Quicksort

Hiba Qutbuddin Habib

Oktober 2022

Introduktion

I fler tidigare uppgifter insåg vi att sortering av data är ganska värdefullt, då lärde vi oss olika typer av sortering såsom urvalssortering, insättningssortering samt merge sortering. I denna uppgiften kommer vi lära oss en yttligare sorterings metod vilket kallas för "Quicksort", då vi kommer att implementera den med hjälp av två andra linjär datastrukturer, vektorer samt länkad lista, implementationen kommer att hjälpa oss att förstå hur denna algoritmen fungerar, vilken tidskomplexitet den här samt skillnaden mellan de både implementationer.

Quicksort, Algoritmen

Quicksort algoritmen sorterar elementerna på plats. Då om man har en sekvens med element, så väljas en element som man kallar för "pivot". Pivot elementet kommer att ha en central roll i sortering, då resten av elementerna i sekvensen kommer att sorteras i jämförelse med just denna element. Då om man väljer en element mitt i sekvensen som pivot så kommer elementer som är mindre än eller lika med pivot elementet att placeras förra den, medans elementerna som är större än den kommer att placeras efter den. Sortering av både sekvensen kommer att fortsätta rekursivt tills hela sekvensen är sorterad.

Quicksort, vektor implementationen

Ett sätt och kunna implementera quicksort algoritmen är att använda den linjära datastrukturen vektorer. Implementation av algoritmen med hjälp av vektorer kommer att ske i två steg, först kommer vi att göra en metod som vi kallar för sort, denna metoden kommer att ha en vilkor, om denna vilkor är ej uppfylld så kommer denna metoden att anropa en annan metod. Metoden ser ut enligt:

```
public static void sort(int[] arr, int low, int high) {
//om både indexer är samma, så behöver vi inte göra något
   if (low == high) return;
```

```
//annars anropas metoden "partition"
     partition(arr, low, high); }
```

Metoden ”Partition” är det den som utför arbetet, alltså den är denna metoden som sorterar elementen på plats,metoden ser ut enligt:

```
 public static void partition(int[] arr, int low, int high) {
 //väljer pivot elementet i mitten av sekvensen
    int middle = low + (high - low) / 2;
    int pivot = arr[middle];
    int i = low, j = high;
    while (i <= j) {
//så länge elementerna som är mindre än pivot, flyttas pekare low fram
        while (arr[i] < pivot) {
            i++; }
//så länge elemterna större än pivot, så flyttas pekare high bakåt
        while (arr[j] > pivot) {
             j--; }
// om pekare i är mindre eller lika med j och vi har element som ska fyttas
        if (i <= j) {
//läggs elementet i en temporär variable
            int temp = arr[i];
//elementerna vid i och j byter plats, samt pekare uppdateras
           arr[i] = arr[j];
            arr[j] = temp;
            i++;
             j--; }}
//resten av sekvensen sorteras rekursivt
   if(low< j)
        partition(arr, low, j);
    if(high>i)
        partition(arr, i, high); }
```

Quicksort, länkade lista

Ett annat sätt och implementera quicksort algoritmen är att använda den linjära datastrukturen länkade lista. För att kunna utföra implementering så behövs det en enkel node klass, som innehåller en pekare till nästa element samt ett värde.En ”add” metod behövs för att sedan kunna lägga in elementerna i listan och på så sätt skapas en lista samt en print metod som printar ut listan.

För att vi ska kunna sortera listan så behövs det två metoder [1], det ena är ”partition”, denna metoden har i uppgift att sortera alla element i listan fram till pivot, då jag väljer pivot som sista element så kommer sortering av restande lista att utföras enligt:

[1] Kod delar implmenterade med hjälp av en del andra källor, vid sökning på google

```
// så länge vi hamnar inte vid sista noden så loopen fortsätter
// då sista noden är pivot
 while (low != high) {
 //om värdet vid den första noden mindre än den valde pivot
    if (low.value < pivot) {
// ändra på pekare
        prev = actual;
// spara värdet av den actual noden i en temp variabel
        int temp = actual.value;
//ändra värde
         actual.value = low.value;
        low.value = temp;
// uppdatera pekare så att vi fortsätter till nästan nod
        actual = actual.next; }
            low = low.next; }
```

Kod stycken ovan kommer att sortera fram tills innan den sista noden, när man hamnar på den sista noden då har man kommit fram till pivot, det man gör då är att byta plats på pivot och noden innan, när detta är utförd så får man en lista med alla element som är mindre än pivot på vänster sidan av den och element som är större är pivot på höger sidan av den. När man har kommit så långt, då det som behövs göras är att sortera den högre delen samt den västra delen av pivot element, då pivot har redan hamnat på sin rätta plats, det kan utföras med hjälp av den andra metoden.

