به نام خدا

ساختمان داده ها

جلسه بیست و دوم

دانشگاه بوعلی سینا

گروه مهندسی کامپیوتر

نیم سال دوم 98-1397

مرتب سازی وجستجو

Searching & Sorting

مطالب این فصل

- جستجو خطی
- جستجوی دودویی
- مرتب سازی درجی
- مرتب سازی سریع
- مرتب سازی ادغام
- مرتب سازی هرمی

مرتب سازي

- [illegible]
- [illegible]
- [illegible]
- [illegible]

اصطلاحات

- زمانی که یک لیست از رکوردها را جستجو می کنیم ، هدف پیدا نمودن رکوردهایی است که دارای فیلدی با مشخصات مورد نیاز باشند. این فیلد را کلید می نامند.

- کارایی روش و خط مشی جستجو بستگی به آرایش و نحوه قرار گرفتن رکوردها در لیست دارد.

جستجوي ترتيبي يا خطی

فرض کنید که لیست و یک کلید جستجو به نام Searchnum داشته باشیم . هدف بازیابی رکوردی است که کلید آن منطبق بر searchnum باشد. اگر این لیست دارای n رکورد باشد ، با list[i].key به مقدار کلید رکورد i دسترسی پیدا می کنیم، لیست را با جستجوی مقادیر کلیدهای list[0].key. . .list[n-1].key مورد بررسی قرار می دهیم تا به رکورد مورد نظر برسیم یا تمام لیست را جستجو کنیم.

جستجوي ترتيبي يا خطی

```
class Element {
  int key;
  // other fields;
public:
  int getKey() const { return key; }
  void setKey(int k) { key = k; }
  // ...
}

int SeqSearch(Element *f, int n, int k) {
// f is [1:n] array, seach if f[i].key == k
  int i = n;
  f[0] = setKey(k); // f[0] is a sentinel
  while(f[i].getKey() != k) --i;
  return i; // return 0 → not found!
}
```

Time complexity: O(n) **Better search method?**

جستجوي دودويي

در جستجوي دودويي بايستي ليست بر اساس فيلد كليد مرتب شود ، يعني:

