The University of Melbourne
School of Computing and Information Systems
COMP90038 Algorithms and Complexity

Assignment 2, Semester 2, 2017

Released: 23 September. Deadline: 13 October at 23:00

Objectives

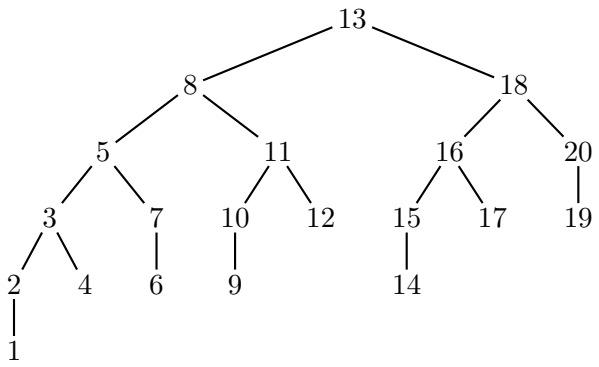
To improve your understanding of data structures and algorithms for sorting and search. To consolidate your knowledge of trees and tree-based algorithms. To develop problem-solving and design skills. To develop skills in analysis and formal reasoning about complex concepts. To improve written communication skills; in particular the ability to use pseudo-code and present algorithms clearly, precisely and unambiguously.

Five Challenges

Challenge 1 **(1 mark)**

Draw a valid AVL tree of height 5, consisting of the 20 integers from 1 to 20 (no repeats). Note that a tree of height 5 has some root-to-leaf path with *six* nodes along it.

ANSWER



For a AVL tree which contains 20 integers, it has the minimum height of 4 when it is a complete binary tree. To find the maxium height, we try to find the minimum nodes number given a specific height. So, if $n(h)$ means the minimum nodes number of height h, then

$$n(h) = n(h-1) + n(h-2) + 1$$

since we want to find the minimum nodes, so we want the heights of subtrees are as small as possible. Then we can find $n(5) = 20$, which means, to achieve height of 5, we need 20 nodes at least. So, 5 is the maxium height of 20 nodes AVL tree. To construct the tree, we obey the rule $n(h) = n(h-1) + n(h-2) + 1$, and after we get the structure, we could use inorder traverse to fit the integers into the structure.

Challenge 2 **(2 marks)**

Snarkbusters Inc. produces and deploys snark detectors, to be used on building facades. These are very expensive devices. One of the company's workers dropped a snark detector while working on the 16th floor of a building in Melbourne's CBD, and the device broke when it hit the ground, in spite of having a reputation as indestructible. The company now wants to know how tough their snark detector really is. They have hired a consultant, Dr. Eva Luator, to determine the largest floor level from which a snark detector can be dropped without breaking. That is, they want the number n such that dropping a snark detector from level n is safe (it does not break), but dropping it from level $n + 1$ is unsafe (it will break). They call that number n the "safe limit". They know that there is no problem when the device is dropped from ground level, that is, they know that $0 < n < 16$.

Since there are 15 floor levels to test, Dr. Luator asks for a handful of snark detectors to use in the experiment. Once a snark detector is broken it cannot be used again in the test. But the company will not give her that many. They argue that one is enough, because Dr. Luator can just drop it from level 1, and then, if it did not break, drop it from level 2, and so on. That process will identify the safe limit. Dr. Luator protests: that could make the testing too time consuming (and expensive) because, in the worst case, she would need to perform 15 experiments (drops). As a compromise, it is decided that Dr. Luator can have two snark detectors to use in her testing.

Eva Luator wants to make the most of the two snark detectors. She wants to minimise the number d of experiments (drops) that she needs to do, *in the worst case*, to determine the safe limit. What is that number d, and what is the testing strategy that achieves it?

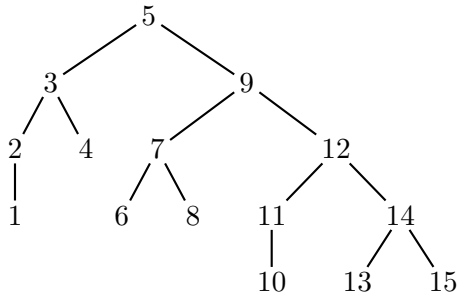
ANSWER

$d = 5$

To solve this problem, we could construct a model, which contains the ordered integers from 1 to 15, and a given float m, and our goal is to fit m into the ordered intergers, but we only have two chance to compare m with a bigger integer.