Den andra metoden kallas för ”sort” denna metoden använder sig av metoden partition vid sortering samt implmenteras den rekursivt, då utifrån viss givna vilkor så kommer den att anropa sig själv enligt:

```
//om pivot är detsamma som start nod och så länge den är inte null:
if (pivot!= null && pivot == low)
//så sorteras listan från nästa nod efter pivot fram till sista nod
    sort(pivot.next, high);
//så länge pivot är inte null och nästa är inte null heller
else if (pivot != null && pivot.next != null)
//så flyttas pekare framåt i listan och sortering sker till sista nod
            sort(pivot.next.next, high);
```

Benchmarks

För att man ska kunna jämför både implemtationen samt förstå vilket tidskomplexitet denna sorterings algoritmen här så behöver man testa både metoderna med samma sekvens av element med samma längd. Benchmarks som jag använde såg ut enligt:

```
//skapa en sekvens med slumpmässiga värde
for(int i= 0; i< sequence.length; i++){
```

```
    sequence[i] = rnd.nextInt(50000);}
//fyll på vektor samt listan med samma värderna
 for(int i=0; i< k ;i++){
     arr[i]= sequence[i]; }
 for(int i =0; i< k;i++)
    int value =sequence[i];
        list.add(value); }
//sättet tidmätningar utfördes
double min1 = Double.POSITIVE_INFINITY;
 for(int j=0 ; j<1000 ;j++) {
        long t2 = System.nanoTime();
//tid mätningar för listans användes
//list.sort(list.head, n), iställer för Quicksort.sort nedan
        Quicksort.sort(arr,low,high);
        long t3 = System.nanoTime();
         double t1 = (t3 - t2);
          if (t1 < min1)
         min1 = t1; }
```

Resultat

n	**10**	**20**	**40**	**80**	**160**	**320**	**640**	**1280**	**2560**	**5120**	**10240**
Tid	0.33	0.92	2.4	2	3.2	29	107	345	460	1370	1802
T/nlog(n)	0.014	0.02	0.02	0.03	0.03	0.02	0.02	0.04	0.02	0.04	0.02

Table 1: Tidmätning vektorer

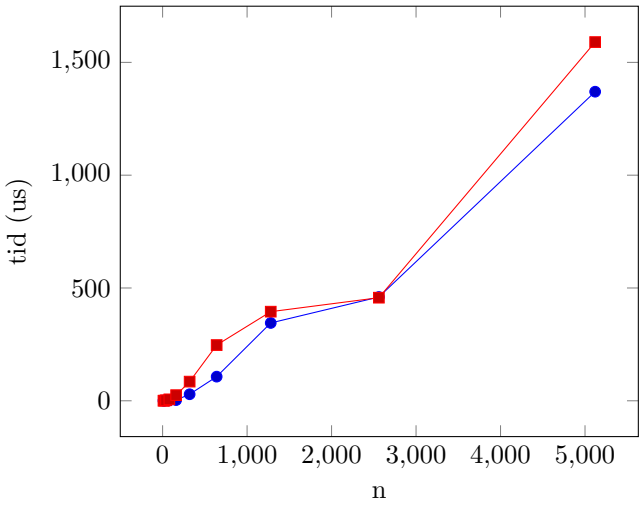
n	**10**	**20**	**40**	**80**	**160**	**320**	**640**	**1280**	**2560**	**5120**	**10240**
Tid	0.25	0.67	2.3	6.4	25	85	247	395	457	1590	1212
T/nlog(n)	0.011	0.011	0.015	0.018	0.03	0.04	0.05	0.04	0.05	0.03	0.013

Table 2: Tidmätning listan

Quicksort är en logaritmiska algoritm alltså den tillhör mängden $O(nlog(n))$ i bästa fall, dock i ett värsta fall så tillhör den mängden $O(n^2)$. Mätningar i tabeller ovan visar att implementering av denna algoritmen med vektorer ger en mer logaritmisk resultat jämfört med resultatet för länkade lista implementationen. Det värsta senario får vi om pivot element är den största eller minsta elementet eller om pivot element väljs i slutet av sekvensen eller i början och sekvensen är omvänd sorterad eller redan sorterad. I både implementering används slumpmässiga värde vilket gör att sekvensen vi får kommer med störst sannolikhet varken att vara omvänt sorterad eller redan färdig sorterad, dock vid vektorn implementering väljs pivot element i mitten av sekvensen vilket gör

när vi placera de mindre och större element i rätt position så kommer vi att bemöts situationen där flera av dessa elementer kommer att ligga på sin rätta position. Däremot vid implementering av länkade listan så valdes den sista elementet som pivot då det är svårt och säga vad är mitten av en länkade lista, i detta fall man kanske ha otur och då är vår pivot den största element eller det minsta dock eftersom pivot är det sista elementen så kommer man behöva sortera ungefär n element flera gånger så att pivot kan hamna rätt, då från början separera de element som är större än eller mindre än pivot för att sedan sorteras de och pivot kan flyttas från sists positionen till den rätta position i sekvensen.

Sammanfattning



Grafen ovan visar resultatet för både algoritmer grafisk, vi kan se att vektor implementation (blå kurva) tar koratare tid jämfört med länkade lista (röda kurvan), vilket tyder på att denna typen av sortering fungerar bättre med den linjära datastrukturer vektor. Både dessa implementeringar har sina fördelar och nackdelar, då det är enklare att implementera denna algoritmen med länkade lista datastrukturer däremot vektor datastruktur ger en mer effektivare resultat.