$$list[0].key \leq list[1].key \leq \ldots \leq list[n-1].key$$

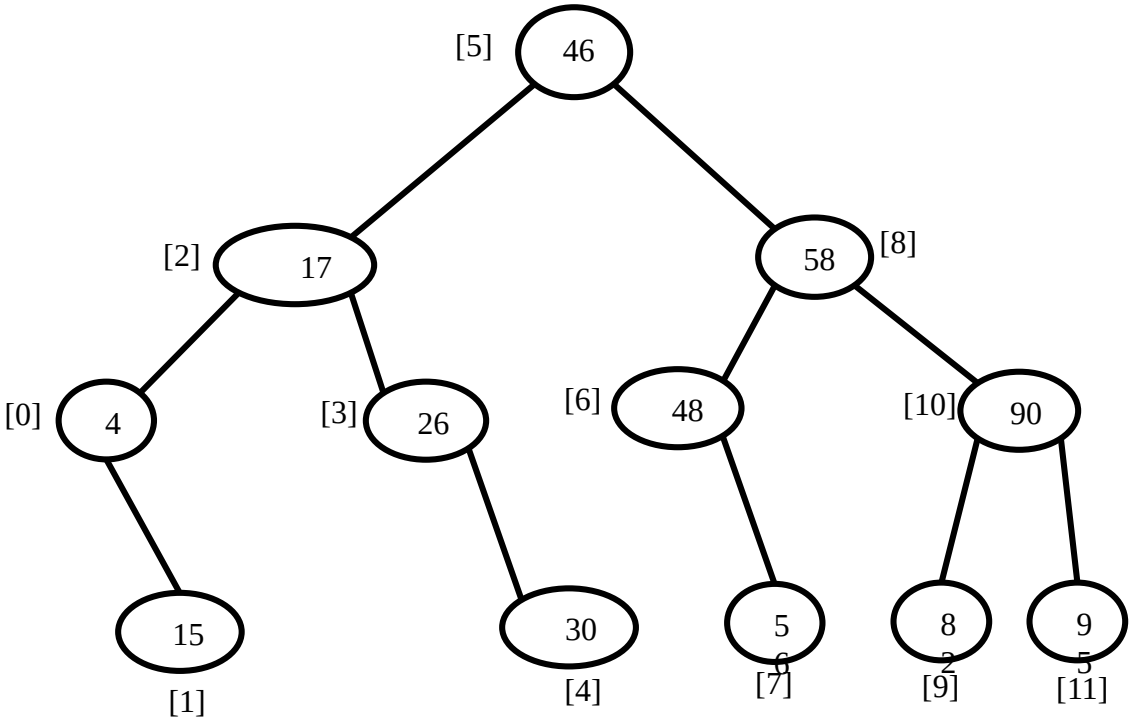
جستجوي دودويي

این جستجو با مقایسه searchnum و list[middle].key ، به ازای middle=(n-1)/2 شروع می شود.

هنگام مقایسه سه حالت ممکن است روی دهد :

1) **searchnum< list[middle].key : در این حالت رکوردهایی بین list[middle] و list[n-1] کنار گذاشته شده و جستجو با رکوردهای List[0] تا List[middle-1] دنبال می شود.**
2) **Searchnum= list[middle].key : در این حالت جستجو با موفقیت به پایان می رسد.**
3) **searchnum> list[middle].key : در این حالت ، رکوردهای بین List[0] و list[middle] کنار گذاشته شده و جستجو با رکوردهای list[middle+1] تا list[n-1] دنبال می شود.**

درخت تصميم گيري براي جستجوي دودويي



مرتب سازي

- مرتب سازی می تواند در موارد زیر مفید باشد:
  - در جستجوی لیستها
  - در بررسی ورودیهای لیستها

- مرتب سازی را می توان به طور کلی به دو دسته تقسیم کرد:
  - مرتب سازی داخلی
    - لیست به اندازه کافی کوچک است و در حافظه اصلی جا می گیرد
    - مرتب سازی درجی, ادغام ، سریع و هرمی
  - مرتب سازی خارجی
    - در این حالت لیست خیلی بزگ است و در حافظه اصلی جا نمی گیرد و در درون دیسک یا نوار عمل مرتب سازی انجام می گیرد.

مرتب سازي درجي Insertion Sort

در اين نوع مرتب سازي ، در هر لحظه فقط يک رکورد در داخل ليست قابل رويت است بنابراين رکور R_i را در بين رکوردهاي مرتد $R_0, R_1, ..., R_{(i-1)}$ طوري قرار مي دهيم که رشته حاصل با اندازه i نيز مر $K_0 \le K_1 \le ... \le K_{i-1}$.

```
void insert(const Element e, Element *list, int i) {
// insert an Element e into an ordered sequence list[1]~
// list[i] and list[0] is a sentinel with the smallest key
  while(e.getKey() < list[i].getKey()) {
    list[i+1] = list[i];
    --i;
  }
  list[i+1] = e;
}
```

Time Complexity: O(i)

```
void InsertSort(Element *list, int n) {
// sort the given list in non-decreasing order of key
  list[0] = MININT; // list[0] works as a sentinel
  for(int j = 2; j <= n; ++j)
    insert(list[j], list, j-1);
}
```

Record R_i is left out of order (LOO) iff $R_i < \max_{1 \le j < i} \{ R_j \}$

Time Complexity: $O(n^2)$

Insertion sort is stable

The simplicity makes it the fastest sorting method for about $n \le 20$

تحلیل تابع insertion sort

زمان محاسباتي جهت درج یک رکورد به داخل لیست مرتب شده ، $O(i)$ خواهد بود.

زمان کل در بدترین حالت برابر است با:

$$O(\sum_{i=0}^{n-1} i) = O(n^2)$$

مرتب سازي درجي(مثال)

Case 1:
worst-case of insertion sort

j	[1]	[2]	[3]	[4]	[5]
-	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

Case 2:
only R_5 is LOO, just need to move R_5

j	[1]	[2]	[3]	[4]	[5]	
-	**2**	3	4	5	1	
2	**2**	**3**	4	5	1	O(1)
3	**2**	**3**	**4**	5	1	O(1)
4	**2**	**3**	**4**	**5**	1	O(1)
5	**1**	**2**	**3**	**4**	**5**	O(n)

مرتب سازي سريع (Quick Sort)

این روش در بین همه مرتب سازیها دارای بهترین متوسط زمانی است.

