Ian Piper
CSCI203

Algorithms and Data Structures

Week 2 – Lecture A

Sorting

- Insertion Sort (review).
- Merge Sort.
- The Heap, a new data structure.
- Heap Sort.

Insertion Sort

Insertion Sort.

- You should already be familiar with insertion sort from CSIT113.
- Insertion sort uses the following strategy:
  1. Start with the second element in the list.
  2. Insert it in the right place in the preceding list.
  3. Repeat with the next unsorted element.
  4. Keep going until we have placed the last element in the list.
- We can see how this works with an example.

Insertion Sort Example

- Starting list:

14	20	7	8	5	15	18	17	6	13

- Start at the second element.

14	20	7	8	5	15	18	17	6	13

- It’s in the right place

14	20	7	8	5	15	18	17	6	13

- We have finished the step.
- Repeat with the next element.

14	20	7	8	5	15	18	17	6	13

Insertion Sort Example

- Starting list:

14	20	7	8	5	15	18	17	6	13

- Start at the next element.

14	20	7	8	5	15	18	17	6	13

- Put it in the right place

7	14	20	8	5	15	18	17	6	13

- We have finished the step.
- Repeat with the next element.

7	14	20	8	5	15	18	17	6	13

Insertion Sort Example

- Starting list:

7	14	20	8	5	15	18	17	6	13

- Start at the next element.

7	14	20	8	5	15	18	17	6	13

- Put it in the right place

7	8	14	20	5	15	18	17	6	13

- We have finished the step.
- Repeat with the next element.

7	8	14	20	5	15	18	17	6	13

Insertion Sort Example

- Starting list:

7	8	14	20	5	15	18	17	6	13

- Start at the next element.

7	8	14	20	5	15	18	17	6	13

- Put it in the right place

5	7	8	14	20	15	18	17	6	13

- We have finished the step.
- Repeat with the next element.

5	7	8	14	20	15	18	17	6	13

Insertion Sort Example

- Just showing the result at the end of each iteration

5	7	8	14	20	15	18	17	6	13
5	7	8	14	15	20	18	17	6	13
5	7	8	14	15	18	20	17	6	13
5	7	8	14	15	17	18	20	6	13
5	6	7	8	14	15	17	18	20	13
5	6	7	8	13	14	15	17	18	20

- And we have finished

Looking Deeper

- To sort a list of n numbers requires $n-1$ iterations.
- Each iteration, however, requires that the selected entry be compared with the already sorted list until its correct location can be found.
- In the worst case this means comparing with every such element.
- So, to sort the i^{th} element requires i comparisons.
- Typically, the entire sort will require around $n^2/2$ comparisons.

Merge Sort

Merge Sort

- All of the different sorting algorithms you have seen have one common feature:
- They all sort the values *in place*, the sorted array is the same array as the unsorted one.
- Merge sort takes a different approach:
- It uses a second array to hold the intermediate results.
- It works recursively by dividing the unsorted array into two parts and merging them in order.

Merge sort

- Because this procedure divides all the way down before merging back up, the final result is a sorted array.
- The pseudocode representation of the merge sort algorithm is as follows:

The Merge Sort Algorithm

```
global X[1..n] // temporary array used in merge procedure

procedure mergesort(T[left..right])
    if left < right then
        centre = (left + right) ÷ 2
        mergesort(T[left..centre])// sort the left half
        mergesort(T[centre+1..right]) // sort the right half
        merge(T[left..centre], T[centre+1..right],T[left..right])
            // join the halves in sorted order
```

The Merge Sort Algorithm

```
procedure merge(A[1..a], B[1..b], C[1..a + b])
    apos = 1; bpos = 1; cpos = 1
    while apos ≤ a and bpos ≤ b do
        if A[apos] ≤ B[bpos] then
            X[cpos] = A[apos]
         apos = apos + 1; cpos = cpos + 1
        else
            X[cpos] = B[bpos]
            bpos = bpos + 1; cpos = cpos + 1
    while apos ≤ a do
        X[cpos] = A[apos]
        apos = apos + 1; cpos = cpos + 1
    while bpos ≤ b do
        X[cpos] = B[bpos]
        bpos = bpos + 1; cpos = cpos + 1
    for cpos = 1 to a + b do
        C[cpos] = X[cpos]
```

Merge Sort: an example

- Starting array

7	2	9	5	1	3	8	4

- Split and recursively call mergesort on both halves

7	2	9	5

1	3	8	4

- Split again

7	2

9	5

1	3

8	4

- And again

7

2

9

5

1

3

8

4

Merge Sort: an example

- The array is now fully split.

