Chapter 4 Arrays

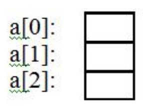
Ground Rules

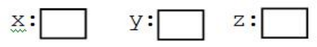
- Switch off your handphone and pager
- Switch off your laptop computer and keep it
- No talking while lecture is going on
- No gossiping while the lecture is going on
- Raise your hand if you have question to ask
- Be on time for lecture
- Be on time to come back from the recess break to continue the lecture
- Bring your lecturenotes to lecture

4.1 Introduction

- Arrays provide means for allocating and accessing memory.
- Instead of declaring variables with different identifiers, we can use the same identifier with indices to represent variables. For instance,
  we use **int a[3]**; to replace **int x, y, z;**
- E.g.

```
int a[3];            int x, y, z;
```

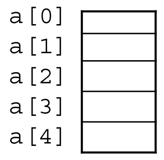




4.2 One-Dimensional Arrays

We declare an array in C by giving the name of the array and the type and number of elements. For example,

```
int a[5];
```



declares `a` as an array of five `int` objects.

Likewise,

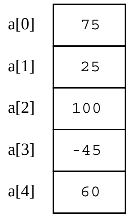
```
long double x[100];
```

declares `x` as an array of 100 `long double` objects:
`x[0], x[1],x[2], ..., x[99].`

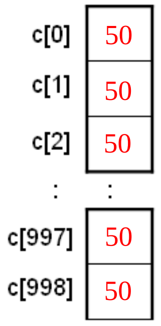
We can initialize an array when it is declared by listing the initial values for its elements.

For example:

int a[5] = {75, 25, 100, -45, 60};

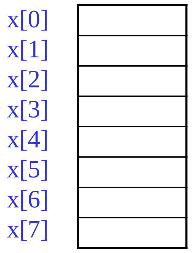


```
# define COUNT 999
# include <stdio.h>
main()
{
   int c[COUNT];
   int i;
   for (i=0;i <COUNT; i++) printf (" %d", c[i]);

   for (i=0;i <COUNT; i++) c[i]=50;

   getch(); // get a character from keyboard

   for (i=0;i <COUNT; i++) printf (" %d", c[i]);
   return 0;
}
```



```
/* demo44.c */
/* find the average and the difference w.r.t. average of 8 real numbers by the use of array */
# include <stdio.h>
int main (void)
{
   float x[8], average, sum;
   int i;
   printf ("Enter 8 real numbers >");
   i = 0;
   sum = 0;
   while (i < 8)
   {
      scanf ("%f", &x[i]);
      sum += x[i];  /* same as  sum += x[i++] */
      i++;
   }
   average = sum/8;
   printf ("\nThe average value is %.2f \n", average);
   printf ("Table of difference between x[i] and the average :\n");
   printf ("\n i     x[i]    x[i]-average");
   printf ("\n-------------------------\n");
   i=-1;
   while (++i < 8)
   {
     printf ("%2d.   %.2f      %6.2f\n", i,    x[i],   x[i]-average);
   }
   return 0;
}
```

```
Screen Output:
Enter 8 real numbers >10 20 30 40 50 60 70 80

The average value is 45.00

Table of difference between x[i] and the average :
 i     x[i]     x[i]-average
-------------------------
 0.   10.00      -35.00
 1.   20.00      -25.00
 2.   30.00      -15.00
 3.   40.00       -5.00
 4.   50.00        5.00
 5.   60.00       15.00
 6.   70.00       25.00
 7.   80.00       35.00
```



Having learnt array, you are ready to learn some algorithms (算法) used in programming. In this chapter you will learn 3 sorting algorithms, namely, selection sort, bubble sort and heap sort.

Sorting Problem

Given an array a, rearrange its elements so that they are in ascending or descending order.

Why should we study sorting ?

- Needed for arranging data for further analysis
  - Sorting the grades of students in a class

    Who will get the SM2 Book Prize ?
  - Sorting the sale of products in a company

    Which product should be abandoned ?
- As an initialization step to other algorithms
  - Pre-processing for Binary Search

Selection Sort

- Move the smallest element to `x[0]`.
- Move the second smallest element to `x[1]`.
- Move the third smallest element to `x[2]`.

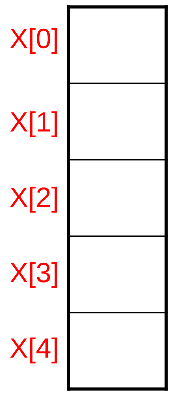
:

- Move the `(n-1)`-th smallest element to `x[n-2]`.

