base de comparaison qui utilise la structure de tas pour trier un tableau.

Structure de tas :

La structure de tas est une structure de donne de type tableau mais qu’on visualise utilisons un arbre binaire, cet arbre binaire est dit Presque complet c-à-dire l’arbre est remplit à tous les niveaux jusqu’au dernier ou il est rempli de gauche a droit.

Dans un tableau chaque élément représente un noeux de notre tas où :

* La racine de l’arbre est première élément du tableau avec indice = 0
* Le fils gauche d’un parent i est calculé avec : fils gauche = i\*2+1
* Le fils droit d’un parent i est calculé avec : fils droit = i\*2+2
* Le parent un noeux est calculer avec : parent = i/2-1

Notre structure de tas peut être un tas-max ou bien un tas-min avec :

* Tas-max : parent > = fils
* Tas-min : parent <= fils

On utilise un tas-max pour un tri ascendant et un tas-min pour un tri descendant.

Dans notre analyse on va utiliser le tas-max.

Le tri par tas a deux phases :

* La première étape est de crée la structure de tas dans notre cas un tas-max à partir de notre tableau.
* Dans la deuxième partie on va commencer à trie notre tas un élément par un. On échange le premier élément du tas avec le dernier élément et on le détache du tas comme il est le plus grand élément du tas courant et comme ça on considère cette élément trie. Mais maintenant notre Tas n’est pas conforme et alors on doit appeler la fonction tamiser qui vas vérifier que notre élément est plus grand que c’est fils et s’il n’est pas on échange notre élément avec le fils le plus grand et on appelle tamiser pour notre élément une autre fois. On refait ça pour change élément du tableau ce qui veux dire n fois.

Tamiser :

|  |  |
| --- | --- |
| **Algorithme tamiser** | |
|  | **Entre** : T : tableau d’entier, index : entier, taille : entier  **Sortie** : T : tableau d’entier trier  **Variable** : index\_fils\_gauche, index\_fils\_droit, max : entier |
| **Debut:**  index\_fils\_gauche = index\*2+1  index\_fils\_droit = index\*2+2  **Si** index\_fils\_gauche < taille **alors**  max = index\_fils\_gauche  **Si** index\_fils\_droit < taille et T[index\_fils\_droit] > T[max] **alors**  max = index\_fils\_droit  **fsi**  **Si** T[max] > T[index] alors  Echanger les element max et index du tableau T  tamiser (T, max, taille)  **fsi**  **fsi**  **Fin** |
|  |  |

Construire le Tas (Tas-max):

|  |
| --- |
| **Algorithme** Tas-max |
| **Entre** : T : tableau d'entier, taille : entier  **Sortie** : T : tableau d'entier trie  **Variable** : index : entier |
| **Debut:**  index = (taille/2)-1 // le dernier élément dans notre tableau qui a un fils dans le Tas  **TANQUE** index >= 0 **alors**  Tamiser (T, index, taille)  index--  **fait**  **Fin** |

Tri par Tas :

|  |  |
| --- | --- |
| Algorithme Tri\_par\_Tas | |
|  | **Entre** : T : tableau d'entier, taille : entier  **Sortie** : T : tableau d'entier trie  **Variable** : index : entier |
| **Debut**  Tas-max (T, taille)  **Pour** index = taille-1 à 1 **alors**  echanger(T[0], T[index])  tamiser (T, 0, index)  **fait**  **Fin** |

Complexité spatiale :

La complexité spatiale du tri par Tas est simplement O(1) car le tri par Tas est un tri sur place c-à-dire il utilise directement le tableau qu’on veut trier sans avoir besoin d’autre espace apart quelques variables temporaires qui n’affecte pas la complexité spatiale globale.

Complexité temporelle :

Le pire cas :

Pour la complexité de Tas-max on vas appeler tamiser n/2, et pour tamiser pour le pire cas on va parcourir la hauteur l’arbre H qui est alors la complexité de Tas-max est

Mais si on analyse un peux plus cette partie on peut trouver que notre complexité est bien plus basse.

On peut utiliser le nombre d’échange que cet algorithme fait pour trouver sa complexité. On a déjà dit que le pire cas de tamiser est log(n) ce qui est vrai mais seulement quand l’élément en question est la racine du tas. Pour les feuilles le pire cas est 0 car il n’y a aucun élément sous eux, pour leur parent le pire cas sera 1, pour le parent de cela le pire cas sera 2….