7	2	9	5	1	3	8	4

- We can now merge each pair.

2	7	5	9	1	3	4	8

- Merge each resulting pair

2	5	7	9	1	3	4	8

- And again to give us the sorted result

1	2	3	4	5	7	8	9

Some Analysis

- At each level, merge operates on all *n* items in the array.
- As each level divides the array in two, there are log *n* levels.
- Overall, mergesort requires $n \times \log n$ operations.

Heaps

Heaps

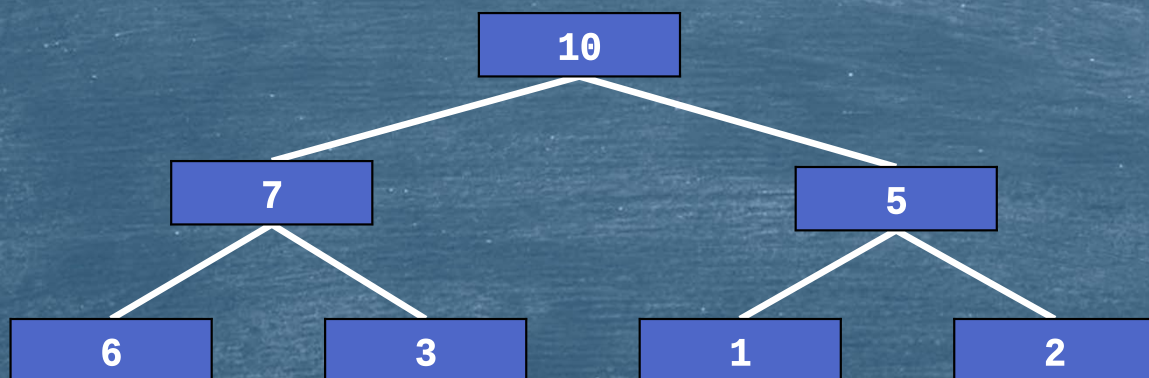
- A heap is an essentially complete binary tree with an additional property
- The value in any node is less than or equal to the value in its parent node. (except for the root node).
- We can store a heap (or any other binary tree) in an array:
  - Heap[1] is the root of the tree
  - Heap[2] and Heap[3] are the children of Heap[1]
  - In general, Heap[*i*] has children Heap[2*i*] and Heap[2*i*+1]

An Example

- The array:

10	7	5	6	3	1	2

- Is the same as the heap:



Operations on Heaps

- If we have a non-heap how can we convert it into one?
- If we have a heap and add a new element at the next leaf how can we restore the heap property?
- If we have a heap and change the root element how can we restore the heap property?
- We need two basic functions to manage heaps:
  - Siftup: Insert a new leaf into the correct position;
  - Siftdown: Insert a new root element into the correct position.
- Each compares an element of the heap with other elements, either its parent or its children.

Siftup

```
Procedure siftup(Heap, i)
// move element i up to its correct position
     if  i = 1 return
     p = i ÷ 2 // integer division
          if Heap[p] > Heap[i]
          return
     else
          swap (Heap[i], Heap[p])
          siftup(Heap,p)
     endif
end
```

Siftup, an example

- Start with the following heap:

10	7	5	6	3	1	

- Add a new element:

10	7	5	6	3	1	9

- Compare with its parent:

10	7	5	6	3	1	9

- Swap:

10	7	9	6	3	1	5

- Compare again

10	7	9	6	3	1	5

- Done

10	7	9	6	3	1	5

Siftdown

```
procedure siftdown(Heap, i)
// move element i down to its correct postion
     c = i * 2
     if Heap[c] < Heap[c + 1] c = c +1
     if Heap[i] < Heap[c]
          swap (Heap[i], Heap[c])
          siftdown(Heap,c)
     endif
end
```

- Note: this procedure is not complete – we need to make sure we don’t fall off the end of the array.

