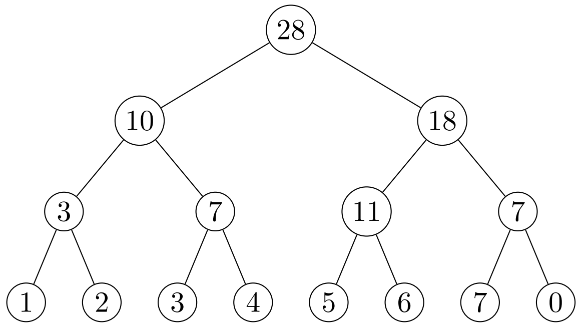
CSC265: Assignment 1

Ruo yi Lin and Junjie Cheng

October 2, 2016

1. (a) Describe in detail a data structure implementing the Partial-Sums ADT. Draw an example with $n = 8$, and values $a_1, \ldots, a_8$ of your choosing, but not $a_1 = \ldots = a_8 = 0$.

Solution: The correct data structure is a full binary tree of height k, B_k. There are $2^k = n$ leaves in the lowest row of this tree. The ith leaf holds value a_i. The values of the parent nodes are defined recursively as the sum of their two children. An example is below with $a_1, \ldots, a_n = 1, 2, 3, 4, 5, 6, 7, 0$.



(b) Describe in clear and precise English how to implement the three operations INIT, ADD(i, t), and SUM(i). Give pseudocode for ADD(i, t), and SUM(i). For each operation argue why it computes the correct output, and show that it runs in the required worst-case running time.

- INIT must run in time $O(n)$
- ADD(i, t) must run in time $O(\log(n))$
- SUM(i) must run in time $O(\log(n))$

Solution:

INIT: Since $n = 2^k$, the binary tree required by the Partial-Sums ADT will always be full. Then the tree can be easily implemented as an array of size $2^{k+1} - 1$, with initial values all set to 0. We assume that the first index of the array is 1.

There is no need to construct pointers, since for a complete binary tree there are simple closed-form expressions for parent and child nodes. The functions are reproduced from CLRS Section 6.1.

Then INIT runs in time $O(2^{k+1} - 1) = O(2(2^k)) = O(2n) = O(n)$.

```
PARENT($i$)
1  return $\lfloor i/2 \rfloor$
```

```
LEFT-CHILD($i$)
1  return $2i$
```

```
RIGHT-CHILD($i$)
1  return $2i + 1$
```

ADD(i, t): Add is simple to implement. The ith leaf of the tree is given by index $2^k + i$. We add t to this leaf, and then add t to each of i's parents, grandparents, great-grandparents ... until we reach the root.

```
ADD($i, t$)
1  node_index $= 2^k + i$
2  while node_index $\neq 0$
3          node_index.value = node_index.value $+ t$
4          node_index = PARENT(node_index)
```

The resultant tree has an updated leaf and satisfies the sum property specified in INIT. This is because only the parents of the leaf specified by i will have their values affected.

ADD(i, t) runs in $O(\log(n))$. One iteration of the while loop is executed for each level of the tree, so the algorithm is proportional to the tree height. This in turn is proportional to the log of the number of nodes in the tree, which is proportional to the number of leaves.

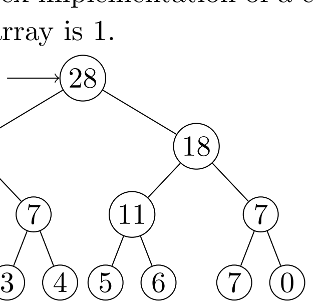
We may reach the same conclusion by noticing that node_index is proportional to n and that it is halved on each iteration when PARENT is called.

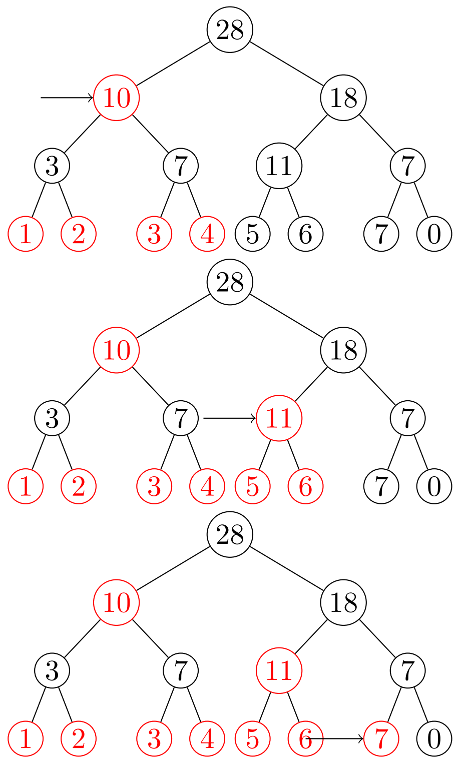
SUM(i): Notice that for any node N at depth h of the tree, there are 2^{n-h} descendants of N which are leaves. We can show by induction that the value contained in N is equal to the sum of its leaf descendants. This is key to our algorithm. We start by initializing a running sum variable, x, at zero. Then we traverse the tree downwards from the root, choosing at each depth h at most one node N. We add the value of N to our running sum x. Our choice of N is based on the value of i, so that

1. No value a_j $(1 \leq j \leq n)$ is added more than once
2. The algorithm terminates once x contains the sum $a_1, \ldots, a_i$

Since we can represent i uniquely in binary, we can decompose it into a unique sum of these "power of two" nodes. Since the algorithm chooses and adds only a single node at each depth h of a tree, it runs in $O(\log(n))$ time.

