HEAPS, PRIORITY QUEUES AND HEAPSORT. HUFFMAN CODING

Chapter 2.4 and 9.3 of GT

DISCLAIMER: These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2016; distribution outside this group of students is NOT allowed.
Few more sorting algorithms

Introduction

In the beginning of this class we introduced a couple of sorting algorithms

- Insertion-Sort, an $O(n^2)$ running time algorithm.
- Selection-Sort, an $O(n^2)$ running time algorithm.
- Bubble-Sort, an $O(n^2)$ running time algorithm.
- Merge-Sort an $O(n \lg n)$ running time algorithm.

In the following lectures we are going to introduce a couple of new sorting algorithms as well as data structures. The algorithms we are going to introduce are

- An algorithm called heapsort with $O(n \lg n)$ worst-case running time that uses a new data structure called a heap.
- An algorithm called quicksort whose average case running time is $O(n \lg n)$ and although its worst-case running time is $O(n^2)$ in practice it performs better than any other sorting algorithm.
- Linear time sorting algorithms that work when the data type of the input is specific.

The data structures we will introduce are

- A priority queue (PQ) $S$ that implements efficiently the following operations: FindMax($S$), Insert($S,x$), RemoveMax($S$). We can also have a PQ $T$ with the following operations FindMin($T$), Insert($T,x$), RemoveMin($T$) implemented similarly.
- The data structure called heap that allows a PQ to be implemented efficiently.

We note that the notion of a heap as defined here has nothing to do with the same term used in compilers to mean an alternative memory area to a stack, where memory is collected when not in use and returned to the running program.
Heaps
An introduction

A (binary) heap is a data structure that has the following properties.

1. Each element of a heap has some priority attached to it.

2. A heap is represented by an array $A[\ldots]$ of length $l(A)$. The contents of $A[i]$ contain the priority of the $i$-th element of the heap. A pointer $A[i].pointer$ may point to a memory area storing other useful information about this element.

3. The size of a heap is denoted by $\text{heapsize} \leq l(A)$. We can forfeit the use of $\text{heapsize}$ if instead we maintain a pointer/reference to the last element of a heap (see GT).

4a. For every node indexed $i$ of the heap we have that $A[i] \geq A[2i+1]$ and $A[i] \geq A[2i+2]$, if the two elements belong to the heap. This is equivalent to $A[\text{floor}((i-1)/2)] \geq A[i]$. We call this property the MaxHeap-Order property.

4a. For every node indexed $i$ of the heap we have that $A[i] \leq A[2i+1]$ and $A[i] \leq A[2i+2]$, if the two elements belong to the heap. This is equivalent to $A[\text{floor}((i-1)/2)] \leq A[i]$. We call this property the MinHeap-Order property.

In the remainder we will be dealing with Max-Heaps (i.e. heaps that maintain the MaxHeap-Order property), although the result we will obtain can be easily translated into equivalent results for Min-Heaps.

Another equivalent representation of a (binary) heap is by a complete binary tree whose all levels but the lowest (with highest depth) are filled with elements and in the last level nodes are filling the tree left to right. Nodes of the tree are numbered top-to-bottom, left-to-right starting from the root (first element with label 1). A parent labeled $i$ has at most two children labeled $2i+1$ and $2i+2$ if they both exist (this is true for an array whose first element is $A[1]$). The equivalence between the array and the tree representation is immediate.

Given the tree representation.

a. parent $(i)$ return $\text{floor}((i-1)/2)$;

b. left $(i)$ return $(2 \times i + 1)$;

c. right $(i)$ return $(2 \times i + 2)$;

Such operations in C/C++ can be implemented with shift operations efficiently.
Heaps
Representation

Array $A$ below is a maxheap. Figure (1) shows the array as a heap-tree that has the MaxHeap-property. Figure (2) is not a heap as nodes indexed 10 and 11 are not filled with elements (the node indexed 9 holds a 5, and the node index 12 holds a 6: there is a gap, i.e. nodes in last level not filled left to right). In Figure (3) in brackets we indicate the index of a node that is the same as the index of that element within array $A$.

$$A = [20, 15, 18, 8, 12, 7, 1, 3, 5, 6]$$

\[
\begin{array}{c}
20 \\
/ \ \\ 
15 \ 18 \ heap \\
/ \ / \\
8 \ 12 \ 7 \ 1 \\
/ \ / \\
3 \ 5 \ 6
\end{array}
\]