درمرتب سازي سريع کليد محور يا مفصل (pivot) که عمل درج را کنترل می کند با توجه به زیر لیست مرتب شده ($R_1,\ldots R_{i-1}$) در مکان صحیح قرار می گیرد. یعنی K_i نسبت به کل لیست در جای صحیح قرار می گیرد.

یعنی اگر $K_j \le K_i$ و $j > s_{(i)}$ در محل s(i) قرار بگیرد ، پس به ازاي $j<=s(i)$ داریم $K_j \ge K_{s(i)}$

این رو بعد از جايگذاري ، ليست اصلي به دو زير ليست که شامل رکوردهاي و مي باشند ، تقسيم مي شوند. از بوده و به ازاي $R_0,\ldots, R_{s(i)-1}$ ، داريم $R_{s(i)+1},\ldots, R_{n-1}$

مرتب سازي سريع

```
void QuickSort(Element *list, int left, int right) {
// sort list[left] ~ list[right]
// put list[left] to the correct position and
// partition the given list into 2 sublists
  if(left < right) {
    int i = left;
    int j = right + 1;
    int pivot = list[left].GetKey();
    do {
      do {++i;} while (i < right && list[i].GetKey() < pivot);
      do (--j;} while (list[j].GetKey() > pivot);
      if(i < j) interchange(i, j);
    } while (i < j);
    interchange(left, j);
    QuickSort(list, left, j-1);
    QuickSort(list, j+1, right);
  }
}
```

مرتب سازي سريع

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	Left	Right
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

مرتب سازي سريع

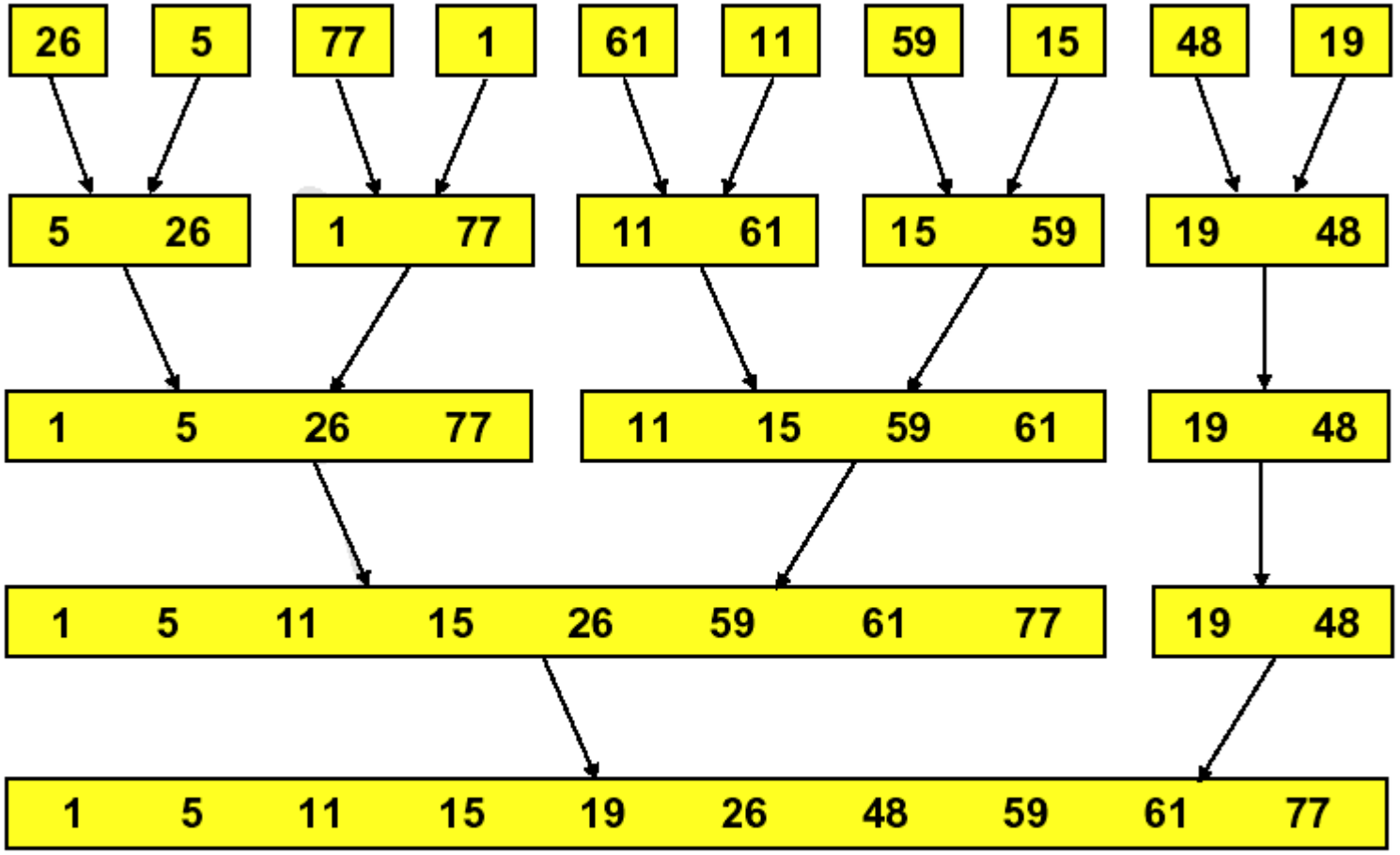
در بد ترین حالت زمان مرتب سازی سریع o(n^2) است.

