Prepared by: [Arun Konagurthu]

Source material acknowledgement

These lecture slides are built on the [source material] developed by [Lloyd Allison].

Preparatory material on topics covered in FIT1008/FIT1045

Priority Queue, Operations on Heap, Heap Sort, and Merge Sort

Faculty of Information Technology, Monash University

What is covered in these slides?

- Slides cover a few recursive sorting algorithms that were covered already FIT1045 and FIT1008, specifically $\mathbf{O(}N\log(N)\mathbf{)}$-time sorting algorithms: Heap and Merge sorts.
- Pseudo code supporting (operations on) these sorting algorithms.
- Will also briefly cover:
  - **heap data-structure**...
  - ...which also implements a data structure called the **priority queue**...
  - ... which will be used in **Heap sort**

Recommended reading

- Priority Queue and Heap data structure reference: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Priority-Q/ *
- Heap sort reference: http://users.monash.edu/~lloyd/tildeAlgDS/Sort/Heap/
- Merge sort reference: http://users.monash.edu/~lloyd/tildeAlgDS/Sort/Merge/

*Note: Priority Queue should not be confused with an ordinary Queue that you studied in FIT1029/FIT1045/FIT1008. They are different data structures. For revision of what an ordinary Queue data structure is, refer: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Queue/

Priority Queue (and not to be confused with an ordinary Queue)

The operations of a [Priority Queue] are:

1. **create** an empty Priority Queue
2. **insert** an element having a certain priority to the Priority Queue
3. **remove** the element having the highest priority.

Heap data structure

- The implementation of a Priority Queue is the (**binary**) **Heap data structure**.
- Heap is so common for Priority Queue implementation that, in context of Priority Queue, when the term **Heap** is used, it generally means an implementation of **Priority Queue**.

Heap's **Shape property** (**This is necessary but NOT sufficient!**)

Shape Property

- A heap is a **binary tree**.
- It is **almost completely filled**, with the **possible exception of the bottom-most level**.
- A heap is **filled left to right**.
- A heap of height $\mathbf{h}$ (or containing $\mathbf{h}$ levels) have between $\mathbf{2^h}$ to $\mathbf{2^{h+1}-1}$ nodes in it.
- Because of the **regularity of the structure of heap**, it can be **represented as an array**.

Heap's **Heap property**

Heap must satisfy the following additional property. (Together with the shape property, these two properties are [necessary and sufficient] for a Heap.

- The children of the element at the **i**-th position, **if they exist**, will be located at:
  - `left(i)  = 2*i`
  - `right(i) = 2*i + 1`
- Any element in the **i**-th position is no smaller than its children (if any) This necessarily implies that **a[1]** must hold the largest value in **a[1...N]**.

---

*Footnotes:

1. We are assuming a **one-based** array, that is, **a[1...N]**. It is possible to redefine the same property with a **zero-based** array, **a[0...N-1]**, but the definitions of **left(i)** and **right(i)** will change – **this is an exercise for you**.
2. There is a similar definition for a heap with **smallest value at the top** (**a[1]**)

Insertion of a new element into a heap

- Let **a[1...i-1]** be an **existing** heap.
- If a **new element** is inserted at **a[i]**, it could potentially **violate** the heap property – that is, it might be greater than the value in its parent node.
- The **parent node** **a[parent]** of the child node **a[i]** will be at the position **parent=floor(i/2)** in the array.
- If **a[parent] < a[i]**, then **a[parent]** and **a[i]** are swapped.
- **But there is still a potential problem**. The new element might still be **larger** than its **new parent**.
- This implies, we will have to work our way **up the Heap** (or **upHeap**, in short).
- This moves **small parents down** until, either **top**, i.e., **a[1]**, is reached or until a **parent is found that is no smaller than the new element**, and the new element can be placed.

Insertion of a new element into a heap

upHeap function

```
function upHeap(variable child)
// PRECONDITION:  a[1..child-1] is a Heap.
// SETUP: New element to be inserted correctly into the heap is ...
//        ... initially placed at a[child] on which this upHeap is run.
// POSTCONDITION: a[1..child]   is a Heap
 { variable newElt = a[child];                //element to be inserted
   variable parent = Math.floor(child/2); // parent index
   while(parent >= 1) // child has a parent
    { // INVARIANT: a[child... ] is a Heap
      if( a[parent] < newElt )
      { a[child] = a[parent]; // move current parent down
        child = parent;
        parent = Math.floor(child/2);
      }
      else break;
    }
   // ASSERT: child == 1 || newElt <= a[parent]
   a[child] = newElt;
 }//upHeap
```

Complexity of insertion is $O(\log N)$-time Worst case

Try to reason why!

Remove Highest Priority Element from heap

- Consider a Heap **a[1...N]**. The highest priority element is **a[1]**.
- If this is **extracted**/**removed** from the heap, then it leaves a **hole** at position 1.
- The hole is filled by moving **a[N]** to **a[1]** and decreasing **N**.
- But, **this rearrangement may violate that heap property** – **in fact, it almost always will violate**! – why?
- The solution is to move the element **down** until it is no smaller than its children (if any).