// selection sort, select.c

```
# include <stdio.h>
# include <stdlib.h>
# define n 50
# define seed 17

int x[n];

main ()
{
  int i, j, temp, index, min;
  long checksum;

  printf("\n\nSelection Sort:\n");
  srand(seed);                    // fix the seed
  for(i=0;i<n;i++) x[i]= rand() % 300;

  printf("\nBefore:");
  for(i=0;i<n;i++) printf(" %d ",x[i]);

  checksum = 0;
  for(i = 0; i <n; i++ ){ checksum +=x[i];}
  printf("\nChecksum before sorting: %ld\n", checksum);
```

```
for (i=0; i<n-1; i++)
{
   index=i;
   min = x[i];
   for (j=i+1; j<n; j++)   // n is the size of array
   {
     if (x[j] < min)
     {
       min = x[j];
       index = j;
     }
   }
   if (index !=i)
   {
     temp = x[i];
     x[i] = x[index];
     x[index] = temp;
   }
}
```



```
  printf("\nAfter:");
  for(i=0;i<n;i++) printf(" %d ",x[i]);

  checksum = 0;
  for(i = 0; i <n; i++ ){ checksum +=x[i];}
  printf("\nChecksum after selection sort: %ld \n", checksum);

  return 0;
}
```

Time Complexity Analysis of Selection Sort in the Worst Case in Term of the Number of Comparison Operations

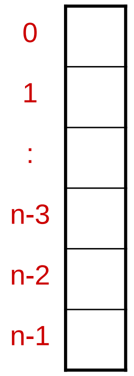
i	Number of Comparisons
0	n-1
1	n-2
2	n-3
:	:
n-3	2
n-2	1

$$\sum_{j=1}^{n-1} j = \frac{(n-1)(n)}{2} = \frac{1}{2}(n^2 - n)$$

$$= O(n^2)$$

Take the highest order and ignore the coefficient

```
for (i=0; i<n-1; i++)
{
    index=i;
    min = x[i];

    for (j=i+1; j<n; j++)   // n is the size of array
    {
       if (x[j] < min)
       {
         min = x[j];
         index = j;
       }
    }

    if (index !=i)
    {
      temp = x[i];
      x[i] = x[index];
      x[index] = temp;
    }
}
```

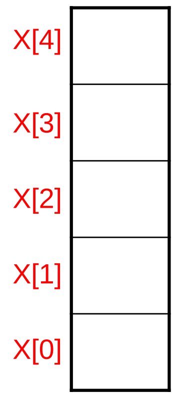


Bubble Sort

- Repeatedly sweep through array, exchanging adjacent pairs of elements that are out of order.
- Stop when one sweep does not produce any changes.
- Visualize the bubbles coming up a beaker of heated water.

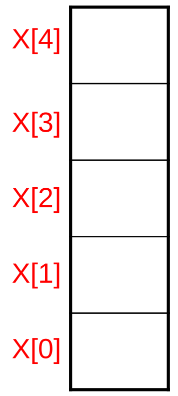
```
// bubble sort for array. In this version the height of comparison is the same for all loops
# include <stdio.h>       // This version is not optimal
# include <stdlib.h>      // bubble1.c
# define n 50
# define seed 29

int x[n];

main ()
{
  int i, exchange, temp;
  long checksum;

  printf("\n\nBubble Sort:\n\n");

  srand(seed);  // fix the seed
  for(i=0;i<n;i++) x[i]= rand() % 300;

  printf("\nBefore:");
  for(i=0;i<n;i++) printf(" %d ",x[i]);
  checksum = 0;
  for(i = 0; i <n; i++ ){ checksum +=x[i];}
  printf("\nChecksum before sorting: %ld\n", checksum);
```

```
do
{
  exchange=0;     // assume no exchange
  for (i=0; i<=n-2; i++)  // n is the size of array
  if (x[i] > x[i+1])
  {
    temp = x[i];
    x[i] = x[i+1];
    x[i+1] = temp;
    exchange=1;
  }
 } while (exchange==1);

printf("\nAfter:");
for(i=0;i<n;i++) printf(" %d ",x[i]);

checksum = 0;
for(i = 0; i <n; i++ ){ checksum +=x[i];}

return 0;
}
```

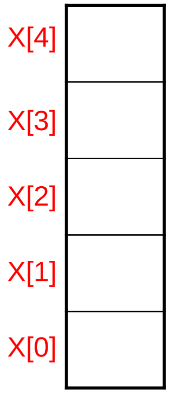


```
// bubble sort for array -  the height of next loop of comparison is shorter by 1 step
// The algorithm will stop as soon as the array is sorted.
// bubbl2a.c
# include <stdio.h>
# include <stdlib.h>
# define n 30
# define seed 29
int x[n];

main ()
{
   int i, height, exchange, temp;
   long checksum;
   int sorted;


   printf("\n\nBubble Sort:\n\n");
   srand(seed);  // fix the seed
   for(i=0;i<n;i++) x[i]= rand() % 300;
   printf("\nBefore:");
   for(i=0;i<n;i++) printf(" %d ",x[i]);
   checksum = 0;
   for(i = 0; i <n; i++ ){ checksum +=x[i];}
   printf("\nChecksum before sorting: %ld\n", checksum);
```

```
exchange=1;
for (height=n-2 ; height>=0 && exchange==1 ; height--)
{
   exchange=0; // assume no exchange
   for (i=0; i<=height; i++)  // rise the number up to height
   if (x[i] > x[i+1])
   {
     temp = x[i];
     x[i] = x[i+1];
     x[i+1] = temp;
     exchange=1;
   }
 }

printf("\nAfter:");
for(i=0;i<n;i++) printf(" %d ",x[i]);
checksum = 0;
for(i = 0; i <n; i++ ){ checksum +=x[i];}
printf("\nChecksum after sorting bubble1: %ld \n", checksum);
```



```
sorted = 1;
for(i = 0; i < n-1; i++ )
{
   if (x[i] > x[i+1])
   {
      sorted = 0;
      break;
   }
}

if (sorted ==1)
    printf ("\n sorted!");
else
    printf ("Not sorted");

return 0;
}
```