Figure (1)

\[
\begin{array}{c}
20 \ [0] \\
/ \\
[1] \ 15 \ 18 \ [2] \\
/ \ / \\
/ \ / \\
\end{array}
\]

Figure (3)
**Heap Operations**

**Description**

Sometimes the tree of the heap that is filled with nodes top to bottom and left to right is referred to as a complete binary tree. Sometimes, the tree representation of a heap (i.e. a complete binary tree with the MinHeap-order or MaxHeap-order property) is referred to as the MinTree or MaxTree (compare to MinHeap and MaxHeap).

The **height** of a tree node, is the size (in edges) of the longest downward path from the node to a descendant leaf of the tree. (or equivalently, it is the height of the tree minus the depth of the node). The height of a tree is the height of the root. A heap with $n$ elements has height $\lceil \log n \rceil$. A heap with $n$ elements has $\lceil \log (n + 1) \rceil$ levels (the root is at level 1 and so on).

Some operations defined on heaps are the following ones.

**Shift/Sift/DownHeap/Heapify** ($A,i$) maintains the heap-order property at the node indexed $i$. The left and right subtree of that node MUST maintain the heap-order property; the tree rooted at $i$ may not maintain the tree property. Depending on the heap we use we can define MaxDownHeap or MinDownHeap. In some other textbooks the DownHeap operation is called **Sift** or **Shift Fix**, **Heapify**.

**Build-MaxHeap** ($A$). Given an unordered array $A$ a heap is built in place.

**RemoveMax** ($A$). Remove the element with the maximum value of the heap; after the removal the heap-property is maintained for the remainder. For a min-heap we can equivalently define **RemoveMin** ($A$).

**Insert** ($A,x$). An element $x$ is inserted into the key.

**HeapSort** ($A$). Given a (possibly unordered) array $A$ this procedure sorts $A$ in place.
Heap operations
MaxDownHeap

When MaxDownHeap is called on the element indexed \( i \) the implicit assumption is that the subtrees rooted at \( \text{left}(i) \) and \( \text{right}(i) \) are heaps; \( A[i] \) however may fail the heap-order property. The purpose of DownHeap is to slide down this key so that the subtree rooted at \( i \) becomes a heap.

```plaintext
// Subtrees at left(i) and right(i) are heaps
// and thus have MaxHeap-Order property
1. largest = index of the maximum key among
   A[i], A[left(i)], A[right(i)];
2. if (largest != i) {
   3. exchange(A[i], A[largest]);
   4. MaxDownHeap(A, largest); //recursively continue
}
```

Analysis. A MaxDownHeap operation takes constant time for steps 1, 2, and 3. Line 4 is run at most \( H \) times where \( h \) is the height of the heap. A heap with \( n \) nodes has height \( H = O(\lg n) \). In general if the height of node labeled \( i \) is \( h \leq H \), then the time for DownHeap is \( T(h) \)

\[
T(h) = T(h - 1) + \Theta(1) \quad \text{and thus, } T(h) = O(h) = O(H) = O(\lg n).
\]
Heap Operations

Build-MaxHeap

//Iterative BuildMaxHeap
Build-MaxHeap(A)
1. heapsize(A)=l(A);
2. for(i=floor((l(A)-1)/2);i>=0;i--)
   { MaxDownHeap(A,i); }

//Recursive BuildMaxHeap
RBuild-MaxHeap(A)
1. heapsize(A)=l(A);
2. if (left(i) < heapsize(A))
   { RBuild-MaxHeap(A,left(i)); }
3. if (right(i) < heapsize(A))
   { RBuild-MaxHeap(A,right(A)); }
4. MaxDownHeap(A,i);

The input to Build-MaxHeap is an unordered array $A$. The outcome of running Build-MaxHeap is to create a MaxHeap out of an unordered array $A$. For an array of size $n$ half its elements already maintain the heap-order property (elements indexed $(l(A) - 1)/2$ through $n - 1$). This is so because these elements do not have any children and thus have the heap-order property by default. Therefore the elements that may not maintain the heap property are the remaining elements located in positions 1 through $\lfloor l(A)/2 \rfloor$.