- در این حالت لیست اولیه مرتب می باشد.

ميانگين زمان محاسبه براي مرتب سازي سريع $O(n \log_2 n)$ **مي باشد.**

مرتب سازي ادغام تكراري(غير بازگشتي)

در نسخه تكراري فرض خواهد شد كه رشته ورودي شامل n ليست مرتب شده به طول يک است. ليست ها را دو به دو (جفتي) ادغام مي كنيم تا تعداد n/2 ليست با اندازه 2به دست آوريم. سپس n/2 ليست حاصل را مجددا دو به دو ادغام مي كنيم و اين عمل را ادامه مي دهيم تا يک ليست باقي بماند.



bottom-up merge

ادغام دو لیست

```
void merge(Element *iList, Element *mList, int l, int m, int n)
{ // iList[l..m] and iList[m+1..n] are 2 sorted lists
  // the sorted result will be mList[l..n]
  int i1, i2, iR, t;
  for(i1 = l, iR = l, i2 = m+1; i1 <= m && i2 <= n; ++iR)
    if(iList[i1].getKey() <= iList[i2].getKey())
      mList[iR] = iList[i1++];
    else
      mList[iR] = iList[i2++];
  if(i1 > m)
    for(t = i2; t <= n; ++t)
      mList[iR+t-i2] = iList[t];
  else
    for(t = i1; t <= m; ++t)
      mList[iR+t-i1] = iList[t];
}
```

Time Complexity: O(n-l)

Space Complexity: O(n-l)

```
void MergePass(Element *iList, Element *rList, int n, int l) {
  int i;
  for(i = 1; i <= n- 2*l +1; i += 2*l)
    merge(iList, rList, i, i + l - 1, i + 2 * l - 1);
  // merge remaining list of length < 2 * l
  if((i + l - 1) < n)  // 2 lists still
    merge(iList, rList, i, i + 1 - l, n);
  else
    for( ; i <= n; ++i) rList[i] = iList[i];
}

void MergeSort(Element *list, int n) { // list[1..n]
  Element *tList = new Element[n+1];
  for(int l = 1; l < n; l *= 2) {
    MergePass(list, tList, n, l);
    l *= 2;
    MergePass(tList, list, n, l);
  }
  delete [] tList;
}
```