Time Complexity Analysis of Bubble Sort (with decreasing height) in the Worst Case in Term of the Number of Comparison Operations

The first bubble will go to the top (position n-1) and this takes n-1 comparisons.

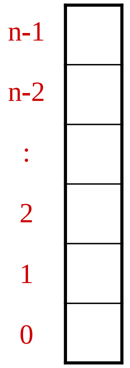
The second bubble will go to position n-2 and this takes n-2 comparisons.

The third bubble will go to position n-3 and this takes n-3 comparisons.

:

The second last bubble will go to position 2 and this takes 2 comparisons.

The last bubble will go to position 1 and this takes 1 comparison.



$$\sum_{j=1}^{n-1} j = \frac{(n-1)(n)}{2} = \frac{1}{2}(n^2 - n)$$

$$= O(n^2)$$

Decreasing height in subsequent iteration

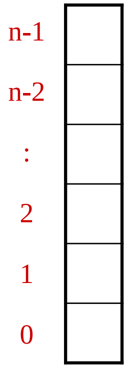
$$\sum_{j=1}^{n-1} (n-1) = (n-1)(n-1)$$

$$= n^2 - 2n + 1$$

$$= O(n^2)$$

Always bubble to the top in all iterations

What is the best time complexity of Bubble Sort to sort n data?



Structured Sorting Algorithm

Heap Sort

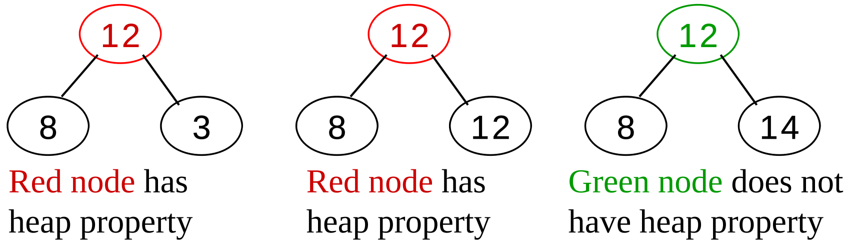
Why do we want a structure in sorting algorithm?

Although bubble sort is slow $O(n^2)$ in the worst case, it can be very fast $O(n)$ for some pre-orderings of data. For example, when the input data are not too messed up initially, bubble sort can stop in a few iterations. But this best performance cannot be guaranteed. We want guarantee in the time complexity for any pre-orderings of data because guarantee is critical in many real-time applications such as in the aircraft control system, aircraft navigation system etc.

Unless there is a structure to constrain or condition the algorithm, there is no way to give any guarantee in the performance. (Likewise in life.)

The heap property

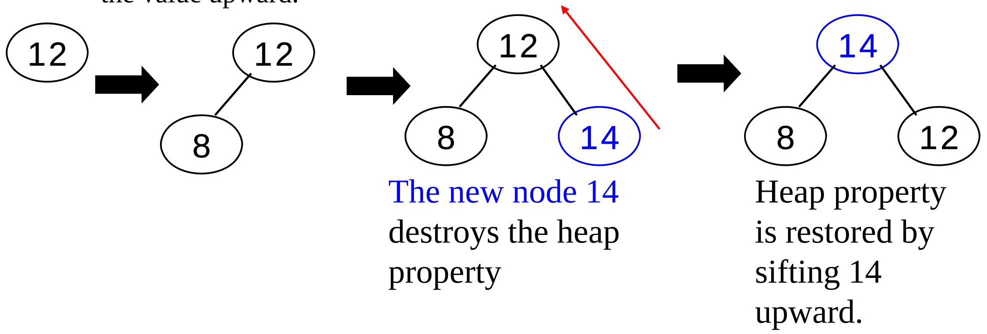
- A node has the heap property if the value in the node (father/parent) is as large as or larger than the values in its sons (children).