The heap construction process moves from the lowest level towards the top of the tree (binary tree representation of a heap). Therefore subtrees rooted at successive levels maintain the heap-order property, and finally the whole tree does.

Analysis. A naive analysis shows that each of $O(n/2)$ elements spends time $O(h) = O(\lg n)$ in line 3 for a total time $O(n \lg n)$. A tighter bound follows from the simple observation that nodes lower on the tree (where the tree is fatter – more populous) require fewer iterations of MaxDownHeap as their height is smaller. The number of nodes at height $h$ is $\text{ceiling}(n/2^{h+1})$ and the time spent on line 3 is $O(hn/2^{h+1})$. The total running time is then

$$O\left(\sum_{h=0}^{\lg n} hn/2^{h}\right) = O\left(n \cdot \sum_{h=0}^{\lg n} h(1/2)^{h}\right) = O\left(n \cdot \sum_{i=0}^{\lg n} i(1/2)^{i}\right) = O(n)$$

where we used the fact (See Appendix) that

$$\sum_{i=0}^{\infty} ix^i \leq x/(1 - x)^2$$

and substitute $h$ for $i$ and $1/2$ for $x$ to give $\sum_{i=0}^{\infty} i(1/2)^i \leq 2$.

Exercise. Analyze the running time of RBuild-MaxHeap. Derive $T(h) = 2T(h - 1) + h$ (i.e. use height information rather than size information) for its running time.

(c) Copyright A. Gerbessiotis. CS610 : Spring 2016 . All rights reserved.
HeapSort
Introduction

The heapsort algorithm utilizes Build-MaxHeap to build a heap from the input keys into an array $A$ of size $n$. The key with the maximum value is stored in the top of the heap (root of heap tree or key in index position 1 of array $A$), and thus it can be removed from the heap and placed in its correct position in the output sequence (the last position). This is achieved by exchanging $A[1]$ and $A[n]$. This exchange, may violate the heap-order property for elements $A[1\ldots n-1]$ (note that the removal of max makes the heap of size one less). It is obvious however that the children of $A[1]$ have the heap-order-property though, as the rest of the heap was untouched by the exchange. A call to MaxDownHeap will maintain the heap-order property for $A[1]$ and the resulting $A[1..n-1]$ becomes a heap. We continue similarly with the removal of the current maximum key from the heap $A[1..n-1]$.

```
Heapsort(A,n)   // A is an unordered array of n keys
1. Build-Max-Heap(A);  // A has become a heap
2. for(i=n-1;i>=1;i--){   // A[0..i] is a heap A[i+1..n-1] is a sorted subarray
3. exchange(A[0], A[i]); // Remove Max and place it into A[i], move A[i] to A[0]
   // A[0..i-1] may not be a heap, A[i..n-1] is a sorted subarray
4. heapsize(A) = heapsize(A)-1; //one key was removed from heap; adjust size
5. MaxDownHeap(A,0);  //Root may not have Heap Property; Run DownHeap
}
```

Step 1 requires time $O(n)$. The loop of lines 2-5 is executed $n-1$ times and each call to MaxDownHeap takes $O(lg n)$ steps. The total running time is thus $O(n \cdot lg n)$ in the worst case.

Note. The algorithms for HeapSort described here is the one corresponding to the in-place algorithm of section 2.4.4 of GT.
Priority Queue
Introduction

The heap data-structure finds however many application in operating system design (scheduling) and (event-driven) simulation. It implements a highly efficient priority queue data structure.

A **priority queue** is a data-structure for maintaining a set \( S \) of elements each with an associated value/priority called a **key**. A priority queue supports the following operations.

- **Insert** \((S,x)\) inserts element \( x \) into queue \( S \).
- **FindMax** \((S)\) returns the element of \( S \) with the highest key value or priority (the element is not removed from \( S \)). This is the Search operation for a priority queue.
- **RemoveMax** \((S)\) removes the element of \( S \) with the highest key value (the element is removed from \( S \)). This is the Delete operation for a priority queue.

We can also define a priority queue with FindMin and RemoveMin operations.

0. The priority queue data-structure can be implemented using a max-heap or a min-heap as appropriate.
1. Operation **FindMax** \((S)\) can be implemented easily by having a heap return its first element \( A[0] \).