An example of the algorithm is illustrated with $i = 7, n = 8$. Our pseudocode is less intuitive, because it uses the array-index implementation of a complete binary tree (given in CLRS 6.1). The first index in the array is 1.





In the pseudocode, $i \text{div} c = \lfloor i/c \rfloor$ and $i \bmod c$ is the usual modulus function. value(a) is a function which returns the value of the node with index a.

```
SUM($i$)
a = 1
b = i
c = n
x = 0
while $b > 0$
    if ($\lfloor b/c \rfloor = 1$)
        $b = b - c$
        $x = x + \text{value}(a)$
        $a = 2a + 2$
    else
        $a = 2a$
    $c = \lfloor c/2 \rfloor$
return $x$
```

Proof of correctness. The following lemma essentially states that the value held by a node is equal to the sum of the values of its leaf descendants.

Lemma 1. *If the index of a node N in an array-implementation of a* Sum *complete binary tree of height k is $2^h + l$, then* N.value *is*

$$\sum_{j=(2^{k-h})l+1}^{(2^{k-h})l+2^{k-h}} a_j.$$

Proof. We fix the size of the tree, k, and proceed by induction *up* the levels of the tree.

Base case: $h = k$. Fix l. Then N is a leaf and

$$\sum_{j=(2^{k-h})l+1}^{(2^{k-h})l+2^{k-h}} a_j = \sum_{j=l+1}^{l+1} a_j = a_{l+1} = \text{N.value}$$

So the base case holds. Note $l = 0$ for the first node at depth h in the tree.

Inductive Case: Suppose that the lemma holds for all nodes at depth $h+1$ of the tree. We show that it holds for nodes at depth h. Fix l.

By construction, N.value = N.left.value + N.right.value. The index of N.left is $2(2^h + l) = 2^{h+1} + 2l$ and the index of N.right is $2(2^h + l) + 1 = 2^{h+1} + 2l + 1$. We apply the induction hypothesis on the children of N to obtain

$$\begin{aligned}
\text{N.value} &= \sum_{j=(2^{k-(h+1)})(2l)+1}^{(2^{k-(h+1)})(2l)+2^{k-(h+1)}} a_j + \sum_{j=(2^{k-(h+1)})(2l+1)+1}^{(2^{k-(h+1)})(2l+1)+2^{k-(h+1)}} a_j \\
&= \sum_{j=(2^{k-h})l+1}^{(2^{k-h})l+2^{k-(h+1)}} a_j + \sum_{j=(2^{k-h})l+2^{k-h}+1}^{(2^{k-h})l+(2^{k-(h+1)})+2^{k-(h+1)}} a_j \\
&= \sum_{j=(2^{k-h})l+1}^{(2^{k-h})l+2^{k-(h+1)}} a_j + \sum_{j=(2^{k-h})l+2^{k-h}+1}^{(2^{k-h})l+2^{k-h}} a_j \\
&= \sum_{j=(2^{k-h})l+1}^{(2^{k-h})l+2^{k-h}} a_j
\end{aligned}$$

So the lemma holds. (Whew!) □

We now list and prove a few loop invariants. Let h be the height of the tree and denote by b_p, c_p, x_p the values of the variables b, c, x after the pth iteration of the while loop at line 4. If $i = \sum_{j=1}^{h} \phi_j 2^j$, then we have

$$\begin{aligned}
b_p &= \sum_{j=p+1}^{h} \phi_j 2^j \\
c_p &= n/2^p \\
x_p &= \sum_{j=1}^{b_p} a_j.
\end{aligned}$$

Lemma 2. *After the pth iteration of the while loop on line 4,*

(c) For each $n = 2^k$, and each setting of $a_1, \ldots, a_n$, give an input i, $1 \le i \le n$, such that your implementation of SUM(i) runs in time $\Omega(\log(n))$. Explain clearly and precisely why the running time is $\Omega(\log(n))$ on this input.

Solution:

2. (a) Give an algorithm for UNION(H_1, H_2) running in worst-case time complexity $O(r_1.d+r_2.d)$ where H_1 and H_2 are R-HEAP structures, r_1 is the root of H_1, r_2 is the root of H_2, and H_1 and H_2 are given to the algorithm by pointers to the roots r_1 and r_2, respectively.

   UNION(H_1, H_2) must return a pointer to the root of an R-HEAP that contains the union of the elements of H_1 and H_2. Describe your algorithm in clear in precise English, and give pseudocode. Show that the running time of the algorithm is $O(r_1.d + r_2.d)$.

   Solution:

   [Insert algorithm here.]

```
UNION($H_1, H_2$)
if ($r_1$ = NIL)
    return $r_2$
elseif ($r_2$ = NIL)
    return $r_1$
else
    if ($r_1.value < r_2.value$)
        small = $r_1$
        big = $r_2$
    else
        small = $r_2$
        big = $r_1$
    new_child = UNION(small.left, biggest)
    if small.right = NIL or small.right.d ¡ small.left.d
    small.left = small.right
    $small.right$ = new_child
else
    small.left = new_child
    return smallest
```

   Partial Correctness

   Assume that the union operation

   Proof of Termination.

   Loop invariant: the algorithm runs in the time complexity ...

   base case

   (b) **Solution:**

   (c) Recall that the height of a tree T with root r equals the number of edges in the longest path from r to a leaf of T. What is the largest possible height of an R-HEAP with n nodes?

   Solution: The largest possible height is $n - 1$.

   Note that in general the largest possible height for a binary tree of n nodes is given by $n-1$: each node in the tree has exactly one child. To construct such a tree which satisfies the R-HEAP properties,