Alors on peut dire qu’on aura n/2 élément au niveau 0 de l’arbre qui vont faire un maximum de 0 échange, n/4 élément au niveau 1 qui peuvent faire un maximum d’un échange, n/8 élément avec deux échange max, extra…

Et alors on aura cette équation :

Alors on voit bien qu’on un d’échange alors notre complexité est de : O(n).

Et pour la deuxième partie de notre algorithme on a une boucle qui se répète n-1 et à chaque fois un appelle à tamiser.

La complexité de tamiser reste la même a (hauteur du Tas)

Et alors on aura

Et une class de complexité de pour cette partie

Et pour la complexité totale du tri par Tas on a :

Et alors une class de complexité temporelle de :

Le meilleur cas :

Première partie Tas-max :

On suppose que le tableau en entre et déjà trie alors avec Tas-max pour chaque appelle à tamiser on va juste vérifier que la structure est conforme au Tas et s’arrête alors une seule instruction c-à-dire :

n/2 appel à tamiser par Tas-max et un temps constant pour change tamiser

Qui nous donne un total de :

La deuxième partie :

On a une boucle qui se répète n-1 fois et à chaque fois on appelle tamiser mais ici la complexité temporelle de tamiser ne va pas changer du pire car on brise volontairement la structure du Tas et alors le meilleur cas pour tamiser vas rester la hauteur de notre Tas qui est de

Et alors en total on aura

Qui nous donnera une complexité temporelle total pour notre algorithme de :

Et une classe de complexité totale pour notre algorithme de tri pas tas de :

Amélioration Buttomup :

Une des améliorations du tri par Tas est buttomup ou l’idée est plu- tôt simple :

Après avoir créé notre Tas on va commencer à trier les éléments par échange le premier élément du tas qui est le plus grand de notre tas le mettre à la fin et le détacher du tas comme il est trié, après on vas appeler tamiser pour notre racine pour réparer notre structure du Tas. Comme l’élément qui est maintenant la racine est venu de la fin du Tas alors il va probablement descendre à un niveau très bas du Tas et pendant notre descente on va faire 2 comparaisons pour change niveau ; entre les deux fils, le max des deux fils et notre élément.

Cette amélioration va essayer de diminuer c’est comparaison par seulement comparer les deux fils jusqu’on attend une feuille. Apres on va vérifier la validité de notre tas par comparer notre élément avec son père et les échanger si notre élément est plus grand jusqu’on trouve sa vraie place dans le tas.

Trouver feuille :

|  |  |
| --- | --- |
| **Algorithme Trouver\_feuille** | |
|  | **Entre** : T : tableau d’entier, index : entier  **Sortie** : max : entier  **Variable**: fils\_gauche, fils\_droit : entier |
| fils\_gauche = index\*2+1  fils\_droit = index\*2+2  **TANQUE** fils\_droit < taille **alors**  index = max(T[fils\_gauche], T[fils\_droit])  fils\_gauche = index\*2+1  fils\_droit = index\*2+2  **fait**  **Si** fils\_gauche < taille **alors**  max = fils\_gauche  **Fsi** |

|  |  |
| --- | --- |
| Algorithme buttomup | |
|  | **Entre** : T : tableau d’entier, index : entier  **Sortie** : T : tableau d’entier tamiser  **Variable**: parent, feuille, tmp:entier |
| **Debut**  feuille = Trouver\_feuille (T, index)  parent = (feuille-1)/2  TANQUE T[feuille] > T[parent] alors  feuille = parent  parent = (feuille-1)/2  **fait**  tmp = T[feuille]  T[feuille] = T[0]  parent = (feuille-1)/2  **TANQUE** feuille > 0 **alors**  échanger ( tmp, T[parent])  feuille = parent  parent = (feuille-1)/2  **fait**  **Fin** |

Le concept de cette amélioration est relié au fait que notre tableau soit conforme à la structure du Tas et comme dans la phase Tas-max où la structure de Tas n’est pas assurée on ne va l’utiliser.

Alors On va seulement utiliser buttomup pour remplacer tamiser dans la deuxième partie de notre algorithme.