**FindMax**\((A)\)
   return\((A[0])\);
2. The **RemoveMax(S)** operation works similarly with the exception that it removes an element from the heap. Its operation is also similar to heapsort.

   RemoveMax(A)
1. if heap-size(A) < 1
   return("Error. No heap elements");
2. max=A[0];
4. heap-size(A) = heap-size(A)-1;
5. MaxDownHeap(A,0);
6. return(max);

   The running time of RemoveMax is $O(\lg n)$.

3. Insert-MaxHeap inserts a node into a heap/priority queue. It first expands a heap by adding a leaf to the tree. Then similarly to the insertion operation of Insertion-Sort it traverses a path from this leaf toward the root to find the proper place for the inserted element.

   Insert-MaxHeap(A,key)
1. heap-size(A) = heap-size(A)+1;
2. i=heap-size(A)-1;
3. while ((i>0) and A[parent(i)] < key) {
   5. i=parent(i);
   }  
6. A[i]=key;

   The running time is $O(\lg n)$ for inserting a single element in a heap or priority queue.
Text Compression

Introduction

In the remainder we deal with yet another application of priority queues, that of text compression. In text compression we are given a string \( T \) of characters drawn from some alphabet \( \Sigma \) and we want to efficiently encode string \( T \) into a smaller (binary) string \( Y \). The way compression is performed is by first assigning a distinct bit sequence to each element of \( \Sigma \) and then storing the characters of the string using this encoding consecutively in computer memory. Since \( n \) bits have \( 2^n \) distinct values an alphabet with \(|\Sigma|\) characters requires \( \lg|\Sigma| \) bits. If each character is equally likely to appear in the documents stored in a computer not much can be done. In real-life however this is not the case. With English certain letters are more likely to appear than others (e.g. \( e \) is more frequent than \( w \) or \( x \)).

The techniques for compression that we will examine are lossless i.e. for a given \( Y \) we can fully recover \( T \) from it. There are also noisy or lossy techniques where full-recovery of \( Y \) from \( T \) is not possible. Such techniques can be used in image or voice compression. The process that converts \( T \) into \( Y \) is called encoding or compression.

The primary method that will be presented in this subject is Huffman encoding that yields what is known as Huffman codes. Huffman codes result in space savings of 20% to 90% depending on the structure of the input string or file.

Huffman’s algorithm/encoding is a greedy algorithm that during its course of execution makes that choice that looks best at that instant.

Huffman encoding relies on the following observation/example. Suppose a string consists not of all ASCII characters but only of few of them, say the numerals (0..9) plus few other characters (delimiters such as space, tab, newline, carriage return), then we can represent each character not with 7 or 8 bits (i.e. ASCII) but with only 4 bits (16 different values). Huffman encoding requires that we know in advance the alphabet \( \Sigma \) and also the frequency of each character. Each character is represented by a different encoding string or binary character code (or code in short) in such a way that no encoding string is a prefix of another encoding string. Codes that have this property are called prefix codes.

The important property of prefix codes is that decoding is quite simple. Since no codeword is a prefix of another one, that codeword at the start of a string or a file is unambiguous and can be removed from the coded file; repeating this process decodes the rest of the string or file uniquely.
Huffman encoding
Prefix codes and prefix trees

We are going to represent prefix codes by binary trees. For each string or file the characters used are collected in set $C$. The frequency $f(c)$ of any character $c$ is also found. Each node of the tree has a left child that represents a zero and a right child that represents an one. Each leaf is a character that appears in the string or file, along with its frequency. A path from the root to a leaf "character" represents the code for that character where every left child traversal corresponds to a zero and every right child one to an one. The binary tree representing a prefix code is full with $|C| - 1$ internal children where $|C|$ is the number of unique leaves/characters that appear in the string/file. An internal node of the tree also has a frequency field. The frequency field of an internal node is the sum of the frequencies of the leaves in the subtree of this node.

The number of bits required to encode the file with a given prefix code tree is $B(T) = \sum_{c \in C} f(c)d(c)$, where $d(c)$ is the depth of character $c$ in the tree, and $f(c)$ is the frequency of character $c$ in the string/file that was used to obtain the tree.

The algorithm that constructs a Huffman code (i.e. an optimal prefix code) is a greedy algorithm. It builds the tree that corresponds to the code bottom to top.