The most straightfoward idea is that we use binary search, and once meet the bigger integer, then we do a linear search on the small part from the lowest integer to the highest integer.

However, this will cost 8 comparisons in the worst case. The optimization is to balance the iterations of linear search and binary search. To achieve that, we construct a binary search tree shown as below. The tree is optimized since the number of nodes in the left subtree is equal to the height of right subtree, which give a minimum time of comparisons, 5.



Challenge 3 **(3 marks)**

Consider the following problem. We are given two arrays, $A[1]..A[n]$ and $B[1]..B[n]$, each holding positive integer keys (possibly with duplicates). The arrays are of the same size, and each is sorted. We are also given a positive integer s, and we want to find a pair (i, j) of indices such that $A[i] + B[j] = s$. If several such pairs exist, we do not care which is produced—any such pair will do. If no such pair exists then we want that indicated, in the form of $(0, 0)$, a non-valid pair of indices, being returned. Using neat, clean pseudo-code, write three different functions to solve the problem:

1. A brute-force function F1$(A[\cdot], B[\cdot], n, s)$. The worst-case running time of F1 should be $\Theta(n^2)$. Make F1 as simple and neat as you can.
   ANSWER

```
function F1(A[·], B[·], n, s)
    for i ← 1, n do
        for j ← 1, n do
            if A[i] + B[j] = s then
                return (i, j)
    return (0, 0)
```

2. A function F2$(A[\cdot], B[\cdot], n, s)$. The worst-case running time of F2 should be $\Theta(n \log n)$. Make F2 as simple and neat as you can.
   ANSWER

```
function F2(A[·], B[·], n, s)
    for i ← 1, n do
        low ← 1
        high ← n
        while low < high do
            m ← floor((high − low)/2) + low
            if A[i] + B[m] = s then
                return (i, m)
            else if A[i] + B[m] < s then
                low ← m + 1
            else
                high ← m − 1
    return (0, 0)
```

3. A function F3$(A[\cdot], B[\cdot], n, s)$. The worst-case running time of F3 should be $\Theta(n)$. Make F3 as simple and neat as you can.
   ANSWER

```
function F3(A[·], B[·], n, s)
    left ← 1
    right ← n
    while left ≤ n and right ≥ 1 do
        if A[left] + B[right] = s then
            return (left, right)
        else if A[i] + B[m] < s then
            left ← left + 1
        else
            right ← right − 1
    return (0, 0)
```

Challenge 4 **(2 marks)**

A d-dimensional vector $\langle u_1, u_2, \ldots, u_d \rangle$ *screens* the vector $\langle v_1, v_2, \ldots, v_d \rangle$ iff there is a permutation π on $\{1, 2, \ldots, d\}$ such that $u_1 > v_{\pi(1)}$, $u_2 > v_{\pi(2)}$, $\ldots$, $u_d > v_{\pi(d)}$. For example, $\langle 7, 9, 4 \rangle$ screens $\langle 7, 6, 3 \rangle$ because one permutation of the latter is $\langle 6, 7, 3 \rangle$. On the other hand, $\langle 7, 9, 4, 5 \rangle$ does not screen $\langle 7, 6, 3, 5 \rangle$. The last example also shows that we can have vectors **u** and **v**, neither of which screens the other.

Using pseudo-code, give an algorithm which determines, given two vectors **u** and **v**, whether **u** screens **v**. It should return True iff **u** screens **v**. If it helps, you an assume that vectors are arrays. Make the algorithm as efficient as you can, and state, as precisely as you can, its worst-case time complexity (and, if possible, average-case as well).

ANSWER

```
function DOESSCREEN(u[·], v[·], d)
    buildHeap(u[·], d)                                  ▷ build the max heap
    buildHeap(v[·], d)
    for i ← 1, d do
        if EJECT(u[·], d − i + 1) ≤ EJECT(v[·], d − i + 1) then
            return false
    return true
function BUILDHEAP(A[·], n)
    for i ← floor(n/2) downto 1 do
        RESTOREHEAP(A[·], i, n)
function EJECT(A[·], n)
    item ← A[1]
    A[1] ← A[n]
    A[n] ← item
    RESTOREHEAP(A[·], 1, n − 1)
    return item
function RESTOREHEAP(A[·], k, n)                        ▷ restore a heap at parent node k
    v ← A[k]
    heap ← false
    while not heap and 2 ∗ k ≤ n do
        j ← 2 ∗ k
        if j < n and A[j] < A[j + 1] then
            j ← j + 1
        if v ≥ A[j] then
            heap ← true
        else
            A[k] ← A[j]
            k ← j
    A[k] ← v
```

To judge whether the vector u screens the v, it could be the same problem of sorting two arrays(treat vector as array), and judge whether every element in one array are bigger than the corresponding element in the other array.