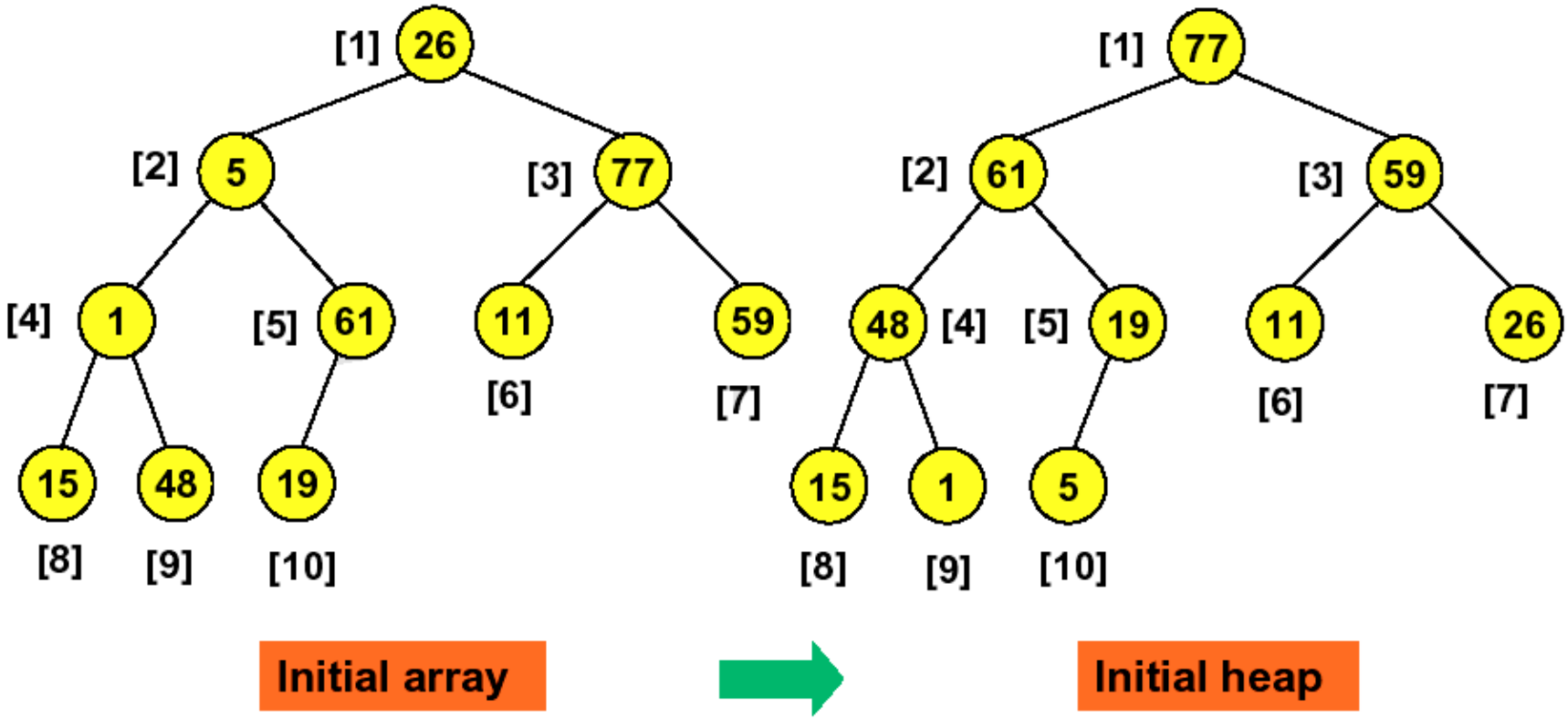
$\log_2 n$ passes in total ➔

Time Complexity: O(n logn)

Iterative merge sort is stable

مرتب سازي heap

روش مرتب سازي heap نيازمند تنها يک مقدار حافظه اضافي است و در همين حال ، بدترين حالت و زمان متوسط آن برابر با O(n log n) خواهد بود. با وجود اينکه مرتب سازي heap کمي کندتر از مرتب سازي ادغام با استفاده از O(n) فضاي اضافي است ولي از مرتب سازي ادغام با استفاده از O(1) فضاي اضافي سريعتر مي باشد.