Initially the $|C|$ characters form individual one-node trees. The single node is the root of its own tree. The information associated with each node is the character representation (e.g. 'A') of the node and the frequency of the character in the string or file. The algorithm works bottom-up by combining two trees into a single one by creating a root node for the new tree whose left child is the root of one of the two nodes and the right child the other node. The frequency field of the new root is the sum of the frequencies of the roots of the two constituent trees (now, children). There is no character representation associated with such an internal node. Therefore after $|C| - 1$ such operations there is only one tree left whose frequency field is the length of the string or file.

The criterion for choosing the two trees to combine is a "greedy" one: the two trees with the lowest frequency fields at their respective roots are chosen.

We use a priority queue to maintain the trees, and implement the priority queue with a MINHEAP. The priority of a tree is the frequency of its root.
Huffman codes

Encoding

The following pseudocode shows the steps for encoding a file and constructing the Huffman code.

```
Encode (uncompressed file)
1. Read file and obtain character array C with frequency of each character i stored in C[i].freq
2. Huffman(C[]);
3. Traverse Tree to obtain prefix codes for each i in C[].
4. Read file replacing character i with a bitstring symbol C[i].symbol
5. Save prefix code table in file.
```

The following pseudocode shows the steps of Huffman’s algorithm.

```
Huffman (C[ ])//Every element of C has a frequency field freq.
0. priorityqueue Q;
1. for(c=0;c<|C|;i++){
2.  create single node bin. tree T: T[c].char=c, T[c].freq=C[c].freq
3.  Q[c] =T[c];
4. }
5. Build-MinHeap(Q,|C|);
6. for(i=0;i<n-1;i++) {
7.   z=ALLOCATENODE();
8.   x=z.left = EXTRACT-MIN(Q);
9.   y=z.right = EXTRACT-MIN(Q);
10.  z.freq = x.freq + y.freq;
11.  z.char = -1;
12.  InsertMinHeap(Q,z);
13. }
14. return(RemoveMin(Q));
```
Huffman’s Algorithm
Decoding

The running time of Huffman is $O(n \lg n)$, where $n = |C|$. Lines 0-4 require $O(|C|) = O(n)$ time. Line 5 (Build-MinHeap operation) requires $O(n)$ time and lines 6-12 require $O(\lg n)$ time per iteration for a total of $O(n \lg n)$ time.

The decoding operation traverses the tree root to leaf using the bitstring of the string /file.

```
HuffmanDecode (compressed file, HuffmanTree T, bitstring B, currentposition pos)
0. Obtain T from file;
1. pointer = T;
2. while (B[pos] != EOF) {
3. if (B[pos] == 0)
4. pointer = pointer.left;
5. else
6. pointer = pointer.right;
7. if (isleaf(pointer) == TRUE) { // ie. pointer.char != -1
8. Output(pointer.char);
9. pointer = T;
10. }
11. }
}
```

The decoding procedure works in time linear to the size in bits of the encoded/compressed file.
Huffman’s Algorithm
Correctness: Part 1

**Theorem Huffman.** Let $C$ be an alphabet in which each character $c$ has frequency $f[c]$. Let $x$ and $y$ be two characters of $C$ with the lowest frequencies. Then there exists an optimal prefix code for $C$ in which $x$ and $y$ have same length codewords that only differ in the last bit.

**Proof.** We start with an optimal but arbitrary prefix code and modify it to obtain another optimal prefix code where $x$ and $y$ are maximum depth sibling in the new tree (i.e. codewords have same length and differ in last/rightmost bit).

Let in the arbitrary optimal prefix code $b$ and $c$ are the siblings of maximum depth. Without loss of generality let $f[b] \leq f[c]$ and also, $f[x] \leq f[y]$. Since $f[x], f[y]$ are lowest leaf frequencies, then $f[b], f[c]$ must be such that $f[x] \leq f[b]$ and $f[y] \leq f[c]$.

We obtain tree $T'$ by exchanging in $T$ $x$ with $b$ and $T''$ by exchanging $y$ with $c$ in $T'$.

