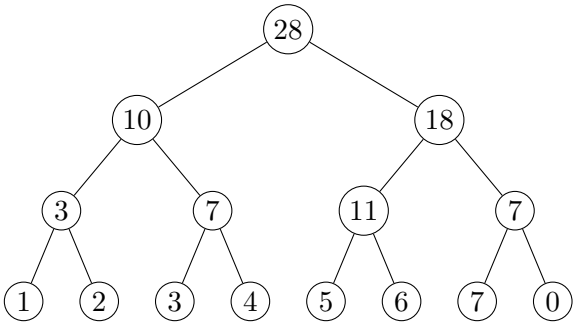
CSC265: Assignment 1

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 ≠ 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 Q1(a). 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. By induction, the value of N is equal to the sum of its leaf descendants. This is key to our algorithm, because we can decompose the sum $\sum_{j=1}^{i} a_j$ into a sum of groups of powers of two. This binary representation of i is unique.
For example, if $i = 7$, we may write $\sum_{j=1}^{i} a_j$ as $\sum_{j=1}^{2^2} a_j + \sum_{j=2^2+1}^{2^2+2^1} a_j + \sum_{j=2^2+2^1+1}^{2^2+2^1+2^0} a_j$. This first sum is a group of 4, the second a group of 2, and the third a group of 1. This corresponds to adding certain nodes on each level of the binary tree.

SUM(i)

```
 1  d = 1
 2  b = i
 3  c = n
 4  x = 0
 5  while b > 0
 6        if (⌊b/c⌋ = 1)
 7              b = b − c
 8              x = x + value(d)
 9              d = 2d + 2
10        else
11              d = 2d
12        c = ⌊c/2⌋
13  return x
```

Note value(d) is a function which returns the value of the node with index d. The pseu-

docode is difficult to parse, so we also illustrate the procedure graphically below.

Only a maximum of one node is added at each level of the tree, so the procedure runs in time proportional to the tree height – $O(\log(n))$.

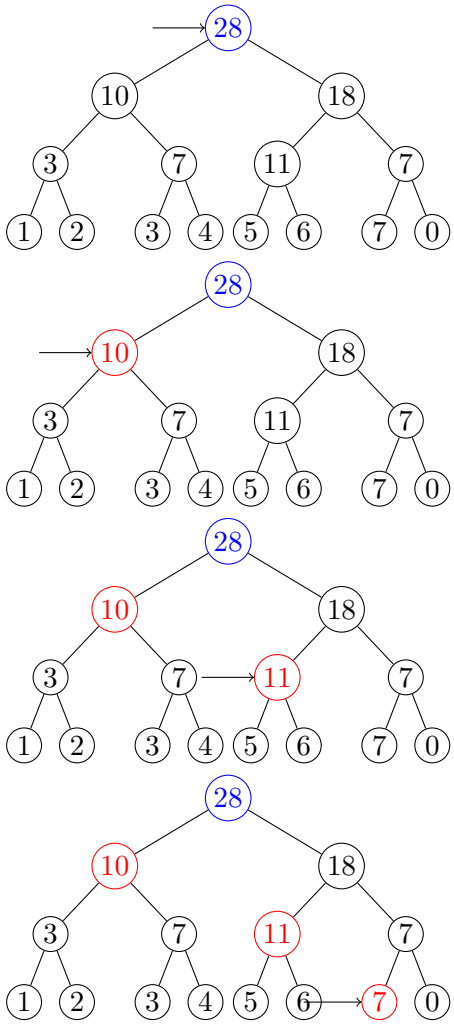


Figure 1: Procedure Sum(i) on the tree in Q1(a) with $i = 7$. The procedure begins at the root of top of the tree. The arrow represents the node being considered. If the node is red, it is added to the running sum x. If it is blue, it has been visited but has not been added.