I use heap sort to achieve this goal. The worst case and average complexity are both $\Theta(n \log n)$, since building heap is $\Theta(n)$ and ejecting is $\Theta(n \log n)$ in the worst case. However, by using the heap, we do not need to sort the entire array, but restore it to a heap every time after ejecting, so the pratical performance could be better than other sorting algorithms.

Challenge 5 **(2 marks)**

An array-based implementation of priority queues is not ideal if we often need to "merge" several priority queues into one, or if we are programming in a language that does not have arrays. Here we are interested in a data structure that is *adaptive* and supports efficient merging, by which we mean taking two priority queues P and P' and forming a new priority queue with elements $P \cup P'$. As is often the case with adaptive data structures, some operations may be worst-case expensive, but when we average their cost across many applications, they are cheap. That will be the case for the operations we want to implement here: MERGE, INSERT, and DELETEMAX; all can be made to have $O(\log n)$ worst-case behaviour, amortised.

Your task is to reconstruct pseudo-code for the three operations (including some helper functions). The appendix contains a scrambled solution, with lines of pseudo-code in random order, without any indentation. You are asked to reconstruct the original code, with proper indentation.

The data structure we use is a kind of *multiway tree*, that is, a node can have any number of subtrees. Since it is a multiway tree and also a (max-)*heap*, we will call it a *meap*. A meap M can be empty, in which case we denote it **void**. If M is non-empty then M has a key, plus a list of non-empty sub-trees (sub-meaps). Note that the list of subtrees can be of any length; in particular it can be empty.

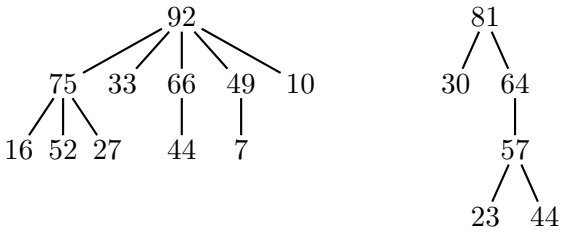
A *tree list* L can be empty, in which case we denote it **null**. If L is non-empty, then L has a first tree, $L.elt$ and a tail, $L.next$, with the remaining trees. In our case, these trees are all meaps.

A meap M (other than **void**) has two attributes:

- $M.key$ is the root node's key.
- $M.subtrees$ is a tree list—the list of M's sub-meaps.

A meap must satisfy the *heap condition*: The key of a child node is smaller than the key of its parent. That is, for every meap M' in $M.subtrees$, we have $M'.key \leq M.key$. The heap condition must hold recursively, for all nodes.

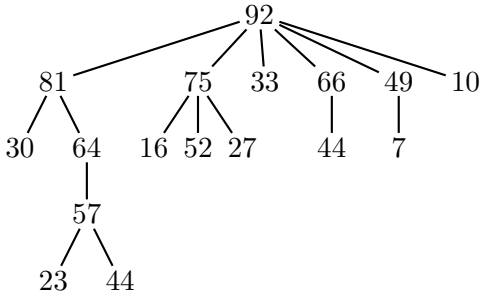
Below are two meaps, M_1 (on the left) and M_2 (on the right):



Here $M_1.key = 92$ and $M_1.subtrees$ is a list of five children. These children are meaps with roots 75, 33, 66, 49, and 10.

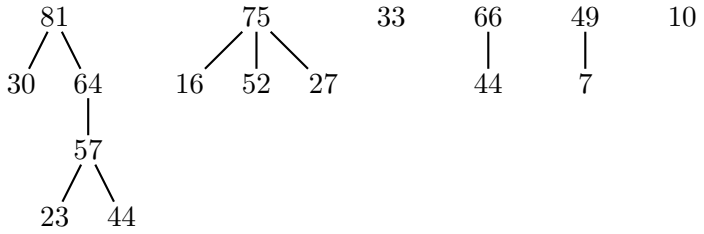
To find and return the largest key in a meap is easy—it is just the root's key; we will not be worried about that operation in this question.