Then we have

$$B(T) - B(T') = \sum_c f(c)d_T(c) - \sum_c f(c)d_{T'}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x)$$

$$= (f(b) - f(x))(d_T(b) - d_T(x))$$

$$\geq 0.$$

Note that since $f(x)$ is minimum frequency leaf $f(b) - f(x) \geq 0$. Since $b$ is maximum depth leaf in $T$ $d_T(b) \geq d_T(x)$.

We can similarly prove that $B(T') - B(T'') \geq 0$. Therefore $B(T'') \leq B(T)$. Since $T$ is optimal $B(T) \leq B(T'')$. Therefore $B(T) = B(T'')$. 

(c) Copyright A. Gerbessiotis. CS610 : Spring 2016. All rights reserved.
Huffman's Algorithm
Correctness: Part 2

**Theorem Huffman2** Let $T$ be a full binary tree representing an optimal prefix code for $C$ with frequencies $f[c]$. Consider two sibling leaves $x$ and $y$ in $T$ and let $z$ be their parent. If $z$ is considered a character with $f[z] = f[x] + f[y]$, the tree $T' = T - \{x, y\}$ represents an optimal prefix code for $C' = C - \{x, y\} \cup \{z\}$.

**Proof.** For every $c$ in $C$ other than $x, y$ we have $d_T(c) = d_{T'}(c)$. For $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_T(z) + 1) = f(z)d_{T'}(z) + f(x) + f(y).$$

Then, $B(T) = B(T') + f(x) + f(y)$. If $B(T')$ is not optimal then there is an optimal tree $O$ such that $B(O) < B(T')$, and its leaves are characters in $C'$. Then $z$ appears as a leaf in $O$. If we add $x, y$ in $O$ we get a prefix code whose total cost is $B(O) + f(x) + f(y) < B(T) = B(T') + f(x) + f(y)$. This, however, contradicts the minimality of $B(T)$. Therefore such an $O$ can not exist because $T'$ is the optimal prefix code for $C'$.

**Theorem Huffman-Algorithm** Huffman produces an optimal prefix code.

**Proof.** A direct result of Theorems Huffman and Huffman2.
Appendix
An equation derivation

We already know the first two derivations: the first one has $n$ terms, the second one has infinite term. In the latter one the right-hand side is to the limit $n \to \infty$. The third one is to be derived from the second one.

\[
\begin{align*}
\sum_{i=0}^{n-1} x^i &= 1 + x + \ldots + x^i + \ldots + x^{n-1} = \frac{x^n - 1}{x - 1} & x \neq 1 \\
\sum_{i=0}^{\infty} x^i &= 1 + x + \ldots + x^i + \ldots = \frac{1}{1 - x} & 0 < x < 1 \\
\sum_{i=0}^{\infty} i \cdot x^i &= 1 \cdot x^1 + 2 \cdot x^2 + \ldots + i \cdot x^i + \ldots = ? & 0 < x < 1
\end{align*}
\]

Starting, from the second equation we work as follows; (a) take the first derivative of both sides, (b) multiply the result (both sides) by $x$. We get:

For any $0 < x < 1$, we have.

\[
\begin{align*}
\sum_{i=0}^{\infty} x^i &= \frac{1}{1 - x} \\
1 + x + \ldots + x^i + \ldots &= \frac{1}{1 - x} \\
(1 + x + \ldots + x^i + \ldots)' &= \left(\frac{1}{1 - x}\right)' \\
0 + 1 \cdot x^0 + \ldots + i \cdot x^{i-1} + \ldots &= \frac{1}{(1 - x)^2} \\
0 \cdot x + 1 \cdot x^1 + \ldots + i \cdot x^i + \ldots &= \frac{x}{(1 - x)^2} \\
\sum_{i=0}^{\infty} i \cdot x^i &= \frac{x}{(1 - x)^2}
\end{align*}
\]

If we substitute $x = 1/2$ we obtain

\[
\sum_{i=0}^{\infty} i/2^i = 2
\]
Appendix
An alternative analysis of RBuild-MaxHeap

The RBuild-MaxHeap works in a strange way. Let $i$ be the index of the element $A[i]$ in the heap array and let $L(i)$ be the elements in the heap-array that belong to the left subtree of the $i$-indexed node and $R(i)$ the elements in the heap-array that belong to the right subtree of the $i$-indexed node. We call $l(i)$ and $r(i)$ the left, right children of the node indexed $i$. So $L(.)$ and $R(.)$ are sets of nodes, whereas $l(.)$, $r(.)$ are nodes.