12
8
3
Red node has heap property
12
8
12
Red node has heap property
12
8
14
Green node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

Sifting Up

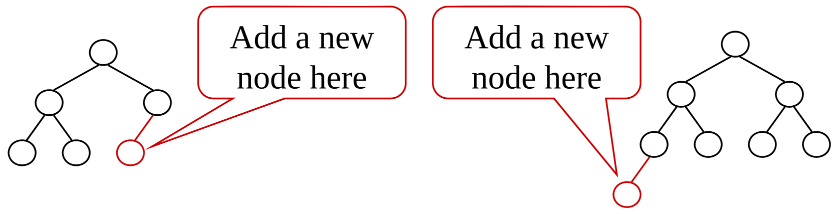
- Given a new node that destroys the heap property of a binary tree, you can restore the heap property by exchanging its value with its parent and cascade the value upward.


12
12
8
12
8
14
The new node 14 destroys the heap property
14
8
12
Heap property is restored by sifting 14 upward.

- This is called sifting up because the new value may have to go all the way up to the root.

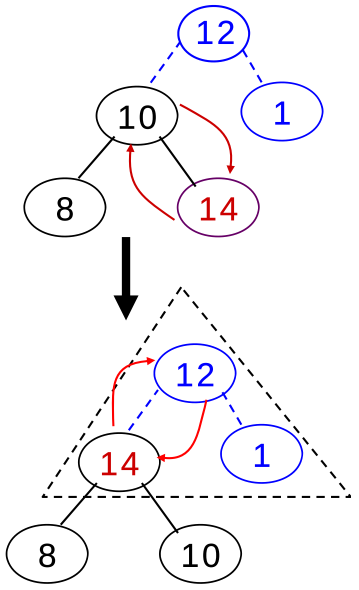
Constructing a heap I

- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:

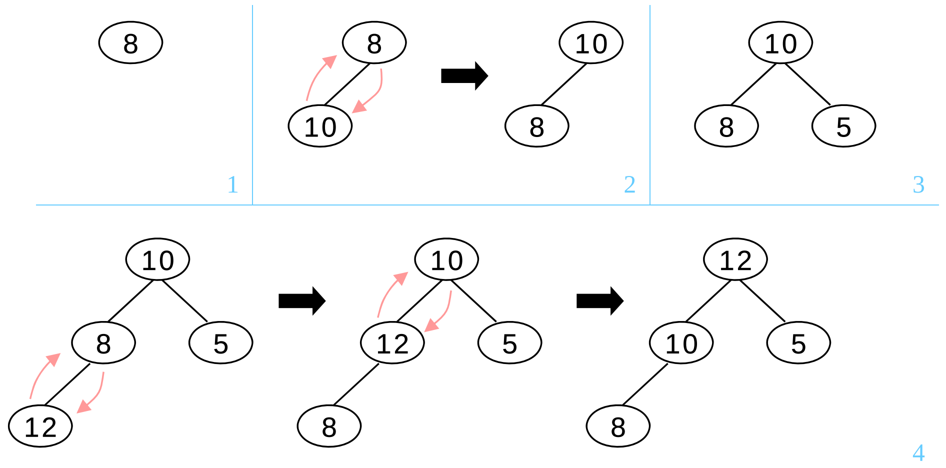


Constructing a heap II

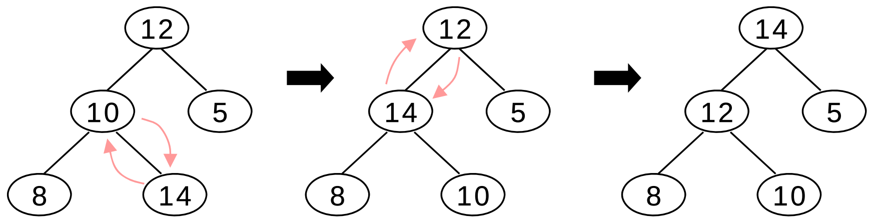
- Each time we add a node, we may destroy the heap property of its parent node because the value of the new node may be smaller than the value of the parent node.
- To fix this, we have to sift up (now it is coasting upward)
- But each time we sift up, the value of the top node in the sift may increase, and this may destroy the heap property of *its* parent node (i.e., the new value of the parent may be greater than the value of the grandparent)
- We repeat the sifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root of the binary tree.



Constructing a heap III



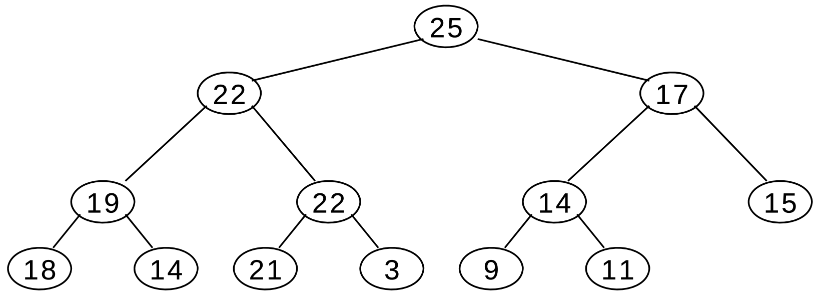
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