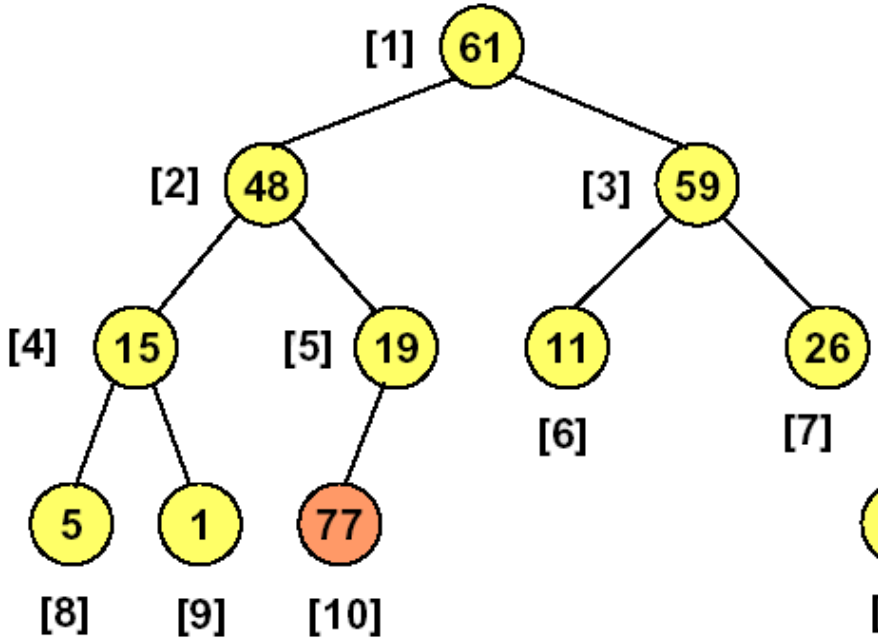

[1] 26
[2] 5
[3] 77
[4] 1
[5] 61
11
59
[6]
[7]
15
48
19
[8]
[9]
[10]
Initial array
[1] 77
[2] 61
[3] 59
48
[4]
[5] 19
11
26
[6]
[7]
15
1
5
[8]
[9]
[10]
Initial heap

```
void adjust(Element *tree, int root, int n) {
// adjust the binary tree rooted at root to be a max heap
  Element e = tree[root];
  int j, k = e.getKey();
  for(j = root * 2; j <= n; j *= 2) {
    if((j < n) && (tree[j].getKey() < tree[j+1].getKey()))
      ++j; // there is a right child and right > left
    if(k >= tree[j].getKey()) // j points to the large child
      break; // key > large child's, find the right place
    tree[j/2] = tree[j]; // or, move up the large child
  }
  tree[j/2] = e;
}
```

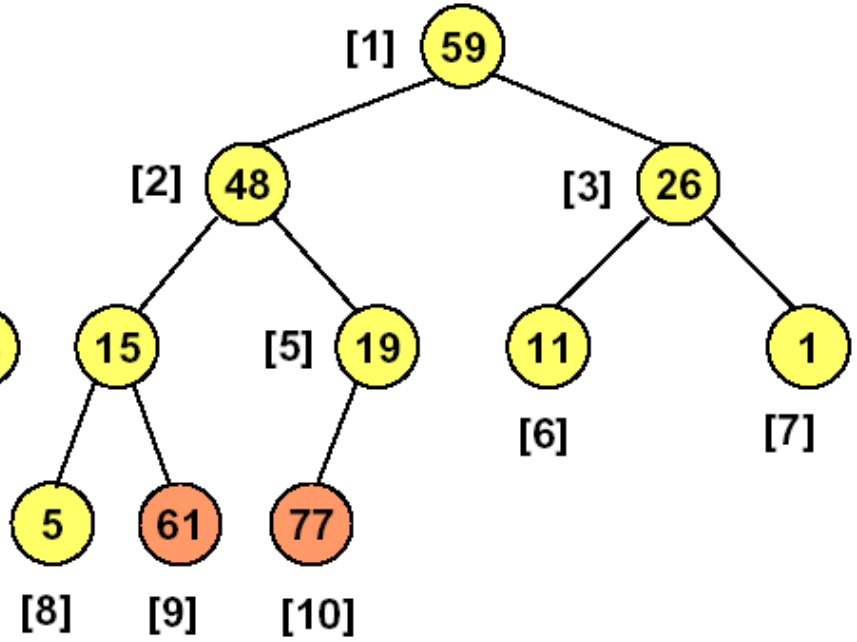
```
void HeapSort(Element *list, int n) {
// sort list[1..n]
  int i;

  for(i = n/2; i >= 1; --i)
    adjust(list, i, n); // turn the list into a max heap

  for(i = n - 1; i >= 1; --i) {
    Element t = list[i+1];  // swap list[1] and list[i+1]
    list[i+1] = list[1];
    list[1] = t;
    adjust(list, 1, i);  // recreate heap after deletion
  }
}
```