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! □

Termination. To prove

Proposition 1. *Write i as $\sum_{j=1}^{k} \phi_j 2^j$. Then the while loop in* SUM(i) *terminates, and does so after the $(k-h)$th iteration, where $h = \{\min(j) | \phi_j = 1\}$.*

Proof. Denote by b_p, c_p the values of the variables b, c after the pth iteration of the while loop. We quickly prove two loop invariants:

$$\begin{aligned}
b_p &= \sum_{j=1}^{k-p} \phi_j 2^j \\
c_p &= 2^{k-p}
\end{aligned}$$

c_p: $c_0 = n = 2^k$, so the base case holds. Inductively, we have $c_p = c_{p-1}/2 = 2^{k-(p-1)} \div 2 = 2^{n-p}$.

b_p: Before the start of the first iteration, $b = b_0 = i$, so the loop invariant holds. Assume that the loop invariant holds on the $p-1$th iteration. Then $b_{p-1} = \sum_{j=1}^{k-(p-1)} \phi_j 2^j$. Notice that $c_{p-1} = 2^{k-(p-1)}$. So $\lfloor b/c \rfloor = 1$ iff $\phi_{k-(p-1)} = 1$. Then $b_p = \sum_{j=1}^{k-p} \phi_j 2^j$ in either branch of the if/else block, so the loop invariant holds.

In particular, $b_{k-h+1} = \sum_{j=1}^{k-(k-h+1)} \phi_j 2^j = \sum_{j=1}^{h-1} \phi_j 2^j = 0$, by construction of h. So the loop quits after the $(k-h)$th iteration of the while loop, and the algorithm exits.

□

Corollary 1. *The worst case running time of* SUM(i) *is* $\mathrm{O}(\log(n))$.

Proof. Choose $i = n - 1 = 2^k - 1 = \sum j = 1k - 12^j$. Then $h = 1$ and the while loop quits only after $k - 1$ iterations. But $k \in \mathrm{O}(\log(n))$. □

Proposition 2. SUM(i) *is partially correct; i.e. it returns* $\sum_{j=1}^{i} a_j$.

Proof. Again, we begin by proving some loop invariants. We have

$$\text{value}(d_p) = \sum_{j=\psi_{p-1}+1}^{2^{k-h}+\psi_{p-1}} a_j$$

$$x_p = \sum_{j=1}^{\psi_{p-1}} a_j$$

$$\texttt{where } \psi_p = \sum_{\alpha=k-p}^{k} \phi_\alpha 2^\alpha$$

value(d_p). Base case: before the start of the loop, value$(d_0) = \sum_{j=1}^{2^k} a_j$ by the lemma. We have $\psi_{-1} = 0$, so the theorem holds.

Inductive case: Suppose the loop invariant holds for value(d_{p-1}).

Now $d_{p-1} = 2^{k-(p-1)} + l$ for some $l, 1 \leq l \leq 2^{k-(p-1)}$. By the induction hypothesis, we can equate the lower bounds on the summations in the lemma and loop invariant. This allows us to solve for l:

$$2^{k-(p-1)} l + 1 = \psi_{p-2} + 1$$

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

$$l = \frac{\sum_{j=k-(p-2)}^{k} a_j}{2^{k-(p-1)}}$$

$$l = \sum_{j=1)}^{p-1} a_j$$

Then there are two cases.

Case 1 $\lfloor b_{p-1}/c_{p-1} \rfloor = 1$. Notice $\phi_{p-1} = 1$ and in particular $\psi_{p-1} = 2^{k-(p-1)} + \psi_{p-2}$.

Write d_p as $2^{k-p} + l'$. Now $d_p = 2d_{p-1} + 2$ so that $l' = 2l + 2$. Using the lemma, we have

$$\text{value}(d_p) = \sum_{j=(2^{k-p})l'+1}^{(2^{k-p})l'+2^{k-p}} a_j$$

The lower bound resolves to

$$\begin{aligned}
&(2^{k-p})l' + 1 \\
&= (2^{k-p})(2l+2) + 1 \\
&= (2^{k-p})(2\sum_{j=1)}^{p-1} a_j + 2) + 1 \\
&= \sum_{j=k-(p-2)}^{k} a_j + 2^{k-(p-1)} + 1 \\
&= \psi_{p-2} + 2^{k-(p-1)} + 1 \\
&= \psi_{p-1} + 1
\end{aligned}$$

Similarly, the upper bound resolves to $2^{k-p} + \psi_{p-1}$.