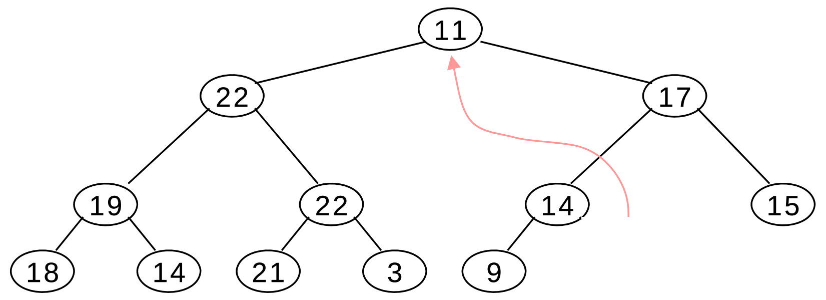
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root

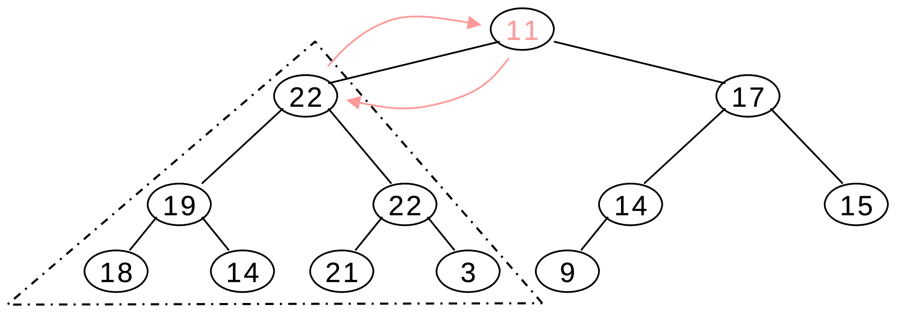
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The ***Re-Heap*** *method I*

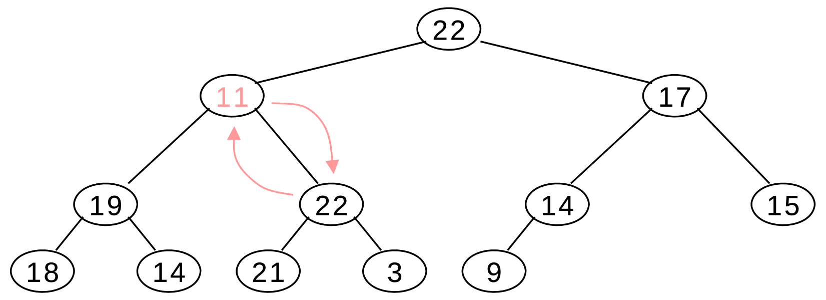
- Our tree is balanced and left-justified, but is no longer a heap
- However, *only the root* lacks the heap property



- We can sift down the root so that the heap property is restored.
- But after doing this, one of its children may have lost the heap property. If it is so, we continue to restore the heap property of the sub-tree downwardly.

The ***reHeap*** *method II*

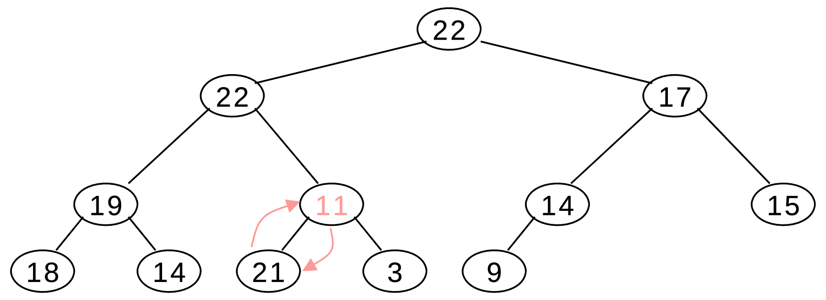
- Now the left child of the root (still the number `11`) lacks the heap property



- We can `sift down` this node
- After doing this, one and only one of its children may have lost the heap property

The ***reHeap*** *method III*

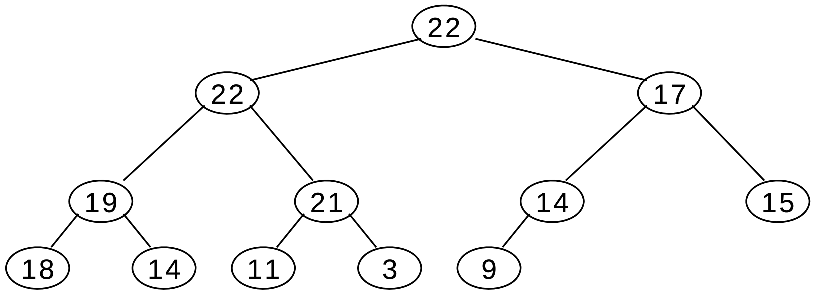
- Now the right child of the left child of the root (still the number `11`) lacks the heap property:



- We can `sift down` this node again
- After doing this, one and only one of its children may have lost the heap property — but it doesn't, because it's a leaf

The `reHeap` method IV

- Our tree is once again a heap, because every node in it has the heap property



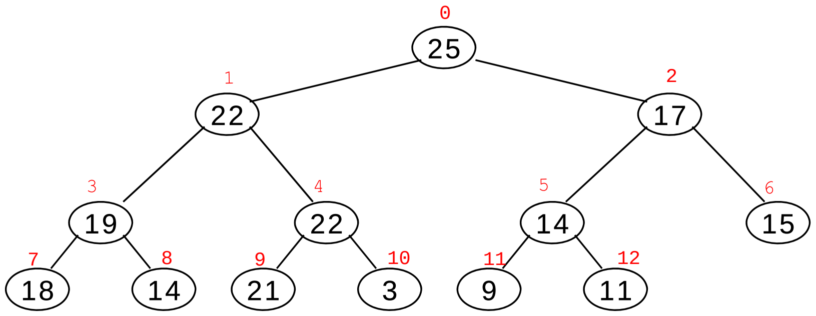
- Once again, the largest of the remaining values is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in the order from the largest to the smallest

Sorting

- What do heaps have to do with sorting an array?
- Here's the neat part:
  - Because the binary tree is *balanced* and *left justified*, it can be represented as an array
  - All our operations on binary trees can be represented as operations on *arrays*
  - To sort:

```
heapify the array;
while the array isn't empty
{
    remove and replace the root;
    reheap the new root node;
}
```

Mapping into an array

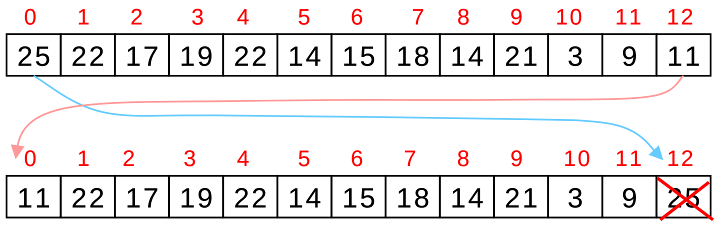


0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

- Notice:
  - The left child of index `i` is at index `2*i+1`
  - The right child of index `i` is at index `2*i+2`
  - Example: the children of node `3` (19) are `7` (18) and `8` (14)

Removing and replacing the root

- The "root" is the first element in the array
- The "rightmost node at the deepest level" is the last element
- Swap them...



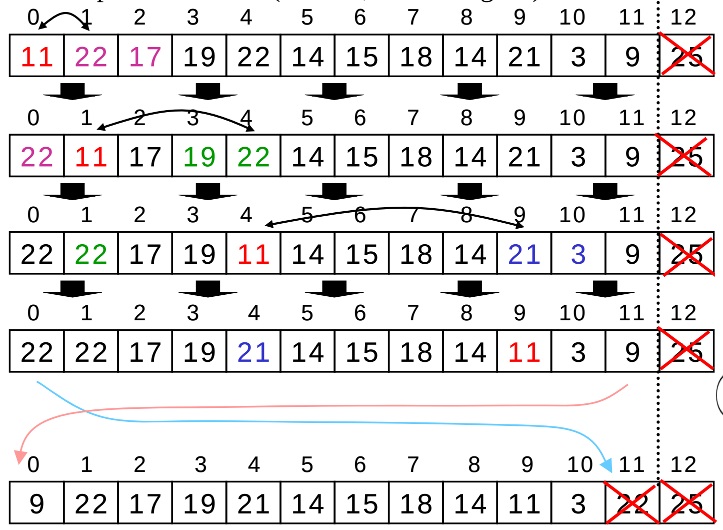
0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

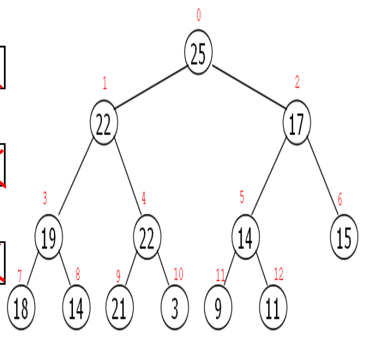
0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	~~25~~

- ...And pretend that the last element in the array no longer exists—that is, the "last index" is `11` (9)

Reheap and repeat

- Reheap the root node (index 0, containing 11)...