Time Complexity: O(n logn)
Extra space: O(1)

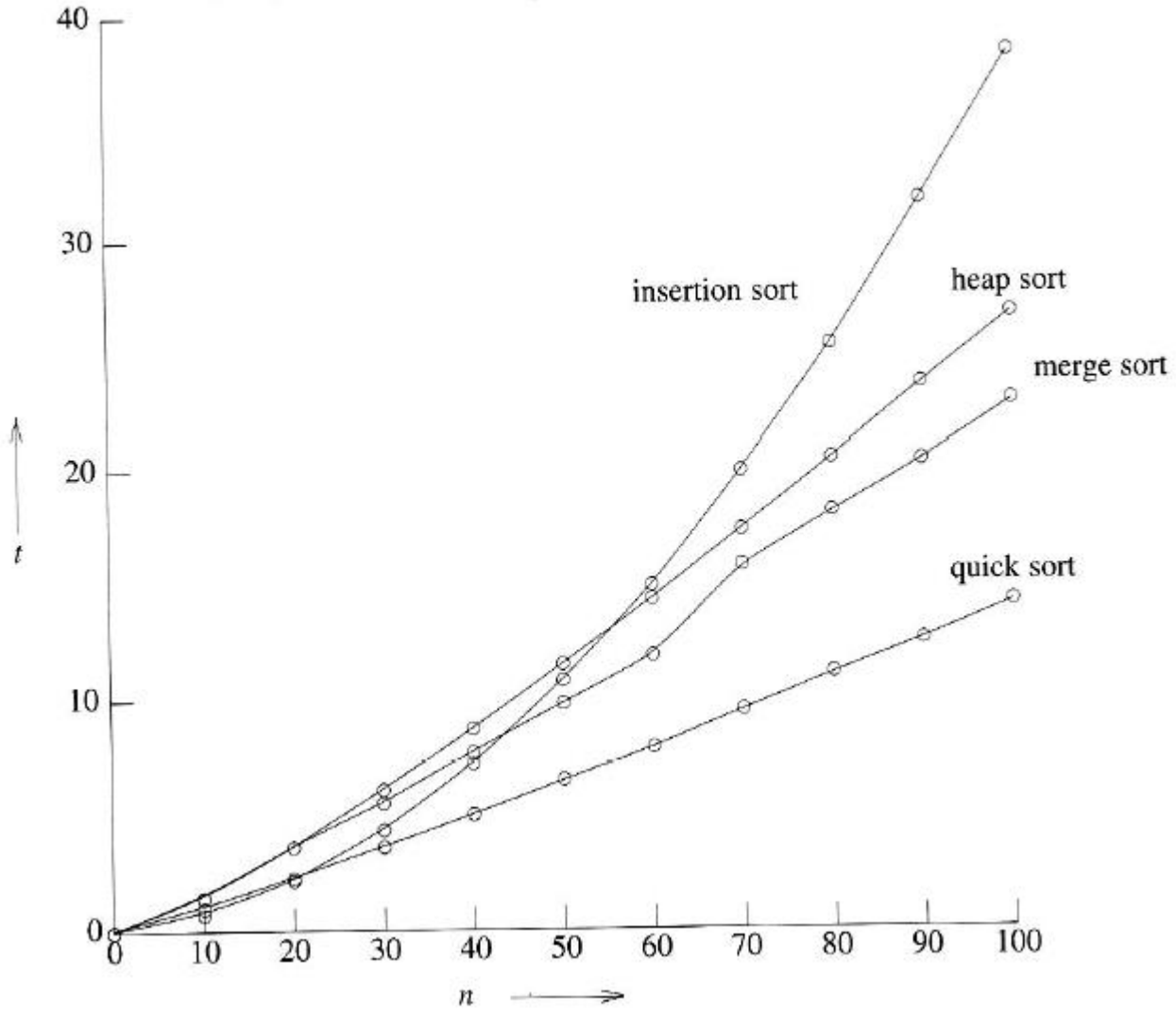
Heap sort is unstable



Heap size = 9,
Sorted = [77]



Heap size = 8,
Sorted = [61, 77]


40
30
20
10
0
t
0
10
20
30
40
50
60
70
80
90
100
n
insertion sort
heap sort
merge sort
quick sort