Case 2 $\lfloor b_{p-1}/c_{p-1} \rfloor = 0$. Then $\phi_{p-1} = 0$ so that $\psi_{p-1} = \psi_{p-2}$. Also, we have $d_p = 2d_{p-1}$ so that $l' = 2l$. Following the same sequence of calculations as in case 1, we see that the loop invariant holds.

x_p:

Base case: Since $\psi_0 = 0$ we have $\sum_{j=1}^{0} a_j = 0 = x_0$, as desired.

Assume that the loop invariant holds before the $(p-1)$th iteration. We show it holds before the pth iteration.

Consider $x_{p-1} = \sum_{j=1}^{\psi_{p-1}} a_j$.

Case 1. If $\lfloor b_{p-1}/c_{p-1} \rfloor = 1$ then

$$\begin{aligned}
x_p &= x_{p-1} + \text{value}(d_{p-1}) \\
&= \sum_{j=1}^{\psi_{p-2}} a_j + \sum_{j=\psi_{p-2}+1}^{2^{k-h}+\psi_{p-2}} a_j \\
&= \sum_{j=1}^{2^{k-h}+\psi_{p-2}} a_j \\
&= \sum_{j=1}^{\psi_{p-1}} a_j
\end{aligned}$$

The second step uses the inductive hypothesis. The last step uses the fact that $\phi_{p-1} = 1$ in this case.

Case 2. If $\lfloor b_{p-1}/c_{p-1} \rfloor = 0$ then $\phi_{p-1} = 1$ so that $\psi_{p-1} = \psi_{p-2}$. Then $x_p = x_{p-1} = \sum_{j=1}^{\psi_{p-2}} a_j = \sum_{j=1}^{\psi_{p-1}} a_j$, so the loop invariant holds.

Now let's consider the value of x when the loop terminates. We showed that this occurs before the $(k-h+1)$th iteration. Then $psi_{k-h} = \sum_{\alpha=k-(k-h)}^{k} \phi_\alpha = \sum_{\alpha=h}^{k} \phi_\alpha = \sum_{\alpha=1}^{k} \phi_\alpha 2^\alpha = i$ (the last step follows from construction of h). But this means $x_{k-h+1} = \sum_{j=1}^{\psi_{(k-h+1)-1}} a_j = \sum_{j=1}^{i} a_j$. So SUM($i$) is partially correct.

□

The total correctness of SUM(i) follows.

Theorem 1. SUM(i) *is correct.*

(c) For each $n = 2^k$, and each setting of $a_1, \ldots, a_n$, give an input i, $1 \leq i \leq n$, 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: Choose $i = 2^k - 1$. Refer to Corollary 1 for a detailed explanation why the run time is $\Omega(\log(n))$ on this input.

Notice that our implementation would (oddly enough) also be $\Omega(\log(n))$ for $i = 1$. To be precise, it is $\Omega(\log(n))$ on all odd inputs.

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)

```
1  if (r1 == NIL)
2      return r2
3  elseif (r2 == NIL)
4      return r1
5  else
6      if (r1.value < r2.value)
7          small = r1
8          large = r2
9      else
10         small = r2
11         large = r1
12     new_child = UNION(small.left, large)
13     new_child.parent = small
14     if new_child.left and new_child.right
15         new_child.d = 1 + min{new_child.left.d, new_child.right.d}
16     else
17         new_child.d = 1
18     if small.right == NIL or small.right.d < small.left.d
19         small.left = small.right
20         small.right = new_child
```

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, make all the children the right-children of the tree, and make all key values equal. More formally, if we want to construct T_n, take a node N with key 1, and set its right child to T_{n-1}. T_1 is the tree with a single node with its key set to 1. Then this tree is an R-HEAP which (trivially) satisfies the min-heap and R-balance properties.