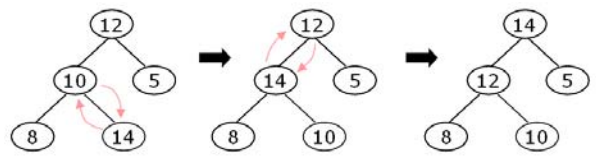


- ...And again, remove and replace the root node
- Remember, though, that the “last” array index is changed
- Repeat until the last becomes first, and the array is sorted!

```
/* Heap Sort       */
#include<stdio.h>
#include<stdlib.h>

# define N 100
# define BASE_START_INDEX  63
// pow(2,0) + pow (2,1) + ... + pow (2, (floor[lg(N)] -1) )
// 1 + 2 + 4 + 8 + 16 + 32 = 63

int x[N];

main()
{
  int i, done;
  int parent, this1, temp;
  long checksum;
  int max_index;
  int j, sorted;

  printf("\n\nHeap Sort:\n\n");
```

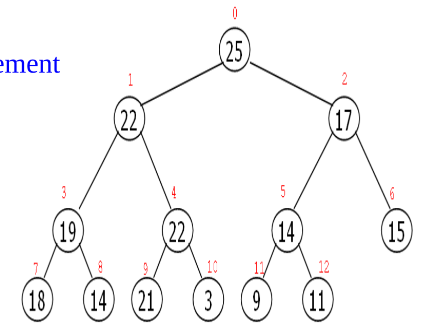
```
srand(17);  // fix the seed
for(i=0;i<N;i++) x[i]= rand() % 250;

printf("\nBefore:");
for(i=0;i<N;i++) printf(" %d ",x[i]);

checksum = 0;
for(i = 0; i <N; i++ ){ checksum +=x[i];}
printf("\nChecksum before sorting: %ld\n", checksum);
```

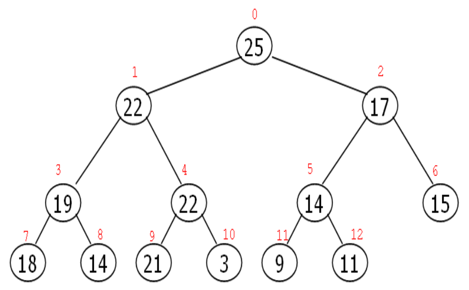
```
// Here I build the real heap.
 for (i=0;i<N;i++) // inspect the array element one by one
 {
   done = 0;
   this1=i;        // the element under investigation
                   // this1 refers to the child when sifting up
   while (!done)
   {
     if (this1==0)    //reach the root thus it is done
            done=1;
     else
     {
            // start sifting up
            parent = (this1-1)/2;
            if (x[parent] < x[this1])
              {
                temp = x[parent];    // exchange child with parent
                x[parent] = x[this1];
                x[this1] = temp;
                this1=parent;        // child is sifted up
              }
            else
              done =1;               // no exchange so sifting completed
     } //else
   } // while
 } // for i
```



```
printf ("\nAfter heapify : ");
for(j=0;j<N;j++) printf(" %d ",x[j]);

checksum = 0;
for(j = 0; j <N; j++ ){ checksum +=x[j];}
printf("\nChecksum after heapify: %ld\n", checksum);
```

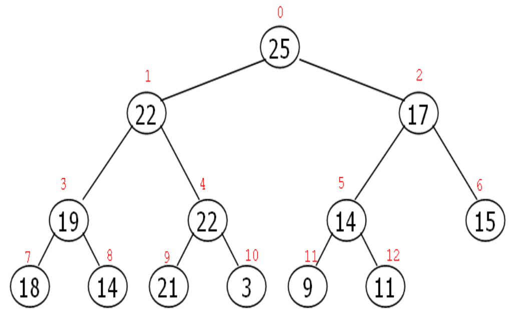
```
// insertion and heap restoration
 for(i=N-1;i>=1;i--)
 {
   temp =  x[0];
   x[0] = x[i];
   x[i] = temp;  // exchange the top with the current_last_element
                 // x[i] to x[n-1] are already committed

  // have to sift down because the top has been distrubed
   done = 0;
   this1 = 0;        // store from root
                     // this1 refers to the parent
                        when sifting down

   while (!done)
   {
     if (this1 >= BASE_START_INDEX)  //reach the bottom
          done=1;                     // thus it is done as it has no child
     else if ( (2*this1+1)>= i )     //if the left child is already committed
          done =1;                    // done bcos right child is also committed
```





```
else if ( (2*this1+2)>= i )
    {  // left child not committed, only right child is committed
       if (x[this1] < x[2*this1+1])
         {
              temp = x[this1];    // exchange parent with left child
              x[this1] = x[2*this1+1];
              x[2*this1+1] = temp;
              this1 = 2*this1+1;   // sifting down
         }
       else
         done=1;          // no exchange with left child, restoration is completed
    }
```

```
   else
        { // both children are not committed
          if (x[2*this1+1] > x[2*this1+2])
             max_index= 2*this1+1;
          else
             max_index = 2*this1+2;
          if (x[this1] < x[max_index])
            {
                 temp = x[this1];   // exchange parent with the max child
                 x[this1] = x[max_index];
                 x[max_index] = temp;
                 this1=max_index;   // sifting down
            }
          else
            done=1;     // no exchange thus heap restoration is complete
        }
  } // while
}  // for i
```



```
printf("\nEnd:");
for(i=0;i<N;i++) printf(" %d ",x[i]);

checksum = 0;
for(i = 0; i < N; i++ ){ checksum +=x[i];}
printf("\nChecksum after sorting: %ld\n", checksum);
```