In the call of line 2 of RBuild-MaxHeap $i = 0$ and RBuild-MaxHeap is entered with $i = 0$. Line 2 of RBuild-MaxHeap builds recursively a MAX-heap out of the $L(i) = L(0)$ nodes, line 4 builds a MAX-heap of the $R(i) = R(0)$ nodes, and now AFTER $L(i) = L(0)$ and $R(i) = R(0)$ have been turned into heaps, it is possible to apply DownHeap at node $i = 0$ to build a MAX-heap out of every node of the tree.

The key to this algorithm is that DownHeap is applied AFTER $L(i)$ have formed a max-heap and after $R(i)$ have also formed a max-heap. Note that if line 5 was the first line of the code, and not the fifth this would NOT have been a CORRECT Build-MaxHeap procedure since in order to apply DownHeap at node indexed $i$, the left subtree and right subtrees MUST BE HEAPS. The latter is guaranteed however after the execution of lines 2 and 4 of RBuild-MaxHeap. Therefore the code given is a recursive Build-Heap method complementing the iterative.

Do they have the same running time? Let’s analyze the running time of RBuild-MaxHeap for a general node indexed $i$. Let the height of node $i$ be $h$ and the running time be expressed in terms of $T(h)$ (this is $h$ not $n$). Then, $l(i)$ and $r(i)$ are of height one less and thus the running time of RBuild-MaxHeap will be $T(h - 1)$ and $T(h - 1)$ respectively. The running time of DownHeap is proportional to the height of node $i$, i.e. it is $O(h)$. Combining all the pieces together we have

$$T(h) = 2T(h - 1) + ch$$

where we write for simplicity the $O(h)$ as $ch$. 

(c) Copyright A. Gerbessiotis. CS610 : Spring 2016 . All rights reserved. 18
Appendix
An alternative analysis of RBuild-MaxHeap

What is the solution to this recurrence? We can use the substitution method or the iteration method to show that $T(h) = O(2^h)$. Let’s try the iteration method. For base case we are going to choose $T(0) = 0$.

$$
T(h) = 2T(h-1) + ch
$$

$$
= 2(2T(h-2) + c(h-1)) + ch
$$

$$
= 2^2T(h-2) + 2^1c(h-1) + 2^0c(h-0)
$$

$$
= 2^2(2T(h-3) + c(h-2)) + 2^1c(h-1) + 2^0c(h-0)
$$

$$
= 2^3T(h-3) + 2^2c(h-2) + 2^1c(h-1) + 2^0c(h-0)
$$

$$
= 2^hT(0) + 2^{h-1}c(h-(h-1)) + \ldots + 2^ic(h-i) + \ldots + 2^1c(h-1) + 2^0c(h-0)
$$

$$
= c \sum_{i=0}^{h-1} 2^i(h-i)
$$

$$
= c \sum_{i=0}^{h-1} 2^ih - c \sum_{i=0}^{h-1} i2^i
$$

$$
= ch(2^h - 1) - c \sum_{i=0}^{h-1} i2^i
$$

$$
= ch(2^{h+1} - 1) - c((h-1)2^{h+1} - (h)2^h + 2)
$$

$$
= c2^{h+1} - ch + 2c.
$$

We used above the base case $T(0) = c$. We also identified the sum $\sum_i i2^i$, as the sum of pages 17/18 of this appendix and also of the formulae collection (available as Handout 5), third line (rightmost), if we set $a = 2$, and $n = h$ there. That formulae can be derived with the methods of the appendix. Thus $\sum_{i=0}^{h-1} i2^i = (h-1)2^{h+1} - (h)2^h + 2$.

For a heap with $n$ elements, the height of the root, i.e. the element indexed $i = 1$, is at most $\log n$. Thus $T(h)$ is $O(2^{h+1}) = O(n)$. This means that the running time of RIGHT-BuildMAX-Heap is $O(n)$, just like the running time of the non-recursive variant presented in class.

**Question.** Why did we use in the running time analysis $T(h)$ instead of $T(n)$? For $i = 1$, how big (i.e. how many nodes does it contain) is the left-subtree $L(0)$ of the root? How big is the right subtree of the $R(0)$ of the root?