To merge two meaps, we include the meap that has the smaller root key as the leftmost subtree of the other meap. Merging the two meaps above, we get MERGE(M_1, M_2), shown here:

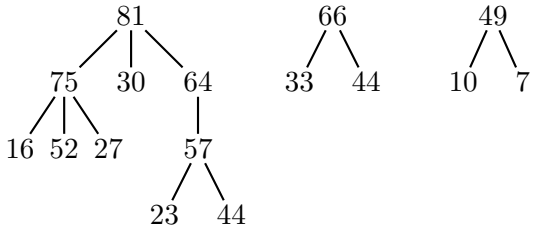


Once MERGE has been implemented, INSERT is easy. DELETEMAX removes the largest element from a meap. It also makes use of MERGE but is more complicated. After discarding the root it must merge the children, to return a single meap. We do the merging in two passes. First we merge the children in pairs from left to right (that is, the first child with the second, the third with the fourth, and so on). Then we merge the resulting meaps in a single sweep from right to left.

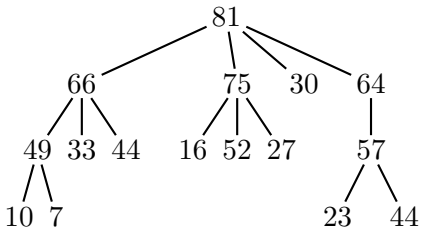
Continuing our example, removal of the root (92) leaves us with six meaps to merge:



Merging pairwise, from left to right, we get:



Now, merging the resulting meaps in a single pass from right to left, we get:



Now have fun re-assembling the code from the appendix.

ANSWER

```
function AddToList(M, L)
    R ← new TreeList
    R.elt ← M
    R.next ← L
    return R
function Merge(M_1, M_2)
    if M_1 = void then
        return M_2
    if M_2 = void then
        return M_1
    M ← new MultiwayTree
    if M_1.key > M_2.key then
        M.key ← M_1.key
        M.subtrees ← AddToList(M_2, M_1.subtrees)
    else
        M.key ← M_2.key
        M.subtrees ← AddToList(M_1, M_2.subtrees)
    return M
function Insert(x, M)
    M' ← new MultiwayTree
    M'.key ← x
    M'.subtrees ← null
    return Merge(M', M)
function MergePairs(L)
    if L = null then
        return void
    M_1 ← L.elt
    L_1 ← L.next
    if L_1 = null then
        return M_1
    M_2 ← L_1.elt
    L_2 ← L_1.next
    return Merge(Merge(M_1, M_2), MergePairs(L_2))
function DeleteMax(M)
    if M = void then
        Error("Cannot delete from empty tree")
    return MergePairs(M.subtrees)
```

Tiancong Li
27 September 2017

Appendix: Scrambled solution for Question 5

else
ERROR("Cannot delete from empty tree")
function ADDTOLIST(M, L) : *TreeList*
function DELETEMAX(M) : *MultiwayTree*
function INSERT(x, M) : *MultiwayTree*
function MERGE(M_1, M_2) : *MultiwayTree*
function MERGEPAIRS(L) : *MultiwayTree*
if $L =$ **null then**
if $L_1 =$ **null then**
if $M =$ **void then**
if $M_1 =$ **void then**
if $M_1.key > M_2.key$ **then**
if $M_2 =$ **void then**
$L_1 \leftarrow L.next$
$L_2 \leftarrow L_1.next$
$M \leftarrow$ **new** *MultiwayTree*
$M.key \leftarrow M_1.key$
$M.key \leftarrow M_2.key$
$M.subtrees \leftarrow$ ADDTOLIST($M_1, M_2.subtrees$)
$M.subtrees \leftarrow$ ADDTOLIST($M_2, M_1.subtrees$)
$M' \leftarrow$ **new** *MultiwayTree*
$M'.key \leftarrow x$
$M'.subtrees \leftarrow$ **null**
$M_1 \leftarrow L.elt$
$M_2 \leftarrow L_1.elt$
$R \leftarrow$ **new** *TreeList*
$R.elt \leftarrow M$
$R.next \leftarrow L$
return M
return M_1
return M_1
return M_2
return MERGE(M', M)
return MERGE(MERGE(M_1, M_2), MERGEPAIRS(L_2))
return MERGEPAIRS($M.subtrees$)
return R
return void