Tri par Tas avec buttomup :

|  |  |
| --- | --- |
| Algorithme Tri\_par\_Tas | |
|  | **Entre** : T : tableau d'entier, taille : entier  **Sortie** : T : tableau d'entier trie  **Variable** : index : entier |
| **Debut**  Tas-max (T, taille)  **Pour** index = taille-1 à 1 **alors**  echanger(T[0], T[index])  buttomup (T, 0)  **fait**  **Fin** |

La complexité spatiale de Tri par tas avec buttomup :

Cette partie ne changera pas comme cette amélioration ne changera rien de la partie spatiale de notre algorithme.

La complexité temporelle du tri par Tas avec buttomup :

L’amélioration buttomup va seulement changer le nombre de comparaison faite pour change itération d’une boucle de n-1.

Dans le meilleur cas on va diviser par 2 le nombre de comparaison qu’on fait par itération a la place de 2 comparaisons par itération on aura 1 seul.

Qui nous donnera a la place e

Mais là class de complexité de notre algorithme de tri de Tas ne change pas :

Pour le pire cas on va seulement diminuer le nombre de comparaison par itération seulement par 1.

Qui nous donnera a la place e

Ce qui ne vas toujours pas changer notre class de complexité temporelle de notre algorithme de :

Ici on peut voir que cette amélioration ne va pas changer la class de complexité de notre algorithme et même peut être contre-productive si les comparaisons qu’on fait sont très facile (temps d’exécution négligeable) et comme on a un appel fonction de plus dans buttomup ça peut engendrer un temps d’exécution pratique qui est pire que la version normale de tri par Tas.

Mais si les comparaisons sont compliquées (grand temps d’exécution) alors la possibilité de diviser les comparaisons qu’on fait par 2 sera très important.

Alors il faut bien étudier votre problème avant d’utilise cette amélioration.

Complexité temporelle pratique :

Avec des comparaisons simples :

Tri par tas itérative sans amélioration :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | N=10000 | N=50000 | N=100000 | N=200000 | N=400000 | N=600000 | N=800000 | N=1000000 |
| Ordre aléatoire | 0.002159 | 0.013 | 0.028 | 0.065 | 0.144 | 0.215 | 0.305 | 0.407 |
| Ordre croissant | 0.0018 | 0.010 | 0.024 | 0.050 | 0.097 | 0.152 | 0.206 | 0.262 |
| Ordre décroissant | 0.001811 | 0.009 | 0.021 | 0.045 | 0.094 | 0.149 | 0.202 | 0.249 |

Tri par tas itérative avec l’amélioration buttomup :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | N=10000 | N=50000 | N=100000 | N=200000 | N=400000 | N=600000 | N=800000 | N=1000000 |
| Ordre aléatoire | 0.0021 | 0.013 | 0.028 | 0.064 | 0.130 | 0.202 | 0.286 | 0.376 |
| Ordre croissant | 0.0018 | 0.010 | 0.021 | 0.045 | 0.090 | 0.138 | 0.191 | 0.246 |
| Ordre décroissant | 0.0017 | 0.010 | 0.020 | 0.044 | 0.087 | 0.138 | 0.187 | 0.234 |

Avec des comparaisons complique :

Tri par tas itérative sans amélioration :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | N=10000 | N=50000 | N=100000 | N=200000 | N=400000 | N=600000 |
| Ordre aléatoire | 0.520 | 3.25 | 7.03 | 15.1 | 32.06 | 49.87 |
| Ordre croissant | 0.533 | 3.35 | 7.27 | 15.55 | 32.9 | 51.07 |
| Ordre décroissant | 0.491 | 3.12 | 6.79 | 14.59 | 31.09 | 48.35 |

Tri par tas itérative avec l’amélioration buttomup :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | N=10000 | N=50000 | N=100000 | N=200000 | N=400000 | N=600000 |
| Ordre aléatoire | 0.323 | 1.89 | 4.03 | 8.5 | 18.04 | 27.9 |
| Ordre croissant | 0.332 | 1.94 | 4.15 | 8.77 | 18.49 | 17.21 |
| Ordre décroissant | 0.308 | 1.8 | 3.84 | 8.16 | 17.21 | 26.65 |

La representation avec des graphe :

Temps d’execution pratique avec des comparaison complex :

Comparaison de la complexité théorique et pratique :

On voit bien que la complexité pratique est proche de la complexité théorique qu’on a calculer.

Et on peut aussi voire l’efficacité de l’amélioration Bottomup quand notre comparaison est compliquée.