Removing element of highest priority from heap

downHeap function

```
function downHeap( a[1...N], parent, N )
 // PRECONDITION:  a[parent+1..N] is a Heap, and parent >= 1
 // POSTCONDITION: a[parent  ..N] is a Heap
  { variable newElt = a[parent];
    variable child  = 2*parent; // left(parent)
    while( child <= N )   // parent has a child
    { // INVARIANT: a[1 .. parent] is a Heap
       if( child < N )    // has 2 children
         if( a[child+1] > a[child] )
           child++;       // right child is bigger
       if( newElt < a[child] )
        { a[parent] = a[child];
          parent = child;
          child = 2*parent;
        }
       else break;
    }
    // ASSERT: child > N || newElt >= a[child]
    a[parent] = newElt;
  }//downHeap
```

Complexity of removing is $O(\log N)$-time worst case

Try to reason why!

Heap Sort – basic ideas

Heap sort's basic idea:

- If a[1....N] are a list of numbers (say!) to be sorted
- Convert a[1...N] into a **`heap`**.
- Remove **biggest** element (found at **`a[1]`**) from the heap.
- Put the removed element in **`a[N]`**
- Remove the next biggest from the heap (again found at **`a[1]`**)
- Put this in **`a[n-1]`**
- $\vdots$ (and so on)

Heap Sort – more details

- Make `a[1...N]` into a heap
- So the biggest element is at `a[1]`
- Move `a[1]` to `a[N]`
  - by swapping `a[1]` and `a[N]`, but...
  - this may destroy heap property, so ...
  - ... use `downHeap` on `a[1...N-1]` to restore the heap property.
  - Now we have the 2nd largest element at `a[1]`
- ⋮

Is the general agenda similar to a sort we have already studied?

Heap Sort

```
heapSort(a[1...N])
/* PRECONDITION:  a[1...N] may not be sorted  */
/* POSTCONDITION: a[1...N] is sorted  */
{ int i, temp;
   /* First make a[1...N] into a heap */
   for (i = N/2; i >= 1; i--)
      downHeap(a, i, N);
  // a[1...N] is now a heap at this stage

  // iteratively remove biggest of a[1...i] and insert
  // ...correctly in a[i]
  for (i = N; i >  1; i--)
   { temp = a[i];
      a[i] = a[1]; // biggest of a[1...i]
      a[1] = temp;
      downHeap(a,1,i-1); // restore a[1..i-1] into a heap
   }
}/*heapSort*/
```

Merge Sort

```
merge_sort(a[...])}

   if (length of a[...] is "small") {
       //sort a[...] by some simple method
       //N.B. a single element is sorted
       ...
   }
   else {
       part1 = merge_sort( 1st half of a[...] )
       part2 = merge_sort( 2nd half of a[...] )

       //merge part1 and part2
       ...
   }
}
```

Top-down Merge sort, e.g. merge_sort([1 .. 8])

```
merge_sort [1 .. 4]
     merge_sort [1 .. 2]
          merge_sort [1...1] // trivial
          merge_sort [2...2] // trivial
          /* merge step */
          merge [1...1] and [2...2 ] into [1...2]
     merge_sort [3...4]
          merge_sort [3...3] // trivial
          merge_sort [4...4] // trivial
          /* merge step */
          merge [3...3] and [4...4] into [3...4]
     /* merge step */
     merge [1...2] and [3...4] into [1...4]
merge_sort([5...8])
    //. . . similarly . . .
    ...
merge [1...4] and [5...8] into [1...8]
```

Merge Sort Wrapper Routine

Wrapper routine

```
function mergeSort(int a[], int N)  /* wrapper routine */
/* NB sorts a[1..N] */

{ int i;
  int b[N];              /* -- the O(N) workspace */

  for(i=1; i <= N; i++)
     b[i]=a[i];          /* -- copy */

  merge(b, 1, N, a); /* -- does the real work . . . */
}
```

Merge Sort – routine where real work is done

```
function merge(int inpA[], int lo, int hi, int outA[])
/* sort (input) inpA[lo...hi] into (output) outA[lo...hi] */
{ int i, j, k, mid;

  if(hi > lo) /* at least 2 elements */
   { int mid = (lo+hi)/2;             /* lo <= mid < hi */

      merge(outA, lo,   mid, inpA);   /*   sort the ... */
      merge(outA, mid+1, hi, inpA);   /*    ... 2 halfs */

      /* and now merge them */
      i = lo;  j = mid+1;  k = lo;
      while( ... )
       {
         ... merge the sorted inpA[lo...mid] and inpA[mid+1...hi]
         ... into outA[lo...hi]
       }/*while */
   }/*if  */
}/*merge */
```

Time and Space complexity

Time Complexity

- Merging N elements takes $O(N)$-time
- There are $\log_2(N)$ levels of recursion
- N elements are merged at each level...
- ...therefore $O(N \log(N))$-time in total, always!

Space complexity

$O(N)$-space, for the extra **work-space** array

Stability

merge sort is stable (with care)

There is also a bottom-up merge sort, e.g. merge sort a[1...8]

```
copy a[ ] into b[ ]

section_length := 1
    merge b[1...1] and b[2..2] into a[1..2]
    merge b[3...3] and b[4..4] into a[3..4]
    merge b[5...5] and b[6..6] into a[5..6]
    merge b[7...7] and b[8..8] into a[7..8]

section_length := 2
    merge a[1..2] and a[3..4] into b[1..4]
    merge a[5..6] and a[7..8] into b[5..8]

section_length := 4
    merge b[1..4] and b[5..8] into a[1..8]
```

By the way! (For the curious among you)

- In-situ merging is possible in
  - O(1) extra space and
  - O(N)-time
- That algorithm is "difficult"
- For those of you who really want to push your learning:

Nicholas Pippenger "Sorting and Selecting in Rounds", Society for Industrial and Applied Mathematics Journal on Computing, 16, 1032 (1987).