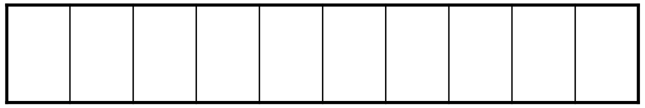
```
  sorted = 1;
  for (i = 0; i < N-1; i++ )
  {
    if (x[i] > x[i+1])
    {
      sorted = 0;
      break;
    }
  }

  if (sorted ==1)
     printf ("\n sorted!");
  else
     printf ("Not sorted");

  return 0;
}
```

```
Heap Sort:

Before: 137  198  22  162  119  28  152  139  89  145  3  123  130  133  80
167  204  209  123  104  8  44  134  159  212  244  106  189  223  92  11  196
17  105  165  189  17  84  182  84  71  146  158  9  80  64  135  94  182  60
86  48  177  0  40  105  7  100  234  86  91  161  91  93  55  234  249  68
135  23  221  87  83  113  117  128  2  118  103  173  77  245  114  162  40
76  197  231  112  220  96  213  11  189  138  168  232  218  91  27

Checksum before sorting: 12012
After heapify :  249  245  234  234  244  232  223  221  189  197  231  218
177  212  161  198  204  167  182  158  173  220  213  209  130  159  106  105
189  91  92  137  196  135  165  89  117  128  123  104  146  145  162  112
135  134  189  168  182  60  86  48  123  0  40  28  7  100  133  80  86  11
91  93  55  17  162  68  105  23  139  87  83  17  113  84  2  118  103  84  77
71  114  8  40  76  119  3  80  9  96  44  11  64  138  22  152  94  91  27
Checksum after heapify: 12012

End: 0  2  3  7  8  9  11  11  17  17  22  23  27  28  40  40  44  48  55  60
64  68  71  76  77  80  80  83  84  84  86  86  87  89  91  91  91  92  93  94
96  100  103  104  105  105  106  112  113  114  117  118  119  123  123  128
130  133  134  135  135  137  138  139  145  146  152  158  159  161  162  162
165  167  168  173  177  182  182  189  189  189  196  197  198  204  209  212
213  218  220  221  223  231  232  234  234  244  245  249
Checksum after sorting: 12012
 sorted!
```

What is the best time complexity and worst time complexity of Heap Sort to sort n data?

We will now learn how to search an array. The first algorithm will be the sequential search.

```
// search1.c    sequential search
# include <stdio.h>
# include <stdlib.h>
# define n 60
# define seed 29

int x[n];

main ()
{
  int search_key, found, i;
  srand(seed);      // fix the seed
  for(i=0;i<n;i++) x[i]= rand() % 150;

  printf("\nEnter a search key: ");
  scanf ("%d", &search_key);

  found = 0;
  for(i = 0; i <n; i++ )
  {
     if (x[i] == search_key)
     {
         found=1;
          break;
     }
  }

  if (found)
     printf ("\n %d is found in the array", search_key);
  else
     printf ("\n %d is not found in the array", search_key);

  printf("\nConents of array:\n");
  for(i=0;i<n;i++) printf(" %d ",x[i]);

  return 0;
}
```



Binary Search: The array will have to be sorted first.

```
// search2.c   binary search

# include <stdio.h>
# include <stdlib.h>

# define n 30
# define seed 29

int x[n];

main ()
{
  int i, height, exchange, temp;
  int search_key, found, left, right, mid;

  srand(seed);  // fix the seed
  for(i=0;i<n;i++) x[i]= rand() % 300;

  exchange=1;

for (height=n-2 ; height>=0 && exchange==1 ; height--)
{
   exchange=0; // assume no exchange

   for (i=0; i<=height; i++)  // rise the number up to height
   if (x[i] > x[i+1])
   {
     temp = x[i];
     x[i] = x[i+1];
     x[i+1] = temp;
     exchange=1;
   }
 }
```

```
  printf ("\nEnter a search key: ");
  scanf ("%d", &search_key);

  left=0;
  right= n-1;
  found=0;

  while (  (left <= right) && !found)
  {
     mid = (left+right)/2;
     if (x[mid] == search_key)
        found=1;
     else if (x[mid] < search_key) left = mid+1;
        else right = mid -1;
  }

  if (found)
     printf ("\n %d is found in the array", search_key);
  else
      printf ("\n %d is not found in the array", search_key);

  printf("\n\nContents of array:");
  for(i=0;i<n;i++) printf(" %d ",x[i]);

  return 0;
}
```

2	6	13	18	23	41	56	74	92	97