Siftdown, an Example

- This array is a heap with the root missing:

	7	9	6	3	1	5

- This is not a heap, fix it:

4	7	9	6	3	1	5

- Is it smaller than one of its children?

4	7	9	6	3	1	5

- Yes: swap it with its larger child

9	7	4	6	3	1	5

- Compare again:

9	7	4	6	3	1	5

- Swap

9	7	5	6	3	1	4

- Done

Heapsort

Heapsort

- Heapsort uses the properties of a heap to sort an array.
- It proceeds as follows:
  1. Convert the array into a heap
  2. Repeatedly:
     a. Swap the first and last elements of the heap
     b. Reduce the size of the heap by 1
     c. Restore the heap property of the smaller heap
  3. Until the heap contains a single element
- The array is now sorted

The Algorithm

```
Procedure heapsort(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
        swap T[1] and T[i]
        siftdown(T[1 .. i – 1], 1)
```

- This uses two functions:
  - **makeheap** which converts the array into a heap
  - **siftdown** which restores the heap property

Makeheap

- The **makeheap** procedure also uses **siftdown**.

```
procedure makeheap(T[1..n])
     for i = n ÷ 2 to 1 step –1 do
          siftdown(T, i)
     end for
end
```

- Each element, other than the leaves, is progressively moved into the correct location.

Makeheap in action

- Start with the following array:

7	2	9	5	1	3	8	4

- Siftdown 5, compare with 4 no swap needed

7	2	9	5	1	3	8	4

7	2	9	5	1	3	8	4

- Siftdown 9, no swaps needed

7	2	9	5	1	3	8	4

7	2	9	5	1	3	8	4

- Siftdown 2, swap with 5

7	2	9	5	1	3	8	4
7	5	9	2	1	3	8	4

- Siftdown 2, swap with 4

7	4	9	2	1	3	8	4
7	5	9	4	1	3	8	2

- Siftdown 7, swap with 9

7	5	9	4	1	3	8	2
9	5	7	4	1	3	8	2

- Siftdown 7, swap with 8

9	5	7	4	1	3	8	2
9	5	8	4	1	3	7	2

- Done

- Starting array:

7	2	9	5	1	3	8	4

- After makeheap:

9	5	8	4	1	3	7	2

- Swap 9 to the end and reduce the size of the heap

2	5	8	4	1	3	7	9

- Siftdown 2

2	5	8	4	1	3	7	9
8	5	2	4	1	3	7	9
8	5	2	4	1	3	7	9
8	5	7	4	1	3	2	9

- So far

8	5	7	4	1	3	2	9

- Swap 8 to the end and reduce the size of the heap

2	5	7	4	1	3	8	9

2	5	7	4	1	3	8	9
7	5	2	4	1	3	8	9
7	5	2	4	1	3	8	9
7	5	3	4	1	2	8	9

- So far

7	5	3	4	1	2	8	9

- Swap 7 to the end and reduce the size of the heap

2	5	3	4	1	7	8	9

-

2	5	3	4	1	7	8	9
5	2	3	4	1	7	8	9
5	2	3	4	1	7	8	9
5	4	3	2	1	7	8	9

- So far

5	4	3	2	1	7	8	9

- Swap 5 to the end and reduce the size of the heap

1	4	3	2	5	7	8	9

1	4	3	2	5	7	8	9

4	1	3	2	5	7	8	9

4	1	3	2	5	7	8	9

4	2	3	1	5	7	8	9

- So far

4	2	3	1	5	7	8	9

- Swap 4 to the end and reduce the size of the heap

1	2	3	4	5	7	8	9

-

1	2	3	4	5	7	8	9

3	2	1	4	5	7	8	9

- So far

3	2	1	4	5	7	8	9

- Swap 3 to the end and reduce the size of the heap

1	2	3	4	5	7	8	9

-

1	2	3	4	5	7	8	9

2	1	3	4	5	7	8	9

- So far

2	1	3	4	5	7	8	9

- Swap 2 to the end and reduce the size of the heap

1	2	3	4	5	7	8	9

- The heap now has a single element and we are done.

1	2	3	4	5	7	8	9

Analysis

- makeheap requires roughly $n \times \log n$ operations.
- siftdown requires roughly $\log n$ operations.
- siftdown is repeated n-1 times.
- Overall, heapsort requires roughly $2 \times n \times \log n$ operations.